ECE 219 Project3

February 21, 2024

1 ECE 219 Project 3

1.1 Team: Kyle Wang(406087325), Nina Cheng(505950945), Sophia Yang(006183357)

[ ]:
```
! pip install gdown
! pip install surprise
```

[ ]:
```
import gdown
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from surprise import Dataset, Reader, accuracy
from surprise.model_selection import cross_validate, train_test_split
from surprise.prediction_algorithms.knns import KNNWithMeans
from sklearn.metrics import roc_curve, auc
```

[ ]:
```
url = "https://drive.google.com/drive/folders/1_JF9plSjE3PAFBuSvUFRkDdftJWo1TFz?
 ↪usp=drive_link"
gdown.download_folder(url, quiet=True, use_cookies=False)
```

[ ]:
```
['/content/Synthetic_Movie_Lens/links.csv',
 '/content/Synthetic_Movie_Lens/movies.csv',
 '/content/Synthetic_Movie_Lens/ratings.csv',
 '/content/Synthetic_Movie_Lens/README.txt',
 '/content/Synthetic_Movie_Lens/tags.csv']
```

1.2 Question 1

1.2.1 A

The sparsity of the movie rating dataset is **0.016999683055613623**

[ ]:
```
data_folder = 'Synthetic_Movie_Lens/'
ratings_df = pd.read_csv(data_folder + "ratings.csv", usecols=['userId',␣
 ↪'movieId', 'rating'])
U_ID = ratings_df.pop('userId').values
M_ID = ratings_df.pop('movieId').values
rating = ratings_df.pop('rating').values
```

```
sparsity = len(rating) / (len(set(M_ID)) * len(set(U_ID)))
print('Sparsity:', sparsity)
```

```
Sparsity: 0.016999683055613623
```

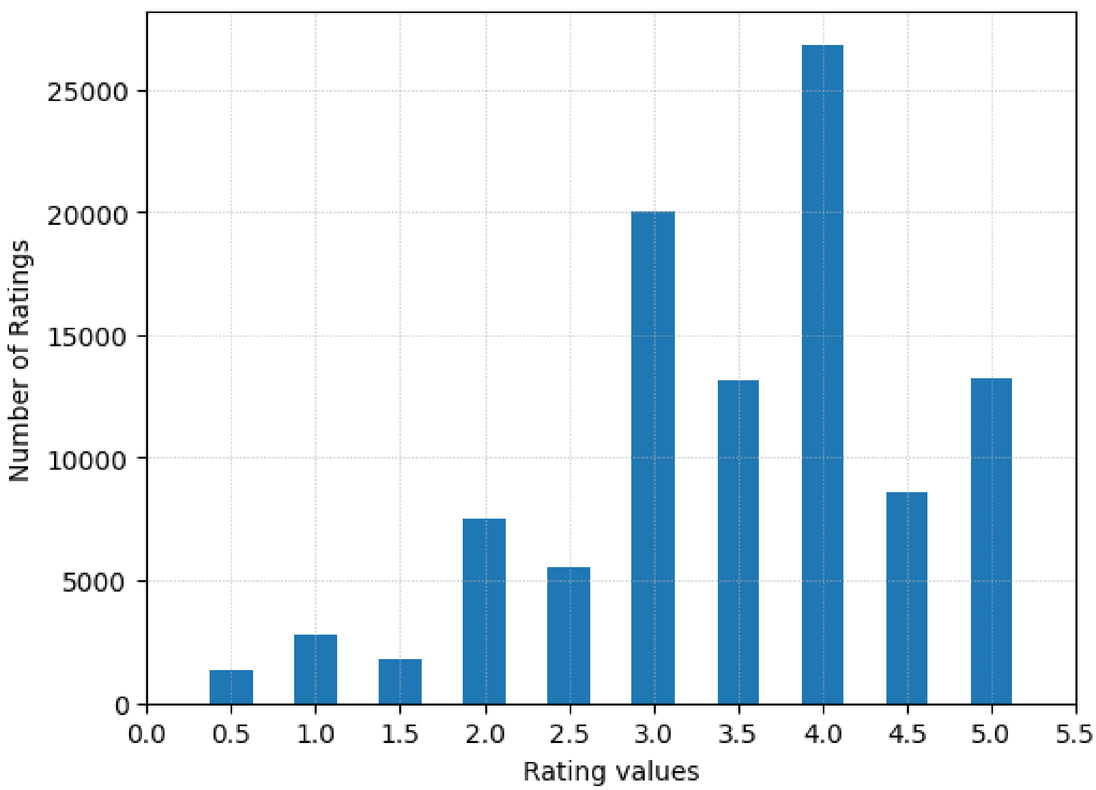
1.2.2 B

This histogram showing the frequency of the rating for each rating interval. Here are the key observations from the graph:

Central Tendency: The ratings are centered around the higher values, with 4.0 being the most common rating. This suggests that users tend to give higher ratings more frequently, which could imply that users are more likely to rate movies that they enjoyed.

Skewness: The distribution appears to be left-skewed, meaning there are fewer low ratings and a longer tail on the lower end of the rating scale. This kind of skewness often indicates that users who decide to rate movies tend to rate movies they feel positively about.

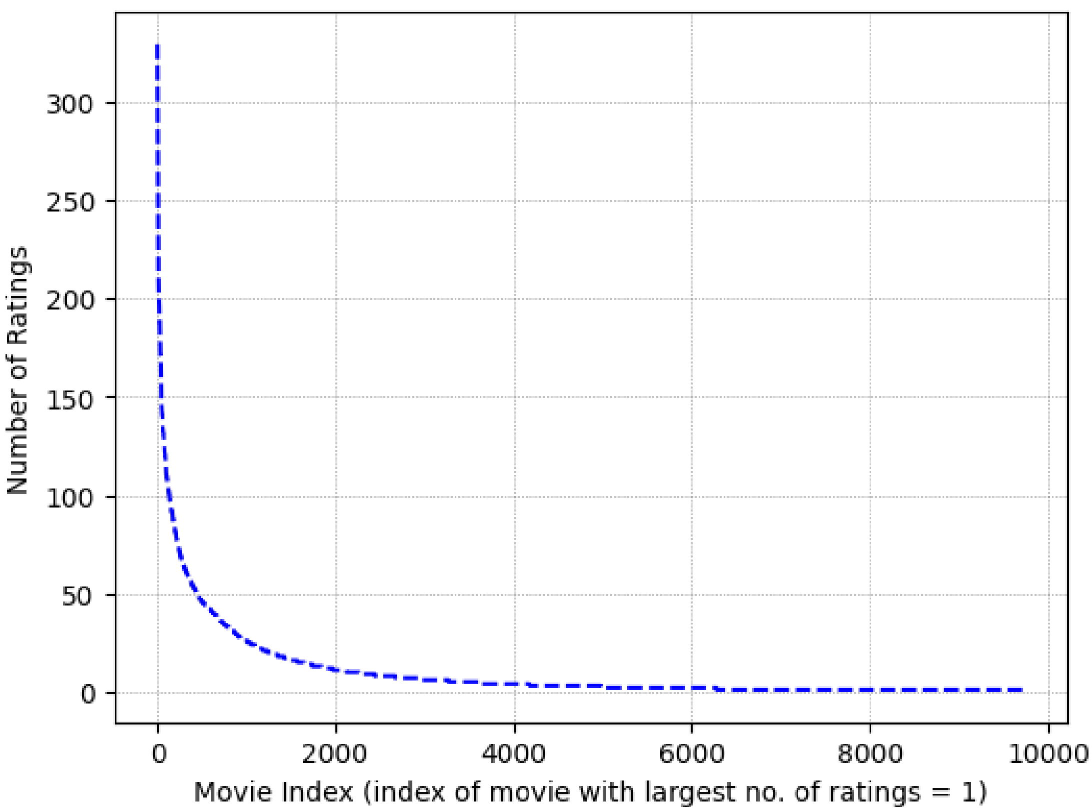
[ ]:
```
unique_vals, inv_indices = np.unique(rating, return_inverse=True)
fig, ax = plt.subplots()
ax.bar(unique_vals, np.bincount(inv_indices), width=0.25)
ax.set_xticks(np.arange(0, 6, 0.5))
ax.set_ylabel('Number of Ratings')
ax.set_xlabel('Rating values')
ax.grid(which='major', linestyle=':', linewidth='0.5')
plt.show()
```



1.2.3 C

The distribution of the number of ratings received among movies. A monotonically decreasing trend is observed.

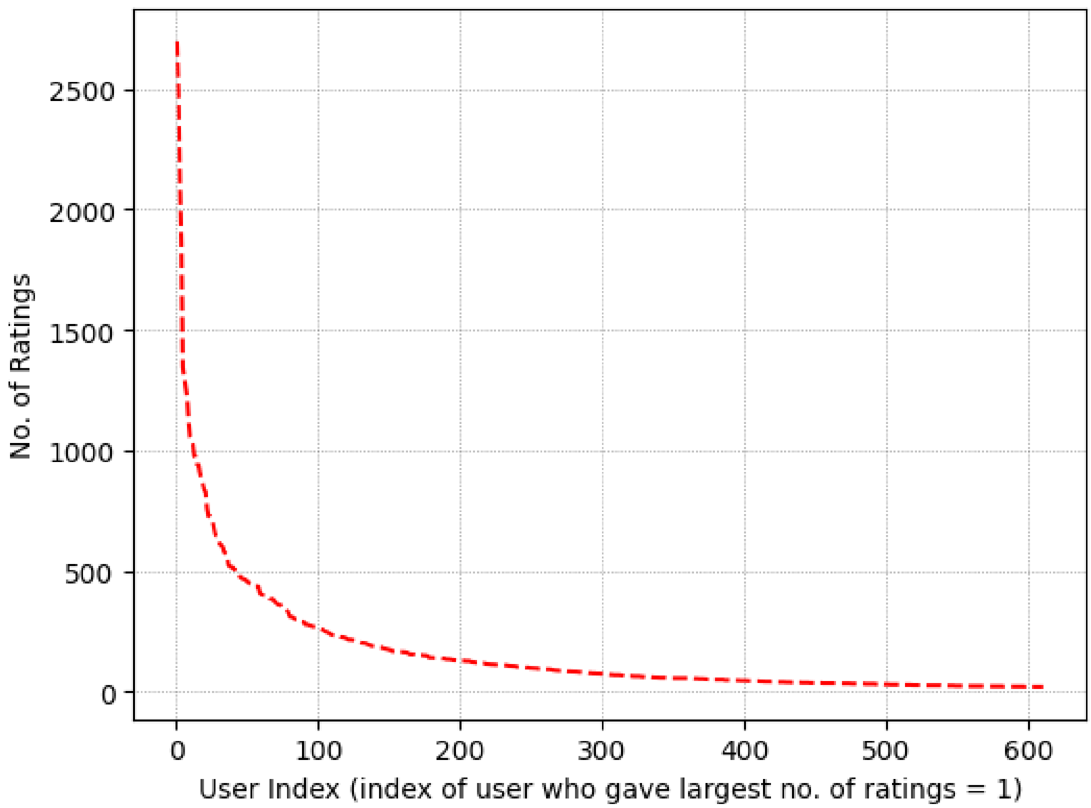
```
[ ]: distinct_movies, rating_counts = np.unique(M_ID, return_counts=True)
     sorted_indices = np.argsort(-rating_counts)
     fig, ax = plt.subplots()
     ax.plot(np.arange(1, len(distinct_movies) + 1), rating_counts[sorted_indices],
      ↪'--b')
     ax.grid(True, which='both', linestyle=':', linewidth='0.5', color='gray')
     ax.set_xlabel('Movie Index (index of movie with largest no. of ratings = 1)')
     ax.set_ylabel('Number of Ratings')
     plt.show()
```



1.2.4 D

The distribution of the number of ratings received among users.

```
[ ]: distinct_users, user_rating_counts = np.unique(U_ID, return_counts=True)
sorted_user_indices = np.argsort(-user_rating_counts)  # Sort indices by␣
 ↪descending order of counts
fig, ax = plt.subplots()
ax.plot(np.arange(1, len(distinct_users) + 1),␣
 ↪user_rating_counts[sorted_user_indices], '--r')
ax.grid(True, which='both', linestyle=':', linewidth='0.5', color='gray')
ax.set_xlabel('User Index (index of user who gave largest no. of ratings = 1)')
ax.set_ylabel('No. of Ratings')
plt.show()
```



1.2.5 E

The curve from graph in C is monotonically decreasing. We also see that only about 500 out of 9700 movies get more than 50 ratings each. The simialr pattern can be observed in graph in D. This means most movies have very few ratings, leading to what's called a sparse matrix – a situation where there's not enough data for many movies.

From a machine learning point of view, sparse data is tricky. It makes it hard for models to learn and classify correctly because they need more data to understand the patterns. This problem is known as the "Curse of Dimensionality."

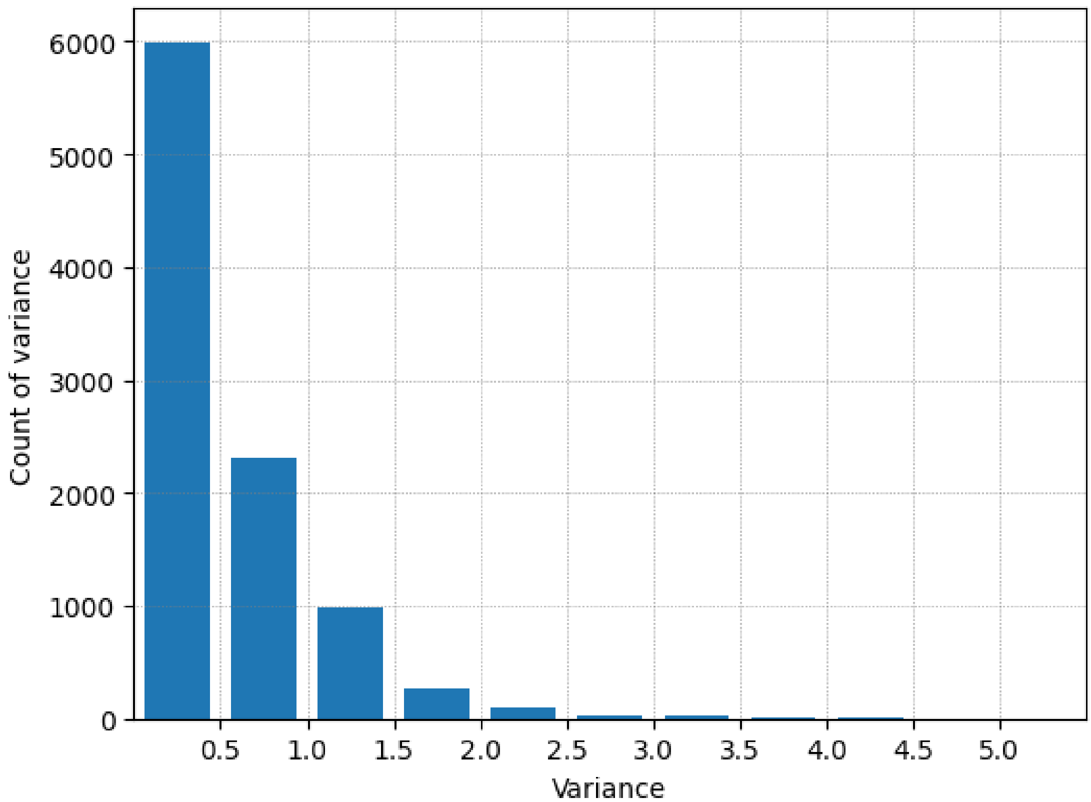
To deal with this, machine learning models often use a technique called regularization. This helps the model to not just memorize the few high-rated movies but to learn more generally, so it can predict well even for movies that don't have many ratings.

1.2.6 F

The histogram below shows that most movies have a variance in ratings between 0 and 1.5, indicating that ratings are generally consistent across different users. This is likely because people often choose movies based on popular reviews, which leads to similar opinions and ratings. The consistency in ratings is also seen in Figure from A, where most ratings fall between 3 to 5.

[ ]:
```
import numpy as np

unique_movie_ID = set(M_ID)
variance_dict = {movie: np.var([rating[i] for i, mid in enumerate(M_ID) if mid␣
↪== movie]) for movie in unique_movie_ID}
movie_ID_list, var_list = zip(*variance_dict.items())
movie_ID_list = list(movie_ID_list)
var_list = list(var_list)
```

[ ]:
```
fig, ax = plt.subplots()
ax.hist(var_list, bins=np.arange(0, 5.5, 0.5), rwidth=0.75)
ax.set_xticks(np.arange(0.5, 5.5, 0.5))
ax.set_xlim([0, 5.5])
ax.grid(True, which='both', linestyle=':', linewidth='0.5', color='grey')
ax.set_xlabel('Variance')
ax.set_ylabel('Count of variance')
plt.show()
```



1.3 Question 2

1.3.1 A

$$\mu_u = \frac{\sum_{k \in I_u} r_{uk}}{|I_u|}$$

1.3.2 B

In plain words, the term $I_u \cap I_v$ represents the set of movies that both user u and user v have rated. This intersection is crucial for calculating similarities between users in recommendation systems, as it identifies the common ground upon which comparisons can be made. When this intersection is empty, denoted by $\emptyset$, it means that there are no movies that both users have rated. This situation can occur in datasets like MovieLens where not all users have rated the same movies, leading to a sparse rating matrix.

1.4 Question 3

Adjusting user ratings to center around the average helps to normalize them. This reduces biases or extreme ratings from users who only rate at the very high or very low end of the scale. By doing this, we eliminate unusual patterns and make the data cleaner. This process, known as mean centering, also reduces the issue of predictor variables in a model being too closely related to each other, which makes it easier to understand the true impact of individual user ratings when we're trying to predict something.

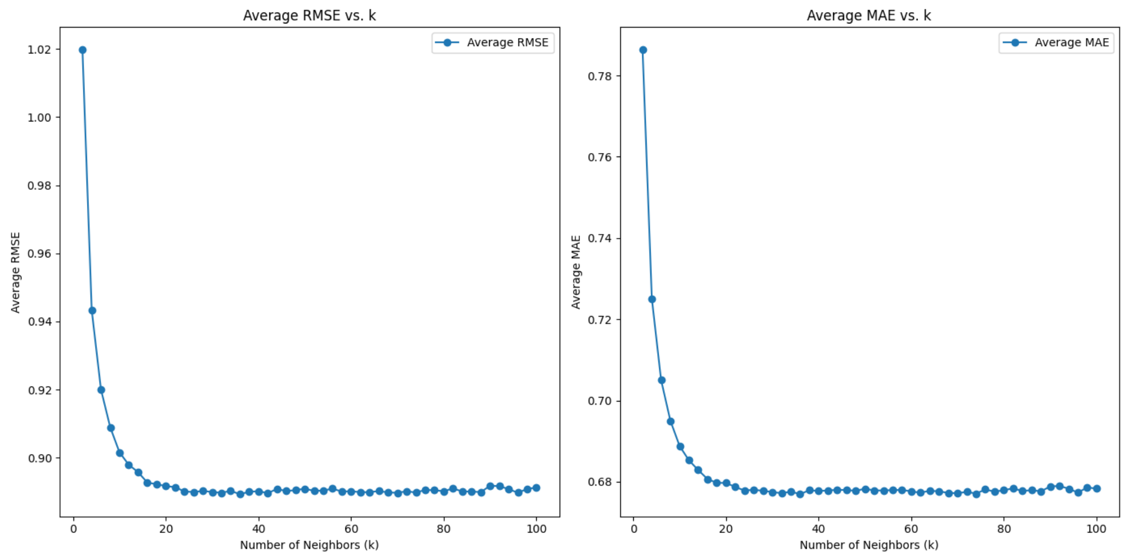
1.5 Question 4

The code and both the average RMSE and MAE v.s. k are shown as below.

[ ]:
```
ratings_df = pd.read_csv(data_folder + "ratings.csv", usecols=['userId',␣
↪'movieId', 'rating'])
reader = Reader(rating_scale=(ratings_df['rating'].min(), ratings_df['rating'].
↪max()))
data = Dataset.load_from_df(ratings_df[['userId', 'movieId', 'rating']], reader)

k_values = range(2, 101, 2)  # From 2 to 100, inclusive, in steps of 2

avg_rmse = []
avg_mae = []

for k in k_values:
    print('Testing for k =',k)
    algo = KNNWithMeans(k=k, sim_options={'name': 'pearson', 'user_based':␣
↪True}, verbose=False)
    results = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=10,␣
↪verbose=False)

    avg_rmse.append(np.mean(results['test_rmse']))
    avg_mae.append(np.mean(results['test_mae']))

plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.plot(k_values, avg_rmse, label='Average RMSE', marker='o')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Average RMSE')
plt.title('Average RMSE vs. k')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(k_values, avg_mae, label='Average MAE', marker='o')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Average MAE')
plt.title('Average MAE vs. k')
plt.legend()

plt.tight_layout()
plt.show()
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
Testing for k = 52
Testing for k = 54
Testing for k = 56
Testing for k = 58
Testing for k = 60
Testing for k = 62
Testing for k = 64
Testing for k = 66
Testing for k = 68
Testing for k = 70
Testing for k = 72
Testing for k = 74
Testing for k = 76
Testing for k = 78
Testing for k = 80
Testing for k = 82
Testing for k = 84
Testing for k = 86
Testing for k = 88
Testing for k = 90
Testing for k = 92
Testing for k = 94
Testing for k = 96
```

```
Testing for k = 98
Testing for k = 100
```



1.6 Question 5

The task was to identify the smallest number of neighbors k needed in a k-NN user-based CF model before the error rates stabilize. Based on Figure in Question 4, it appears that the error rates level out when **k equals 24**. At this point, the **RMSE is 0.8901** and the **MAE is 0.6778**. This suggests that increasing the number of neighbors beyond **24** does not significantly improve the prediction accuracy of the model.

[ ]:
```
avg_rmse[11]
```

[ ]: 0.8901031747310851

[ ]:
```
avg_mae[11]
```

[ ]: 0.6778398221783927

1.7 Question 6

The code and the graph for this quesiont is as below. The minimum average RMSE for K-NN on popular movie trimmed test set is **0.8684** at k = 34. The minimum average RMSE for K-NN on unpopular movie trimmed test set is **1.0527** at k = 76. The minimum average RMSE for K-NN on high variance movie trimmed test set is **1.3893** at k = 96.

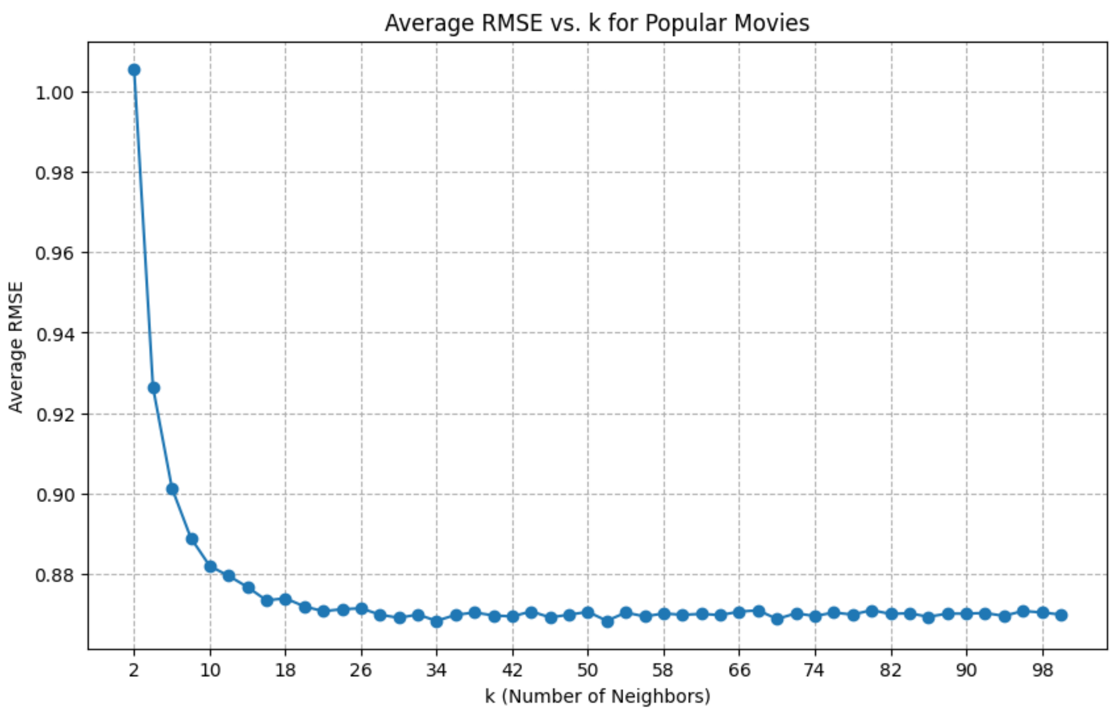
[ ]:
```
def popular_movies_trim(ratings):
    """Trim dataset to contain movies with more than 2 ratings."""
    movie_counts = ratings['movieId'].value_counts()
    popular_movies = movie_counts[movie_counts > 2].index
```

```
    return ratings[ratings['movieId'].isin(popular_movies)]

def unpopular_movies_trim(ratings):
    """Trim dataset to contain movies with 2 or fewer ratings."""
    movie_counts = ratings['movieId'].value_counts()
    unpopular_movies = movie_counts[movie_counts <= 2].index
    return ratings[ratings['movieId'].isin(unpopular_movies)]

def high_variance_movies_trim(ratings):
    """Trim dataset to contain movies with variance >= 2 and at least 5 ratings.
 ↪"""
    sufficient_ratings = ratings.groupby('movieId').filter(lambda x: len(x) >=␣
 ↪5)
    high_variance_movies = sufficient_ratings.groupby('movieId').filter(lambda␣
 ↪x: x['rating'].var() >= 2)
    return high_variance_movies
```

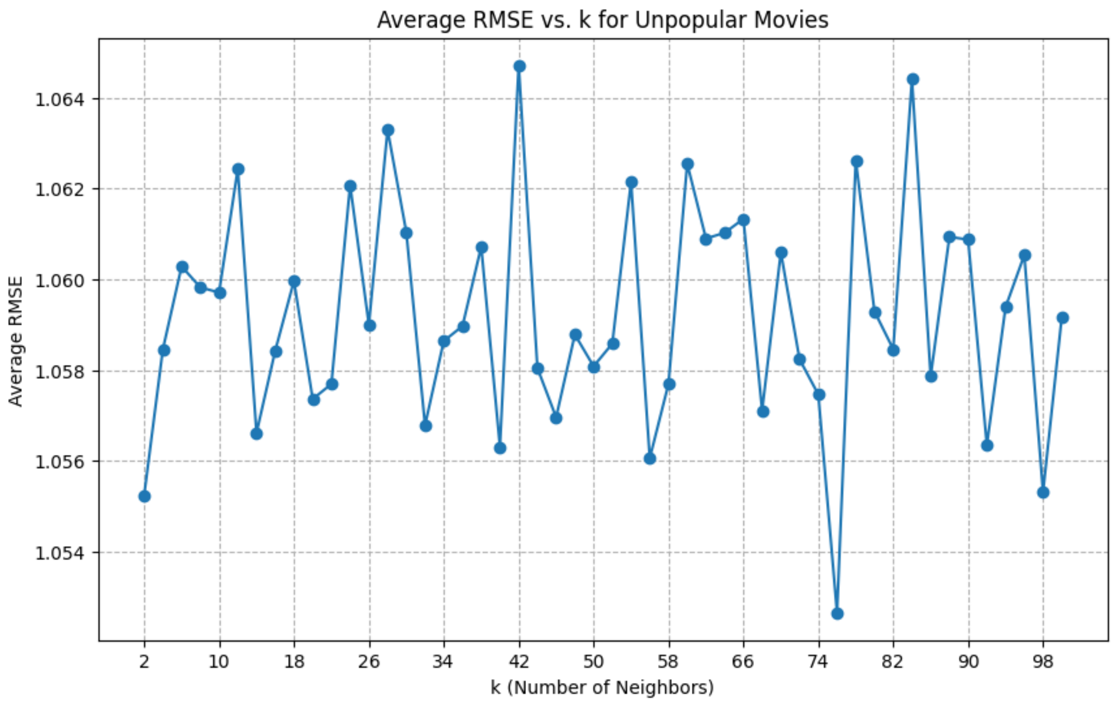
[ ]:
```
from surprise import KNNWithMeans, Dataset, Reader, accuracy
from surprise.model_selection import cross_validate, KFold
import matplotlib.pyplot as plt
import numpy as np

def evaluate_rmse_with_k_sweep(ratings, dataset_name):
    reader = Reader(rating_scale=(0.5, 5))
    data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']],␣
 ↪reader)
    kf = KFold(n_splits=10)

    k_values = range(2, 101, 2)
    avg_rmse_results = []

    for k in k_values:
        # print(f'Testing for k = {k} in {dataset_name} dataset')
        algo = KNNWithMeans(k=k, sim_options={'name': 'pearson', 'user_based':␣
 ↪True}, verbose=False)
        rmse_results = []
        for trainset, testset in kf.split(data):
            algo.fit(trainset)
            predictions = algo.test(testset)
            rmse_results.append(accuracy.rmse(predictions, verbose=False))
        avg_rmse = np.mean(rmse_results)
        avg_rmse_results.append(avg_rmse)

    plt.figure(figsize=(10, 6))
    plt.plot(k_values, avg_rmse_results, marker='o')
    plt.title(f'Average RMSE vs. k for {dataset_name} Movies')
    plt.xlabel('k (Number of Neighbors)')
```

```python
    plt.ylabel('Average RMSE')
    plt.xticks(k_values[::4])
    plt.grid(ls='--')
    plt.show()

    min_avg_rmse = min(avg_rmse_results)
    optimal_k = k_values[avg_rmse_results.index(min_avg_rmse)]
    print(f'Minimum Average RMSE: {min_avg_rmse:.4f} at k = {optimal_k} for␣
 ↪{dataset_name} dataset')

    return optimal_k, min_avg_rmse

trimmed_datasets = {
    "Popular": popular_movies_trim(ratings_df),
    "Unpopular": unpopular_movies_trim(ratings_df),
    "High Variance": high_variance_movies_trim(ratings_df)
}

for name, dataset in trimmed_datasets.items():
    print(f"Evaluating {name} Movies")
    evaluate_rmse_with_k_sweep(dataset, name)  # Pass the name of the dataset␣
 ↪as an argument
```

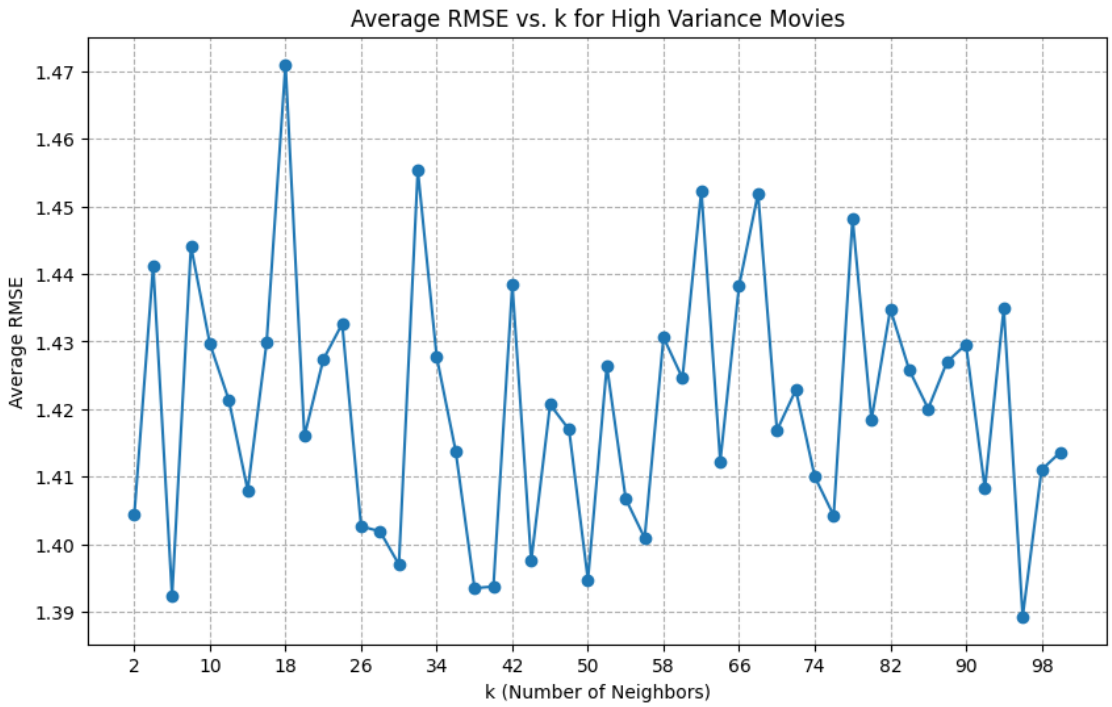
```
Evaluating Popular Movies
```



```
Minimum Average RMSE: 0.8684 at k = 34 for Popular dataset
Evaluating Unpopular Movies
```




```
Minimum Average RMSE: 1.0527 at k = 76 for Unpopular dataset
Evaluating High Variance Movies
```




```
Minimum Average RMSE: 1.3893 at k = 96 for High Variance dataset
```

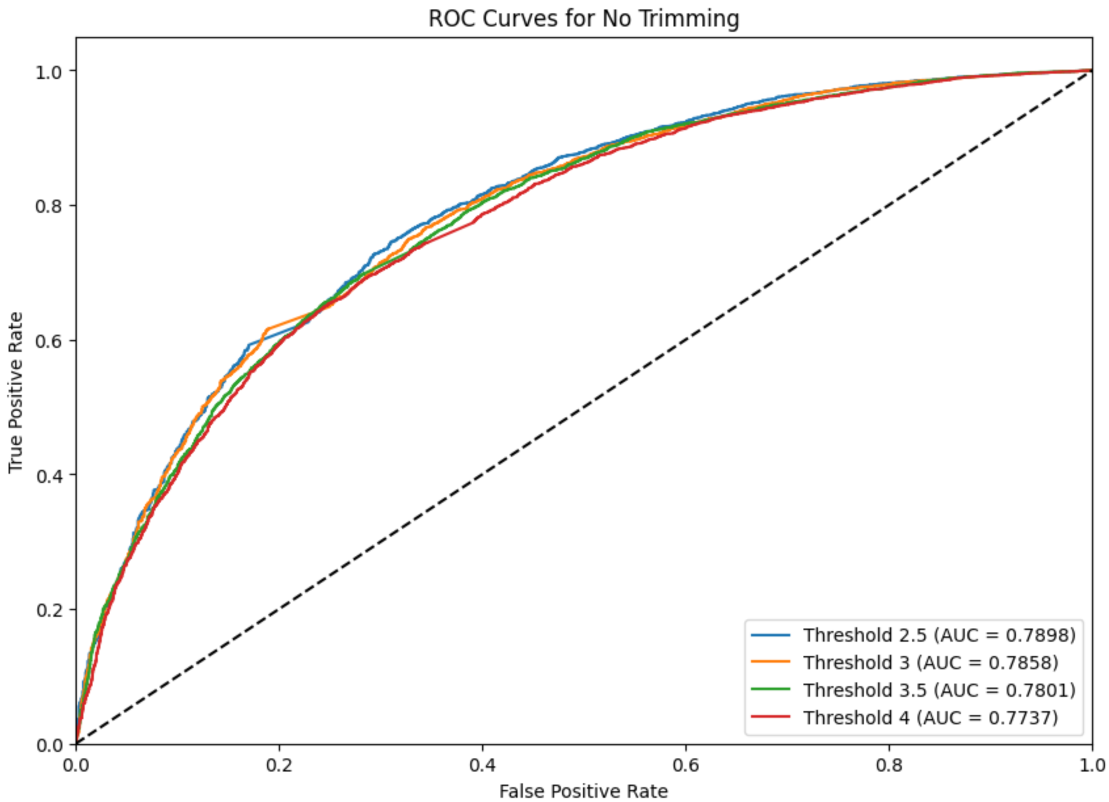
In this section we plot the ROC curves for No Trimming, Popular, Unpopular, and High variance with threshold = [2.5, 3, 3.5, 4]. The 4 plots and code are as follows.

[ ]:
```
ratings_df = pd.read_csv(data_folder + "ratings.csv", usecols=['userId',␣
↪'movieId', 'rating'])
```

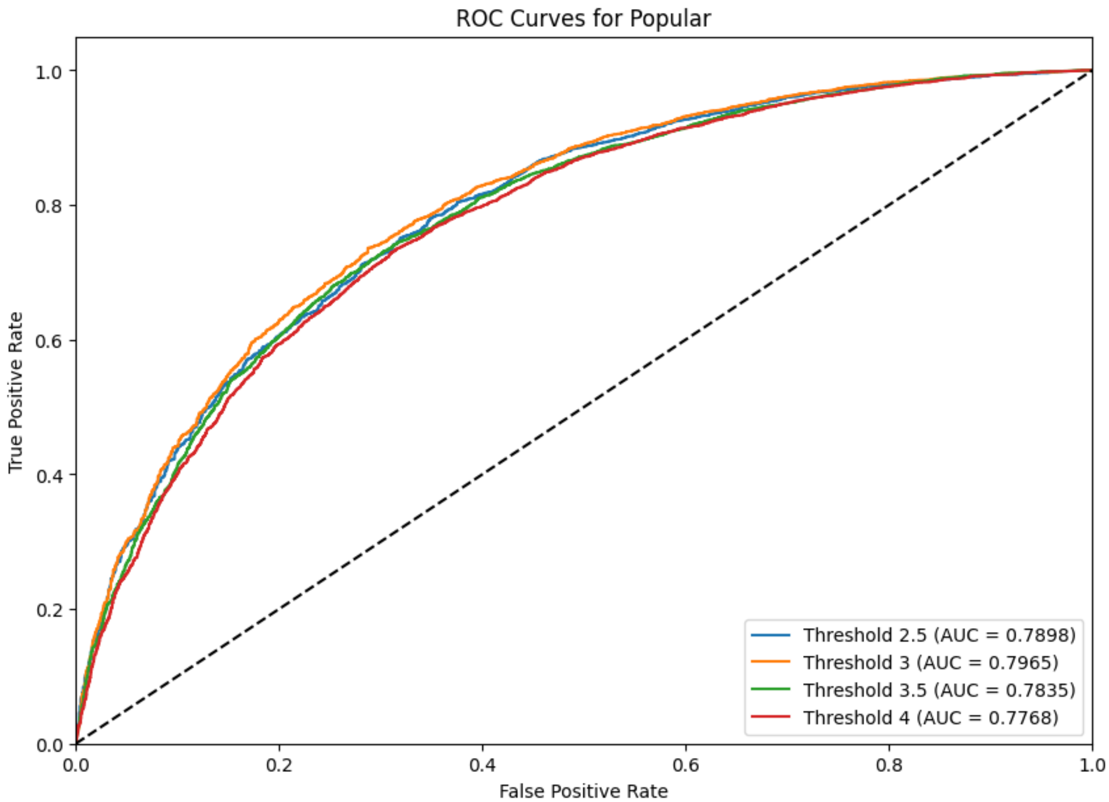
[ ]:
```
optimal_k = 24
algo = KNNWithMeans(k=optimal_k, sim_options={'name': 'pearson', 'user_based':␣
↪True})

thresholds = [2.5, 3, 3.5, 4]

def plot_roc_curves(data, algo, thresholds, title):
    trainset, testset = train_test_split(data, test_size=0.1)
    algo.fit(trainset)
    predictions = algo.test(testset)
    actual_ratings = np.array([pred.r_ui for pred in predictions])
    estimated_ratings = np.array([pred.est for pred in predictions])

    plt.figure(figsize=(10, 7))

    for threshold in thresholds:
        actual_binary = actual_ratings >= threshold
```

```
        fpr, tpr, _ = roc_curve(actual_binary, estimated_ratings)
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'Threshold {threshold} (AUC = {roc_auc:.4f})')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curves for {title}')
    plt.legend(loc="lower right")
    plt.show()

trimmed_datasets = {
    "No Trimming": ratings_df,
    "Popular": popular_movies_trim(ratings_df),
    "Unpopular": unpopular_movies_trim(ratings_df),
    "High Variance": high_variance_movies_trim(ratings_df)
}

reader = Reader(rating_scale=(0.5, 5))
for name, dataset in trimmed_datasets.items():
    data = Dataset.load_from_df(dataset[['userId', 'movieId', 'rating']],␣
 ↪reader)
    print(f"ROC Curves for {name} Movies")
    plot_roc_curves(data, algo, thresholds, name)
```

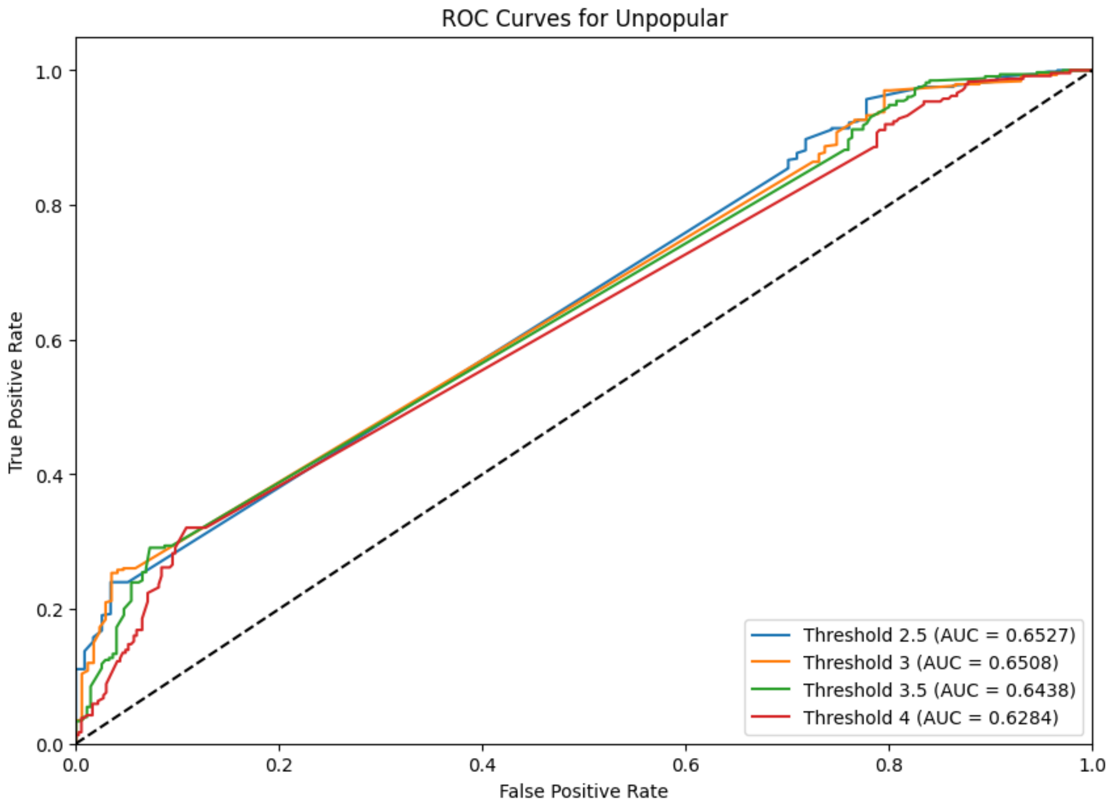
```
ROC Curves for No Trimming Movies
Computing the pearson similarity matrix…
Done computing similarity matrix.
```



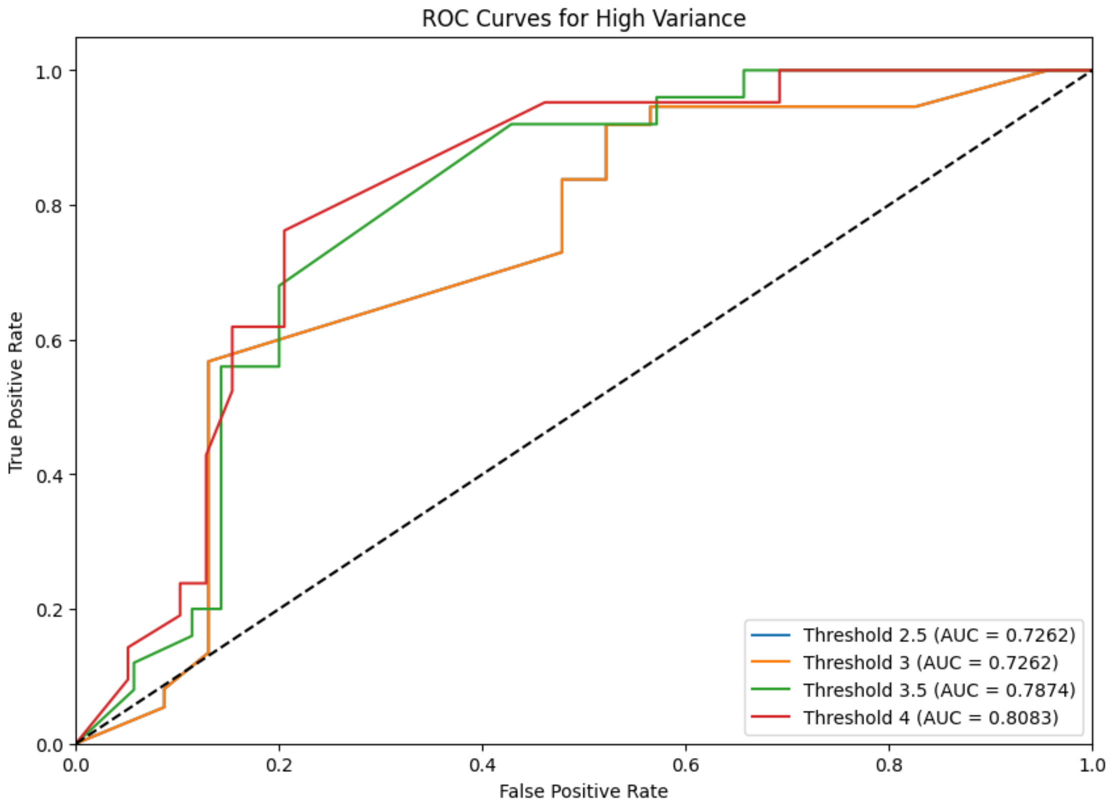

```
ROC Curves for Popular Movies
Computing the pearson similarity matrix…
Done computing similarity matrix.
```




```
ROC Curves for Unpopular Movies
Computing the pearson similarity matrix…
Done computing similarity matrix.
```




```
ROC Curves for High Variance Movies
Computing the pearson similarity matrix…
Done computing similarity matrix.
```



[ ]:
```
!gdown 1T_UBGy1lhRft1c74iqASUFsDH5EmWAeV
!gdown 1rt8wkW9mBG-z5Wlu2iGTkcja0a5hZDwU
!gdown 1wkgBqP29fCzv0KDVbBKPYlXgWTjQfdN0
!gdown 1Xm0hYDNO2HMgUnZ-DerchhBcjX5TsCPM
!gdown 1I4JuDdo2TBw-BVtNfdXv9uRagaVeyahs
```

[ ]:
```
!pip install pandas numpy joblib scipy matplotlib scikit-surprise
!pip install -U scikit-learn
```

[5]:
```
import os
import pandas as pd

modified_fn = './ratings_modified.csv'
if not os.path.exists(modified_fn):
  df = pd.read_csv('./ratings.csv')
  print(df.columns)
  df = df.drop(columns=['Unnamed: 0'])
  df.to_csv(modified_fn, index=False)
```

```
Index(['Unnamed: 0', 'userId', 'movieId', 'rating', 'timestamp'],
dtype='object')
```

[6]:
```
import pandas as pd
import numpy as np

# Trim dataset
groupby_movies = {}
df_ratings = pd.read_csv('./ratings.csv')
for idx in df_ratings.index:
  movieId = df_ratings['movieId'][idx]
  item = [df_ratings['userId'][idx], df_ratings['movieId'][idx],␣
 ↪df_ratings['rating'][idx], df_ratings['timestamp'][idx]]
  if movieId not in groupby_movies:
    groupby_movies[movieId] = [item]
  else:
    groupby_movies[movieId].append(item)

# Popular
with open('ratings_popular.csv', 'w') as f:
  f.write('userId,movieId,rating,timestamp\n')
  for k, v in groupby_movies.items():
    if len(v) > 2:
      for row in v:
        f.write(','.join([str(x) for x in row]) + '\n')

# Unpopular
with open('ratings_unpopular.csv', 'w') as f:
  f.write('userId,movieId,rating,timestamp\n')
  for k, v in groupby_movies.items():
    if len(v) <= 2:
      for row in v:
        f.write(','.join([str(x) for x in row]) + '\n')

# High variance
with open('ratings_high_var.csv', 'w') as f:
  f.write('userId,movieId,rating,timestamp\n')
  for k, v in groupby_movies.items():
    if len(v) >= 5 and np.var([x[2] for x in v]) >= 2.0:
      for row in v:
        f.write(','.join([str(x) for x in row]) + '\n')
```

[ ]:
```
import pandas as pd

df_ratings = pd.read_csv('./ratings.csv')
df_links = pd.read_csv('./links.csv')
df_movies = pd.read_csv('./movies.csv')
df_tags = pd.read_csv('./tags.csv')
```

[ ]:
```python
import numpy as np

user_lut = {}
movie_lut = {}

cnt = 0
for ele in df_ratings['userId']:
  if ele not in user_lut:
    user_lut[ele] = cnt
    cnt += 1

cnt = 0
for ele in df_ratings['movieId']:
  if ele not in movie_lut:
    movie_lut[ele] = cnt
    cnt += 1

def create_rating_matrix(data):
  R = np.zeros((len(user_lut), len(movie_lut)))
  for ele in data:
    R[user_lut[ele[0]], movie_lut[ele[1]]] = ele[2]
  return R

data = df_ratings[['userId', 'movieId', 'rating']].values.tolist()
data = list(map(lambda x:(int(x[0]), int(x[1]), x[2]), data))
```

1.8 Question 7

Understanding the NMF cost function: Is the optimization problem given by equation 5 convex? Consider the optimization problem given by equation 5. For U fixed, formulate it as a least-squares problem.

1.8.1 Answer

No, the optimization problem in equation 5 is not convex since it contains the mulitiplaction of U and V.

If we have U fixed, eq5 can be formulated as $\underset{V}{\text{minimize}} \|W(UV^T - r)\|_F^2$, which can be seen as a weighted least-squares problem. $\|\cdot\|_F$ is the Frobenius norm.

1.9 Question 8.A

Design a NMF-based collaborative filter to predict the ratings of the movies in the original dataset and evaluate its performance using 10-fold cross-validation. Sweep k (number of latent factors) from 2 to 50 in step sizes of 2, and for each k compute the average RMSE and average MAE obtained by averaging the RMSE and MAE across all 10 folds. If NMF takes too long, you can increase the step size. Increasing it too much will result in poorer granularity in your results. Plot the average RMSE (Y-axis) against k (X-axis) and the average MAE (Y- axis) against k (X-axis). For solving this question, use the default value for the regularization parameter.

1.9.1 Answer

The code and figures are shown below.

```
[ ]: import multiprocessing as mp
from surprise import BaselineOnly, Dataset, Reader, NMF
from surprise.model_selection import GridSearchCV
from surprise.model_selection import cross_validate

num_folds = 10

file_path = './ratings_modified.csv'
reader = Reader(line_format="user item rating timestamp", sep=",", skip_lines=1)
data = Dataset.load_from_file(file_path, reader=reader)

print(f'Start to grid search on {mp.cpu_count()} cores...')
param_grid = {
    'n_factors': range(2, 51, 2)
}

grid = GridSearchCV(NMF, param_grid, measures=['RMSE', 'MAE'], cv=num_folds,␣
 ↪n_jobs=mp.cpu_count(), joblib_verbose=10)

grid.fit(data)
print(grid.best_score["rmse"])
results_df = pd.DataFrame.from_dict(grid.cv_results)

print('Finish searching')
```

```
Start to grid search on 8 cores…
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    2.1s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    3.6s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    5.3s
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:    8.0s
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:   11.4s
[Parallel(n_jobs=8)]: Done  45 tasks      | elapsed:   15.0s
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:   19.2s
[Parallel(n_jobs=8)]: Done  69 tasks      | elapsed:   25.1s
[Parallel(n_jobs=8)]: Done  82 tasks      | elapsed:   31.2s
[Parallel(n_jobs=8)]: Done  97 tasks      | elapsed:   38.5s
[Parallel(n_jobs=8)]: Done 112 tasks      | elapsed:   47.0s
[Parallel(n_jobs=8)]: Done 129 tasks      | elapsed:   56.5s
[Parallel(n_jobs=8)]: Done 146 tasks      | elapsed:  1.1min
[Parallel(n_jobs=8)]: Done 165 tasks      | elapsed:  1.3min
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:  1.5min
[Parallel(n_jobs=8)]: Done 205 tasks      | elapsed:  1.7min
[Parallel(n_jobs=8)]: Done 226 tasks      | elapsed:  2.0min
```

```
0.9129704550301316
Finish searching
```

```
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:  2.3min finished
```

[ ]:
```python
min_rmse_idx = np.argmin(np.asarray(grid.cv_results['mean_test_rmse']))
min_rmse_k = grid.cv_results['param_n_factors'][min_rmse_idx]
min_rmse = grid.cv_results['mean_test_rmse'][min_rmse_idx]
min_mae_idx = np.argmin(grid.cv_results['mean_test_mae'])
min_mae_k = grid.cv_results['param_n_factors'][min_mae_idx]
min_mae = grid.cv_results['mean_test_mae'][min_mae_idx]

print(f'{min_rmse_idx=}, {min_rmse_k=}, {min_rmse=}, {min_mae_idx=},
 ↪{min_mae_k=}, {min_mae=}')

results_df
```

```
min_rmse_idx=8, min_rmse_k=18, min_rmse=0.9129704550301316, min_mae_idx=11,
min_mae_k=24, min_mae=0.6932347809702486
```

[ ]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	split3_test_rmse	\
0	1.149377	1.157664	1.136367	1.147862	
1	1.035449	1.050471	1.031990	1.039156	
2	0.972752	0.996457	0.971418	0.981631	
3	0.943250	0.951580	0.935739	0.952087	
4	0.927559	0.942838	0.925675	0.927949	
5	0.908603	0.930107	0.914457	0.917770	
6	0.907262	0.920527	0.917602	0.910209	
7	0.913581	0.922876	0.916136	0.913091	
8	0.904787	0.924278	0.913938	0.911603	
9	0.908167	0.919715	0.917969	0.916641	
10	0.905889	0.923501	0.917206	0.916595	
11	0.908993	0.927530	0.915499	0.916788	
12	0.911827	0.931515	0.920512	0.915709	
13	0.917691	0.934119	0.932031	0.922256	
14	0.919532	0.937872	0.932258	0.928002	
15	0.917576	0.940392	0.937097	0.933391	
16	0.928976	0.945799	0.938079	0.930809	
17	0.933987	0.948214	0.947042	0.939495	
18	0.931495	0.951677	0.956333	0.942627	
19	0.938462	0.957961	0.954563	0.944607	
20	0.943821	0.958578	0.959971	0.946308	
21	0.946882	0.963844	0.955338	0.953009	
22	0.950162	0.964931	0.965085	0.953434	
23	0.951054	0.969565	0.970236	0.962670	
24	0.955290	0.971383	0.975374	0.965056	

	split4_test_rmse	split5_test_rmse	split6_test_rmse	split7_test_rmse	\

0	1.150475	1.161105	1.140962	1.146763
1	1.045580	1.043648	1.040543	1.039267
2	0.978221	0.984893	0.975844	0.982181
3	0.949438	0.951961	0.945082	0.955533
4	0.936257	0.936324	0.920052	0.939142
5	0.924544	0.921377	0.913815	0.926255
6	0.918272	0.918823	0.913676	0.921390
7	0.922352	0.918554	0.919233	0.917951
8	0.918413	0.915090	0.907757	0.920971
9	0.918574	0.908033	0.908961	0.916652
10	0.922293	0.918508	0.913560	0.919283
11	0.925491	0.916170	0.911902	0.920605
12	0.925394	0.923268	0.920163	0.930994
13	0.929655	0.927326	0.921985	0.931629
14	0.937811	0.926641	0.924181	0.937988
15	0.938760	0.936479	0.929928	0.940738
16	0.942343	0.935369	0.935438	0.944217
17	0.945497	0.943843	0.937953	0.944260
18	0.951760	0.942611	0.938487	0.952755
19	0.955564	0.941442	0.940479	0.954924
20	0.962928	0.950963	0.946713	0.961083
21	0.963102	0.958647	0.953414	0.959973
22	0.963283	0.957091	0.959010	0.967614
23	0.971590	0.961193	0.959029	0.972335
24	0.979496	0.968803	0.964226	0.975716

	split8_test_rmse	split9_test_rmse	…	split9_test_mae	mean_test_mae \
0	1.139458	1.135377	…	0.953703	0.965033
1	1.025092	1.034107	…	0.846654	0.849238
2	0.974121	0.969185	…	0.777544	0.783613
3	0.940234	0.939247	…	0.741699	0.745482
4	0.922864	0.928309	…	0.724779	0.725402
5	0.911594	0.915216	…	0.710365	0.709276
6	0.906916	0.909951	…	0.700829	0.701157
7	0.907234	0.905639	…	0.696756	0.700000
8	0.908395	0.904475	…	0.692359	0.694521
9	0.913441	0.908513	…	0.693648	0.693420
10	0.908502	0.913269	…	0.694799	0.693620
11	0.908989	0.912984	…	0.694186	0.693235
12	0.911611	0.922027	…	0.701774	0.696045
13	0.921698	0.927031	…	0.703701	0.699232
14	0.927056	0.928606	…	0.704040	0.701600
15	0.922782	0.930597	…	0.704758	0.702622
16	0.927220	0.932258	…	0.705081	0.704347
17	0.939660	0.937029	…	0.708559	0.708847
18	0.935976	0.946892	…	0.714258	0.711656
19	0.944247	0.943073	…	0.713862	0.714091

20	0.949419	0.945477	…	0.715881	0.717443
21	0.945222	0.956220	…	0.721743	0.719579
22	0.951731	0.955121	…	0.721587	0.722240
23	0.956261	0.960475	…	0.728448	0.726051
24	0.962762	0.965802	…	0.729126	0.729874

	std_test_mae	rank_test_mae	mean_fit_time	std_fit_time	mean_test_time \
0	0.007533	25	1.496163	0.166414	0.073673
1	0.006223	24	1.901850	0.120049	0.063885
2	0.007288	23	2.016857	0.060329	0.063875
3	0.005361	22	2.495944	0.152524	0.065526
4	0.006631	19	2.639536	0.075517	0.062580
5	0.006173	13	2.881351	0.125798	0.072273
6	0.005247	8	3.182049	0.132757	0.072756
7	0.004198	7	3.491193	0.226571	0.064533
8	0.005308	4	3.704223	0.240034	0.081014
9	0.002652	2	3.678815	0.188139	0.084438
10	0.005151	3	3.888518	0.117377	0.077923
11	0.004953	1	4.146256	0.172582	0.080060
12	0.006244	5	4.130042	0.218948	0.069801
13	0.005202	6	4.366433	0.179371	0.083165
14	0.005054	9	4.358084	0.102025	0.061966
15	0.005959	10	4.482469	0.072573	0.076990
16	0.005593	11	4.759442	0.084450	0.078747
17	0.003708	12	4.834383	0.089027	0.079023
18	0.005704	14	5.063588	0.081765	0.078641
19	0.005494	15	5.237170	0.093588	0.078020
20	0.005957	16	5.449093	0.101413	0.077758
21	0.005458	17	5.600993	0.134582	0.077227
22	0.005503	18	5.644697	0.105914	0.075726
23	0.006019	20	5.970446	0.048860	0.084752
24	0.005879	21	5.576076	0.895412	0.060684

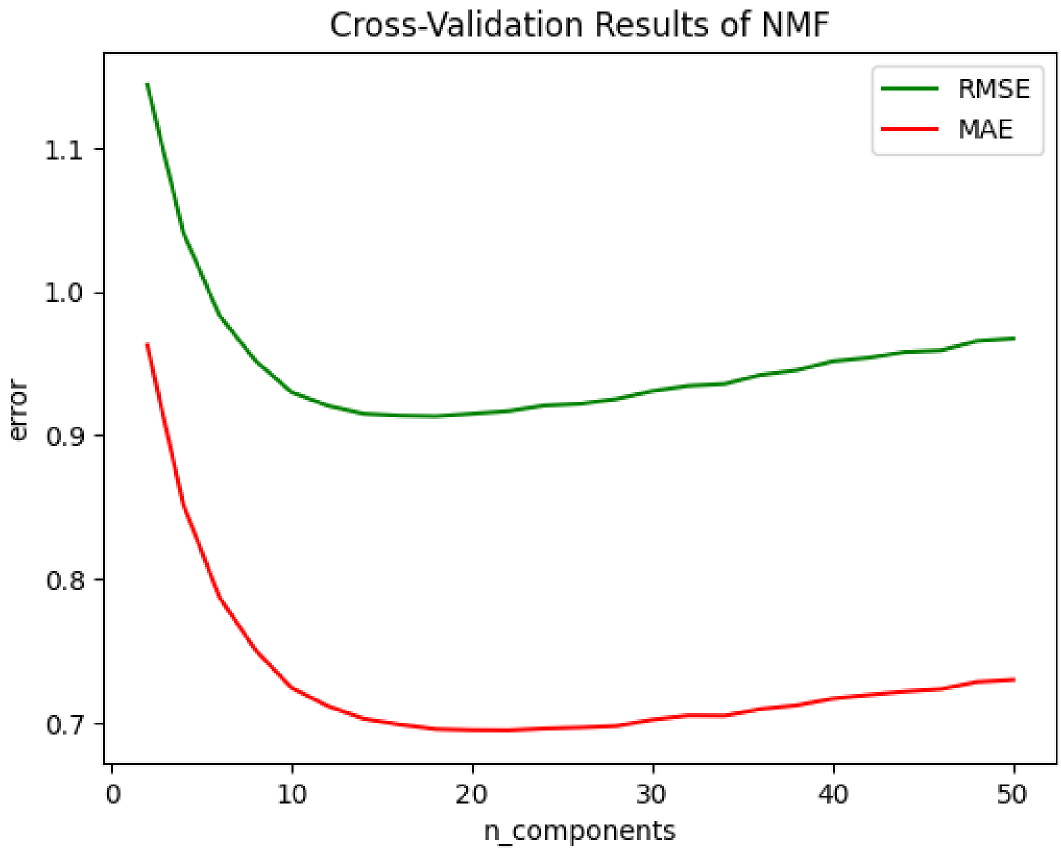
	std_test_time	params	param_n_factors
0	0.024500	{'n_factors': 2}	2
1	0.004661	{'n_factors': 4}	4
2	0.002954	{'n_factors': 6}	6
3	0.004092	{'n_factors': 8}	8
4	0.001604	{'n_factors': 10}	10
5	0.018874	{'n_factors': 12}	12
6	0.031590	{'n_factors': 14}	14
7	0.002076	{'n_factors': 16}	16
8	0.032945	{'n_factors': 18}	18
9	0.014466	{'n_factors': 20}	20
10	0.027924	{'n_factors': 22}	22
11	0.025811	{'n_factors': 24}	24
12	0.012538	{'n_factors': 26}	26

```
13       0.020101  {'n_factors': 28}              28
14       0.000611  {'n_factors': 30}              30
15       0.002692  {'n_factors': 32}              32
16       0.001595  {'n_factors': 34}              34
17       0.002710  {'n_factors': 36}              36
18       0.001672  {'n_factors': 38}              38
19       0.003334  {'n_factors': 40}              40
20       0.002798  {'n_factors': 42}              42
21       0.002404  {'n_factors': 44}              44
22       0.000823  {'n_factors': 46}              46
23       0.015484  {'n_factors': 48}              48
24       0.018910  {'n_factors': 50}              50

[25 rows x 32 columns]
```

[ ]:
```python
# Save results

import os
import pickle as pkl

nmf_cv_filepath = './nmf_cv.pkl'
with open(nmf_cv_filepath, 'wb') as f:
  pkl.dump(grid.cv_results, f)
from google.colab import files
files.download(nmf_cv_filepath)

# load
# with open(nmf_cv_filepath, 'rb') as f:
#   results = pkl.load(f)
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.Javascript object>
```

[ ]:
```python
# Plot figures
import matplotlib
import matplotlib.pyplot as plt
import statistics

plt.title('Cross-Validation Results of NMF')
plt.plot(grid.cv_results['param_n_factors'], grid.cv_results['mean_test_rmse'],␣
 ↪color='green', label='RMSE')
plt.plot(grid.cv_results['param_n_factors'], grid.cv_results['mean_test_mae'],␣
 ↪color='red', label='MAE')
plt.legend()
plt.xlabel('n_factors')
plt.ylabel('error')
plt.show()
```



1.10 Question 8.B

Use the plot from the previous part to find the optimal number of latent factors. Optimal number of latent factors is the value of k that gives the minimum average RMSE or the minimum average MAE. Please report the minimum average RMSE and MAE. Is the optimal number of latent factors same as the number of movie genres?

1.10.1 Answer

According to RMSE, the optimal k = 18 and the minimum average RMSE is 0.913.

According to MAE, the optimal k = 24 and the minimum average MAE is 0.693.

The number of movie genres is 20(code is shown below), which is very close to the optimal k.

[ ]:
```
import pandas as pd

df_movies = pd.read_csv('./movies.csv')
genres = df_movies['genres'].to_list()

genres_set = set()
```

```
for row in genres:
  l = row.split('|')
  for ele in l:
    if ele not in genres_set:
      genres_set.add(ele)

print(f'Unique genres: {genres_set}')
print(f'Num of unique genres: {len(genres_set)}')
```

```
Unique genres: {'Sci-Fi', 'Mystery', 'War', 'Musical', 'Fantasy', 'Action',
'Film-Noir', '(no genres listed)', 'Crime', 'IMAX', 'Adventure', 'Horror',
'Drama', 'Thriller', 'Documentary', 'Romance', 'Western', 'Children', 'Comedy',
'Animation'}
Num of unique genres: 20
```

1.11 Question 8.C

Performance on trimmed dataset subsets

1.11.1 Answer

The code and figures are shown below.

Results:

Subset	minimum average RMSE
Popular	0.892
Unpopular	1.12
High Variance	1.56

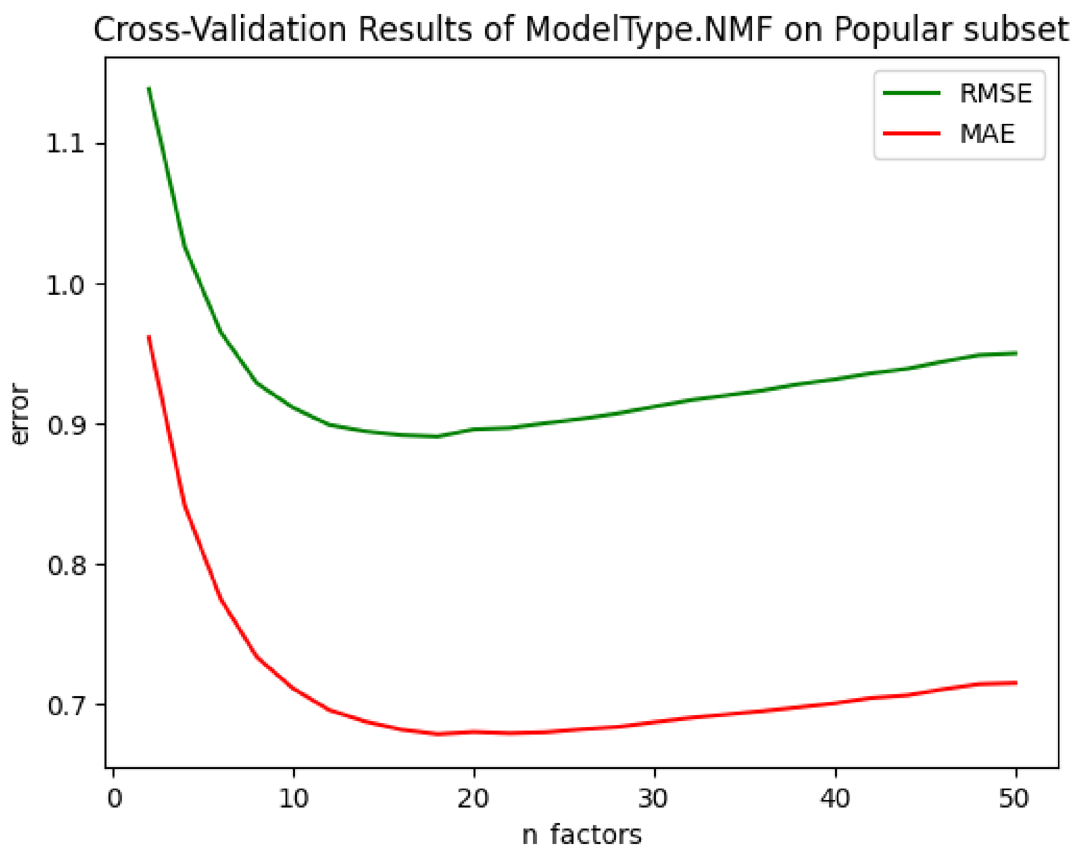
[ ]:
```
import multiprocessing as mp
from surprise import BaselineOnly, Dataset, Reader, NMF, SVD
from surprise.model_selection import GridSearchCV
import enum
import matplotlib
import matplotlib.pyplot as plt
import statistics

class ModelType(enum.Enum):
  NMF: str = 'NMF'
  MFBiased: str = 'MF w Bias'

file_path = './ratings_popular.csv'
MODEL_MAP = {
    ModelType.NMF: NMF,
    ModelType.MFBiased: SVD
}
```

```
num_folds = 10

def grid_search_and_report(model_name, dataset_filepath, dataset_cat):
  reader = Reader(line_format="user item rating timestamp", sep=",",␣
↪skip_lines=1)
  data = Dataset.load_from_file(dataset_filepath, reader=reader)

  print(f'Start to grid search on {mp.cpu_count()} cores...')
  param_grid = {
      'n_factors': range(2, 51, 2)
  }

  grid = GridSearchCV(MODEL_MAP[model_name], param_grid, measures=['RMSE',␣
↪'MAE'], cv=num_folds, n_jobs=mp.cpu_count(), joblib_verbose=10)

  grid.fit(data)
  print(grid.best_score["rmse"])
  results_df = pd.DataFrame.from_dict(grid.cv_results)

  print('Finish searching')

  # Plot figures
  plt.title(f'Cross-Validation Results of {model_name} on {dataset_cat} subset')
  plt.plot(grid.cv_results['param_n_factors'], grid.
↪cv_results['mean_test_rmse'], color='green', label='RMSE')
  plt.plot(grid.cv_results['param_n_factors'], grid.
↪cv_results['mean_test_mae'], color='red', label='MAE')
  plt.legend()
  plt.xlabel('n_factors')
  plt.ylabel('error')
  plt.show()

  min_rmse_idx = np.argmin(np.asarray(grid.cv_results['mean_test_rmse']))
  min_rmse_k = grid.cv_results['param_n_factors'][min_rmse_idx]
  min_rmse = grid.cv_results['mean_test_rmse'][min_rmse_idx]
  min_mae_idx = np.argmin(grid.cv_results['mean_test_mae'])
  min_mae_k = grid.cv_results['param_n_factors'][min_mae_idx]
  min_mae = grid.cv_results['mean_test_mae'][min_mae_idx]

  print(f'{min_rmse_idx=}, {min_rmse_k=}, {min_rmse=}, {min_mae_idx=},␣
↪{min_mae_k=}, {min_mae=}')

  return results_df
```

[ ]:

```
# Popular
file_path = './ratings_popular.csv'
results_df = grid_search_and_report(ModelType.NMF, file_path, 'Popular')
```

```
Start to grid search on 8 cores…

[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    1.7s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    2.8s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    4.4s
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:    6.3s
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    8.8s
[Parallel(n_jobs=8)]: Done  45 tasks      | elapsed:   12.0s
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:   15.2s
[Parallel(n_jobs=8)]: Done  69 tasks      | elapsed:   19.4s
[Parallel(n_jobs=8)]: Done  82 tasks      | elapsed:   23.7s
[Parallel(n_jobs=8)]: Done  97 tasks      | elapsed:   29.2s
[Parallel(n_jobs=8)]: Done 112 tasks      | elapsed:   35.0s
[Parallel(n_jobs=8)]: Done 129 tasks      | elapsed:   42.2s
[Parallel(n_jobs=8)]: Done 146 tasks      | elapsed:   49.3s
[Parallel(n_jobs=8)]: Done 165 tasks      | elapsed:   58.4s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:  1.1min
[Parallel(n_jobs=8)]: Done 205 tasks      | elapsed:  1.3min
[Parallel(n_jobs=8)]: Done 226 tasks      | elapsed:  1.5min
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:  1.7min finished

0.8906051397245609
Finish searching
```




```
min_rmse_idx=8, min_rmse_k=18, min_rmse=0.8906051397245609, min_mae_idx=8,
min_mae_k=18, min_mae=0.6790342469464664
```

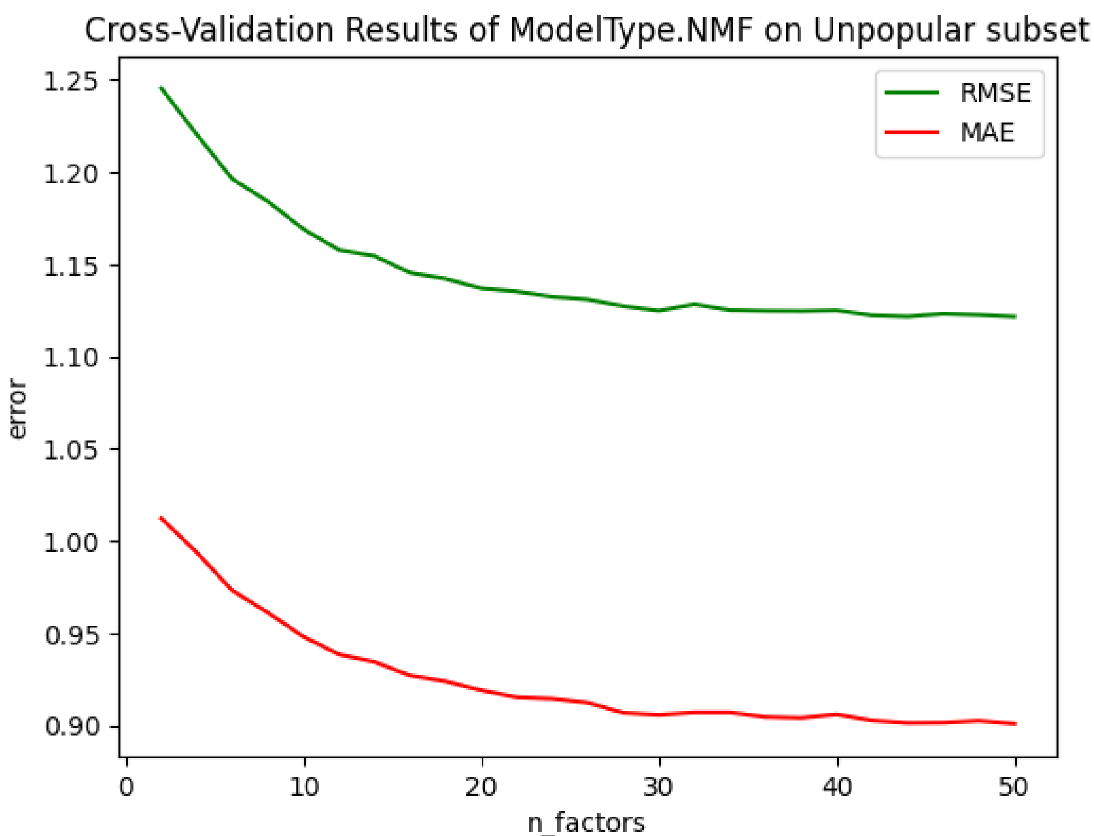
[ ]:
```python
# Unpopular
file_path = './ratings_unpopular.csv'
results_df = grid_search_and_report(ModelType.NMF, file_path, 'Unpopular')
```

```
Start to grid search on 8 cores…

[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Batch computation too fast (0.15151739120483398s.) Setting
batch_size=2.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.2s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    0.3s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    0.5s
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    1.4s
[Parallel(n_jobs=8)]: Done  52 tasks      | elapsed:    2.3s
[Parallel(n_jobs=8)]: Done  74 tasks      | elapsed:    3.7s
[Parallel(n_jobs=8)]: Batch computation too slow (2.0258554813671417s.) Setting
batch_size=1.
[Parallel(n_jobs=8)]: Done  96 tasks      | elapsed:    5.3s
```

```
[Parallel(n_jobs=8)]: Done 122 tasks      | elapsed:    7.8s
[Parallel(n_jobs=8)]: Done 138 tasks      | elapsed:    9.2s
[Parallel(n_jobs=8)]: Done 153 tasks      | elapsed:   10.8s
[Parallel(n_jobs=8)]: Done 168 tasks      | elapsed:   12.7s
[Parallel(n_jobs=8)]: Done 185 tasks      | elapsed:   14.9s
[Parallel(n_jobs=8)]: Done 202 tasks      | elapsed:   17.5s
[Parallel(n_jobs=8)]: Done 221 tasks      | elapsed:   20.7s
```

```
1.1215856803790163
Finish searching
```

```
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:   25.1s finished
```




```
min_rmse_idx=24, min_rmse_k=50, min_rmse=1.1215856803790163, min_mae_idx=24,
min_mae_k=50, min_mae=0.9007586812564747
```

[ ]:
```
# High Variance
file_path = './ratings_high_var.csv'
results_df = grid_search_and_report(ModelType.NMF, file_path, 'High Variance')
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Batch computation too fast (0.014300346374511719s.)
```

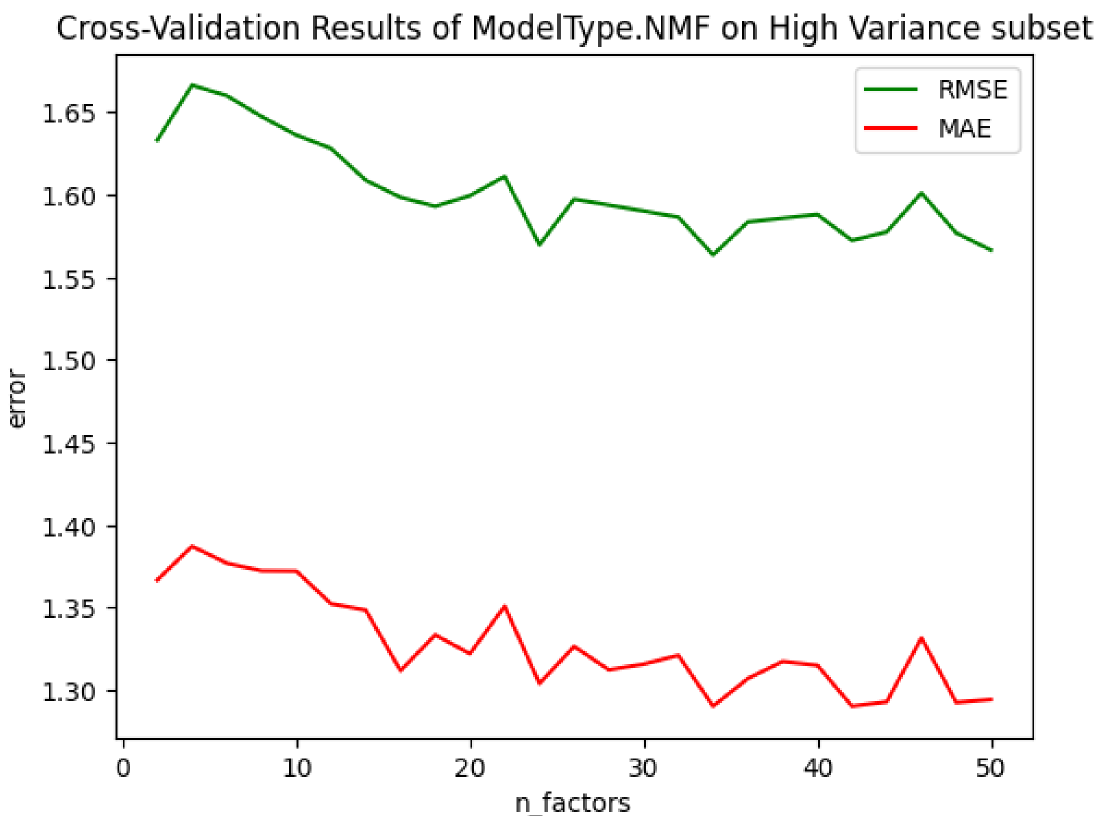
```
Setting batch_size=2.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    0.0s
```

```
Start to grid search on 8 cores…
```

```
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Batch computation too fast (0.017053842544555664s.)
Setting batch_size=4.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Batch computation too fast (0.07892823219299316s.) Setting
batch_size=8.
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 100 tasks      | elapsed:    0.2s
[Parallel(n_jobs=8)]: Done 176 tasks      | elapsed:    0.5s
```

```
1.5636009606836163
Finish searching
```

```
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:    0.9s finished
```




```
min_rmse_idx=16, min_rmse_k=34, min_rmse=1.5636009606836163, min_mae_idx=16,
min_mae_k=34, min_mae=1.290087939765576
```

1.12 Question 8.D

Plot the ROC curves for the NMF-based collaborative filter and also report the area under the curve (AUC) value

1.12.1 Answer

The code and figure are shown below.

AUC:

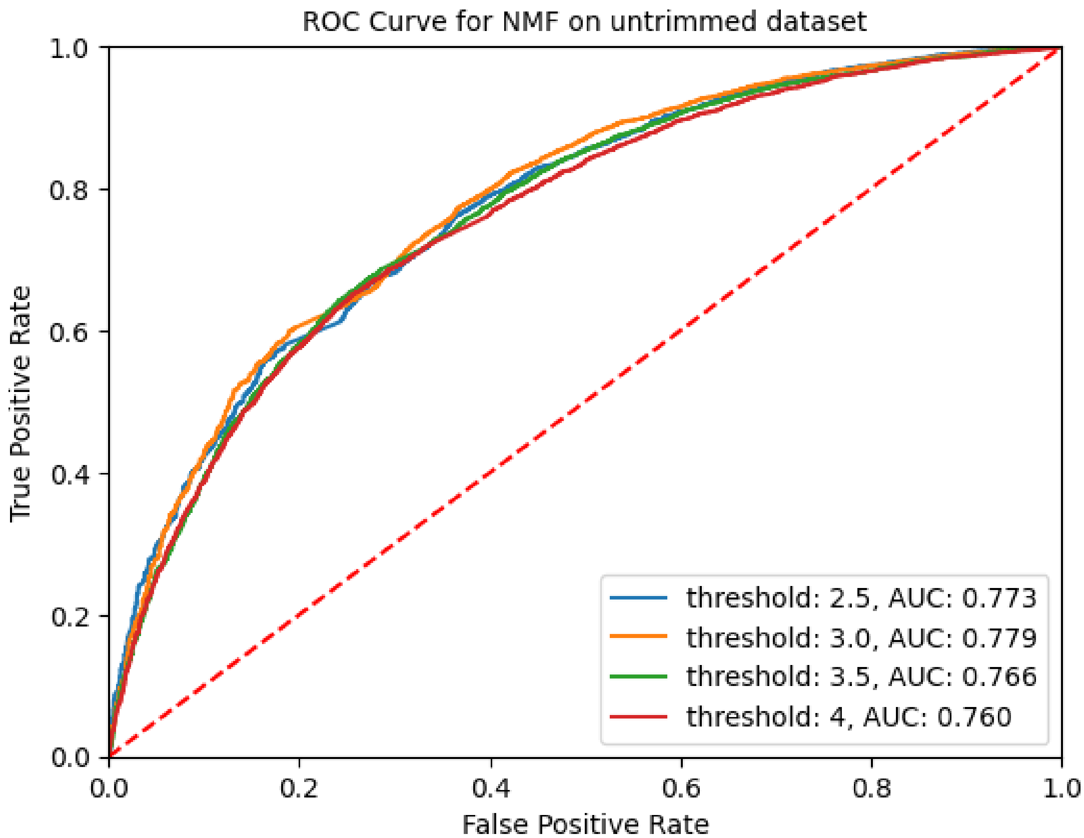
Dataset	k	threshold	AUC
untrimmed	18	2.5	0.773
untrimmed	18	3	0.779
untrimmed	18	3.5	0.766
untrimmed	18	4	0.760
popular	18	2.5	0.776
popular	18	3	0.796
popular	18	3.5	0.772
popular	18	4	0.774
unpopular	50	2.5	0.602
unpopular	50	3	0.602
unpopular	50	3.5	0.637
unpopular	50	4	0.605
high variance	34	2.5	0.633
high variance	34	3	0.659
high variance	34	3.5	0.753
high variance	34	4	0.633

[ ]:
```
from sklearn.metrics import roc_curve, auc, RocCurveDisplay
from surprise import BaselineOnly, Dataset, Reader, NMF, SVD
from surprise.model_selection import train_test_split
import matplotlib.pyplot as plt

def get_roc_and_auc(model_name: ModelType, k: int, dataset_filepath: str,␣
 ↪dataset_cat: str, threshold: float):
  reader = Reader(line_format="user item rating timestamp", sep=",",␣
 ↪skip_lines=1)
  data = Dataset.load_from_file(dataset_filepath, reader=reader)
  trainset, testset = train_test_split(data, test_size=0.1)
  model = MODEL_MAP[model_name](n_factors=k, random_state=42)
  model.fit(trainset)
  pred = []
  for ele in testset: # (uid, iid, gt)
    pred.append(model.predict(uid=ele[0], iid=ele[1], r_ui=ele[2]))

  y_gt = [x.r_ui for x in pred]
  y_pred = [x.est for x in pred]
```

```
def apply_threshold(score):
  return 1 if score >= threshold else 0

def normalize(score):
  return score / 5.0

y_gt_binary = list(map(apply_threshold, y_gt))
y_pred_norm = list(map(normalize, y_pred))

fpr, tpr, thresholds = roc_curve(y_gt_binary, y_pred_norm)
roc_auc = auc(fpr, tpr)
print(f'AUC for {model_name.value} on {dataset_cat} dataset with {threshold=}:
↪ {roc_auc:.3f}')
plt.plot(fpr, tpr, label=f'threshold: {threshold}, AUC: {roc_auc:.3f}')
```

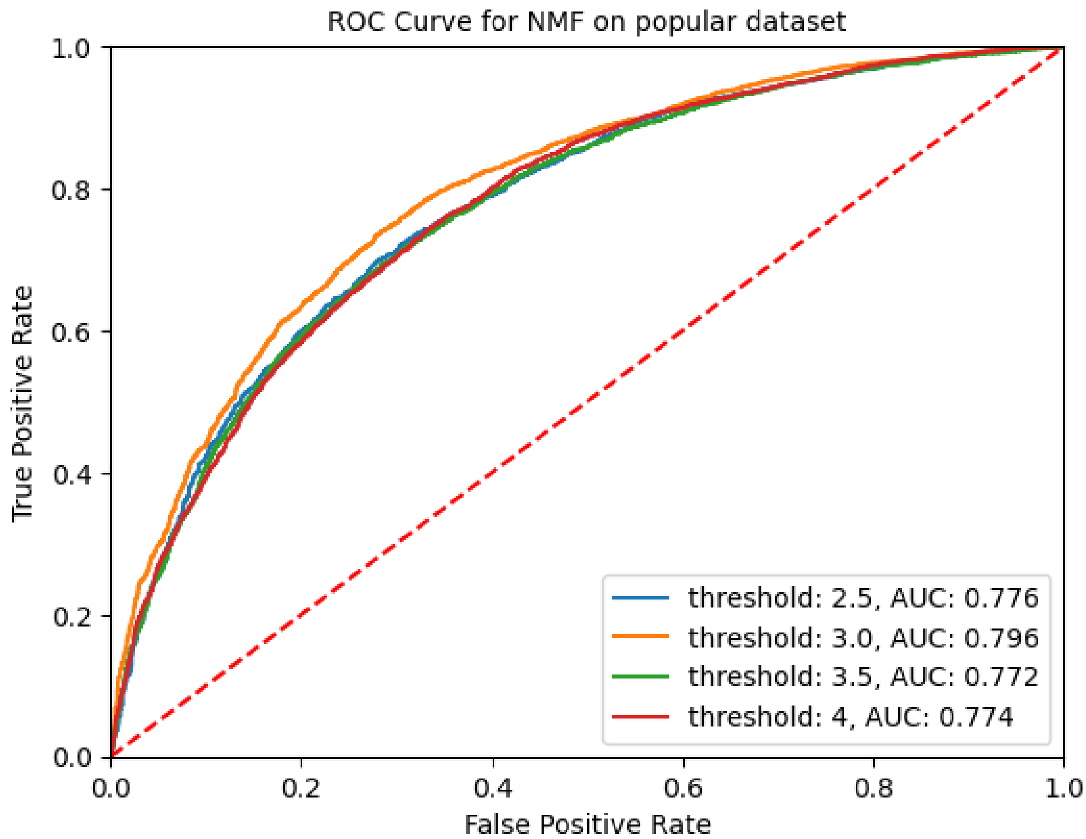
[ ]:
```
k = 18
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.NMF, k, './ratings_modified.csv', 'untrimmed', t)
plt.title(f'ROC Curve for NMF on untrimmed dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for NMF on untrimmed dataset with threshold=2.5: 0.773
AUC for NMF on untrimmed dataset with threshold=3.0: 0.779
AUC for NMF on untrimmed dataset with threshold=3.5: 0.766
AUC for NMF on untrimmed dataset with threshold=4: 0.760
```



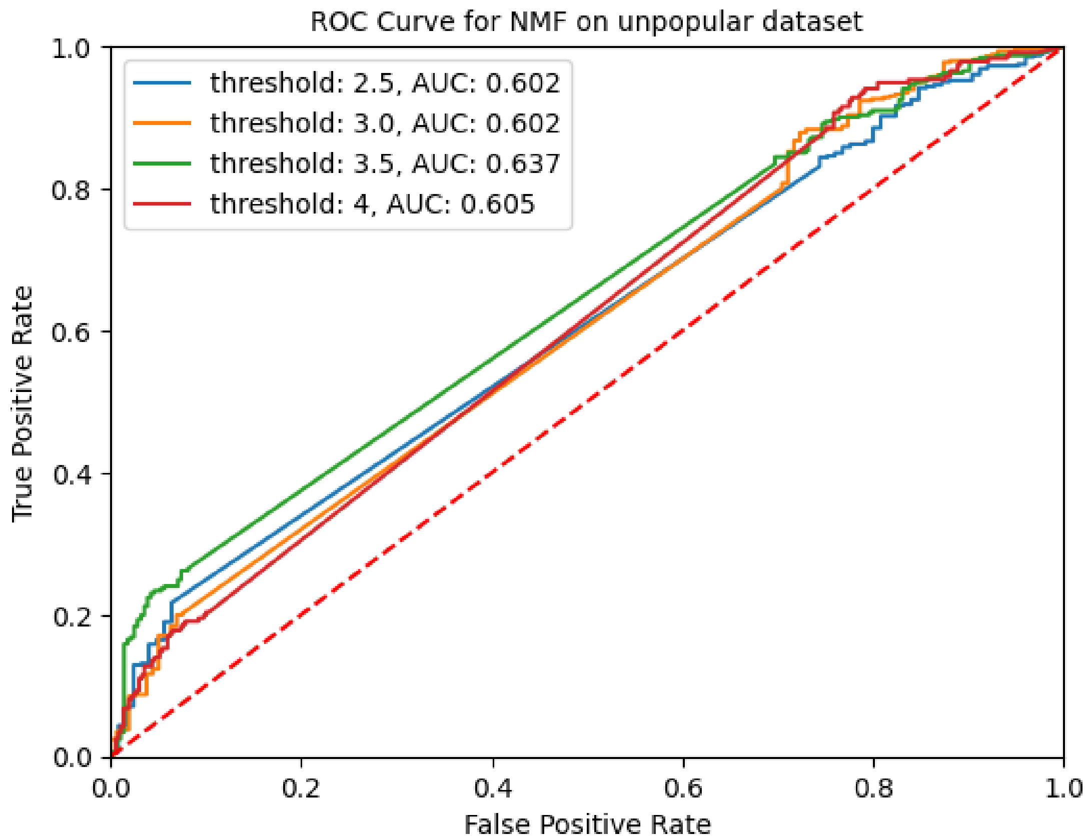
[ ]:
```
k = 18
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.NMF, k, './ratings_popular.csv', 'popular', t)
plt.title(f'ROC Curve for NMF on popular dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for NMF on popular dataset with threshold=2.5: 0.776
AUC for NMF on popular dataset with threshold=3.0: 0.796
AUC for NMF on popular dataset with threshold=3.5: 0.772
AUC for NMF on popular dataset with threshold=4: 0.774
```



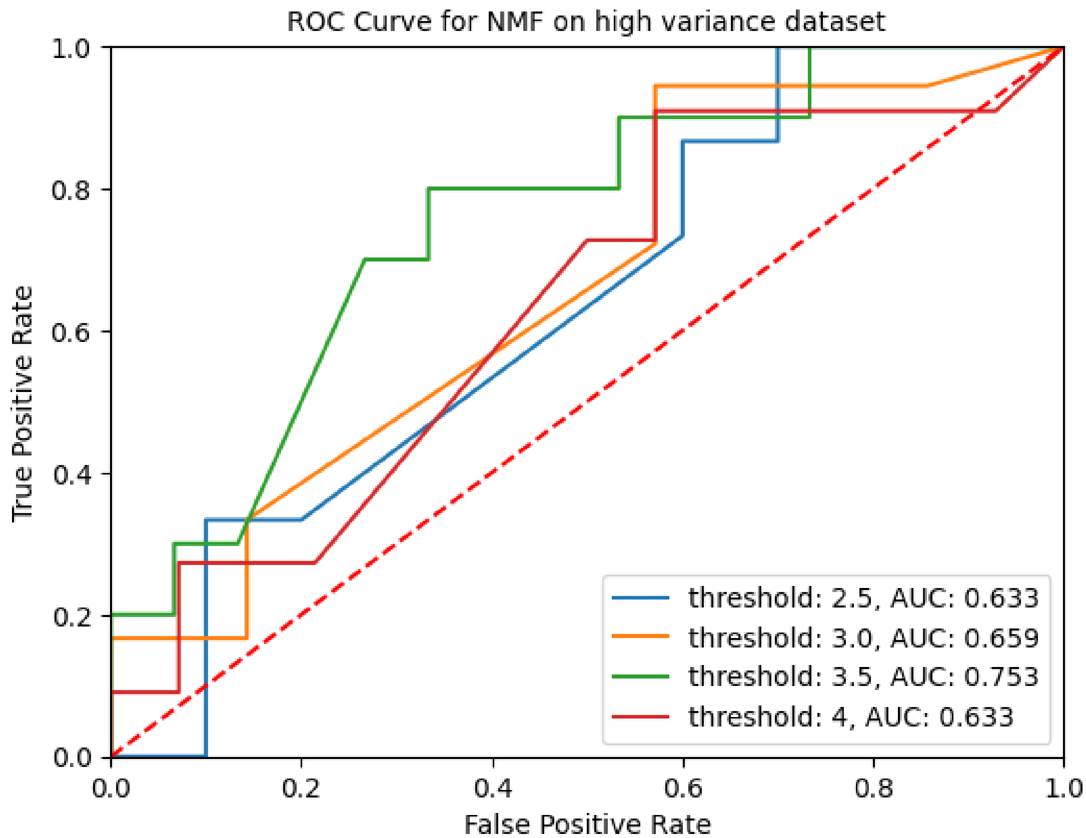
[ ]:
```
k = 50
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.NMF, k, './ratings_unpopular.csv', 'unpopular', t)
plt.title(f'ROC Curve for NMF on unpopular dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for NMF on unpopular dataset with threshold=2.5: 0.602
AUC for NMF on unpopular dataset with threshold=3.0: 0.602
AUC for NMF on unpopular dataset with threshold=3.5: 0.637
AUC for NMF on unpopular dataset with threshold=4: 0.605
```



[ ]:
```python
k = 34
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.NMF, k, './ratings_high_var.csv', 'high variance',␣
↪t)
plt.title(f'ROC Curve for NMF on high variance dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for NMF on high variance dataset with threshold=2.5: 0.633
AUC for NMF on high variance dataset with threshold=3.0: 0.659
AUC for NMF on high variance dataset with threshold=3.5: 0.753
AUC for NMF on high variance dataset with threshold=4: 0.633
```



1.13 Question 9

Interpreting the NMF model: Perform Non-negative matrix factorization on the ratings matrix R to obtain the factor matrices U and V , where U represents the user-latent factors interaction and V represents the movie-latent factors interaction (use k = 20). For each column of V , sort the movies in descending order and report the genres of the top 10 movies. Do the top 10 movies belong to a particular or a small collection of genre? Is there a connection between the latent factors and the movie genres?

1.13.1 Answer

The code is shown below.

We can see some of the factors do have relation to specific genres, like drama of factor 12, thriller of factor 17, children of factor 0, etc.

There is a connection between the latent factors and the movie genres.

[ ]:
```
from surprise import BaselineOnly, Dataset, Reader, NMF

file_path = './ratings_modified.csv'
```

```python
reader = Reader(line_format="user item rating timestamp", sep=",", skip_lines=1)
data = Dataset.load_from_file(file_path, reader=reader)
trainset = data.build_full_trainset()

model = NMF(n_factors=20)
model.fit(trainset)
V = model.qi

n_movies, n_factors = V.shape

TOPK = 10
topk_movies = []
for i in range(n_factors):
  Vi = V[:, i]
  sorted_index = np.argsort(Vi)[::-1][:TOPK]
  topk_movies.append([int(trainset.to_raw_iid(x)) for x in sorted_index])

df_movies = pd.read_csv('./movies.csv')

movies_map = {}
for idx in df_movies.index:
  movies_map[df_movies['movieId'][idx]] = df_movies['genres'][idx].split('|')

for i in range(n_factors):
  movies_genres = []
  for mid in topk_movies[i]:
    movies_genres.append(movies_map[mid])
  print(f'Genres for factor {i}: {movies_genres}')
```

```
Genres for factor 0: [['Animation', 'Children', 'Comedy'], ['Adventure',
'Children', 'Comedy'], ['Drama'], ['Children', 'Comedy', 'Drama', 'Mystery'],
['Comedy'], ['Drama', 'Thriller'], ['Comedy', 'Romance'], ['Action',
'Adventure', 'Animation', 'Children', 'Comedy'], ['Comedy', 'Drama'],
['Adventure', 'Children']]
Genres for factor 1: [['Comedy', 'Documentary'], ['Drama', 'Mystery',
'Romance'], ['Action', 'Comedy', 'Crime'], ['Action', 'Crime', 'Thriller'],
['Drama'], ['Horror', 'Thriller'], ['Romance'], ['Action', 'Adventure',
'Comedy', 'Sci-Fi'], ['Drama', 'Thriller'], ['Action', 'Sci-Fi', 'Thriller',
'IMAX']]
Genres for factor 2: [['Fantasy', 'Western'], ['Drama', 'Fantasy', 'Mystery'],
['Action', 'Fantasy', 'Horror', 'Sci-Fi', 'Thriller'], ['Children', 'Comedy',
'Drama'], ['Action', 'Crime', 'Drama', 'Thriller'], ['Drama', 'Romance'],
['Crime', 'Drama', 'Mystery', 'Thriller'], ['Action', 'Sci-Fi'], ['Drama'],
['Action', 'Comedy', 'Drama']]
Genres for factor 3: [['Adventure', 'Thriller'], ['Horror', 'Mystery',
'Thriller'], ['Crime', 'Horror', 'Mystery'], ['Comedy', 'Drama', 'Romance'],
['Children', 'Comedy'], ['Horror', 'Mystery', 'Thriller'], ['Comedy',
'Romance'], ['Crime', 'Horror', 'Thriller'], ['Action', 'Sci-Fi'], ['Crime',
```

```
'Drama', 'Sci-Fi', 'Thriller']]
Genres for factor 4: [['Drama', 'Horror', 'Thriller'], ['Action', 'Comedy',
'Romance'], ['Adventure', 'Drama'], ['Musical'], ['Comedy', 'Crime'], ['Comedy',
'Musical'], ['Drama', 'War'], ['Comedy', 'Drama', 'Romance'], ['Sci-Fi'],
['Drama']]
Genres for factor 5: [['Crime', 'Drama'], ['Adventure', 'Children'], ['Drama',
'Romance'], ['Action', 'Adventure', 'Comedy'], ['Horror', 'Mystery',
'Thriller'], ['Crime', 'Drama', 'Thriller'], ['Comedy', 'Romance'], ['Drama'],
['Adventure', 'Children'], ['Crime', 'Drama']]
Genres for factor 6: [['Horror'], ['Drama'], ['Drama', 'Fantasy', 'Mystery'],
['Drama', 'Mystery'], ['Action', 'Adventure', 'Children', 'IMAX'], ['Comedy',
'Drama'], ['Comedy', 'Drama', 'Romance'], ['Drama', 'Thriller'], ['Drama'],
['Comedy']]
Genres for factor 7: [['Comedy'], ['Drama', 'Sci-Fi'], ['Adventure', 'Children',
'Comedy'], ['Action', 'Comedy'], ['Action', 'Adventure', 'Animation',
'Children', 'Comedy', 'Romance'], ['Comedy', 'Crime', 'Mystery', 'Romance'],
['Action', 'Drama'], ['Comedy'], ['Action', 'Comedy'], ['Action', 'Adventure',
'Comedy', 'Sci-Fi']]
Genres for factor 8: [['Comedy', 'Drama'], ['Comedy'], ['Drama', 'Sci-Fi'],
['Animation', 'Comedy', 'Drama', 'Fantasy', 'Sci-Fi'], ['Horror'], ['Horror',
'Sci-Fi'], ['Action', 'Drama', 'Fantasy'], ['Fantasy', 'Horror'], ['Comedy',
'Drama'], ['Comedy']]
Genres for factor 9: [['Drama', 'Horror'], ['Action', 'Comedy', 'Horror',
'Thriller'], ['Comedy', 'Fantasy', 'Horror', 'Musical', 'Thriller'], ['Drama',
'Horror'], ['Action', 'Adventure', 'Children', 'Comedy'], ['Horror'], ['Horror',
'Mystery', 'Thriller'], ['Comedy'], ['Comedy', 'Drama'], ['Drama']]
Genres for factor 10: [['Action', 'Drama'], ['Action', 'Crime', 'Drama',
'Thriller'], ['Drama', 'Thriller'], ['Action', 'Adventure'], ['Comedy', 'Drama',
'Horror'], ['Comedy', 'Drama', 'Romance'], ['Drama', 'Romance'], ['Comedy',
'Fantasy', 'Horror'], ['Comedy', 'Romance'], ['Comedy']]
Genres for factor 11: [['Action', 'Animation', 'Drama', 'Sci-Fi', 'Thriller'],
['Comedy'], ['Comedy', 'Musical'], ['Sci-Fi'], ['Comedy'], ['Documentary'],
['Crime', 'Horror', 'Thriller'], ['Drama', 'War'], ['Comedy', 'Drama'],
['Horror', 'Sci-Fi', 'Thriller']]
Genres for factor 12: [['Drama'], ['Crime', 'Drama', 'Romance'], ['Action',
'Drama'], ['Crime', 'Film-Noir', 'Thriller'], ['Comedy', 'Drama', 'Romance'],
['Action', 'Adventure', 'Drama', 'Thriller'], ['Drama', 'Romance'], ['Drama',
'Musical', 'Romance'], ['Crime', 'Drama', 'Mystery'], ['Action', 'Adventure',
'Children', 'Fantasy']]
Genres for factor 13: [['Adventure', 'Comedy', 'Fantasy'], ['Drama'], ['Drama'],
['Action', 'Horror', 'Sci-Fi'], ['Comedy', 'Romance'], ['Drama', 'Fantasy',
'Thriller'], ['Drama', 'Romance'], ['Drama'], ['Drama', 'Horror', 'Mystery',
'Thriller'], ['Comedy']]
Genres for factor 14: [['Drama', 'Film-Noir'], ['Comedy', 'Drama', 'Romance'],
['Drama', 'Romance'], ['Drama'], ['Documentary'], ['Drama', 'Romance',
'Western'], ['Action', 'Adventure', 'Drama', 'Thriller'], ['Adventure',
'Children', 'Comedy'], ['Crime', 'Drama', 'Thriller'], ['Action', 'Horror',
'Sci-Fi']]
```

```
Genres for factor 15: [['Animation', 'Children', 'Comedy', 'Musical'],
['Horror', 'Sci-Fi'], ['Sci-Fi'], ['Fantasy', 'Horror'], ['Comedy', 'Drama',
'Romance'], ['Children', 'Comedy'], ['Action', 'Horror', 'Mystery', 'Sci-Fi'],
['Documentary', 'Musical'], ['Action', 'Comedy', 'Crime', 'Fantasy'],
['Adventure', 'Animation', 'Children', 'Comedy', 'IMAX']]
Genres for factor 16: [['Drama'], ['Drama', 'Romance', 'Thriller'], ['Horror',
'Sci-Fi'], ['Comedy'], ['Action', 'Fantasy', 'Horror', 'Sci-Fi', 'Thriller'],
['Action', 'Sci-Fi'], ['Crime', 'Drama', 'Fantasy', 'Mystery', 'Thriller'],
['Comedy'], ['Comedy'], ['Drama']]
Genres for factor 17: [['Drama', 'Thriller'], ['Romance', 'Thriller'], ['Drama',
'Thriller'], ['Drama'], ['Horror', 'Mystery', 'Thriller'], ['Horror',
'Thriller'], ['Drama', 'Fantasy', 'Sci-Fi'], ['Comedy', 'Drama', 'Romance'],
['Sci-Fi'], ['Drama', 'Romance', 'Thriller']]
Genres for factor 18: [['Action', 'Crime', 'Thriller'], ['Drama', 'Sci-Fi'],
['Comedy', 'Drama'], ['Action', 'Comedy', 'Crime', 'Thriller'], ['Action',
'Drama'], ['Adventure', 'Animation', 'Children', 'Fantasy', 'IMAX'], ['Action',
'Drama', 'Fantasy'], ['Action', 'Comedy', 'Sci-Fi', 'Thriller'], ['Drama'],
['Action', 'Comedy']]
Genres for factor 19: [['Children', 'Comedy'], ['Comedy', 'Documentary'],
['Horror', 'Thriller'], ['Drama', 'Horror', 'Thriller'], ['Horror'], ['Drama'],
['Action', 'Adventure', 'Animation', 'Sci-Fi'], ['Comedy', 'Romance'],
['Action', 'Adventure', 'Animation'], ['Crime', 'Drama']]
```

1.14 Question 10.A

Design a MF-based collaborative filter to predict the ratings of the movies in the original dataset and evaluate it's performance using 10-fold cross-validation. Sweep k (number of latent factors) from 2 to 50 in step sizes of 2, and for each k compute the average RMSE and average MAE obtained by averaging the RMSE and MAE across all 10 folds. Plot the average RMSE (Y-axis) against k (X-axis) and the average MAE (Y-axis) against k (X-axis). For solving this question, use the default value for the regularization parameter.

1.14.1 Answer

The code and figure are shown below.

[ ]:
```python
import multiprocessing as mp
from surprise import BaselineOnly, Dataset, Reader, SVD
from surprise.model_selection import GridSearchCV

num_folds = 10

file_path = './ratings_modified.csv'
reader = Reader(line_format="user item rating timestamp", sep=",", skip_lines=1)
data = Dataset.load_from_file(file_path, reader=reader)

print(f'Start to grid search on {mp.cpu_count()} cores...')
param_grid = {
```

```
    'n_factors': range(2, 51, 2)
}

grid = GridSearchCV(SVD, param_grid, measures=['RMSE', 'MAE'], cv=num_folds,␣
 ↪n_jobs=mp.cpu_count(), joblib_verbose=10)

grid.fit(data)
results_df = pd.DataFrame.from_dict(grid.cv_results)

print('Finish searching')
```

```
Start to grid search on 8 cores…
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.5s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    1.3s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    1.9s
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:    2.9s
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    4.1s
[Parallel(n_jobs=8)]: Done  45 tasks      | elapsed:    5.2s
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:    6.5s
[Parallel(n_jobs=8)]: Done  69 tasks      | elapsed:    8.2s
[Parallel(n_jobs=8)]: Done  82 tasks      | elapsed:    9.8s
[Parallel(n_jobs=8)]: Done  97 tasks      | elapsed:   12.2s
[Parallel(n_jobs=8)]: Done 112 tasks      | elapsed:   14.2s
[Parallel(n_jobs=8)]: Done 129 tasks      | elapsed:   16.5s
[Parallel(n_jobs=8)]: Done 146 tasks      | elapsed:   18.8s
[Parallel(n_jobs=8)]: Done 165 tasks      | elapsed:   21.7s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:   24.5s
[Parallel(n_jobs=8)]: Done 205 tasks      | elapsed:   27.7s
[Parallel(n_jobs=8)]: Done 226 tasks      | elapsed:   31.3s
```

```
Finish searching
```

```
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:   34.8s finished
```

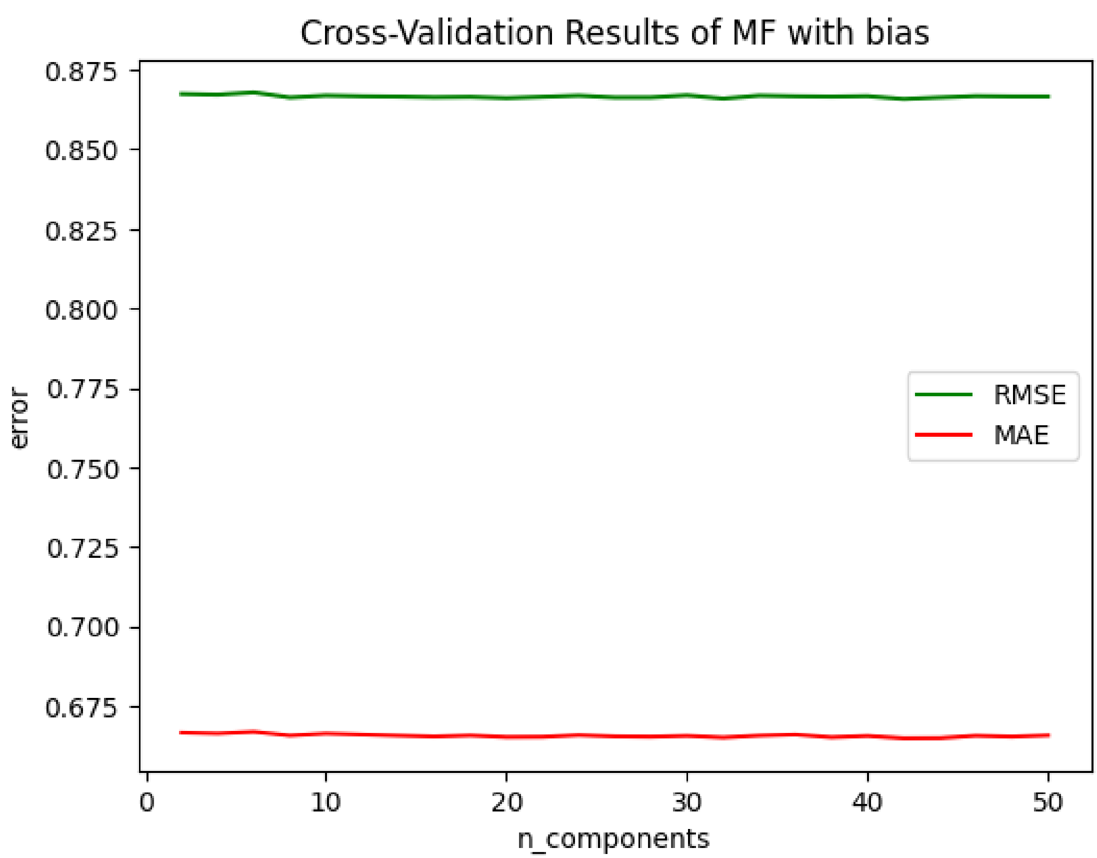
[ ]:
```
# Save results

import os
import pickle as pkl

svd_cv_filepath = './svd_cv.pkl'
with open(svd_cv_filepath, 'wb') as f:
  pkl.dump(grid.cv_results, f)
from google.colab import files
files.download(svd_cv_filepath)

# load
# with open(svd_cv_filepath, 'rb') as f:
```

```
#   results = pkl.load(f)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

[ ]:
```
# Plot figures
import matplotlib
import matplotlib.pyplot as plt
import statistics

plt.title('Cross-Validation Results of MF with bias')
plt.plot(grid.cv_results['param_n_factors'], grid.cv_results['mean_test_rmse'],␣
 ↪color='green', label='RMSE')
plt.plot(grid.cv_results['param_n_factors'], grid.cv_results['mean_test_mae'],␣
 ↪color='red', label='MAE')
plt.legend()
plt.xlabel('n_factors')
plt.ylabel('error')
plt.show()
```



1.15 Question 10.B

Use the plot from the previous part to find the optimal number of latent factors. Optimal number of latent factors is the value of k that gives the minimum average RMSE or the minimum average MAE. Please report the minimum average RMSE and MAE. Is the optimal number of latent factors same as the number of movie genres?

1.15.1 Answer

The code is shown below.

According to RMSE, the optimal k = 42 and the minimum average RMSE is 0.866.

According to MAE, the optimal k = 42 and the minimum average MAE is 0.665.

The number of movie genres is 20, which is not close to the optimal k.

[ ]:
```
min_rmse_idx = np.argmin(np.asarray(grid.cv_results['mean_test_rmse']))
min_rmse_k = grid.cv_results['param_n_factors'][min_rmse_idx]
min_rmse = grid.cv_results['mean_test_rmse'][min_rmse_idx]
min_mae_idx = np.argmin(grid.cv_results['mean_test_mae'])
min_mae_k = grid.cv_results['param_n_factors'][min_mae_idx]
min_mae = grid.cv_results['mean_test_mae'][min_mae_idx]

print(f'{min_rmse_idx=}, {min_rmse_k=}, {min_rmse=}, {min_mae_idx=},␣
 ↪{min_mae_k=}, {min_mae=}')

results_df
```

```
min_rmse_idx=20, min_rmse_k=42, min_rmse=0.8658194521968439, min_mae_idx=20,
min_mae_k=42, min_mae=0.6648765165625126
```

[ ]:

	split0_test_rmse	split1_test_rmse	split2_test_rmse	split3_test_rmse	\
0	0.856818	0.860873	0.869859	0.864091	
1	0.855676	0.861226	0.869586	0.865549	
2	0.856806	0.861971	0.869303	0.864360	
3	0.854246	0.860631	0.868907	0.862626	
4	0.856893	0.862267	0.870016	0.864099	
5	0.856615	0.861748	0.870866	0.864797	
6	0.852667	0.862412	0.869079	0.863535	
7	0.857682	0.861058	0.869679	0.863040	
8	0.854778	0.861901	0.869123	0.865623	
9	0.853331	0.860356	0.869283	0.863474	
10	0.853133	0.863193	0.870203	0.861645	
11	0.856144	0.861175	0.867871	0.865765	
12	0.854895	0.858581	0.868081	0.863313	
13	0.854210	0.860648	0.868352	0.865414	
14	0.854656	0.858581	0.870121	0.865027	
15	0.855587	0.859602	0.868960	0.862791	
16	0.857442	0.860467	0.869713	0.863967	

17	0.855435	0.858722	0.869192	0.863267
18	0.857177	0.861093	0.867933	0.864455
19	0.856143	0.860491	0.868738	0.863899
20	0.855042	0.859866	0.868517	0.862555
21	0.854886	0.861198	0.869760	0.864564
22	0.857089	0.862501	0.867324	0.864394
23	0.856557	0.860473	0.868122	0.862761
24	0.854343	0.862542	0.866527	0.862381

	split4_test_rmse	split5_test_rmse	split6_test_rmse	split7_test_rmse	\
0	0.858268	0.868882	0.884139	0.872449	
1	0.858600	0.867188	0.885630	0.871961	
2	0.858256	0.867446	0.885776	0.873733	
3	0.856717	0.867266	0.882903	0.872074	
4	0.858307	0.867316	0.883262	0.871396	
5	0.857676	0.868501	0.884497	0.868278	
6	0.856945	0.866494	0.883116	0.873233	
7	0.854187	0.866837	0.886208	0.871901	
8	0.854495	0.867827	0.885312	0.867919	
9	0.859429	0.865492	0.880813	0.871431	
10	0.860779	0.868127	0.883313	0.869771	
11	0.857228	0.867714	0.884962	0.871382	
12	0.857031	0.866568	0.885607	0.870292	
13	0.855137	0.867216	0.886548	0.871168	
14	0.856082	0.869630	0.884121	0.871328	
15	0.856074	0.865664	0.883019	0.869594	
16	0.857509	0.869356	0.881423	0.873645	
17	0.858265	0.866513	0.886724	0.872322	
18	0.858480	0.869327	0.883863	0.868765	
19	0.857938	0.871659	0.881234	0.870485	
20	0.856681	0.866374	0.884596	0.869509	
21	0.858861	0.866889	0.882152	0.867197	
22	0.855885	0.868077	0.883287	0.872560	
23	0.858165	0.865997	0.885428	0.872721	
24	0.858754	0.866547	0.889372	0.868326	

	split8_test_rmse	split9_test_rmse	…	split9_test_mae	mean_test_mae	\
0	0.865232	0.872935	…	0.670329	0.666662	
1	0.864714	0.871255	…	0.668748	0.666411	
2	0.867167	0.873462	…	0.670406	0.666919	
3	0.866276	0.871187	…	0.669826	0.665780	
4	0.863694	0.871522	…	0.668597	0.666385	
5	0.864805	0.868791	…	0.667055	0.666065	
6	0.864727	0.873338	…	0.670075	0.665746	
7	0.862534	0.870499	…	0.667154	0.665436	
8	0.864368	0.873385	…	0.670770	0.665787	
9	0.865249	0.872192	…	0.669039	0.665260	

10	0.863615	0.871100	…	0.668028	0.665321
11	0.864509	0.871752	…	0.668196	0.665891
12	0.865327	0.873260	…	0.669164	0.665475
13	0.863920	0.870500	…	0.667966	0.665363
14	0.867814	0.872364	…	0.667862	0.665651
15	0.866362	0.871713	…	0.668546	0.665121
16	0.865090	0.870049	…	0.667939	0.665746
17	0.864063	0.872061	…	0.668693	0.666073
18	0.863044	0.871439	…	0.667269	0.665197
19	0.865405	0.870926	…	0.668658	0.665621
20	0.864500	0.870555	…	0.666767	0.664877
21	0.865495	0.871928	…	0.668110	0.664922
22	0.866054	0.869982	…	0.668521	0.665695
23	0.865365	0.870461	…	0.667064	0.665393
24	0.865091	0.871964	…	0.669108	0.665812

	std_test_mae	rank_test_mae	mean_fit_time	std_fit_time	mean_test_time	\
0	0.006209	24	0.211396	0.042952	0.052285	
1	0.006110	23	0.223882	0.051099	0.044378	
2	0.006541	25	0.266599	0.040214	0.051747	
3	0.006422	16	0.405859	0.096560	0.062913	
4	0.005696	22	0.323615	0.053884	0.064438	
5	0.005953	20	0.320187	0.060719	0.054556	
6	0.006182	15	0.378984	0.101211	0.058823	
7	0.006826	9	0.505629	0.070298	0.068088	
8	0.006522	17	0.642190	0.040551	0.095004	
9	0.005685	5	0.656008	0.024609	0.076189	
10	0.005754	6	0.672362	0.062826	0.079897	
11	0.005911	19	0.677138	0.056967	0.076303	
12	0.006226	10	0.735488	0.009455	0.074345	
13	0.006865	7	0.758515	0.012367	0.076944	
14	0.006467	12	0.785583	0.012201	0.078575	
15	0.006127	3	0.817727	0.020935	0.091104	
16	0.005420	14	0.827303	0.024292	0.090229	
17	0.006471	21	0.875024	0.011501	0.089724	
18	0.005465	4	0.899685	0.016722	0.088519	
19	0.004986	11	0.920482	0.015121	0.089451	
20	0.006226	1	0.956555	0.023860	0.088969	
21	0.005344	2	0.977093	0.011350	0.094109	
22	0.005708	13	0.986926	0.010844	0.089897	
23	0.006206	8	1.034077	0.018811	0.092362	
24	0.006918	18	0.954922	0.151253	0.068182	

	std_test_time	params	param_n_factors
0	0.012199	{'n_factors': 2}	2
1	0.008530	{'n_factors': 4}	4
2	0.012504	{'n_factors': 6}	6

```
3        0.016011   {'n_factors': 8}             8
4        0.020763  {'n_factors': 10}            10
5        0.016094  {'n_factors': 12}            12
6        0.012324  {'n_factors': 14}            14
7        0.016071  {'n_factors': 16}            16
8        0.014062  {'n_factors': 18}            18
9        0.013837  {'n_factors': 20}            20
10       0.033913  {'n_factors': 22}            22
11       0.006501  {'n_factors': 24}            24
12       0.007485  {'n_factors': 26}            26
13       0.003745  {'n_factors': 28}            28
14       0.003383  {'n_factors': 30}            30
15       0.011206  {'n_factors': 32}            32
16       0.005246  {'n_factors': 34}            34
17       0.004192  {'n_factors': 36}            36
18       0.004890  {'n_factors': 38}            38
19       0.003532  {'n_factors': 40}            40
20       0.004816  {'n_factors': 42}            42
21       0.014279  {'n_factors': 44}            44
22       0.004165  {'n_factors': 46}            46
23       0.009281  {'n_factors': 48}            48
24       0.017934  {'n_factors': 50}            50

[25 rows x 32 columns]
```

1.16 Question 10.C

Performance on dataset subsets

1.16.1 Answer

The code and figures are shown below.

Results:

Subset	minimum average RMSE
Popular	0.856
Unpopular	0.897
High Variance	1.553

[ ]:
```
# Popular
file_path = './ratings_popular.csv'
results_df = grid_search_and_report(ModelType.MFBiased, file_path, 'Popular')
```

```
Start to grid search on 8 cores…

[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.5s
```

```
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    1.1s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    1.8s
[Parallel(n_jobs=8)]: Done  25 tasks      | elapsed:    2.6s
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    3.4s
[Parallel(n_jobs=8)]: Done  45 tasks      | elapsed:    4.5s
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:    5.8s
[Parallel(n_jobs=8)]: Done  69 tasks      | elapsed:    7.2s
[Parallel(n_jobs=8)]: Done  82 tasks      | elapsed:    8.8s
[Parallel(n_jobs=8)]: Done  97 tasks      | elapsed:   10.6s
[Parallel(n_jobs=8)]: Done 112 tasks      | elapsed:   12.4s
[Parallel(n_jobs=8)]: Done 129 tasks      | elapsed:   14.9s
[Parallel(n_jobs=8)]: Done 146 tasks      | elapsed:   17.4s
[Parallel(n_jobs=8)]: Done 165 tasks      | elapsed:   20.1s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:   22.7s
[Parallel(n_jobs=8)]: Done 205 tasks      | elapsed:   25.8s
[Parallel(n_jobs=8)]: Done 226 tasks      | elapsed:   28.8s
```

```
0.8557901459317451
Finish searching
```

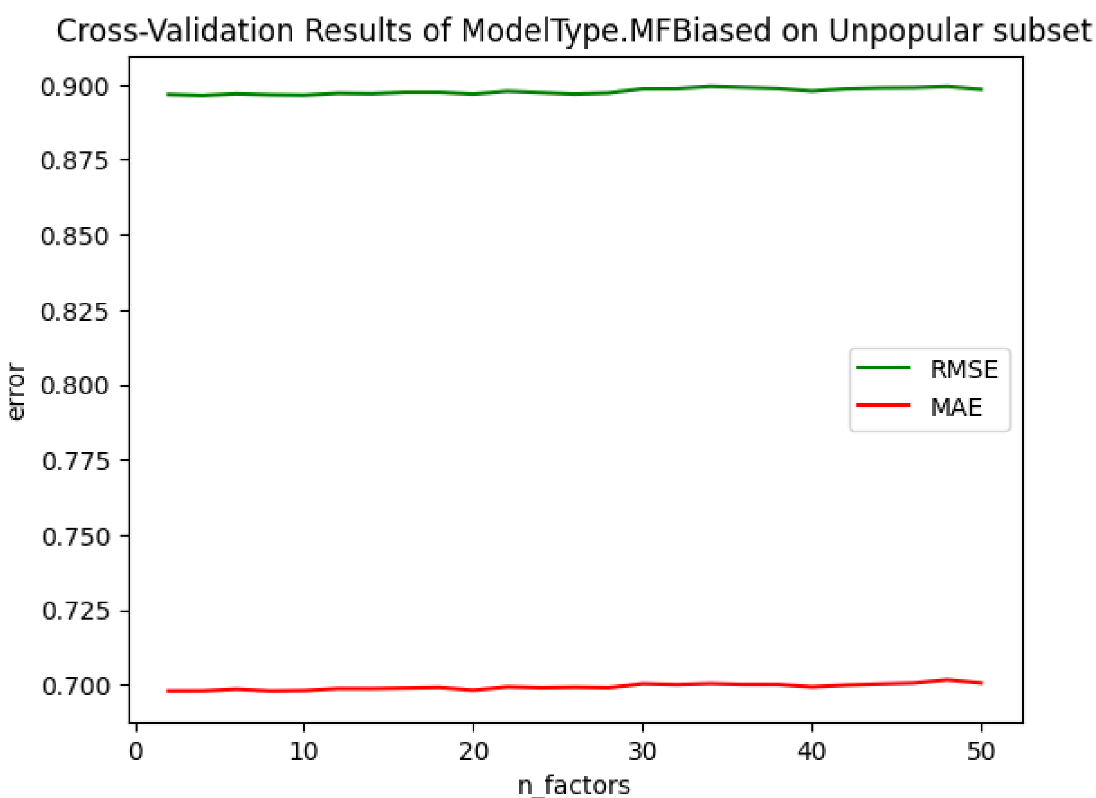
```
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:   32.1s finished
```


```
min_rmse_idx=15, min_rmse_k=32, min_rmse=0.8557901459317451, min_mae_idx=18,
```

```
min_mae_k=38, min_mae=0.6565735099832046
```

[ ]:
```
# Unpopular
file_path = './ratings_unpopular.csv'
results_df = grid_search_and_report(ModelType.MFBiased, file_path, 'Unpopular')
```

```
Start to grid search on 8 cores…

[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Batch computation too fast (0.046787261962890625s.)
Setting batch_size=2.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Batch computation too fast (0.14458990097045898s.) Setting
batch_size=4.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.3s
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:    0.5s
[Parallel(n_jobs=8)]: Done 100 tasks      | elapsed:    0.8s
[Parallel(n_jobs=8)]: Done 144 tasks      | elapsed:    1.3s
[Parallel(n_jobs=8)]: Done 196 tasks      | elapsed:    1.7s

0.8962927178677791
Finish searching

[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:    2.2s finished
```



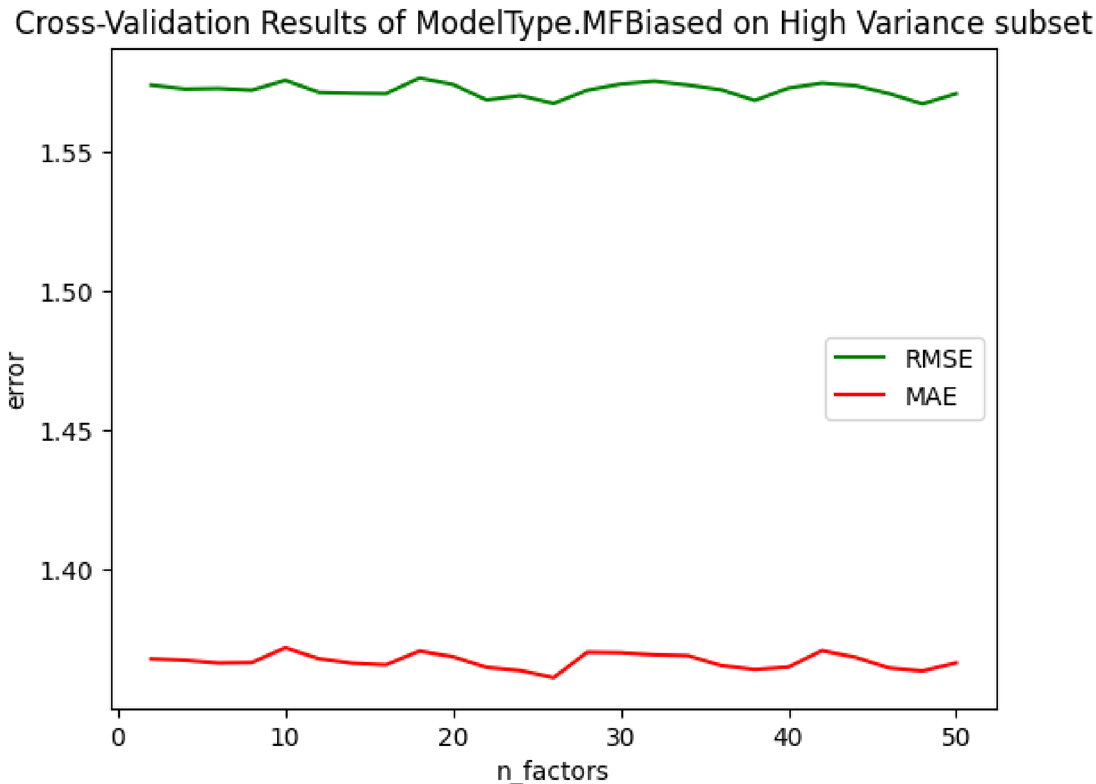
```
min_rmse_idx=1, min_rmse_k=4, min_rmse=0.8962927178677791, min_mae_idx=3,
min_mae_k=8, min_mae=0.6979247525432892
```

[ ]:
```python
# High Variance
file_path = './ratings_high_var.csv'
results_df = grid_search_and_report(ModelType.MFBiased, file_path, 'High␣
↪Variance')
```

```
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Batch computation too fast (0.0027716159820556640s.)
Setting batch_size=2.
[Parallel(n_jobs=8)]: Done   2 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done   9 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Batch computation too fast (0.009077310562133789s.)
Setting batch_size=4.
```

```
Start to grid search on 8 cores…
1.567049408307865
Finish searching
```

```
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Batch computation too fast (0.018002033233642578s.)
Setting batch_size=8.
[Parallel(n_jobs=8)]: Done  56 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Batch computation too fast (0.029534578323364258s.)
Setting batch_size=16.
[Parallel(n_jobs=8)]: Done 176 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 250 out of 250 | elapsed:    0.1s finished
```




```
min_rmse_idx=23, min_rmse_k=48, min_rmse=1.567049408307865, min_mae_idx=12,
min_mae_k=26, min_mae=1.3613933744377442
```

1.17 Question 10.D

Plot the ROC curves for the MF-based collaborative filter and also report the area under the curve (AUC) value

1.17.1 Answer

The code and figure are shown below.

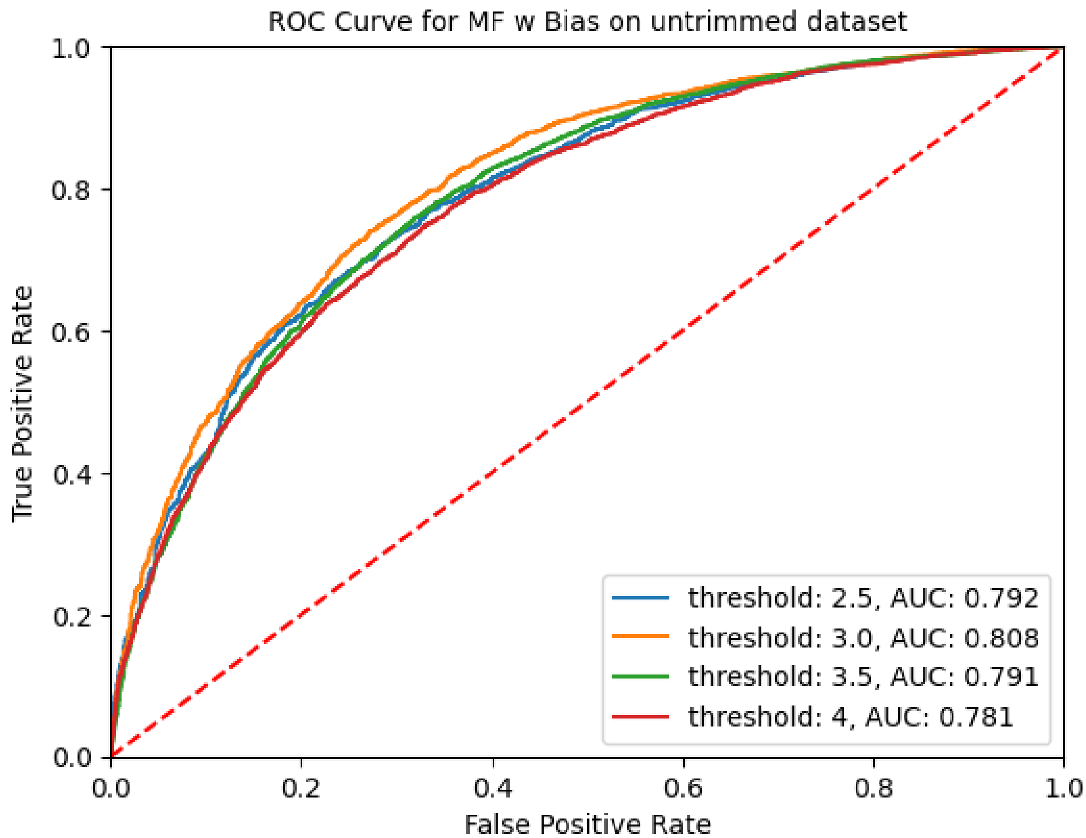
AUC:

Dataset	k	threshold	AUC
untrimmed	42	2.5	0.792
untrimmed	42	3	0.808
untrimmed	42	3.5	0.791
untrimmed	42	4	0.781
popular	42	2.5	0.777
popular	32	3	0.803
popular	32	3.5	0.779
popular	32	4	0.787

Dataset	k	threshold	AUC
unpopular	4	2.5	0.830
unpopular	4	3	0.793
unpopular	4	3.5	0.824
unpopular	4	4	0.782
high variance	26	2.5	0.643
high variance	26	3	0.766
high variance	26	3.5	0.683
high variance	26	4	0.691

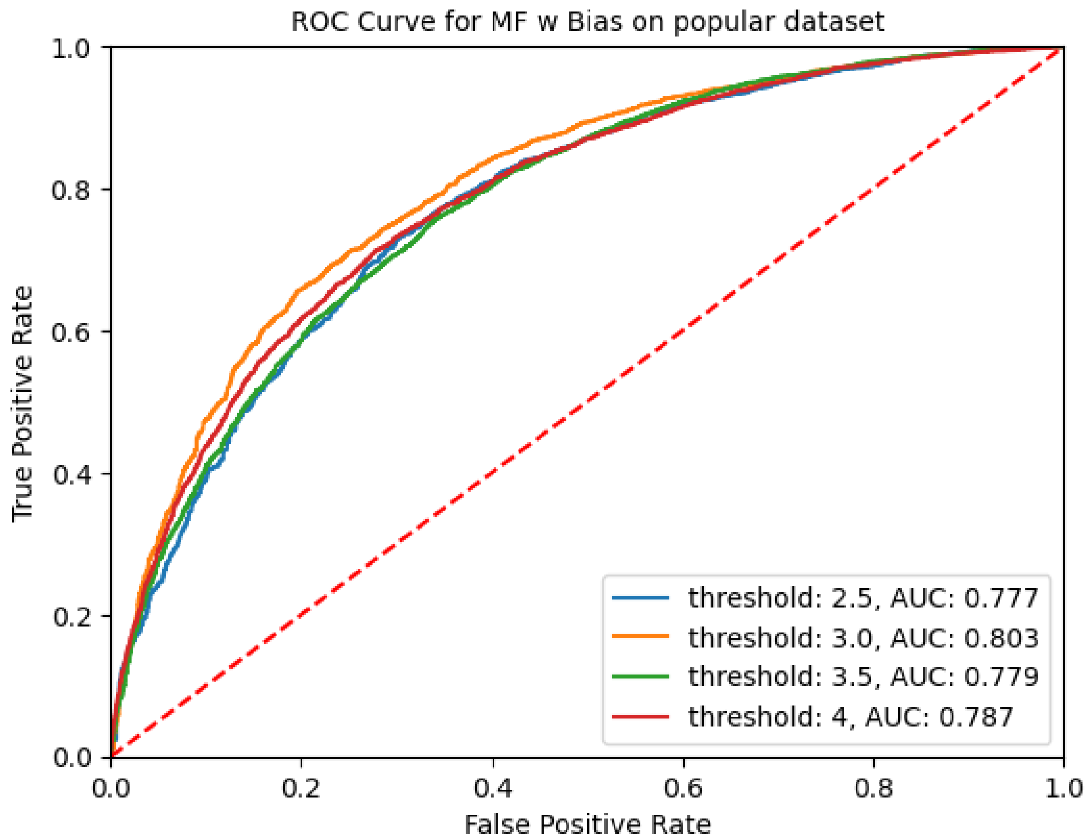
[ ]:
```
k = 42
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.MFBiased, k, './ratings_modified.csv', 'untrimmed',␣
↪t)
plt.title(f'ROC Curve for MF w Bias on untrimmed dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for MF w Bias on untrimmed dataset with threshold=2.5: 0.792
AUC for MF w Bias on untrimmed dataset with threshold=3.0: 0.808
AUC for MF w Bias on untrimmed dataset with threshold=3.5: 0.791
AUC for MF w Bias on untrimmed dataset with threshold=4: 0.781
```



[ ]:
```
k = 32
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.MFBiased, k, './ratings_popular.csv', 'popular', t)
plt.title(f'ROC Curve for MF w Bias on popular dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

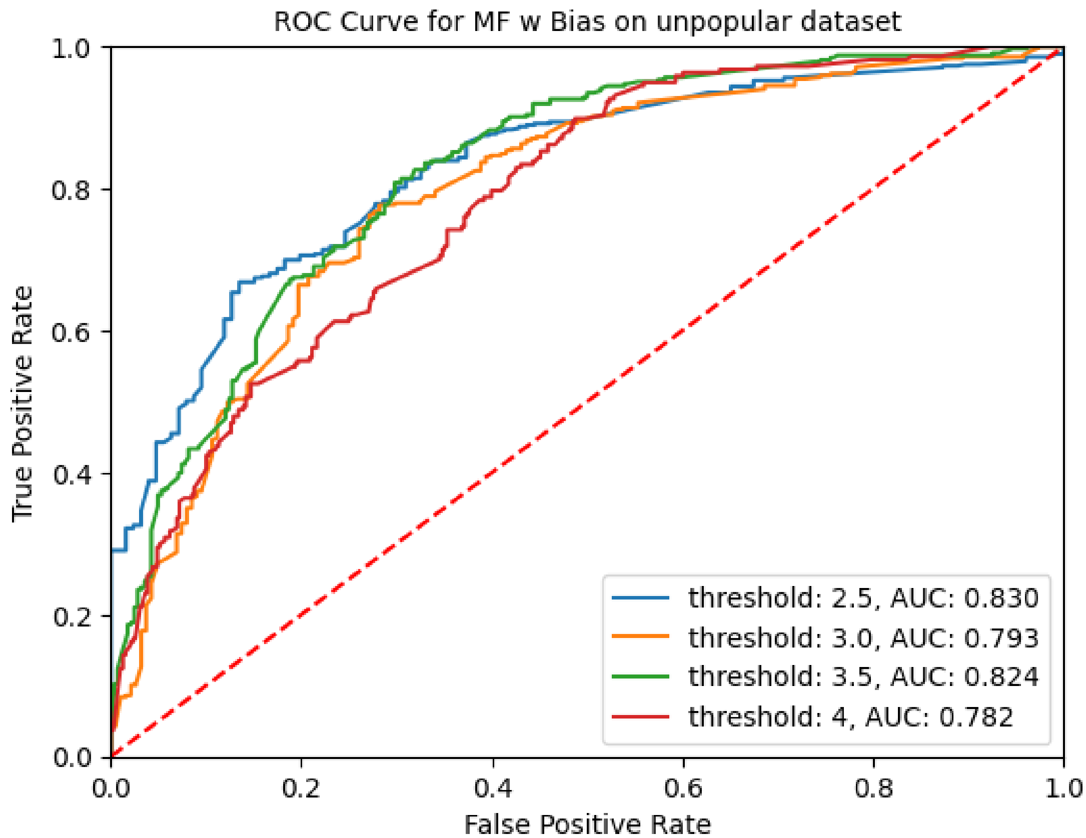
```
AUC for MF w Bias on popular dataset with threshold=2.5: 0.777
AUC for MF w Bias on popular dataset with threshold=3.0: 0.803
AUC for MF w Bias on popular dataset with threshold=3.5: 0.779
AUC for MF w Bias on popular dataset with threshold=4: 0.787
```



[ ]:

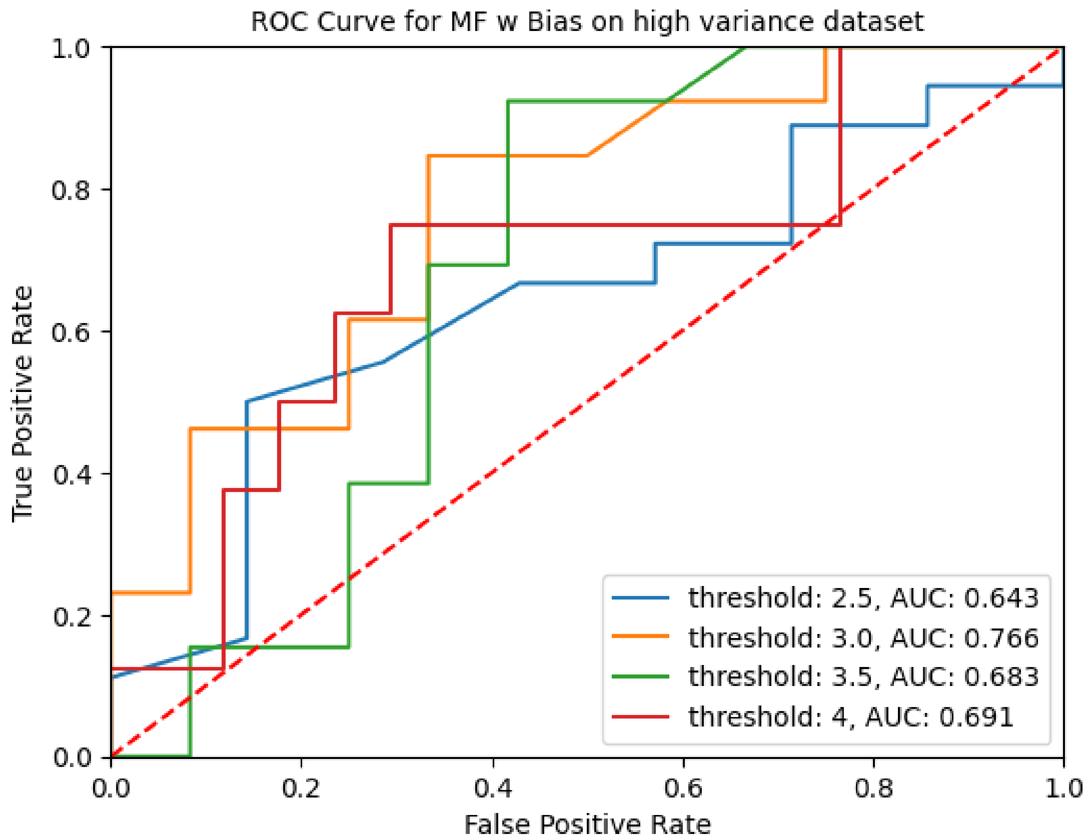
```
k = 4
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.MFBiased, k, './ratings_unpopular.csv',␣
 ↪'unpopular', t)
plt.title(f'ROC Curve for MF w Bias on unpopular dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for MF w Bias on unpopular dataset with threshold=2.5: 0.830
AUC for MF w Bias on unpopular dataset with threshold=3.0: 0.793
AUC for MF w Bias on unpopular dataset with threshold=3.5: 0.824
AUC for MF w Bias on unpopular dataset with threshold=4: 0.782
```



[ ]:
```python
k = 26
thresholds = [2.5, 3., 3.5, 4]
for i, t in enumerate(thresholds):
  get_roc_and_auc(ModelType.MFBiased, k, './ratings_high_var.csv', 'high␣
↪variance', t)
plt.title(f'ROC Curve for MF w Bias on high variance dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for MF w Bias on high variance dataset with threshold=2.5: 0.643
AUC for MF w Bias on high variance dataset with threshold=3.0: 0.766
AUC for MF w Bias on high variance dataset with threshold=3.5: 0.683
AUC for MF w Bias on high variance dataset with threshold=4: 0.691
```



1.18 Question 11

Designing a Naive Collaborative Filter

1.18.1 Answer

The code is shown below.

Results:

Dataset	RMSE	MAE
untrimmed	0.935	0.729
popular	0.932	0.728
unpopular	0.971	0.749
high variance	1.466	1.182

[54]:
```
import pandas as pd
import numpy as np
import statistics
import random
```

```python
import warnings
from sklearn.metrics import root_mean_squared_error, mean_absolute_error

'''
Define Naive Collaborative Filter
'''
class NaiveCF(object):
  def __init__(self):
    self.U = None
    self.uid_lut = {}
    self.uid_lut_reverse = {}
    self.mid_lut = {}
    self.mid_lut_reverse = {}

  def embedding(self, dataset: list[list]):
    '''
      Generate uid and mid look up table
    '''
    uid_cnt = 0
    mid_cnt = 0
    for uid, mid, rating in dataset:
      if uid not in self.uid_lut:
        self.uid_lut[uid] = uid_cnt
        self.uid_lut_reverse[uid_cnt] = uid
        uid_cnt += 1
      if mid not in self.mid_lut:
        self.mid_lut[mid] = mid_cnt
        self.mid_lut_reverse[mid_cnt] = mid
        mid_cnt += 1

    return uid_cnt

  def fit(self, dataset: list[list]):
    num_users = self.embedding(dataset)
    self.U = [[] for _ in range(num_users)]
    for uid, _, rating in dataset:
      self.U[self.uid_lut[uid]].append(rating)

    self.U = list(map(statistics.mean, self.U))

  def predict(self, X: list[list]):
    return [self.U[self.uid_lut[uid]] for uid, _ in X]

  def score(self, X: list[list], y: list[float]):
    pred = self.predict(X)

    rmse = root_mean_squared_error(y, pred)
```

```
    mae = mean_absolute_error(y, pred)
    return rmse, mae


'''
Generate kfold datasets
'''
def kfold(data: list[list], num_folds: int):
  random.shuffle(data)

  p = 1. / num_folds
  batchsize = int(len(data) * p)
  for i in range(num_folds):
    trainset = []
    testset = []
    if i == 0:
      trainset = data[batchsize:]
      testset = data[:batchsize]
    else:
      trainset = data[:(i * batchsize)] + data[((i + 1) * batchsize):]
      testset = data[(i * batchsize):((i + 1) * batchsize)]

    yield trainset, testset



def load_dataset(filepath: str):
  df_ratings = pd.read_csv(filepath)
  print(f'total data size: {len(df_ratings)}')

  data = []
  for idx in df_ratings.index:
    userid = df_ratings['userId'][idx]
    data.append([userid, df_ratings['movieId'][idx], df_ratings['rating'][idx]])

  return data


def load_and_split(filepath: str, num_folds: int):
  df_ratings = pd.read_csv(filepath)
  print(f'total data size: {len(df_ratings)}')

  data = []
  for idx in df_ratings.index:
    userid = df_ratings['userId'][idx]
    data.append([userid, df_ratings['movieId'][idx], df_ratings['rating'][idx]])
```

```python
  yield from kfold(data, num_folds)


# separate inputs and labels
def separate_xy(dataset: list[list]):
  x = []
  y = []
  for uid, mid, rating in dataset:
    x.append([uid, mid])
    y.append(rating)

  return x, y
```

[55]:
```python
model = NaiveCF()
model.fit(load_dataset('./ratings_modified.csv'))
```

```
total data size: 100836
```

[57]:
```python
# untrimmed

rmse_list = []
mae_list = []
file_path = './ratings_modified.csv'
for trainset, testset in load_and_split(file_path, 10):
  test_x, test_y = separate_xy(testset)
  rmse, mae = model.score(test_x, test_y)
  rmse_list.append(rmse)
  mae_list.append(mae)

print(f'{rmse_list=}')
print(f'{mae_list=}')

print(f'[Untrimmed] Average RMSE: {statistics.mean(rmse_list)}; Average MAE:␣
 ↪{statistics.mean(mae_list)}')
```

```
total data size: 100836
rmse_list=[0.940478582210219, 0.9277912724114626, 0.940918758094466,
0.9265149534812046, 0.9317419017774499, 0.9361118804640998, 0.9434515249837891,
0.9419110005551677, 0.9284656789723668, 0.9298061418957644]
mae_list=[0.7341448479370711, 0.726455859401225, 0.7339103228844339,
0.7235946673300209, 0.7237648857814186, 0.7317763091009439, 0.7332301053099151,
0.7321469863416844, 0.7233830919139342, 0.7269778793988385]
[Untrimmed] Average RMSE: 0.934719169484599; Average MAE: 0.7289384955399486
```

[58]:
```python
rmse_list = []
mae_list = []
file_path = './ratings_popular.csv'
for trainset, testset in load_and_split(file_path, 10):
```

```python
    test_x, test_y = separate_xy(testset)
    rmse, mae = model.score(test_x, test_y)
    rmse_list.append(rmse)
    mae_list.append(mae)

print(f'{rmse_list=}')
print(f'{mae_list=}')

print(f'[Popular] Average RMSE: {statistics.mean(rmse_list)}; Average MAE: {statistics.mean(mae_list)}')
```

```
total data size: 94794
rmse_list=[0.9319156174715126, 0.9295284012485507, 0.9312461760595184,
0.9449823594927991, 0.931276147577174, 0.9129004343141289, 0.9338255648188031,
0.9379707657632441, 0.9332362977270618, 0.9361614523202161]
mae_list=[0.7267171835931441, 0.7254379443238842, 0.7279265472353758,
0.7364541165848963, 0.7241991441171766, 0.7169082075225771, 0.7319351765405503,
0.7285951956570422, 0.7263749789829711, 0.7321440558415352]
[Popular] Average RMSE: 0.9323043216793009; Average MAE: 0.7276692550399153
```

[59]:
```python
rmse_list = []
mae_list = []
file_path = './ratings_unpopular.csv'
for trainset, testset in load_and_split(file_path, 10):
    test_x, test_y = separate_xy(testset)
    rmse, mae = model.score(test_x, test_y)
    rmse_list.append(rmse)
    mae_list.append(mae)

print(f'{rmse_list=}')
print(f'{mae_list=}')

print(f'[Unpopular] Average RMSE: {statistics.mean(rmse_list)}; Average MAE: {statistics.mean(mae_list)}')
```

```
total data size: 6042
rmse_list=[0.9241272249588524, 0.9752081141368454, 0.9614429119190228,
1.0484995761067502, 0.9330703028820446, 0.9724269319153618, 0.9347954080432576,
0.9983194983719261, 0.9362580654143215, 1.0223196032891513]
mae_list=[0.7131352163106783, 0.7426895806531257, 0.7324556341854607,
0.8145719233280024, 0.7237635458617655, 0.7544883896291348, 0.7276049414802067,
0.7579644959559309, 0.7327212418590506, 0.7859851993954291]
[Unpopular] Average RMSE: 0.9706467637037534; Average MAE: 0.7485380168658785
```

[60]:
```python
rmse_list = []
mae_list = []
file_path = './ratings_high_var.csv'
for trainset, testset in load_and_split(file_path, 10):
```

```
  test_x, test_y = separate_xy(testset)
  rmse, mae = model.score(test_x, test_y)
  rmse_list.append(rmse)
  mae_list.append(mae)

print(f'{rmse_list=}')
print(f'{mae_list=}')

print(f'[High Variance] Average RMSE: {statistics.mean(rmse_list)}; Average MAE:
 ↪ {statistics.mean(mae_list)}')
```

```
total data size: 250
rmse_list=[1.2038925570180217, 1.4474061317000875, 1.6810820092453402,
1.7769234986667286, 1.2500137850027706, 1.4254056598023537, 1.2204950296401045,
1.7981133564744913, 1.437424005760509, 1.4217527323071506]
mae_list=[0.9622631539006377, 1.1927212721871068, 1.3983758416861334,
1.4678929352600196, 1.0111500218443552, 1.0845254607892856, 0.92356990742214,
1.4359880850239295, 1.1714026104461277, 1.17021564421123]
[High Variance] Average RMSE: 1.4662508765617557; Average MAE:
1.1818104932770965
```

1.19 Question 12

Comparing the most performant models across architecture: Plot the best ROC curves (threshold = 3) for the k-NN, NMF, and MF with bias based collaborative filters in the same figure. Use the figure to compare the performance of the filters in predicting the ratings of the movies.

1.19.1 Answer

The code and figure are shown below.

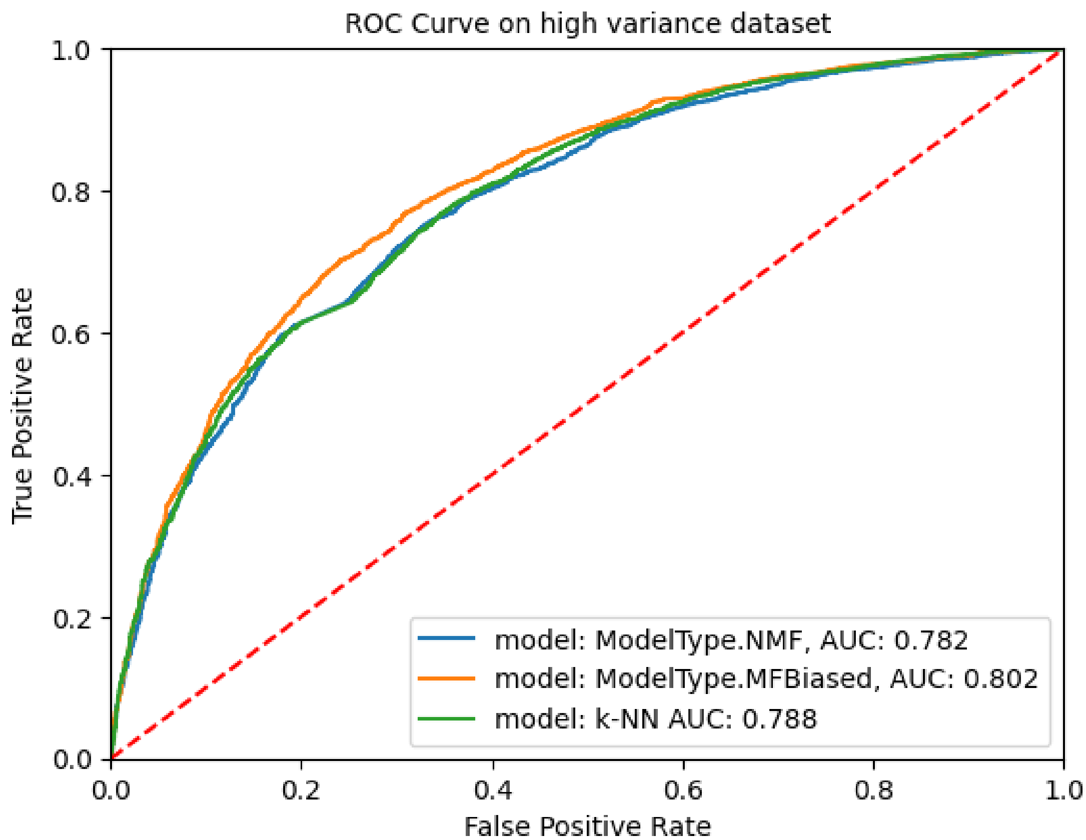
As we can see from the ROC curve and AUC score, the Matrix Factorization with Bias method outperform other methods.

[ ]:
```
from sklearn.metrics import roc_curve, auc, RocCurveDisplay
from surprise import BaselineOnly, Dataset, Reader, NMF, SVD, KNNWithMeans
from surprise.model_selection import train_test_split
import matplotlib.pyplot as plt

def get_roc_and_auc_various_model(model_name: ModelType, k: int,␣
 ↪dataset_filepath: str, dataset_cat: str, threshold: float):
  reader = Reader(line_format="user item rating timestamp", sep=",",␣
 ↪skip_lines=1)
  data = Dataset.load_from_file(dataset_filepath, reader=reader)
  trainset, testset = train_test_split(data, test_size=0.1)
  model = MODEL_MAP[model_name](n_factors=k, random_state=42)
  model.fit(trainset)
  pred = []
  for ele in testset: # (uid, iid, gt)
```

```python
        pred.append(model.predict(uid=ele[0], iid=ele[1], r_ui=ele[2]))

    y_gt = [x.r_ui for x in pred]
    y_pred = [x.est for x in pred]

    def apply_threshold(score):
        return 1 if score >= threshold else 0

    def normalize(score):
        return score / 5.0

    y_gt_binary = list(map(apply_threshold, y_gt))
    y_pred_norm = list(map(normalize, y_pred))

    fpr, tpr, thresholds = roc_curve(y_gt_binary, y_pred_norm)
    roc_auc = auc(fpr, tpr)
    print(f'AUC for {model_name.value} on {dataset_cat} dataset with {threshold=}:
 ↪ {roc_auc:.3f}')
    plt.plot(fpr, tpr, label=f'model: {str(model_name)}, AUC: {roc_auc:.3f}')


get_roc_and_auc_various_model(ModelType.NMF, 18, './ratings_modified.csv',␣
 ↪'untrimmed', 3)
get_roc_and_auc_various_model(ModelType.MFBiased, 42, './ratings_modified.csv',␣
 ↪'untrimmed', 3)


ratings_df = pd.read_csv("./ratings.csv", usecols=['userId', 'movieId',␣
 ↪'rating'])
reader = Reader(rating_scale=(0.5, 5))
data = Dataset.load_from_df(ratings_df[['userId', 'movieId', 'rating']], reader)

trainset, testset = train_test_split(data, test_size=0.1)

k = 24
algo = KNNWithMeans(k=k, sim_options={'name': 'pearson', 'user_based': True})
algo.fit(trainset)
predictions = algo.test(testset)
threshold = 3
actual = np.array([pred.r_ui for pred in predictions])
actual_binary = (actual >= threshold).astype(int)
estimated = np.array([pred.est for pred in predictions])
fpr, tpr, _ = roc_curve(actual_binary, estimated)
roc_auc = auc(fpr, tpr)
print(f'AUC for k-NN on untrimmed dataset with {threshold=}: {roc_auc:.3f}')

plt.plot(fpr, tpr, label=f'model: k-NN AUC: {roc_auc:.3f}')
```

```
plt.title(f'ROC Curve on high variance dataset', fontsize=10)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

```
AUC for NMF on untrimmed dataset with threshold=3: 0.782
AUC for MF w Bias on untrimmed dataset with threshold=3: 0.802
Computing the pearson similarity matrix…
Done computing similarity matrix.
AUC for k-NN on untrimmed dataset with threshold=3: 0.788
```



[ ]:
```
# mount to google drive to get data
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
from sklearn.datasets import load_svmlight_file
from sklearn.metrics import ndcg_score
import numpy as np
import pandas as pd

# Load the dataset for one fold
def load_one_fold(data_path):
    X_train, y_train, qid_train = load_svmlight_file(str(data_path + 'train.
 ↪txt'), query_id=True)
    X_test, y_test, qid_test = load_svmlight_file(str(data_path + 'test.txt'),␣
 ↪query_id=True)
    X_valid, y_valid, qid_valid = load_svmlight_file(str(data_path + 'vali.
 ↪txt'), query_id=True)
    y_train = y_train.astype(int)
    y_test = y_test.astype(int)
    _, group_train = np.unique(qid_train, return_counts=True) # counts of the␣
 ↪unique values
    _, group_test = np.unique(qid_test, return_counts=True) # counts of the␣
 ↪unique values
    _, group_valid = np.unique(qid_valid, return_counts=True) # counts of the␣
 ↪unique values
    return X_train, y_train, qid_train, group_train, X_test, y_test, qid_test,␣
 ↪group_test, X_valid, y_valid, qid_valid, group_valid

def ndcg_single_query(y_score, y_true, k):
    order = np.argsort(y_score)[::-1]
    y_true = np.take(y_true, order[:k])

    gain = 2 ** y_true - 1

    discounts = np.log2(np.arange(len(y_true)) + 2)
    return np.sum(gain / discounts)

# calculate NDCG score given a trained model
def compute_ndcg_all(model, X_test, y_test, qids_test, k=10):
    unique_qids = np.unique(qids_test)
    ndcg_ = list()
    for i, qid in enumerate(unique_qids):
        y = y_test[qids_test == qid]

        if np.sum(y) == 0:
            continue

        p = model.predict(X_test[qids_test == qid])

        idcg = ndcg_single_query(y, y, k=k)
```

```
        ndcg_.append(ndcg_single_query(p, y, k=k) / idcg)
    return np.mean(ndcg_)

# get importance of features
# def get_feature_importance(model, importance_type='gain'):
#     return model.booster_.feature_importance(importance_type=importance_type)
def get_feature_importance(model, reduced_indices=None, importance_type='gain'):

    if reduced_indices:
      feature_order = reduced_indices
    else:
      feature_order = model.feature_name()

    importance_df = (
        pd.DataFrame({
            'feature_name': feature_order,
            'importance_gain': model.feature_importance(importance_type='gain'),
            'importance_split': model.
 ↪feature_importance(importance_type='split'),
        })
        .sort_values('importance_gain', ascending=False)
        .reset_index(drop=True)
    )
    return importance_df
```

1.20 QUESTION 13: Data Understanding and Preprocessing

- Loading and pre-processing Web10k data.
- Print out the number of unique queries in total and show distribution of relevance labels

[ ]:
```
fold = 1
data_path = '/content/drive/MyDrive/UCLA Course/Winter 2024/ECE 219/
 ↪Project3-Recommender Systems/MSLR-WEB10K'
X_train, y_train, qid_train, group_train, X_test, y_test, qid_test, group_test,␣
 ↪X_valid, y_valid, qid_valid, group_valid = load_one_fold(data_path + '/
 ↪Fold'+str(fold)+'/')
```

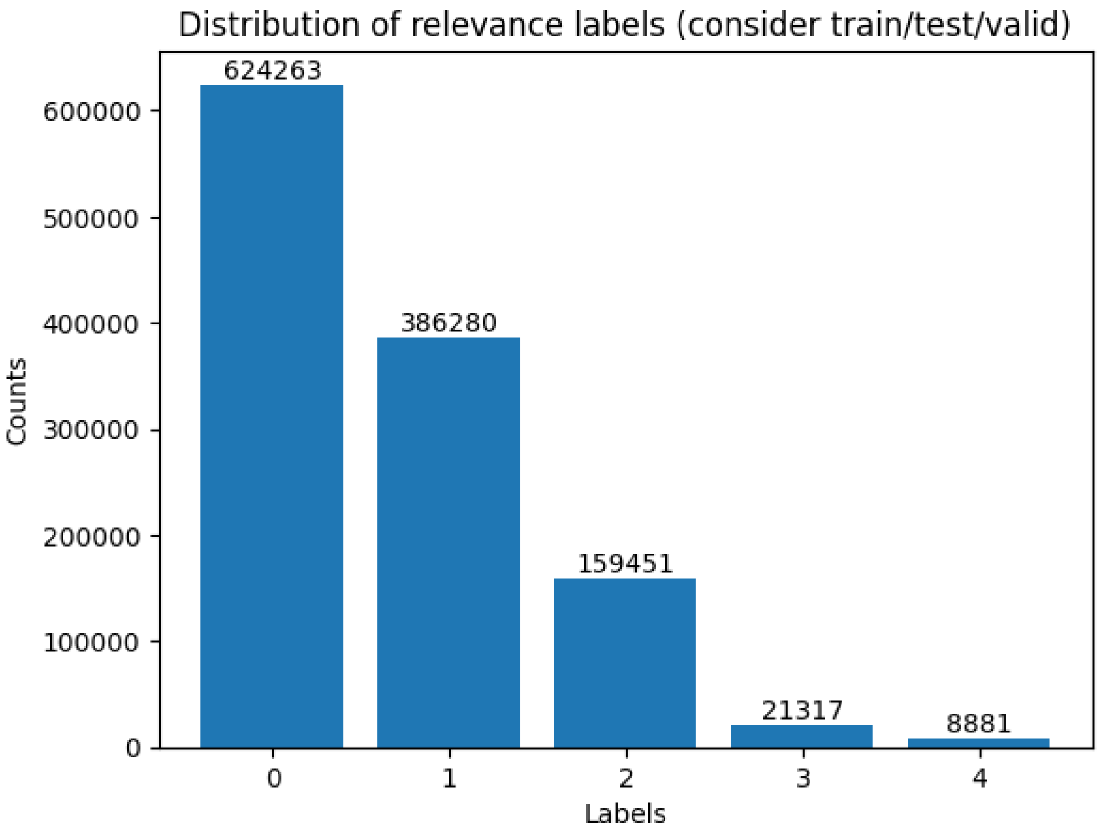
[ ]:
```
# calculate the number of unique queries
unique_qids = np.unique(np.concatenate((qid_train, qid_test, qid_valid)))
num_unique_queries = len(unique_qids)
print(f"Number of unique queries: {num_unique_queries}")
```

```
Number of unique queries: 10000
```

[ ]:
```
import matplotlib.pyplot as plt

# show distribution of relevance labels
all_labels = np.concatenate((y_train, y_test, y_valid))
```

```python
labels, counts = np.unique(all_labels, return_counts=True)
label_distribution = {label: counts[i] for i, label in enumerate(labels)}

print(f"Distribution of relevance labels (consider train/test/valid):")
for label, count in label_distribution.items():
    print(f"Label {int(label)}: {count}")

# plot the distribution
plt.bar(label_distribution.keys(), label_distribution.values())

for k, v in label_distribution.items():
  plt.text(k, v, v, ha = 'center', va = 'bottom')

plt.xlabel("Labels")
plt.ylabel("Counts")
plt.title("Distribution of relevance labels (consider train/test/valid)")
plt.show()
```

```
Distribution of relevance labels (consider train/test/valid):
Label 0: 624263
Label 1: 386280
Label 2: 159451
Label 3: 21317
Label 4: 8881
```



1.21 QUESTION 14 & QUESTION 15

1.21.1 QUESTION 14: LightGBM Model Training

For each of the five provided folds, train a LightGBM model using the `lambdarank` objective. After training, evaluate and report the model's performance on the test set using nDCG@3, nDCG@5 and nDCG@10.

Answer:

nDCG	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.4446	0.4487	0.4425	0.4514	0.4607
nDCG@5	0.4546	0.4521	0.4509	0.4588	0.4649
nDCG@10	0.4702	0.4675	0.4692	0.4791	0.4842

1.21.2 QUESTION 15: Result Analysis and Interpretation

For each of the five provided folds, list top 5 most important features of the model based on the importance score. Use importance_type='gain'.

Answer:

No.	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
1	Column_133	Column_133	Column_133	Column_133	Column_133
2	Column_7	Column_107	Column_107	Column_54	Column_54
3	Column_54	Column_54	Column_54	Column_107	Column_7
4	Column_107	Column_129	Column_129	Column_129	Column_107
5	Column_129	Column_7	Column_7	Column_128	Column_128

[ ]:
```
!pip install lightgbm
```

[ ]:
```
import lightgbm as lgb

def train_and_evaluate_one_fold_with_validation(data_path, X_train, y_train,␣
 ↪group_train, X_test, y_test, qid_test, group_test, X_valid, y_valid,␣
 ↪qid_valid, group_valid):

  # prepare train, valid, test datasets
  train_data = lgb.Dataset(X_train, label=y_train, group=group_train,␣
 ↪free_raw_data=False)
  valid_data = lgb.Dataset(X_valid, label=y_valid, group=group_valid,␣
 ↪free_raw_data=False)
  test_data = lgb.Dataset(X_test, label=y_test, group=group_test,␣
 ↪free_raw_data=False, reference=train_data)

  # parameters for LightGBM
  params = {
      'objective': 'lambdarank',
      'metric': 'ndcg',
      'ndcg_eval_at': [3, 5, 10],
      'learning_rate': 0.05,
      'num_leaves': 31,
      'verbose': -1
  }

  # train the model
  num_boost_round = 100
  lgb.cv(params, train_data, num_boost_round, nfold=5) # cv
  gbm = lgb.train(params, train_data, num_boost_round, valid_sets=[valid_data],␣
 ↪callbacks=[lgb.early_stopping(stopping_rounds=10)])

  # evaluate the model
  print('\nModel performance on the test set (nDCG@3, nDCG@5, nDCG@10):')
  for k in [3, 5, 10]:
    ndcg_scores = compute_ndcg_all(gbm, X_test, y_test, qid_test, k)  # adjust␣
 ↪this for k=3, 5, 10
```

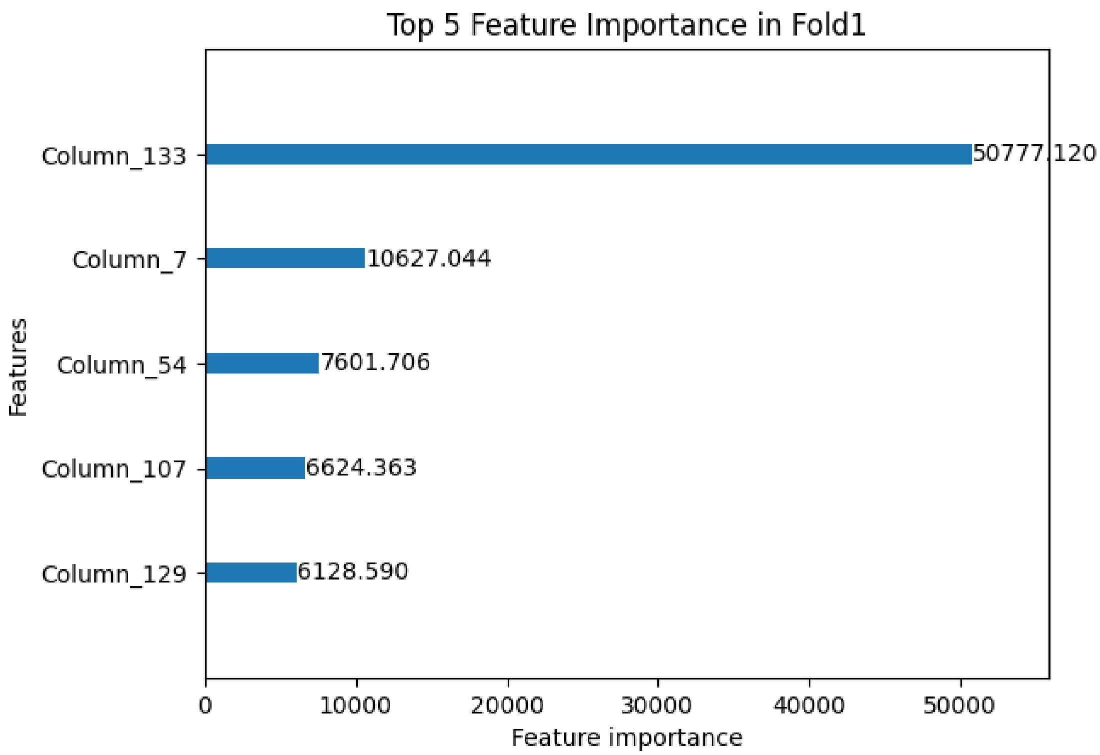
```
    print(f'k = {k}, testing ndcg score: {ndcg_scores}')

  return gbm

def plot_feature_importance(gbm, fold):
  # feature importance
  feature_importance = get_feature_importance(gbm)
  print(f'\nTop 5 feature importance in Fold{fold}:\n{feature_importance.
 ↪head(5)}\n')

  # plot top 5 most important features
  lgb.plot_importance(gbm, importance_type='gain', max_num_features=5,␣
 ↪title=f'Top 5 Feature Importance in Fold{fold}', grid=False)
  plt.show()
```

[ ]:

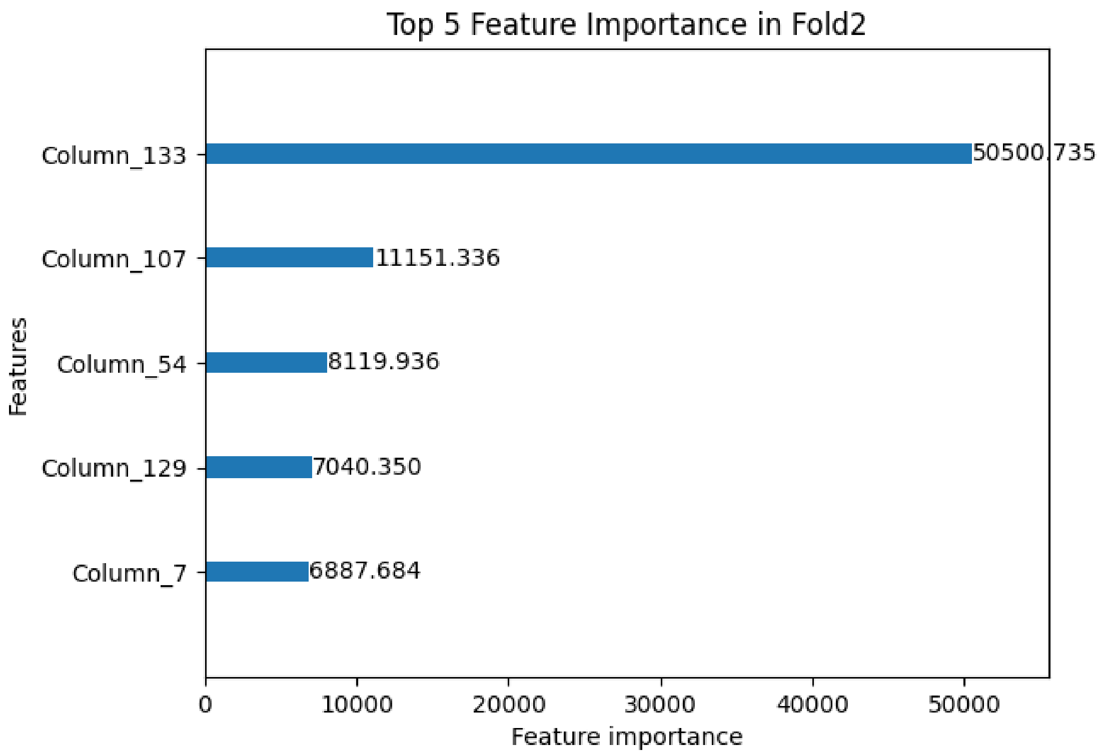
```
data_path = '/content/drive/MyDrive/UCLA Course/Winter 2024/ECE 219/
 ↪Project3-Recommender Systems/MSLR-WEB10K'

for fold in range(1, 6):
  print(f'\n============= Fold{fold} =============')
  # load one fold data
  X_train, y_train, qid_train, group_train, X_test, y_test, qid_test,␣
 ↪group_test, X_valid, y_valid, qid_valid, group_valid =␣
 ↪load_one_fold(data_path + '/Fold'+str(fold)+'/')
  # train and evaluate model
  gbm = train_and_evaluate_one_fold_with_validation(data_path, X_train,␣
 ↪y_train, group_train, X_test, y_test, qid_test, group_test, X_valid,␣
 ↪y_valid, qid_valid, group_valid)
  plot_feature_importance(gbm, fold)
```

```
============= Fold1 =============
Training until validation scores don't improve for 10 rounds
Early stopping, best iteration is:
[73]	valid_0's ndcg@3: 0.473367	valid_0's ndcg@5: 0.476424
valid_0's ndcg@10: 0.493763

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.44456992515667343
k = 5, testing ndcg score: 0.45456467531943034
k = 10, testing ndcg score: 0.4701929292191055

Top 5 feature importance in Fold1:
  feature_name  importance_gain  importance_split
0   Column_133     50777.120073               103
1     Column_7     10627.043671                30
2    Column_54      7601.705648                27
```

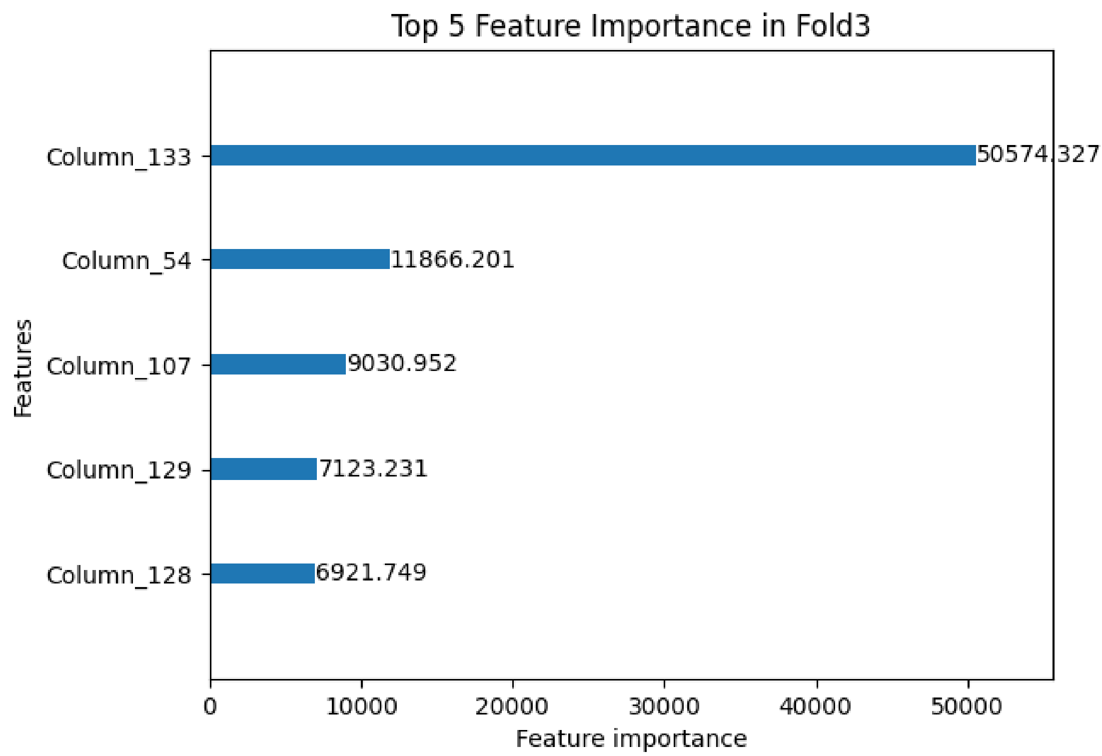
```
3    Column_107       6624.363126                130
4    Column_129       6128.589797                114
```



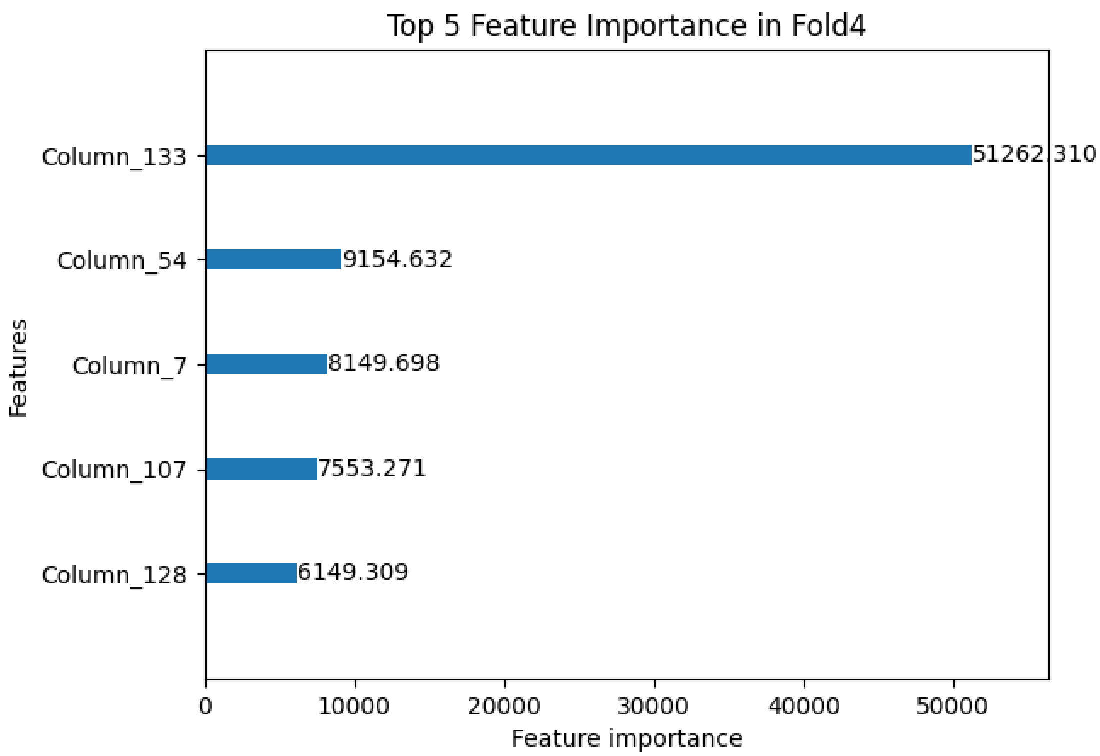

```
============= Fold2 =============
Training until validation scores don't improve for 10 rounds
Early stopping, best iteration is:
[75]    valid_0's ndcg@3: 0.462326      valid_0's ndcg@5: 0.469716
valid_0's ndcg@10: 0.488841

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.44874691316801657
k = 5, testing ndcg score: 0.4520912281087201
k = 10, testing ndcg score: 0.46754763319425136

Top 5 feature importance in Fold2:
  feature_name  importance_gain  importance_split
0   Column_133     50500.734923                98
1   Column_107     11151.335602               157
2    Column_54      8119.935635                37
3   Column_129      7040.350111               143
4     Column_7      6887.684011                22
```



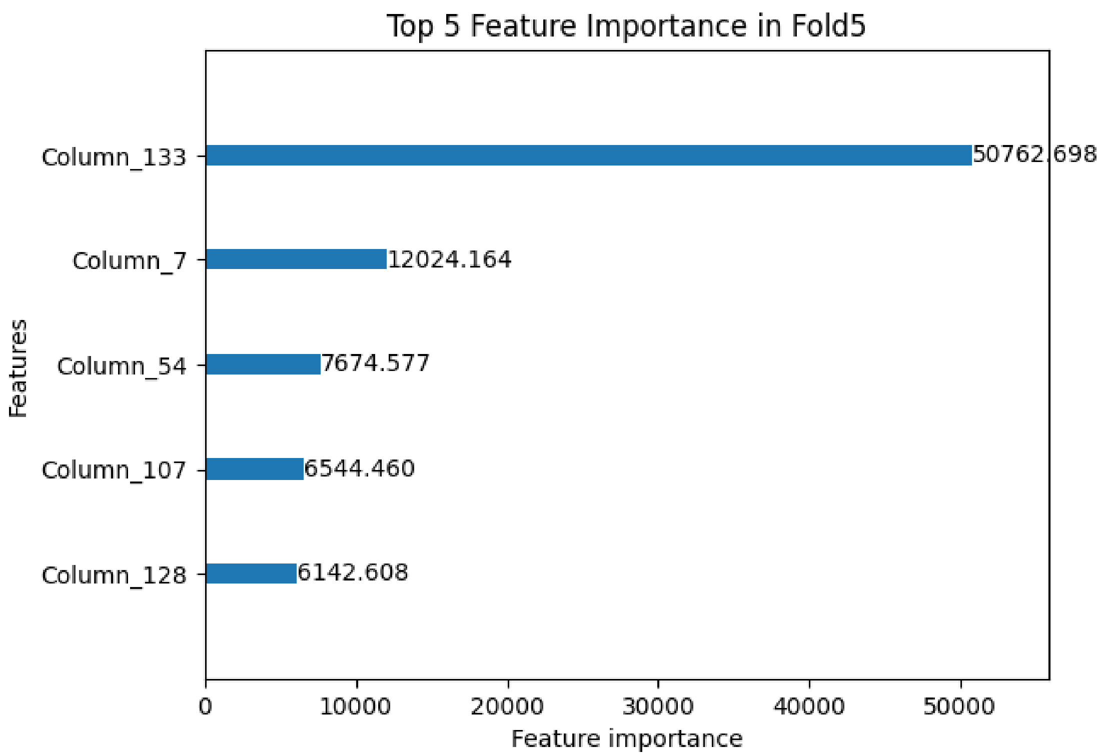

```
============= Fold3 =============
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[100]   valid_0's ndcg@3: 0.474074      valid_0's ndcg@5: 0.476085
valid_0's ndcg@10: 0.492215

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.44251536463540597
k = 5, testing ndcg score: 0.4508952689136397
k = 10, testing ndcg score: 0.46923858038012356

Top 5 feature importance in Fold3:
  feature_name  importance_gain  importance_split
0   Column_133     50574.326808               103
1    Column_54     11866.200945                34
2   Column_107      9030.951607               144
3   Column_129      7123.230594               179
4   Column_128      6921.748647               179
```




```
============= Fold4 =============
Training until validation scores don't improve for 10 rounds
Early stopping, best iteration is:
[87]    valid_0's ndcg@3: 0.458525      valid_0's ndcg@5: 0.464746
valid_0's ndcg@10: 0.48288

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.4513869012433184
k = 5, testing ndcg score: 0.45881548888493456
k = 10, testing ndcg score: 0.47908493872692465

Top 5 feature importance in Fold4:
  feature_name  importance_gain  importance_split
0   Column_133     51262.309673               103
1    Column_54      9154.632476                29
2     Column_7      8149.698136                23
3   Column_107      7553.270632               136
4   Column_128      6149.309267               156
```




```
============= Fold5 =============
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[99]    valid_0's ndcg@3: 0.471239      valid_0's ndcg@5: 0.477973
valid_0's ndcg@10: 0.496325

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.4607164493091444
k = 5, testing ndcg score: 0.464918580972127 96
k = 10, testing ndcg score: 0.48418702980777395

Top 5 feature importance in Fold5:
  feature_name  importance_gain  importance_split
0   Column_133     50762.697981               113
1     Column_7     12024.163998                23
2    Column_54      7674.576921                38
3   Column_107      6544.460454               136
4   Column_128      6142.607745               174
```



1.22 QUESTION 16: Experiments with Subset of Features

1.22.1 16-1 Remove top 20

Remove **the top 20** most important features according to the computed importance score in the question 15. Then train a new LightGBM model on the resulted 116 dimensional query-url data. Evaluate the performance of this new model on the test set using nDCG. Does the outcome align with your expectations? If not, please share your hypothesis regarding the potential reasons for this discrepancy.

Answer:

- The model performance on testing dataset of nDCG@3, nDCG@5, and nDCG@10 are:
  - k = 3, testing ndcg score: 0.3843961691376027
  - k = 5, testing ndcg score: 0.3924111354260531
  - k = 10, testing ndcg score: 0.41529167826546537
- As anticipated, there was a decline in model performance compared to the original 136-feature model. This outcome was expected due to the removal of the 20 most significant features, which likely contributed essential information for the model's prediction accuracy.

[ ]:
```
get_feature_importance(gbm)[:20] # show to 20 features
```

[ ]:
```
# get top 20 features
```

```
important_feature_indices = [int(i.split('_')[-1]) for i in␣
↪get_feature_importance(gbm)[:20]['feature_name'].tolist()]
reduced_indices = [i for i in range(136) if i not in important_feature_indices]

# remove top 20 features
if X_train[:, reduced_indices].shape[1] == 116:
  X_train_reduced = X_train[:, reduced_indices]
  X_test_reduced = X_test[:, reduced_indices]
  X_valid_reduced = X_valid[:, reduced_indices]
  print(X_train_reduced.shape, X_test_reduced.shape, X_valid_reduced.shape)
```

```
(722602, 116) (235259, 116) (242331, 116)
```

[ ]:
```
# train model with reduced features
gbm_remove_top_20 = train_and_evaluate_one_fold_with_validation(data_path,␣
↪X_train_reduced, y_train, group_train, X_test_reduced, y_test, qid_test,␣
↪group_test, X_valid_reduced, y_valid, qid_valid, group_valid)
```

```
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[99]    valid_0's ndcg@3: 0.401609      valid_0's ndcg@5: 0.41011
valid_0's ndcg@10: 0.427587

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.3843961691376027
k = 5, testing ndcg score: 0.3924111354260531
k = 10, testing ndcg score: 0.41529167826546537
```

[ ]:
```
# top 5 features
get_feature_importance(gbm_remove_top_20, reduced_indices)[:5]
```

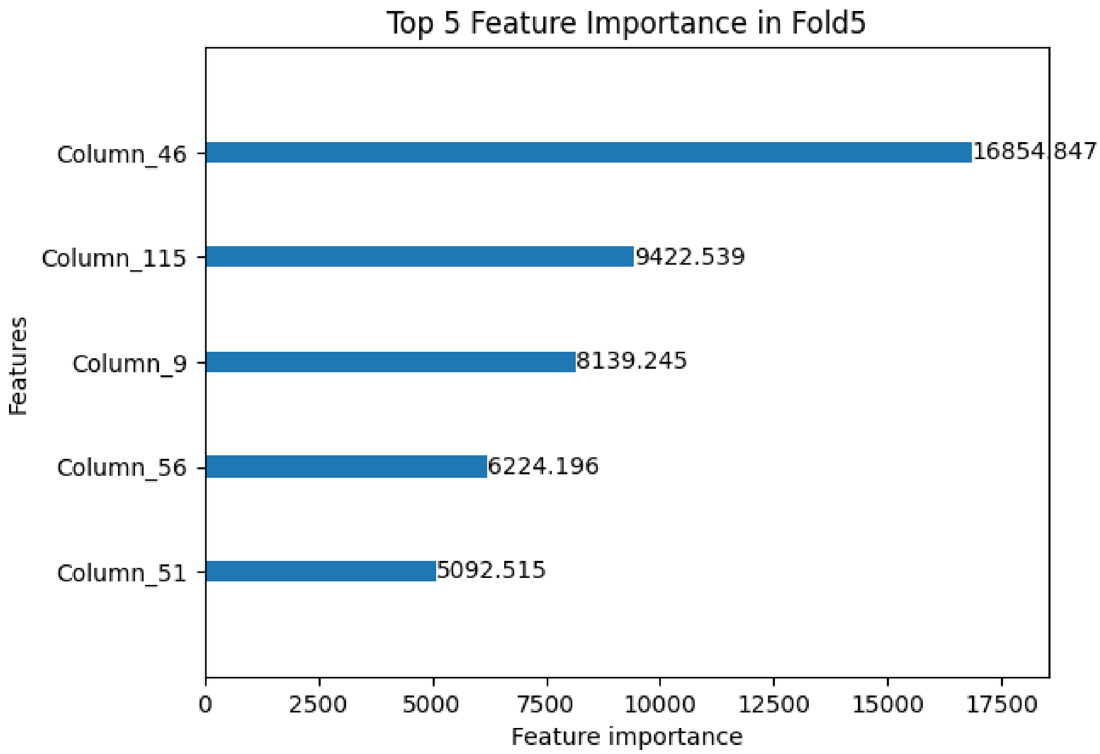
[ ]:

	feature_name	importance_gain	importance_split
0	52	16854.846956	30
1	135	9422.539188	179
2	10	8139.244966	230
3	63	6224.195507	69
4	58	5092.515440	182

[ ]:
```
# show new feature importance
plot_feature_importance(gbm_remove_top_20, 5)

# Map Top 5 features into the original feature order
print(f'\nTop 5 features in original feature order: {reduced_indices[46],␣
↪reduced_indices[115], reduced_indices[9], reduced_indices[56],␣
↪reduced_indices[51]}')
```

```
Top 5 feature importance in Fold5:
  feature_name  importance_gain  importance_split
```

0	Column_46	16854.846956	30
1	Column_115	9422.539188	179
2	Column_9	8139.244966	230
3	Column_56	6224.195507	69
4	Column_51	5092.515440	182




```
Top 5 features in original feature order: (52, 135, 10, 63, 58)
```

1.22.2 16-2 Remove least 60

Remove **the 60 least** important features according to the computed importance score in the question 15. Then train a new LightGBM model on the resulted 76 dimensional query-url data. Evaluate the performance of this new model on the test set using nDCG. Does the outcome align with your expectations? If not, please share your hypothesis regarding the potential reasons for this discrepancy.

Answer:

- The model performance on testing dataset of nDCG@3, nDCG@5, and nDCG@10 are:
  - k = 3, testing ndcg score: 0.4583990018460936
  - k = 5, testing ndcg score: 0.4659233889614642
  - k = 10, testing ndcg score: 0.4847601836863461

- This slight improvement in model performance compared to the original 136-feature model was aligned with my expectations. The enhancement can be attributed to the elimination of the least significant features, which may have been causing noise in the model. Removing the unnecessary useless information probably simplifies the dataset for more effective learning and prediction.

[ ]:
```
get_feature_importance(gbm)[-60:] # show least 60 features
```

[ ]:
```
# get least 60 features
important_feature_indices = [int(i.split('_')[-1]) for i in␣
↪get_feature_importance(gbm)[-60:]['feature_name'].tolist()]
reduced_indices = [i for i in range(136) if i not in important_feature_indices]

# remove least 60 features
if X_train[:, reduced_indices].shape[1] == 136-60 :
  X_train_reduced = X_train[:, reduced_indices]
  X_test_reduced = X_test[:, reduced_indices]
  X_valid_reduced = X_valid[:, reduced_indices]
  print(X_train_reduced.shape, X_test_reduced.shape, X_valid_reduced.shape)
```

```
(722602, 76) (235259, 76) (242331, 76)
```

[ ]:
```
# train model with reduced features
gbm_remove_least_60 = train_and_evaluate_one_fold_with_validation(data_path,␣
↪X_train_reduced, y_train, group_train, X_test_reduced, y_test, qid_test,␣
↪group_test, X_valid_reduced, y_valid, qid_valid, group_valid)
```

```
Training until validation scores don't improve for 10 rounds
Did not meet early stopping. Best iteration is:
[100]   valid_0's ndcg@3: 0.472033      valid_0's ndcg@5: 0.478094
valid_0's ndcg@10: 0.497149

Model performance on the test set (nDCG@3, nDCG@5, nDCG@10):
k = 3, testing ndcg score: 0.4583990018460936
k = 5, testing ndcg score: 0.4659233889614642
k = 10, testing ndcg score: 0.4847601836863461
```

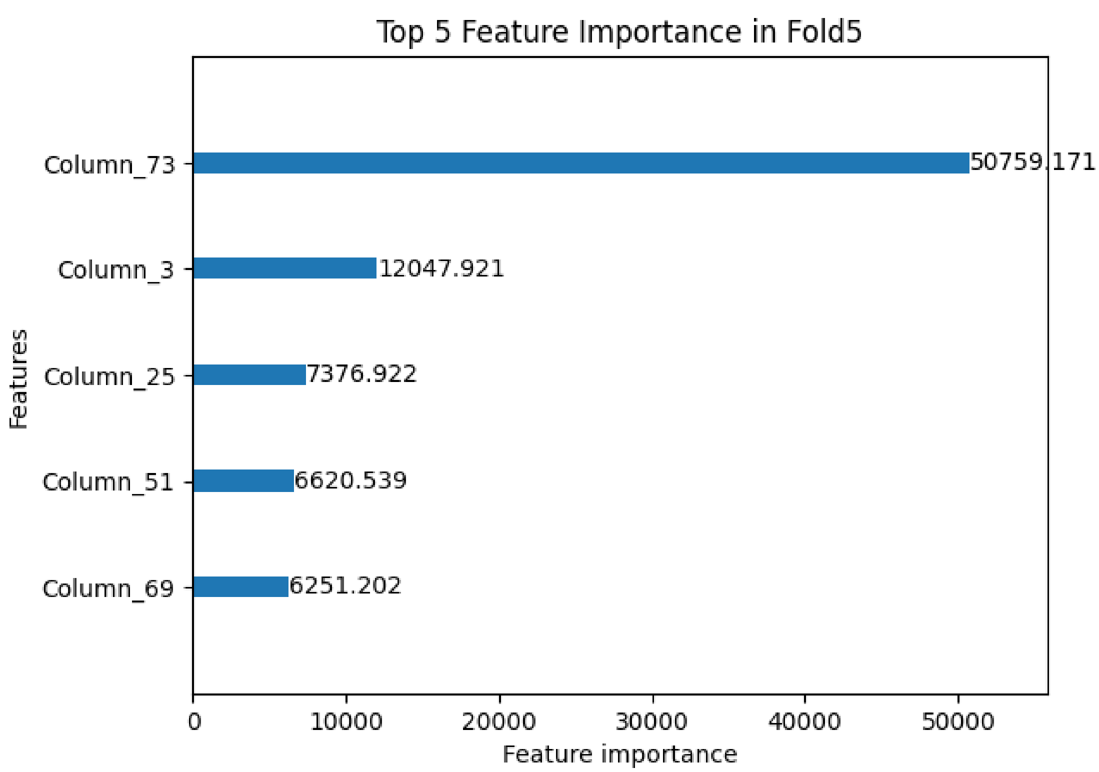
[ ]:
```
# top 5 features
get_feature_importance(gbm_remove_least_60, reduced_indices)[:5]
```

[ ]:

	feature_name	importance_gain	importance_split
0	133	50759.171127	112
1	7	12047.921001	20
2	54	7376.922251	39
3	107	6620.538874	148
4	129	6251.201645	177

[ ]:
```
# show new feature importance
plot_feature_importance(gbm_remove_least_60, 5)

# Map Top 5 features into the original feature order
print(f'\nTop 5 features in original feature order: {reduced_indices[73],␣
↪reduced_indices[3], reduced_indices[25], reduced_indices[51],␣
↪reduced_indices[69]}')
```

```
Top 5 feature importance in Fold5:
  feature_name  importance_gain  importance_split
0    Column_73     50759.171127               112
1     Column_3     12047.921001                20
2    Column_25      7376.922251                39
3    Column_51      6620.538874               148
4    Column_69      6251.201645               177
```




```
Top 5 features in original feature order: (133, 7, 54, 107, 129)
```

[ ]: