Project 3

Kuei-Tzu Hu 206300553

Sreya Muppalla 505675909

Christina Lee 406299676

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/My Drive/ECE ENGR 219/
```

```
Mounted at /content/drive
/content/drive/.shortcut-targets-by-id/1FqG9_tNYj-2BTn9RFaOQCLxMO_q609zf/ECE ENGR 219
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

QUESTION 1: Explore the Dataset:

In this question, we explore the structure of the data.

A Compute the sparsity of the movie rating dataset:

```
ratings = pd.read_csv("Synthetic_Movie_Lens/ratings.csv")

userId = ratings['userId'].values
movieId = ratings['movieId'].values
rating = ratings['rating'].values

sparsity = len(rating)/(len(set(movieId))*len(set(userId)))
print("Sparsity =", sparsity)
```

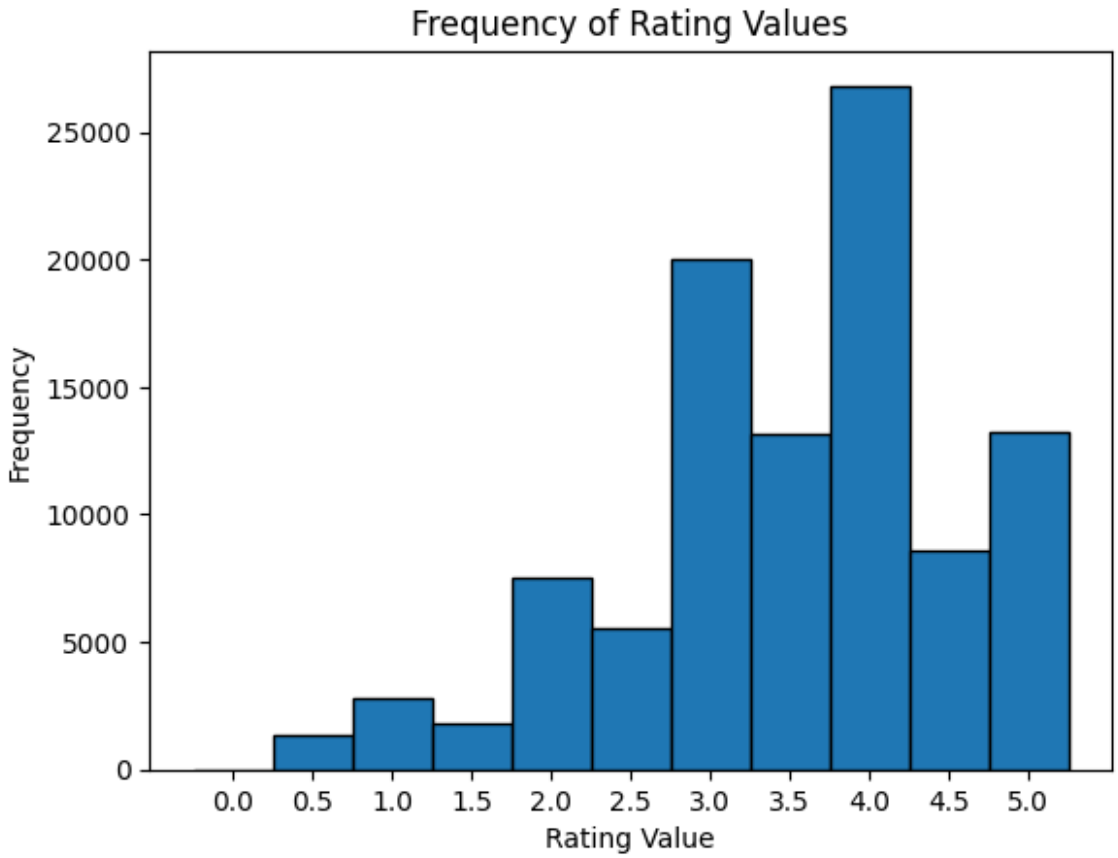
```
Sparsity = 0.016999683055613623
```

B Plot a histogram showing the frequency of the rating values: Bin the raw rating values into intervals of width 0.5 and use the binned rating values as the horizontal axis. Count the number of entries in the ratings matrix R that fall within each bin and use this count as the height of the vertical axis for that particular bin. Comment on the shape of the histogram.

- The histogram is slightly skewed to the right, with 4.0 being the most common rating. Ratings ending in .0 are generally more popular than ratings ending in .5.

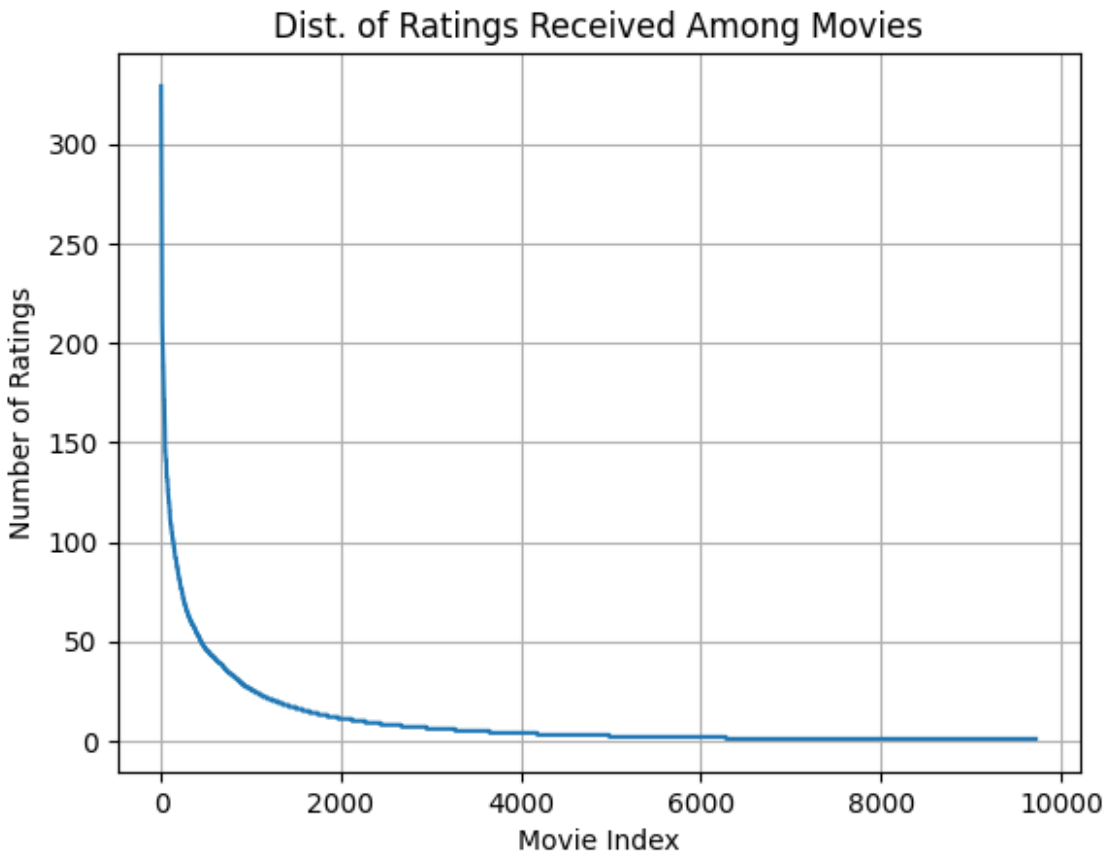
```
bins = np.arange(0, 6, 0.5) - 0.25
plt.hist(rating, bins=bins, edgecolor = "black")
plt.xticks(np.arange(0, 5.5, 0.5))
plt.xlabel("Rating Value")
plt.ylabel("Frequency")
plt.title("Frequency of Rating Values")
```

```
Text(0.5, 1.0, 'Frequency of Rating Values')
```



C Plot the distribution of the number of ratings received among movies: The X-axis should be the movie index ordered by decreasing frequency and the Y -axis should be the number of ratings the movie has received; ties can broken in any way. A monotonically decreasing trend is expected.

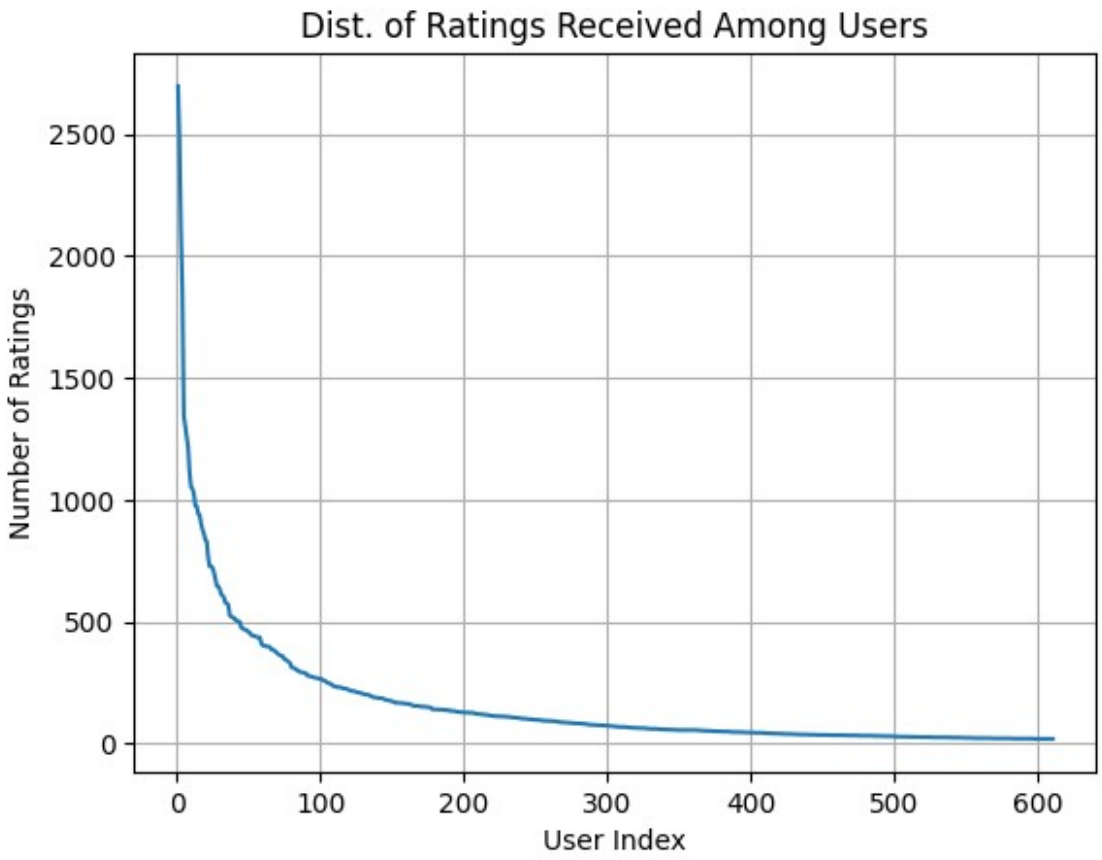
```
index, count =  np.unique(movieId, return_counts=True)
plt.plot(range(1,len(index)+1), sorted(count, reverse=True))
plt.grid()
plt.xlabel("Movie Index")
plt.ylabel("Number of Ratings")
plt.title("Dist. of Ratings Received Among Movies")

Text(0.5, 1.0, 'Dist. of Ratings Received Among Movies')
```



D Plot the distribution of ratings among users: The X-axis should be the user index ordered by decreasing frequency and the Y -axis should be the number of movies the user has rated. The requirement of the plot is similar to that in Question C.

```
index, count =  np.unique(userId, return_counts=True)
plt.plot(range(1,len(index)+1), sorted(count, reverse=True))
plt.grid()
plt.xlabel("User Index")
plt.ylabel("Number of Ratings")
plt.title("Dist. of Ratings Received Among Users")
```

```
Text(0.5, 1.0, 'Dist. of Ratings Received Among Users')
```



E Discuss the salient features of the distributions from Questions C,D and their implications for the recommendation process.

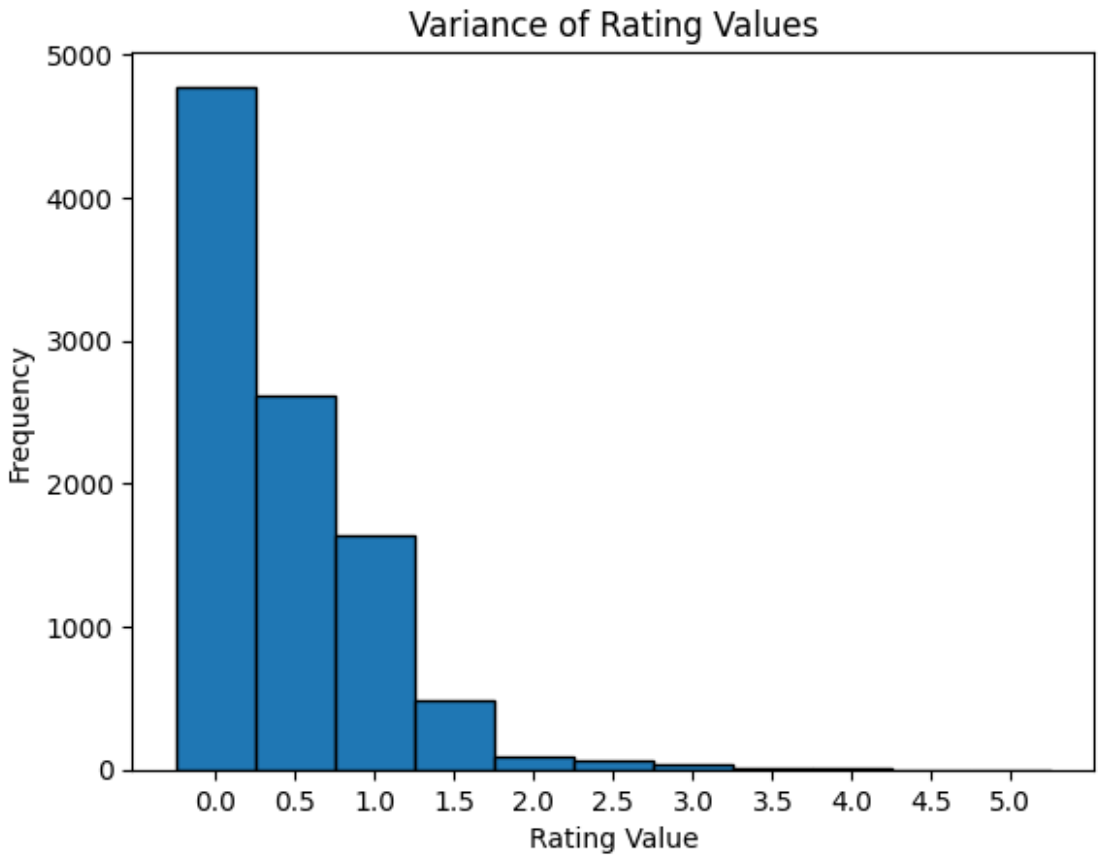
Number of ratings has an inverse relationship with movie indices because the graph shows that a lot smaller amount of movies received most of the ratings whereas the rest(hence majority) of the movies received a smaller number of ratings. So the rating matrix R is sparse so heavy regularization needs to be used to precent overfitting and false links.

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

F Compute the variance of the rating values received by each movie: Bin the variance values into intervals of width 0.5 and use the binned variance values as the horizontal axis. Count the number of movies with variance values in the binned intervals and use this count as the vertical axis. Briefly comment on the shape of the resulting histogram.

- The shape of the histogram is skewed to the left. Most of the variances are 0 and the freqeuncy decreases as rating value increases.

```
movie_rating = {}
for i in range(len(rating)):
    if movieId[i] not in movie_rating:
        movie_rating[movieId[i]] = [rating[i]]
    else:
        movie_rating[movieId[i]].append(rating[i])

variance = [np.var(movie_rating[m]) for m in movie_rating]

bins = np.arange(0, 6, 0.5) - 0.25
plt.hist(variance, bins=bins, edgecolor = "black")
plt.xticks(np.arange(0, 5.5, 0.5))
plt.xlabel("Rating Value")
plt.ylabel("Frequency")
plt.title("Variance of Rating Values")

Text(0.5, 1.0, 'Variance of Rating Values')
```



QUESTION 2: Understanding the Pearson Correlation Coefficient:

A Write down the formula for μu in terms of Iu and ruk;

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

B In plain words, explain the meaning of Iu ∩ Iv. Can Iu ∩ Iv = ∅? (Hint: Rating matrix R is sparse)

The intersection represents movies rated by users u and v. Because rating matrix R is sparse, the intersection would be equal to an empty set since it makes sense that both types of users didn't watch and rate the same movie.

QUESTION 3: Understanding the Prediction function:

Can you explain the reason behind mean-centering the raw ratings (rvj − μv) in the prediction function? (Hint: Consider users who either rate all items highly or rate all items poorly and the impact of these users on the prediction function.)

The reason is because it helps get rid of extreme data points and reduces bias which can lead to more accurate predictions.

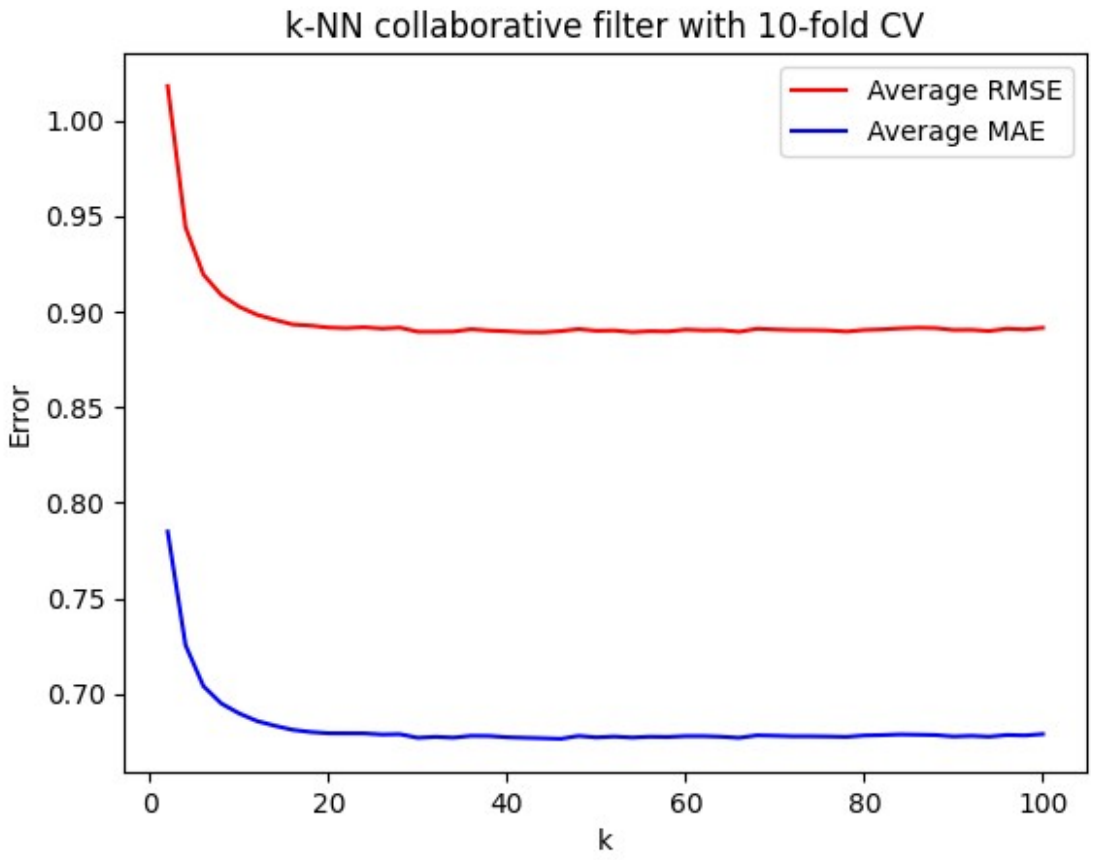
QUESTION 4: Design a k-NN collaborative filter to predict the ratings of the movies in the original dataset and evaluate its performance using 10-fold cross validation.

Sweep k (number of neighbors) from 2 to 100 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 average RMSE (Y-axis) against k (X-axis) and average MAE (Y-axis) against k (X-axis).

```
!pip install surprise

Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Collecting scikit-surprise (from surprise)
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 772.0/772.0 kB 4.3 MB/s eta
0:00:00
etadata (setup.py) ... ent already satisfied: joblib>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
>surprise) (1.3.2)
Requirement already satisfied: numpy>=1.17.3 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
>surprise) (1.25.2)
Requirement already satisfied: scipy>=1.3.2 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
```

```
>surprise) (1.11.4)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... e=scikit_surprise-
1.1.3-cp310-cp310-linux_x86_64.whl size=3163006
sha256=87286772f751b37d142430223dfb7ddd0c7c3219d99688c1d8942aa71606687
3
  Stored in directory:
/root/.cache/pip/wheels/a5/ca/a8/4e28def53797fdc4363ca4af740db15a9c2f1
595ebc51fb445
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.3 surprise-0.1

from surprise import Dataset, Reader
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms.knns import KNNWithMeans

reader = Reader(line_format='user item rating timestamp',
sep=',',skip_lines=1, rating_scale=(0.5, 5))
data = Dataset.load_from_file('Synthetic_Movie_Lens/ratings.csv',
reader=reader)

k = np.arange(2,102,2)
rmse = []
mae = []
for i in k:
    print('Test k =',i)
    knn = KNNWithMeans(k=i, sim_options={'name':'pearson'},
verbose=False);
    cv = cross_validate(knn, data,cv=10,n_jobs=1)

    rmse.append(np.mean(cv['test_rmse']))
    mae.append(np.mean(cv['test_mae']))

fig, ax = plt.subplots()
ax.plot(k, rmse, 'r', label='Average RMSE')
ax.plot(k, mae, 'b', label='Average MAE')
ax.legend(loc='best')
plt.xlabel("k")
plt.ylabel("Error")
plt.title("k-NN collaborative filter with 10-fold CV")

Test k = 2
Test k = 4
Test k = 6
Test k = 8
Test k = 10
Test k = 12
Test k = 14
Test k = 16
```

```
Test k = 18
Test k = 20
Test k = 22
Test k = 24
Test k = 26
Test k = 28
Test k = 30
Test k = 32
Test k = 34
Test k = 36
Test k = 38
Test k = 40
Test k = 42
Test k = 44
Test k = 46
Test k = 48
Test k = 50
Test k = 52
Test k = 54
Test k = 56
Test k = 58
Test k = 60
Test k = 62
Test k = 64
Test k = 66
Test k = 68
Test k = 70
Test k = 72
Test k = 74
Test k = 76
Test k = 78
Test k = 80
Test k = 82
Test k = 84
Test k = 86
Test k = 88
Test k = 90
Test k = 92
Test k = 94
Test k = 96
Test k = 98
Test k = 100

Text(0.5, 1.0, 'k-NN collaborative filter with 10-fold CV')
```



QUESTION 5:

Use the plot from question 4, to find a 'minimum k'.

Note: The term 'minimum k' in this context means that increasing k above the minimum value would not result in a significant decrease in average RMSE or average MAE. If you get the plot correct, then 'minimum k' would correspond to the k value for which average RMSE and average MAE converges to a steady-state value. Please report the steady state values of average RMSE and average MAE.

- From the plot above, we can see that minimum k is 20.

```
print('Steady state value of average RMSE:', rmse[9])
print('Steady state value of average MAE:',mae[9])

Steady state value of average RMSE: 0.89161141380354
Steady state value of average MAE: 0.6793447484329692
```

QUESTION 6:

Within EACH of the 3 trimmed subsets in the dataset, design (train andvalidate): A k-NN collaborative filter on the ratings of the movies (i.e Popular, Unpopular or High-Variance) and evaluate each of the three models' performance using 10-fold cross validation:

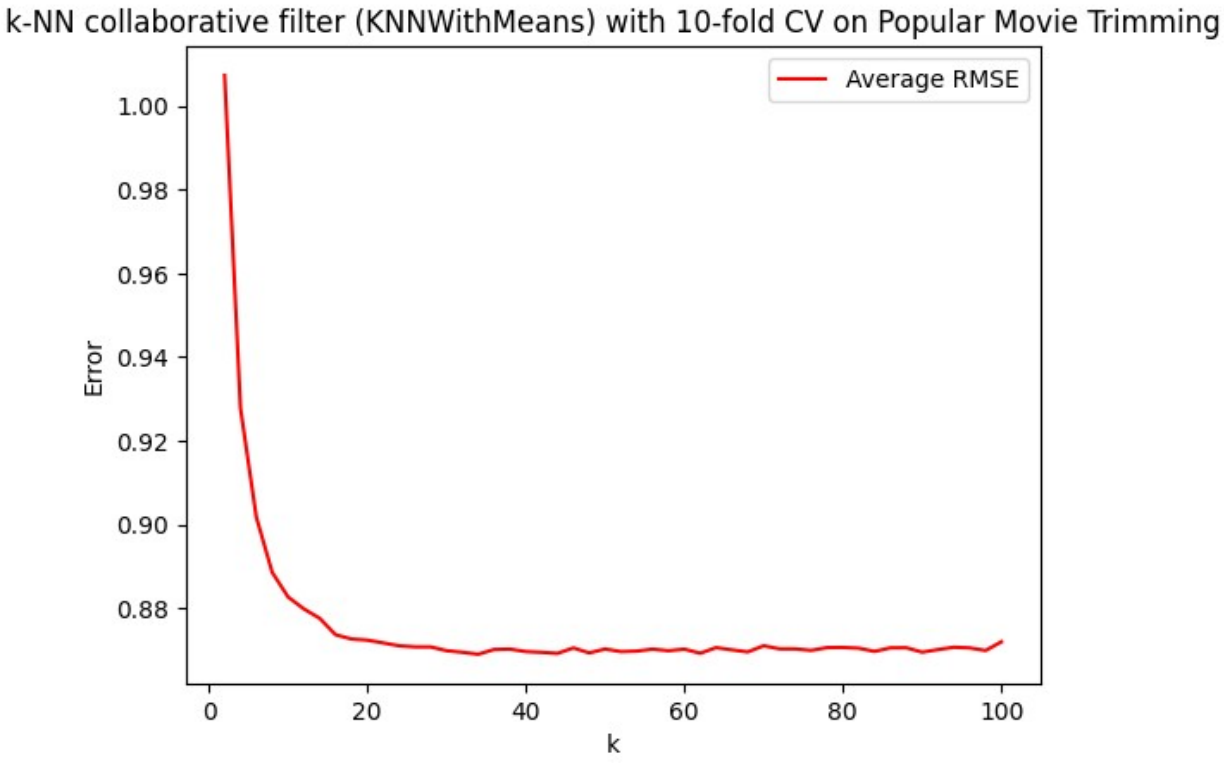
• Sweep k (number of neighbors) from 2 to 100 in step sizes of 2, and for each k compute the average RMSE obtained by averaging the RMSE across all 10 folds. Plot average RMSE (Y-axis) against k (X-axis). Also, report the minimum average RMSE.

• Plot the ROC curves for the k-NN collaborative filters for threshold values [2.5, 3, 3.5, 4]. These thresholds are applied only on the ground truth labels in held-out validation set. For each of the plots, also report the area under the curve (AUC) value. You should have 4 × 4 plots in this section (4 trimming options – including no trimming times 4 thresholds) - all thresholds can be condensed into one plot per trimming option yielding only 4 plots.

```
# Popular Movie Trimming
from surprise.model_selection import KFold
from surprise import accuracy

kf = KFold(n_splits=10)
k = np.linspace(2,100,dtype=int)
avg_rmse = []
ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
        ref[j[1]].append(j[2])

pop_trim = [j for j in data.raw_ratings if len(ref[j[1]]) > 2]
df = pd.DataFrame(pop_trim)
df = df.drop(df.columns[3], axis=1)
data_trim = Dataset.load_from_df(df, reader)

for i in k:
    print('k = ', i)
    rmse = 0
    for trainset, testset in kf.split(data_trim):
        pred =
KNNWithMeans(k=i,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)
        rmse += accuracy.rmse(pred,verbose=False)
    avg_rmse.append(rmse/10.0)

print("Minimum Average RMSE for Popular Movie Trimming: ",
min(avg_rmse))
fig, ax = plt.subplots()
ax.plot(k,avg_rmse, 'r', label='Average RMSE')
ax.legend(loc='best')
plt.xlabel("k"); plt.ylabel("Error"); plt.title("k-NN collaborative
filter (KNNWithMeans) with 10-fold CV on Popular Movie Trimming")
plt.show()
```

```
k =  2
k =  4
k =  6
k =  8
k =  10
k =  12
k =  14
k =  16
k =  18
k =  20
k =  22
k =  24
k =  26
k =  28
k =  30
k =  32
k =  34
k =  36
k =  38
k =  40
k =  42
k =  44
k =  46
k =  48
k =  50
k =  52
k =  54
k =  56
k =  58
k =  60
k =  62
k =  64
k =  66
k =  68
k =  70
k =  72
k =  74
k =  76
k =  78
k =  80
k =  82
k =  84
k =  86
k =  88
k =  90
k =  92
k =  94
k =  96
k =  98
```

```
k =  100
Minimum Average RMSE for Popular Movie Trimming:  0.8689899677791469
```

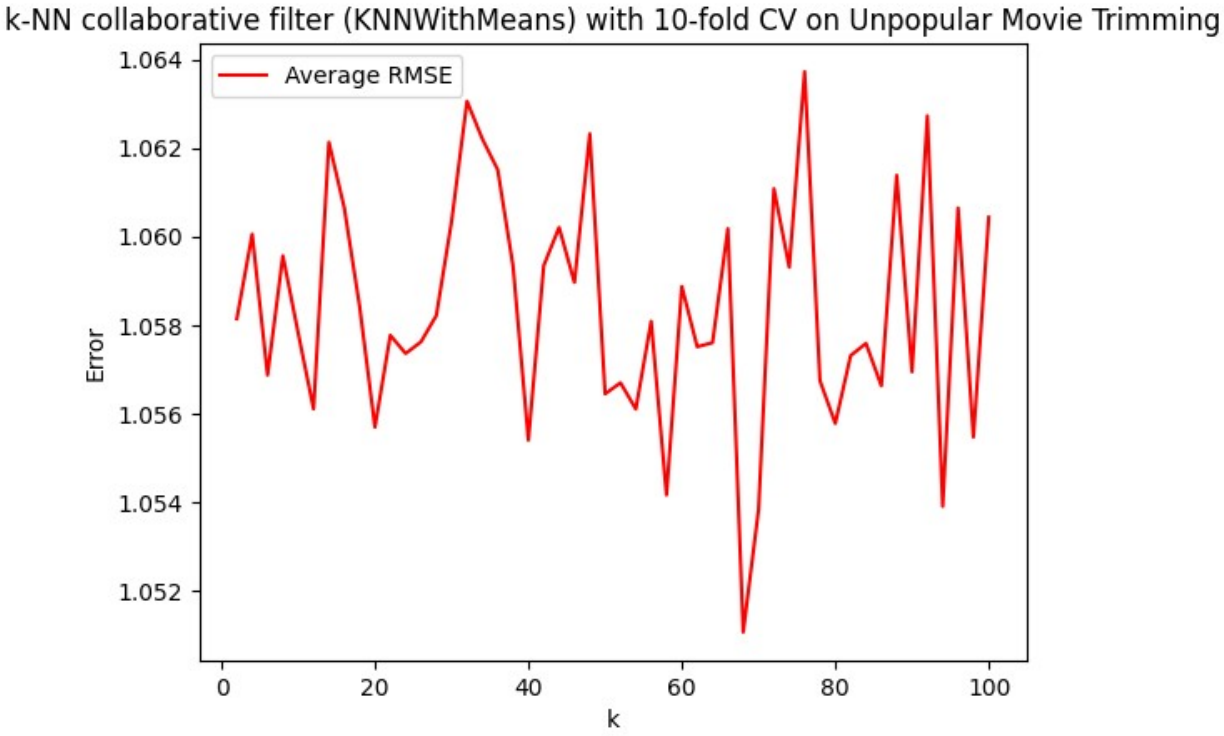



```
# Unpopular Movie Trimming
kf = KFold(n_splits=10)
k = np.linspace(2,100,dtype=int)
avg_rmse = []
ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
        ref[j[1]].append(j[2])

unpop_trim = [j for j in data.raw_ratings if len(ref[j[1]]) <= 2]
df = pd.DataFrame(unpop_trim)
df = df.drop(df.columns[3], axis=1)
data_trim = Dataset.load_from_df(df, reader)

for i in k:
    print('k = ', i)
    rmse = 0
    for trainset, testset in kf.split(data_trim):
        pred =
```

```
KNNWithMeans(k=i,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)
        rmse += accuracy.rmse(pred,verbose=False)
    avg_rmse.append(rmse/10.0)

print("Minimum Average RMSE for Unpopular Movie Trimming: ",
min(avg_rmse))
fig, ax = plt.subplots()
ax.plot(k,avg_rmse, 'r', label='Average RMSE')
ax.legend(loc='best')
plt.xlabel("k"); plt.ylabel("Error"); plt.title("k-NN collaborative
filter (KNNWithMeans) with 10-fold CV on Unpopular Movie Trimming")
plt.show()
```

```
k =  2
k =  4
k =  6
k =  8
k =  10
k =  12
k =  14
k =  16
k =  18
k =  20
k =  22
k =  24
k =  26
k =  28
k =  30
k =  32
k =  34
k =  36
k =  38
k =  40
k =  42
k =  44
k =  46
k =  48
k =  50
k =  52
k =  54
k =  56
k =  58
k =  60
k =  62
k =  64
k =  66
k =  68
k =  70
k =  72
```

```
k =  74
k =  76
k =  78
k =  80
k =  82
k =  84
k =  86
k =  88
k =  90
k =  92
k =  94
k =  96
k =  98
k =  100
Minimum Average RMSE for Unpopular Movie Trimming:  1.051062534411778
```

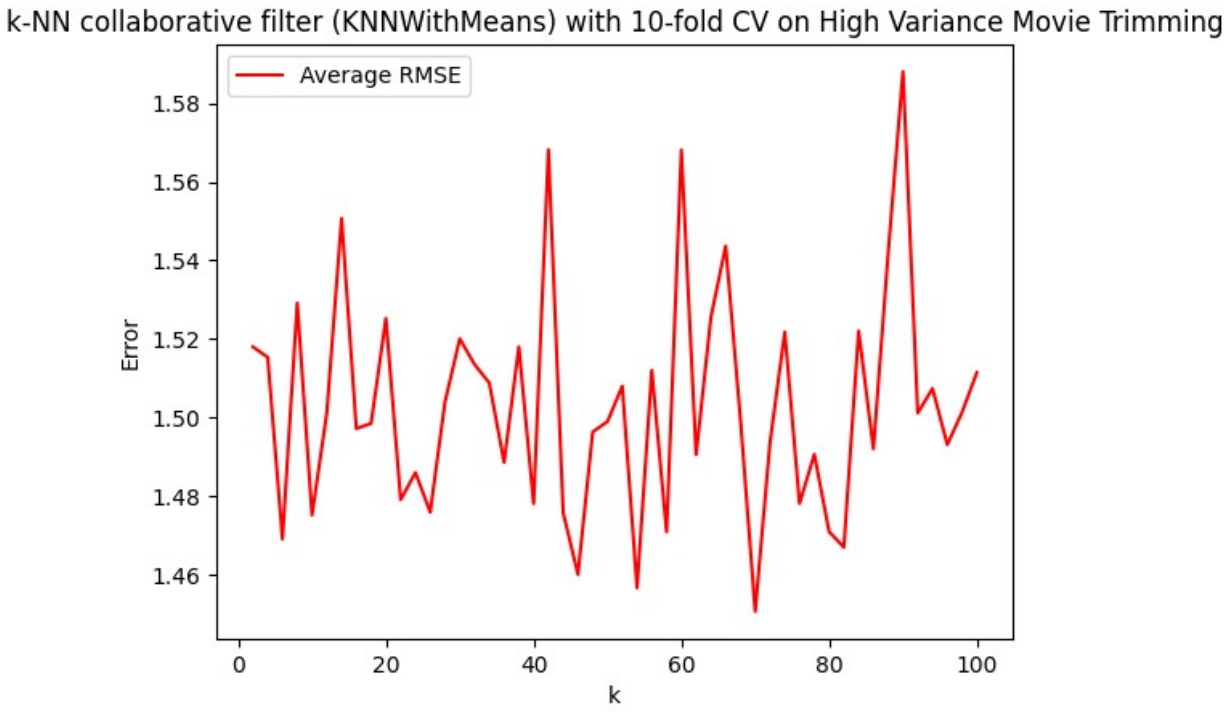



```
# High Variance Movie Trimming
kf = KFold(n_splits=10)
k = np.linspace(2,100,dtype=int)
avg_rmse = []
ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
```

```
        ref[j[1]].append(j[2])

highvar_trim = [j for j in data.raw_ratings if (len(ref[j[1]]) >= 5
and np.var(ref[j[1]]) >= 2)]
df = pd.DataFrame(highvar_trim)
df = df.drop(df.columns[3], axis=1)
data_trim = Dataset.load_from_df(df, reader)

for i in k:
    print('k = ', i)
    rmse = 0
    for trainset, testset in kf.split(data_trim):
        pred =
KNNWithMeans(k=i,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)
        rmse += accuracy.rmse(pred,verbose=False)
    avg_rmse.append(rmse/10.0)

print("Minimum Average RMSE for High Variance Movie Trimming: ",
min(avg_rmse))
fig, ax = plt.subplots()
ax.plot(k,avg_rmse, 'r', label='Average RMSE')
ax.legend(loc='best')
plt.xlabel("k"); plt.ylabel("Error"); plt.title("k-NN collaborative
filter (KNNWithMeans) with 10-fold CV on High Variance Movie
Trimming")
plt.show()
```

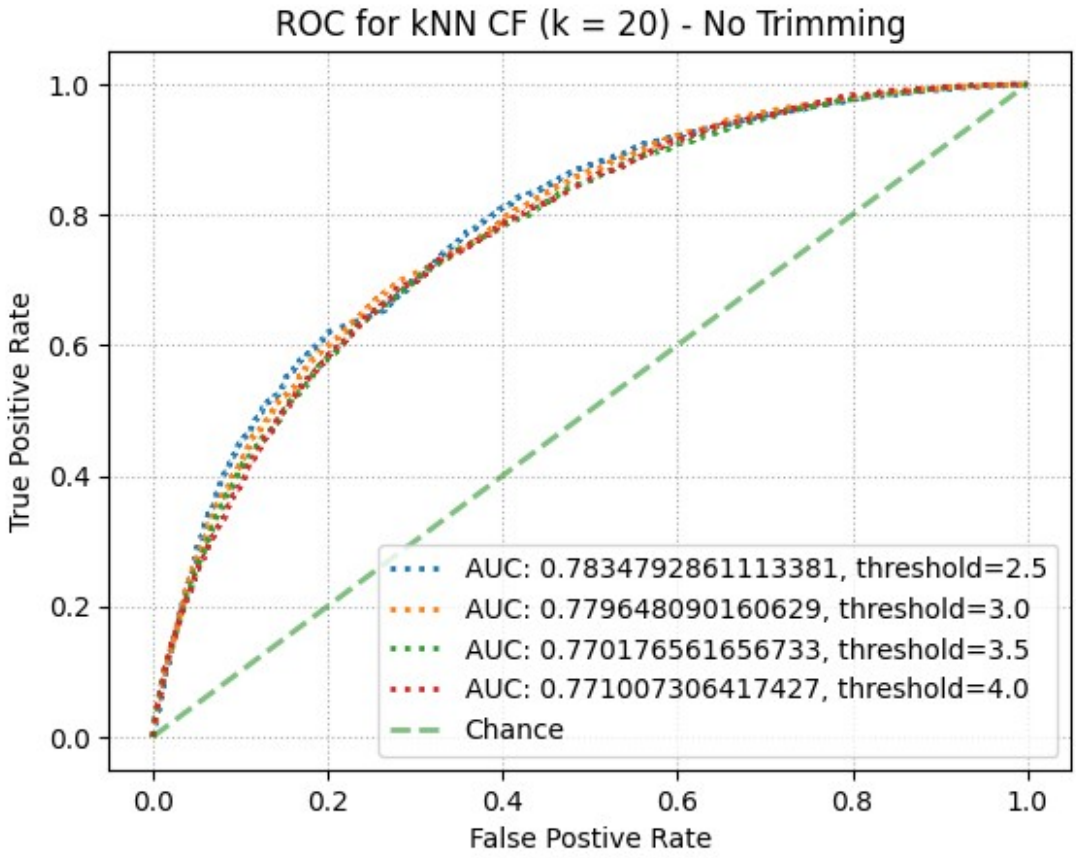
```
k =  2
k =  4
k =  6
k =  8
k =  10
k =  12
k =  14
k =  16
k =  18
k =  20
k =  22
k =  24
k =  26
k =  28
k =  30
k =  32
k =  34
k =  36
k =  38
k =  40
k =  42
k =  44
```

```
k =  46
k =  48
k =  50
k =  52
k =  54
k =  56
k =  58
k =  60
k =  62
k =  64
k =  66
k =  68
k =  70
k =  72
k =  74
k =  76
k =  78
k =  80
k =  82
k =  84
k =  86
k =  88
k =  90
k =  92
k =  94
k =  96
k =  98
k =  100
Minimum Average RMSE for High Variance Movie Trimming:
1.4507260778831574
```




```
from surprise.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
k = 20
thres = [2.5, 3.0, 3.5, 4.0]
trainset, testset = train_test_split(data, test_size=0.1)
res  =
KNNWithMeans(k=k,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)

fig, ax = plt.subplots()
for item in thres:
    thresholded_out = []
    for row in res:
        if row.r_ui > item:
            thresholded_out.append(1)
        else:
            thresholded_out.append(0)
    fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row
in res])
    ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC:
"+str(auc(fpr,tpr))+', threshold='+str(item))
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='g',
label='Chance', alpha=.5)
plt.legend(loc='best')
plt.grid(linestyle=':')
plt.title('ROC for kNN CF (k = 20) - No Trimming')
plt.xlabel('False Postive Rate')
```

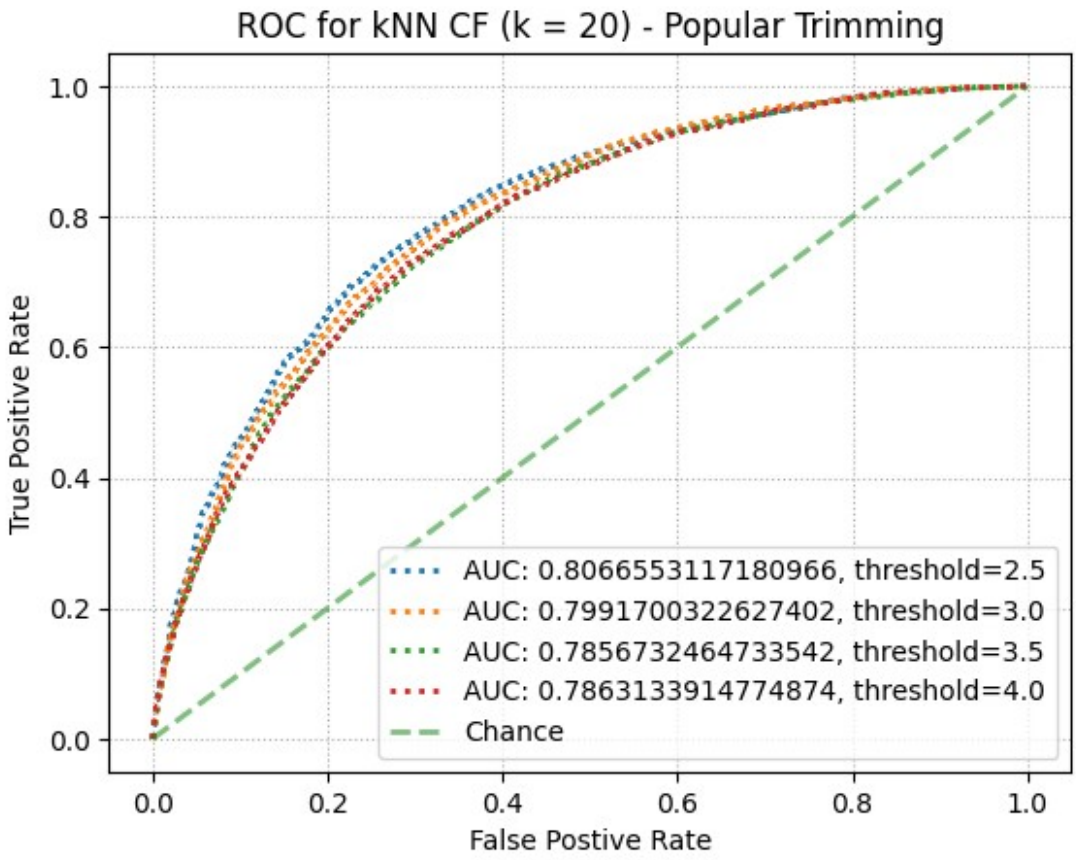
```
plt.ylabel('True Positive Rate')
plt.show()
```




```
k = 20
thres = [2.5, 3.0, 3.5, 4.0]
ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
        ref[j[1]].append(j[2])

pop_trim = [j for j in data.raw_ratings if len(ref[j[1]]) > 2]
df = pd.DataFrame(pop_trim)
df = df.drop(df.columns[3], axis=1)
print(df.shape)
data_trim = Dataset.load_from_df(df, reader)

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

res =
```

```
KNNWithMeans(k=k,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)

fig, ax = plt.subplots()
for item in thres:
    thresholded_out = []
    for row in res:
        if row.r_ui > item:
            thresholded_out.append(1)
        else:
            thresholded_out.append(0)
    fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row
in res])
    ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC:
"+str(auc(fpr,tpr))+', threshold='+str(item))
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='g',
label='Chance', alpha=.5)
plt.legend(loc='best')
plt.grid(linestyle=':')
plt.title('ROC for kNN CF (k = 20) - Popular Trimming')
plt.xlabel('False Postive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

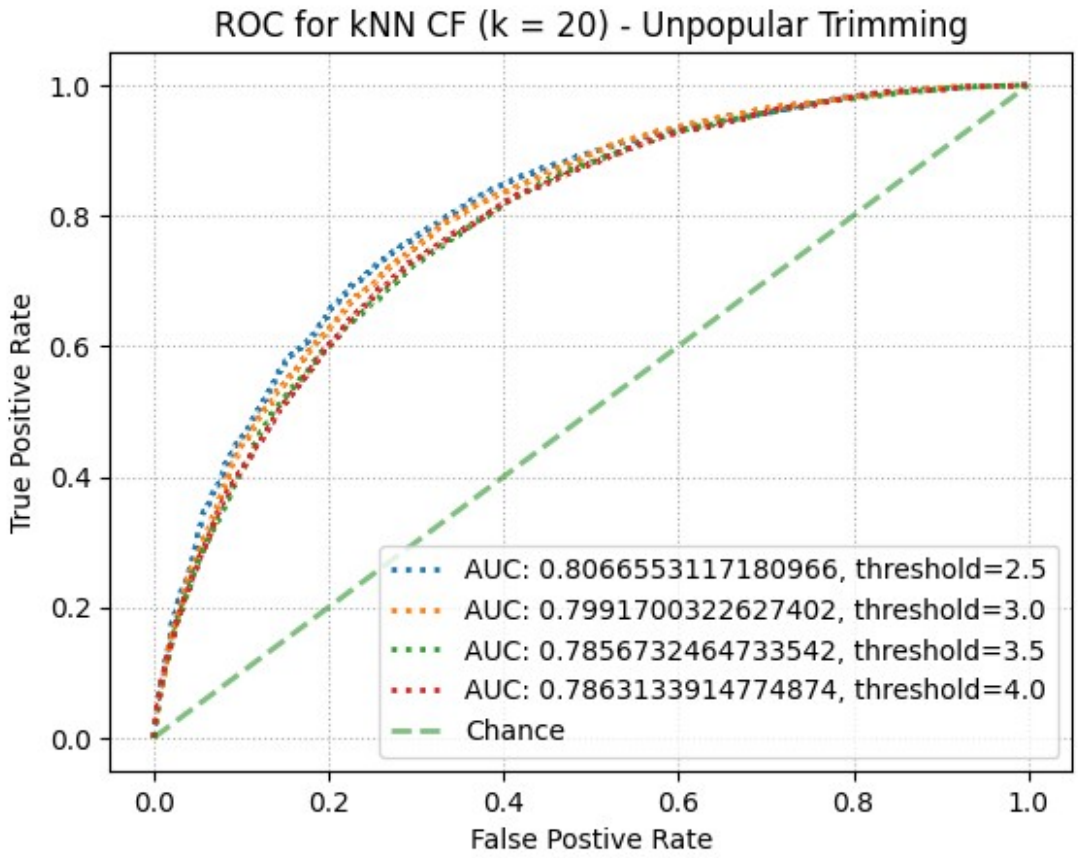
```
(94794, 3)
```




```
k = 20
thres = [2.5, 3.0, 3.5, 4.0]
ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
        ref[j[1]].append(j[2])
unpop_trim = [j for j in data.raw_ratings if len(ref[j[1]]) <= 2]
df = pd.DataFrame(unpop_trim)
print(df.shape)
df = df.drop(df.columns[3], axis=1)
data_trim = Dataset.load_from_df(df, reader)

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

pred =
KNNWithMeans(k=k,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)

fig, ax = plt.subplots()
for item in thres:
```

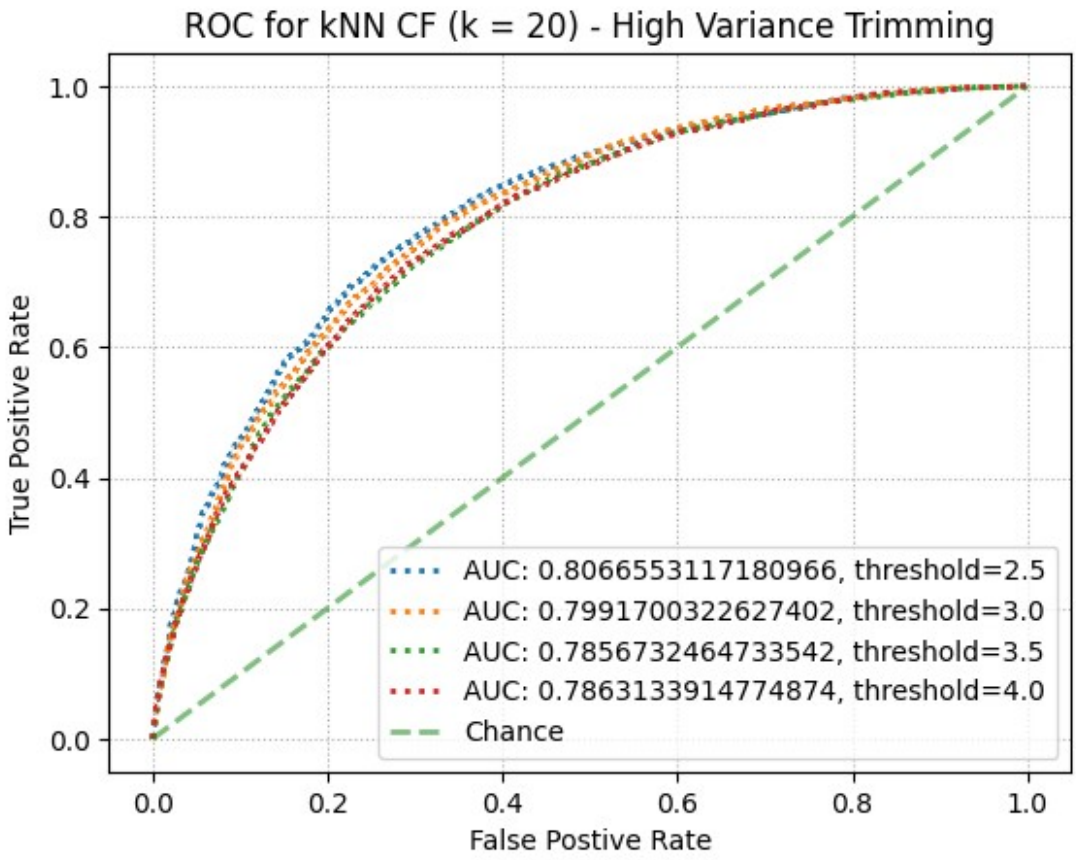
```
    thresholded_out = []
    for row in res:
        if row.r_ui > item:
            thresholded_out.append(1)
        else:
            thresholded_out.append(0)
    fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row
in res])
    ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC:
"+str(auc(fpr,tpr))+', threshold='+str(item))
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='g',
label='Chance', alpha=.5)
plt.legend(loc='best')
plt.grid(linestyle=':')
plt.title('ROC for kNN CF (k = 20) - Unpopular Trimming')
plt.xlabel('False Postive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

```
(6042, 4)
```



```
k = 20
thres = [2.5, 3.0, 3.5, 4.0]

ref = {}
for j in data.raw_ratings:
    if j[1] in ref.keys():
        ref[j[1]].append(j[2])
    else:
        ref[j[1]] = []
        ref[j[1]].append(j[2])

highvar_trim = [j for j in data.raw_ratings if (len(ref[j[1]]) >= 5
and np.var(ref[j[1]]) >= 2)]
df = pd.DataFrame(highvar_trim)
print(df.shape)
df = df.drop(df.columns[3], axis=1)
data_trim = Dataset.load_from_df(df, reader)

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

pred =
KNNWithMeans(k=k,sim_options={'name':'pearson'},verbose=False).fit(tra
inset).test(testset)

fig, ax = plt.subplots()
for item in thres:
    thresholded_out = []
    for row in res:
        if row.r_ui > item:
            thresholded_out.append(1)
        else:
            thresholded_out.append(0)
    fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row
in res])
    ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC:
"+str(auc(fpr,tpr))+', threshold='+str(item))
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='g',
label='Chance', alpha=.5)
plt.legend(loc='best')
plt.grid(linestyle=':')
plt.title('ROC for kNN CF (k = 20) - High Variance Trimming')
plt.xlabel('False Postive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

```
(250, 4)
```



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

- The optimization problem is not convex in both U and V simultaneuosly, but if either U or V is fixed, the problem becomes convex in the other.
- When U is fixed, the problem can be treated as a least squares problem for optimizing V. The goal is to find the matrix V that minimizes the sum of squared differences between the entries of the original matrix R and the corresponding entries of $U\,V^T$, weighted by the values of W.

Project 3

Kuei-Tzu Hu 206300553

Sreya Muppalla 505675909

Christina Lee 406299676

```
!pip install surprise
```

```
Collecting surprise
  Downloading surprise-0.1-py2.py3-none-any.whl (1.8 kB)
Collecting scikit-surprise (from surprise)
  Downloading scikit-surprise-1.1.3.tar.gz (771 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 772.0/772.0 kB 10.4 MB/s eta
0:00:00
etadata (setup.py) ... ent already satisfied: joblib>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
>surprise) (1.3.2)
Requirement already satisfied: numpy>=1.17.3 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
>surprise) (1.25.2)
Requirement already satisfied: scipy>=1.3.2 in
/usr/local/lib/python3.10/dist-packages (from scikit-surprise-
>surprise) (1.11.4)
Building wheels for collected packages: scikit-surprise
  Building wheel for scikit-surprise (setup.py) ... e=scikit_surprise-
1.1.3-cp310-cp310-linux_x86_64.whl size=3163001
sha256=2474329d8a331b20c076745ce6a011862fe7d961c9654356e88ea3e501f2eed
b
  Stored in directory:
/root/.cache/pip/wheels/a5/ca/a8/4e28def53797fdc4363ca4af740db15a9c2f1
595ebc51fb445
Successfully built scikit-surprise
Installing collected packages: scikit-surprise, surprise
Successfully installed scikit-surprise-1.1.3 surprise-0.1
```

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/My Drive/ECE ENGR 219/Synthetic_Movie_Lens
```

```
Mounted at /content/drive
/content/drive/My Drive/ECE ENGR 219/Synthetic_Movie_Lens
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive')
```

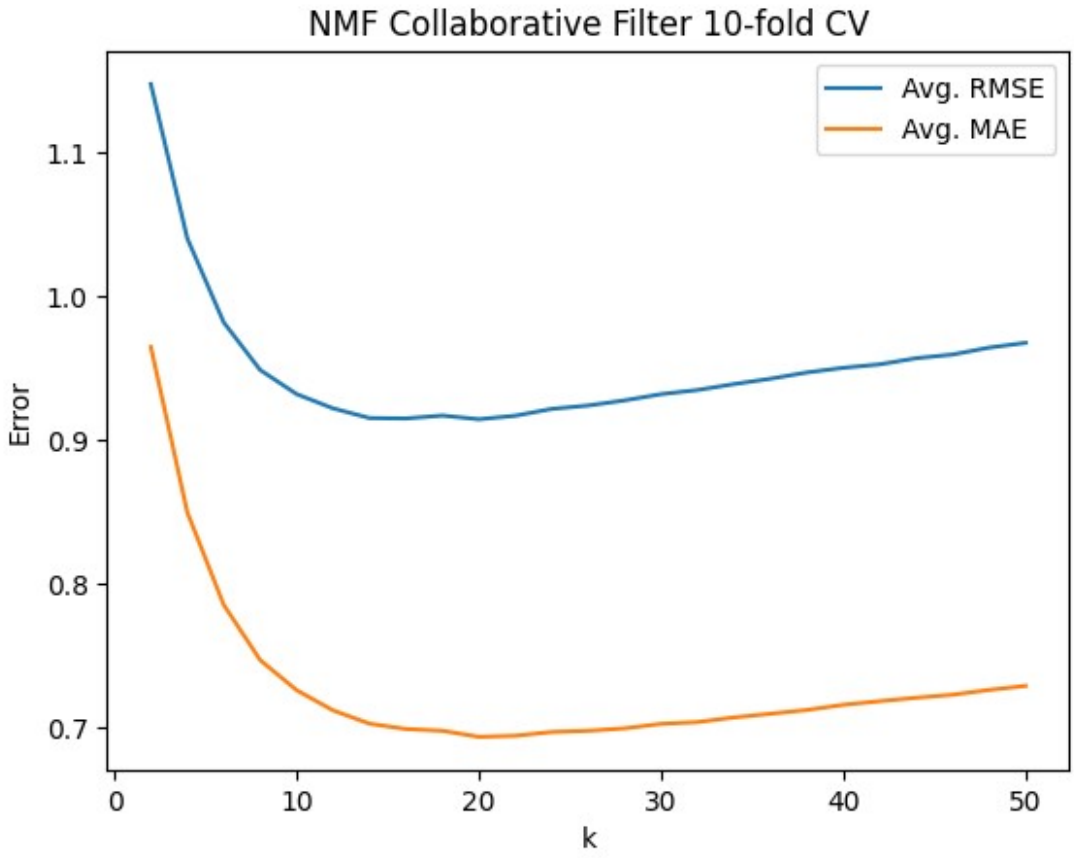
```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).

ratings = pd.read_csv("ratings.csv")
userId = ratings['userId'].values
movieId = ratings['movieId'].values
rating = ratings['rating'].values

from surprise import Dataset, Reader
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms.knns import KNNWithMeans

reader = Reader(line_format='user item rating timestamp',
sep=',',skip_lines=1, rating_scale=(0.5, 5))
data = Dataset.load_from_file('ratings.csv', reader=reader)
```

###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.

```
from surprise.prediction_algorithms.matrix_factorization import NMF

maeAvg = []
rmseAvg = []
#ADD MORE STEPS LATER
k = np.linspace(2,50,num=25,dtype=int)
for idx in k:
    print(idx)
    nmf = NMF(n_factors=idx)
    cv = cross_validate(nmf,data,cv=10)
    rmse_mean = np.mean(cv['test_rmse'])
    mae_mean = np.mean(cv['test_mae'])
    rmseAvg.append(rmse_mean)
    maeAvg.append(mae_mean)

fig, ax = plt.subplots()
ax.plot(k,rmseAvg, label='Avg. RMSE')
ax.plot(k, maeAvg,label='Avg. MAE')
plt.xlabel("k")
plt.ylabel("Error")
plt.legend()
plt.title("NMF Collaborative Filter 10-fold CV")
```

```
Text(0.5, 1.0, 'NMF Collaborative Filter 10-fold CV')
```



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?

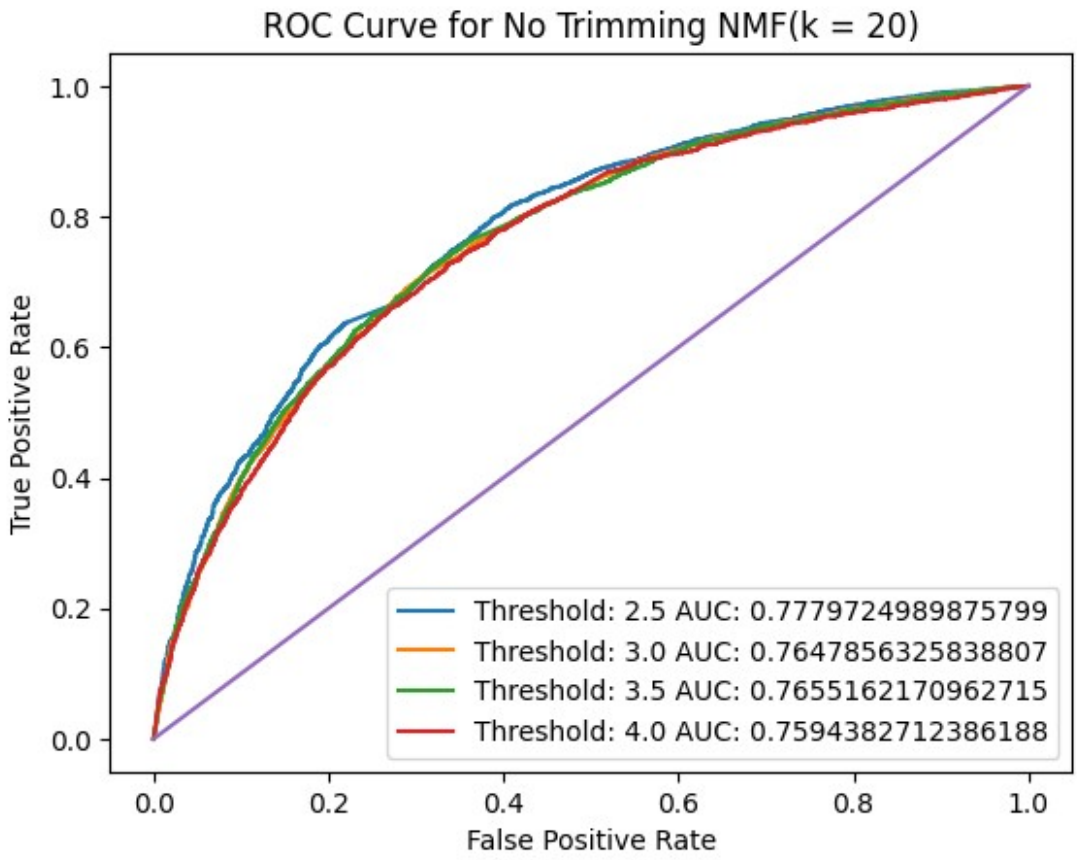
The optimal number of latent factors(26 for MAE and 14 for RMSE) is around the same as the number of movie genres(19).

```
comb_mae = list(zip(k,maeAvg))
comb_rmse = list(zip(k,rmseAvg))
min_mae = min(comb_mae, key = lambda t: t[1])
min_rmse = min(comb_rmse, key = lambda t: t[1])

print("Minimum average MAE: ", min_mae[1], " k: ", min_mae[0])
print("Minimum average RMSE: ", min_rmse[1], " k: ", min_rmse[0])
```

```
Minimum average MAE:  0.6932836558584488  k:  20
Minimum average RMSE:  0.9141154964129221  k:  20
```

```
from surprise.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
```

```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_pop = list(zip(k,rmseAvg))
min_rmse_pop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
nmf = NMF(n_factors=min_rmse_pop[0])
nmf = nmf.fit(train)
nmf = nmf.test(test)
fig, ax = plt.subplots()
for threshold in threshold_vals:
    predicted = []
    for row in nmf:
        if row.r_ui <= threshold:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in nmf]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(threshold) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for No Trimming NMF(k = ' + str(min_rmse_pop[0]) +
')'
plt.title(title)
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



C) Performance on trimmed dataset subsets: For each of Popular, Unpopular and High-Variance subsets:

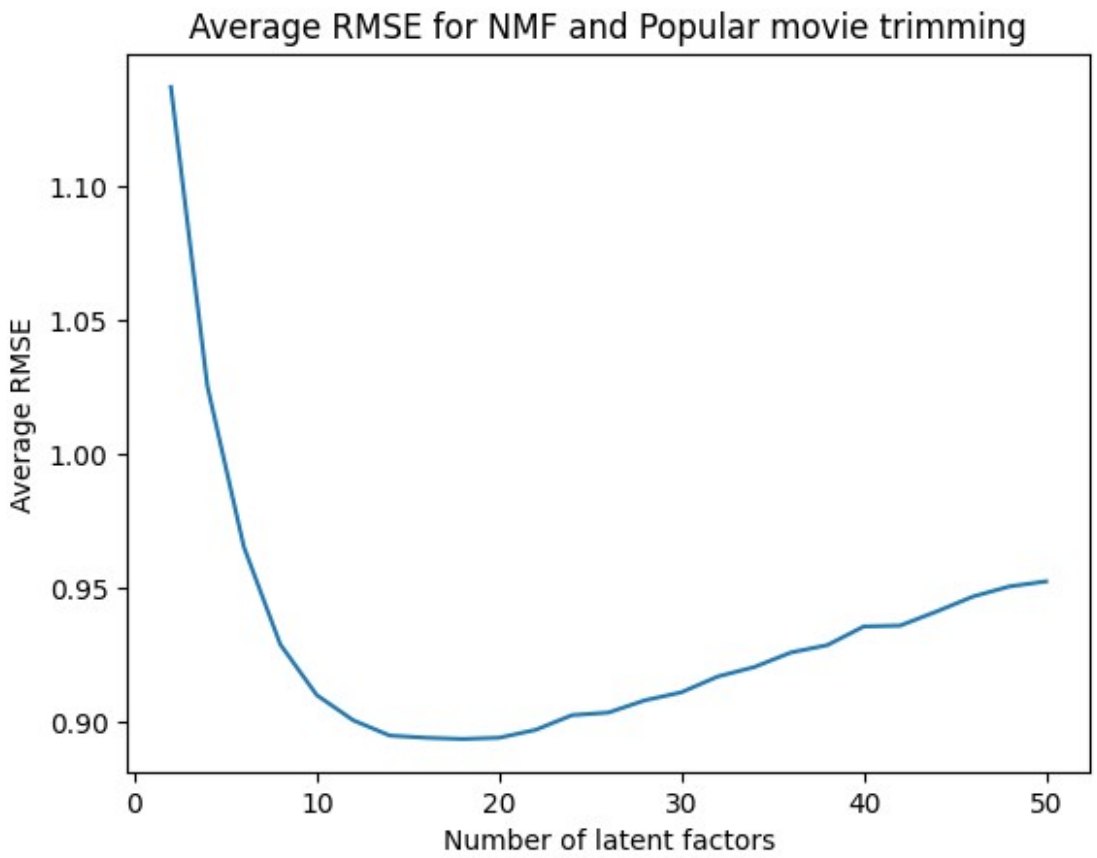
- **Design a NMF collaborative filter for each trimmed subset and evaluate its performance using 10-fold cross validation. Sweep k (number of latent factors) from 2 to 50 in step sizes of 2, and for each k compute the average RMSE obtained by averaging the RMSE across all 10 folds.**
- **Plot average RMSE (Y-axis) against k (X-axis). Report the minimum average RMSE.**
- **Plot the ROC curves for the MF-based collaborative filter and also report the area under the curve (AUC) value as done in Question 6**

```
from surprise.model_selection import KFold
from surprise import accuracy

tracker = {}
for row in data.raw_ratings:
  if row[1] not in tracker:
    tracker[row[1]] = []
  tracker[row[1]].append(row[2])
```

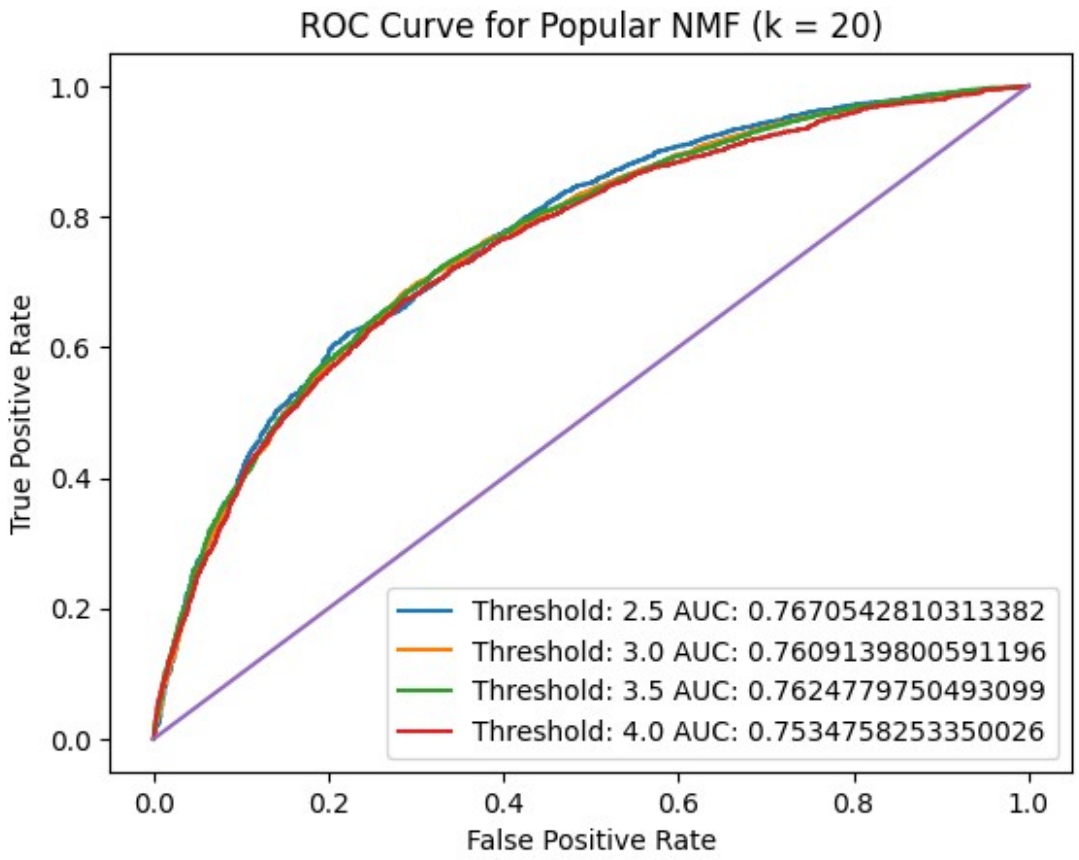
```
#Popular
RMSENMFpop = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
      trimTest = [row for row in test if len(tracker[row[1]]) > 2]
      nmf = NMF(n_factors=idx)
      nmf = nmf.fit(train)
      nmf = nmf.test(trimTest)
      currRMSE.append(accuracy.rmse(nmf))
    RMSENMFpop.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSENMFpop))
```

```
Minimum Average RMSE: 0.8934364643638464
```

```
plt.plot(k,RMSENMFpop)
plt.title('Average RMSE for NMF and Popular movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



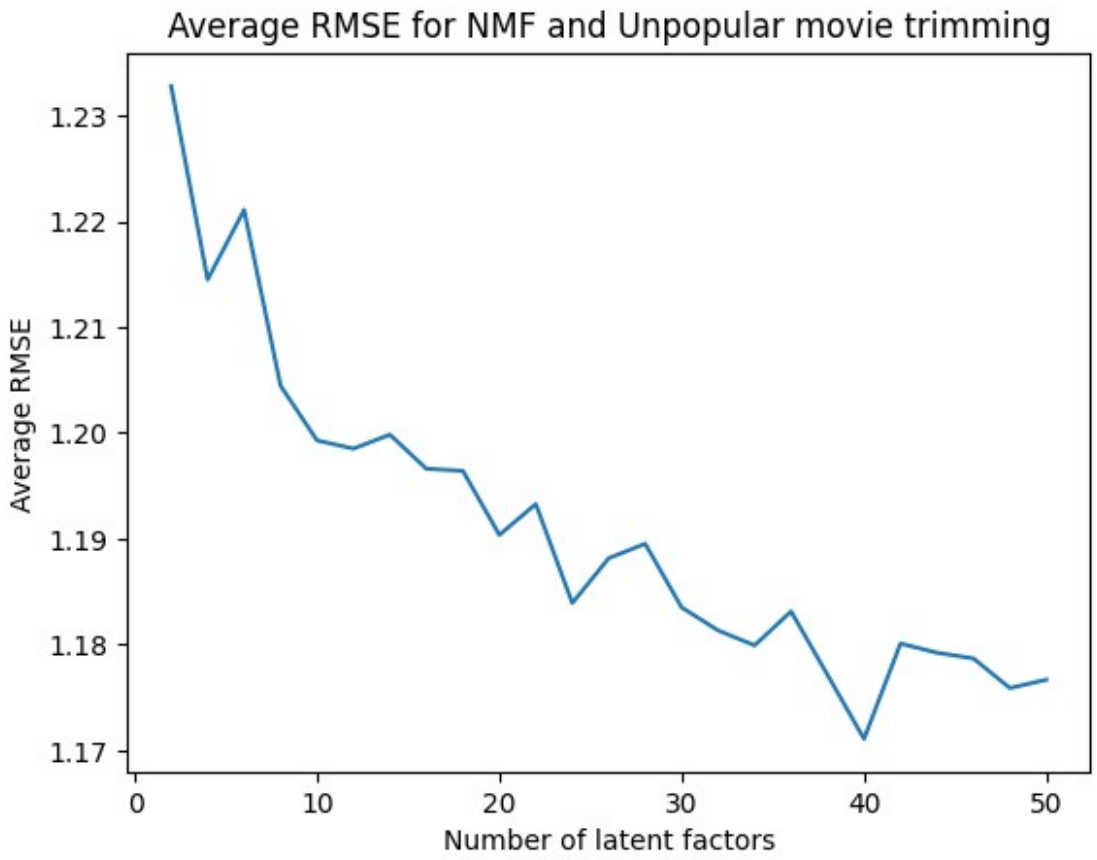
```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_pop = list(zip(k.tolist(),RMSENMFpop))
min_rmse_pop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
nmf = NMF(n_factors=min_rmse_pop[0])
nmf = nmf.fit(train)
nmf = nmf.test(test)
fig, ax = plt.subplots()
for thresh in threshold_vals:
    predicted = []
    for row in nmf:
        if row.r_ui <= thresh:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in nmf]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(thresh) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for Popular NMF (k = ' + str(min_rmse_pop[0]) + ')'
plt.title(title)
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```




```
#Unopular
RMSENMFunpop = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
      trimTest = [row for row in test if len(tracker[row[1]]) <= 2]
      nmf = NMF(n_factors=idx)
      nmf = nmf.fit(train)
      nmf = nmf.test(trimTest)
      currRMSE.append(accuracy.rmse(nmf))
    RMSENMFunpop.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSENMFunpop))
```

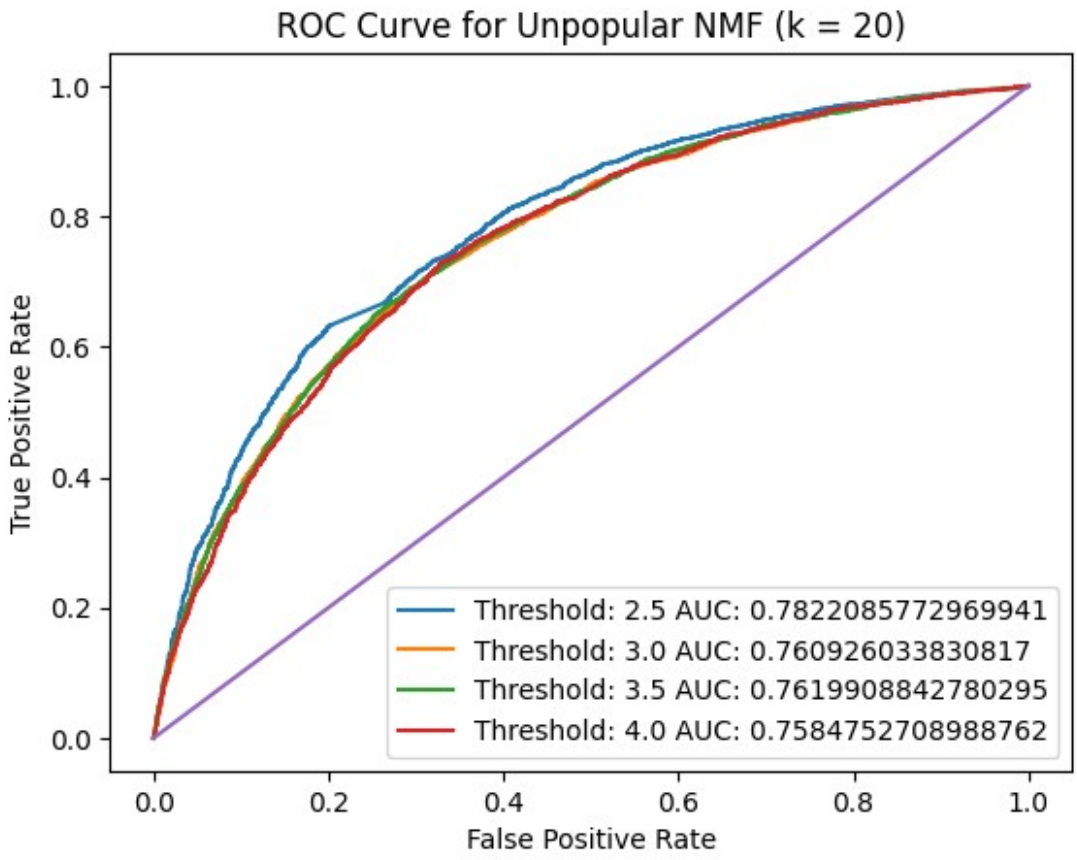
```
Minimum Average RMSE: 1.1710796049085808
```

```
plt.plot(k,RMSENMFunpop)
plt.title('Average RMSE for NMF and Unpopular movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```




```
from surprise.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_pop = list(zip(k.tolist(),RMSENMFunpop))
min_rmse_pop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
nmf = NMF(n_factors=min_rmse_pop[0])
nmf = nmf.fit(train)
nmf = nmf.test(test)
fig, ax = plt.subplots()
for thresh in threshold_vals:
    predicted = []
    for row in nmf:
        if row.r_ui <= thresh:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in nmf]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(thresh) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
```

```
title = 'ROC Curve for Unpopular NMF (k = ' + str(min_rmse_pop[0]) +
')'
plt.title(title)
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

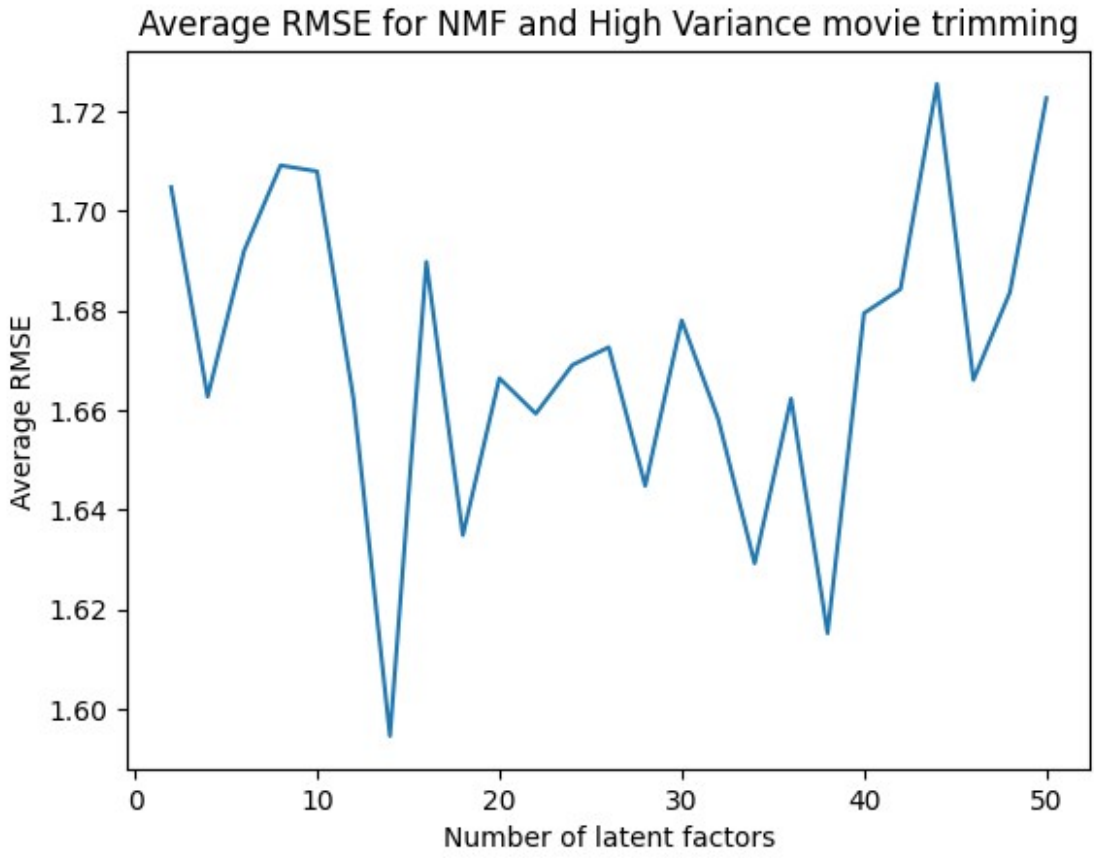



```
#High Variance
tracker = {}
for row in data.raw_ratings:
    if row[1] not in tracker:
      tracker[row[1]] = []
    tracker[row[1]].append(row[2])

RMSENMFhivar = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
      trimTest = [row for row in test if (len(tracker[row[1]]) >= 5
```

```
and np.var(tracker[row[1]]) >= 2)]
      nmf = NMF(n_factors=idx)
      nmf = nmf.fit(train)
      nmf = nmf.test(trimTest)
      currRMSE.append(accuracy.rmse(nmf))
    RMSENMFhivar.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSENMFhivar))
```

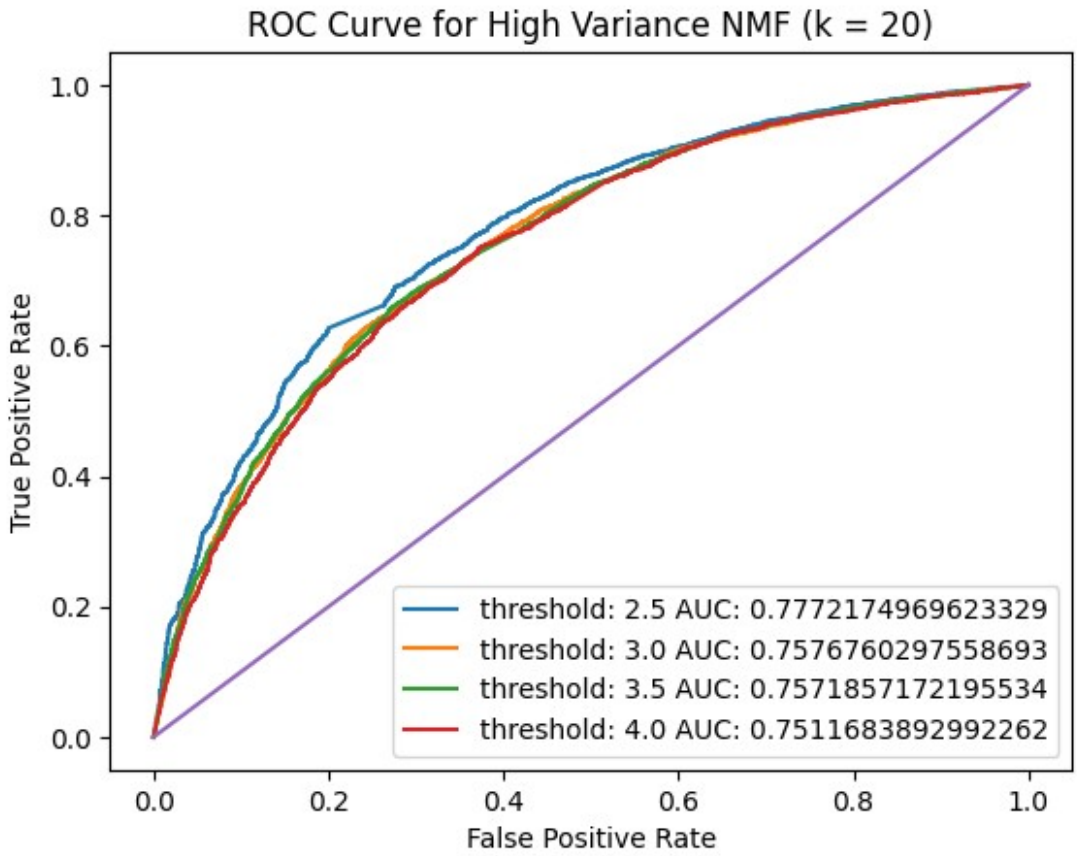
```
Minimum Average RMSE: 1.5946200381286004
```

```
plt.plot(k,RMSENMFhivar)
plt.title('Average RMSE for NMF and High Variance movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```




```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_hivar = list(zip(k,RMSENMFhivar))
min_rmse_hivar = min(comb_rmse, key = lambda t: t[1])
trainset, testset = train_test_split(data, test_size=0.1)
nmf = NMF(n_factors=min_rmse_hivar[0])
nmf = nmf.fit(train)
```

```
nmf = nmf.test(test)
fig, ax = plt.subplots()
for thresh in threshold_vals:
    predicted = []
    for row in nmf:
        if row.r_ui <= thresh:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in nmf]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'threshold: ' + str(thresh) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for High Variance NMF (k = ' +
str(min_rmse_hivar[0]) + ')'
plt.title(title)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```



###Question 9

The output has the genres of the top 10 movies for each number of latent factors. As you can see, for all the latent factors, the top 10 movies belong to a small collection of genres. A pattern that arises is when the number of latent factors increases, the number of distinct movie genres decreases. This is because movie genres are clustered more the higher the number of latent factors is.

```
from surprise.model_selection import train_test_split
from surprise.prediction_algorithms.matrix_factorization import NMF

train, test = train_test_split(data, test_size=.1)
df = pd.read_csv('movies.csv',names=['movieid','title','genres'])
df = df['genres']
nmf = NMF(n_factors=20)
nmf.fit(train)
nmf.test(test)
k = [0,4,8,12,16,19]
for i in k:
  print(i)
  tmp = nmf.qi[:,i]
  tmp2 = [(a,b) for a,b in enumerate(tmp)]
  tmp2.sort(key = lambda x:x[1], reverse=True)
  for idx in tmp2[:10]:
      print(df[idx[0]])
```

```
0
Drama
Action|Adventure|Fantasy|Sci-Fi
Drama|Mystery|Thriller
Action|Sci-Fi
Comedy
Action|Drama
Drama|Romance
Action
Action|Adventure|Sci-Fi
Action|Drama|IMAX
4
Comedy
Action
Drama
Drama|Romance
Action|Drama|War
Action|Crime
Comedy|Drama
Adventure|Comedy
Drama
Comedy|Drama
8
Comedy
```

```
Comedy|Drama
Comedy|Romance
Drama
Horror|Thriller
Children|Comedy
Comedy|Crime|Drama|Thriller
Comedy|Drama|Romance
Drama|Romance
Horror|Sci-Fi|Thriller
12
Adventure|Animation|Comedy
Drama|Mystery|Romance
Children|Comedy|Musical
Action|Crime|Drama|Thriller
Action|Drama|Sci-Fi|Thriller
Drama|Horror|Sci-Fi|Thriller
Action|Crime|Drama|Thriller
Thriller
Comedy
Action|Sci-Fi
16
Drama|War
Drama
Horror|Romance
Drama
Animation|Sci-Fi
Comedy|Drama
Thriller
Adventure|Children
Drama
Comedy|Drama|Romance
19
Romance
Crime|Drama
Drama
Comedy|Drama
Comedy|Crime
Comedy|Romance
Action|Drama|Sci-Fi|Thriller
Comedy
Sci-Fi
Comedy|Drama
```

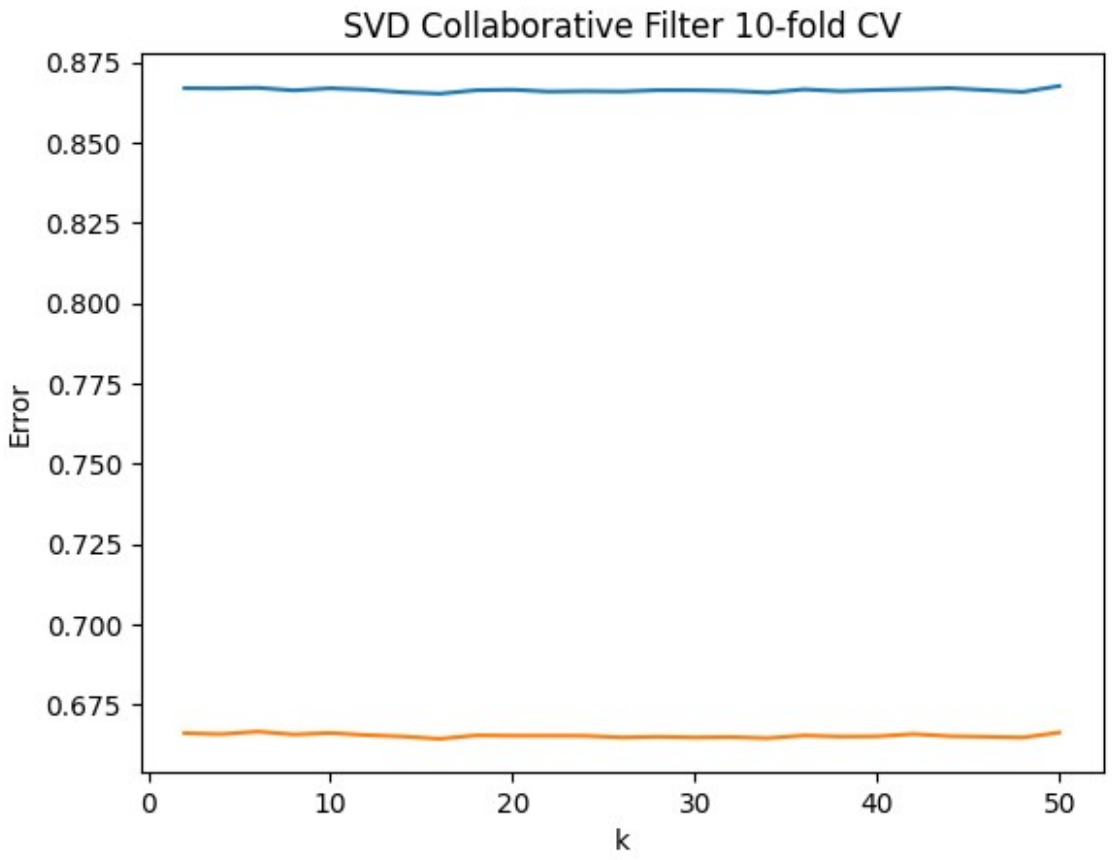
###Question *10*

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.

```
from surprise.prediction_algorithms.matrix_factorization import SVD
k = np.linspace(2,50,num=25,dtype=int)
maeAvg = []
rmseAvg = []
for idx in k:
    print(idx)
    svd = SVD(n_factors=idx)
    cv = cross_validate(svd,data,cv=10)
    rmse_mean = np.mean(cv['test_rmse'])
    mae_mean = np.mean(cv['test_mae'])
    rmseAvg.append(rmse_mean)
    maeAvg.append(mae_mean)

fig, ax = plt.subplots()
ax.plot(k,rmseAvg, label='Avg. RMSE')
ax.plot(k, maeAvg,label='Avg. MAE')
plt.xlabel("k");
plt.ylabel("Error");
plt.title("SVD Collaborative Filter 10-fold CV")
```

```
Text(0.5, 1.0, 'SVD Collaborative Filter 10-fold CV')
```



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?

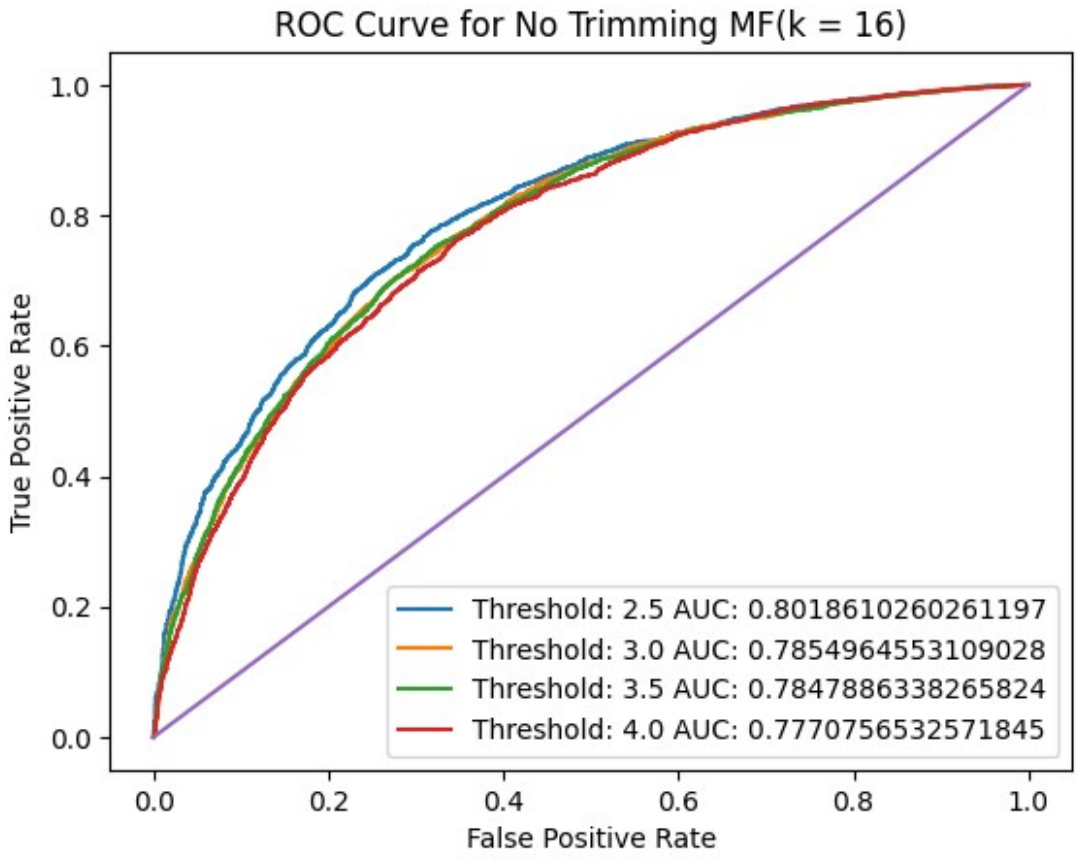
The minimum avg of both MAE and RMSE show that k=16. This is pretty close to the number of movie genres which is 19.

```
comb_mae = list(zip(k,maeAvg))
comb_rmse = list(zip(k,rmseAvg))
min_mae = min(comb_mae, key = lambda t: t[1])
min_rmse = min(comb_rmse, key = lambda t: t[1])

print("Minimum average MAE: ", min_mae[1], " k: ", min_mae[0])
print("Minimum average RMSE: ", min_rmse[1], " k: ", min_rmse[0])
```

```
Minimum average MAE:  0.6643649579801207  k:  16
Minimum average RMSE:  0.8651467248215328  k:  16
```

```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_pop = list(zip(k,rmseAvg))
min_rmse_pop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
svd = SVD(n_factors=min_rmse_pop[0])
```

```
svd = svd.fit(train)
svd = svd.test(test)
fig, ax = plt.subplots()
for threshold in threshold_vals:
    predicted = []
    for row in svd:
        if row.r_ui <= threshold:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in svd]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(threshold) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for No Trimming MF(k = ' + str(min_rmse_pop[0]) +
')'
plt.title(title)
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



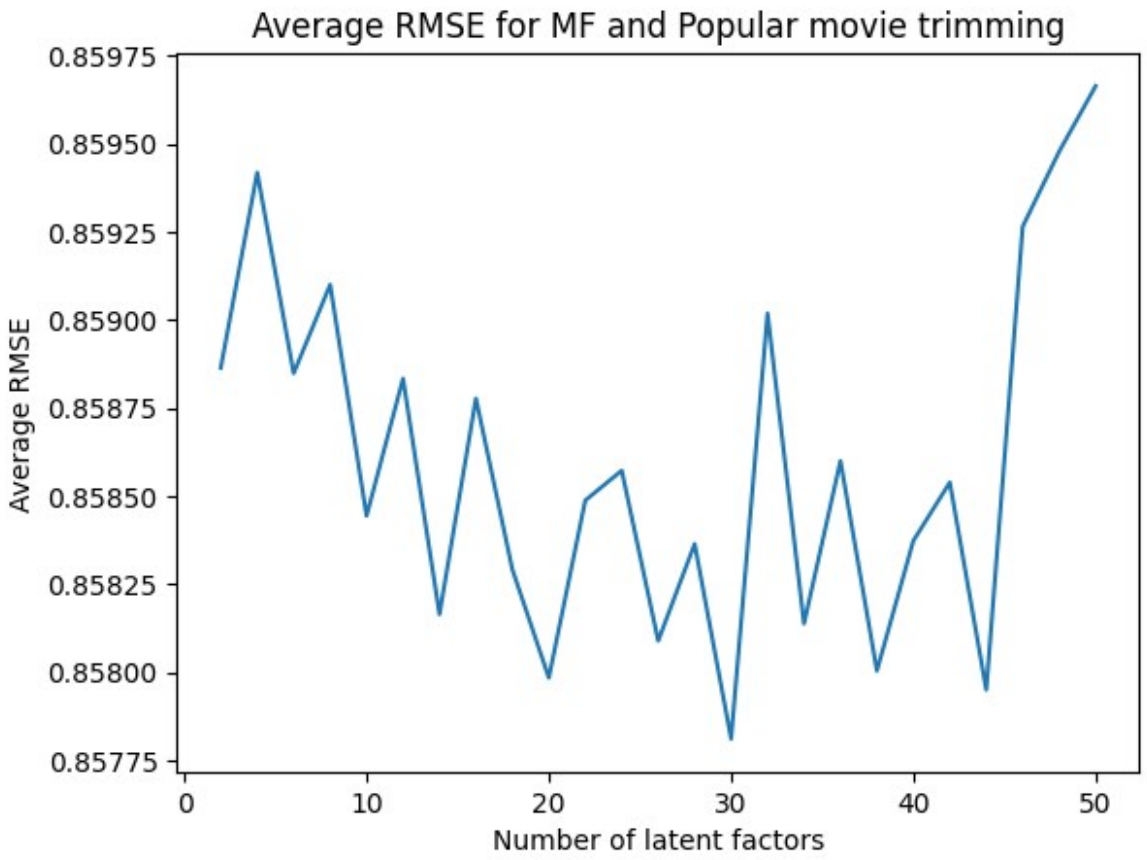
C) Performance on trimmed dataset subsets: For each of Popular, Unpopular and High-Variance subsets:

- **Design a MF collaborative filter for each trimmed subset 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 obtained by averaging the RMSE across all 10 folds.**
- **Plot average RMSE (Y-axis) against k (X-axis). Report the minimum average RMSE.**
- **Plot the ROC curves for the MF-based collaborative filter and also report the area under the curve (AUC) value as done in Question 6.**

```
from surprise.model_selection import KFold
from surprise import accuracy

tracker = {}
for row in data.raw_ratings:
  if row[1] not in tracker:
    tracker[row[1]] = []
  tracker[row[1]].append(row[2])

#Popular
RMSEMFpop = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
      trimTest = [row for row in test if len(tracker[row[1]]) > 2]
      svd = SVD(n_factors=idx)
      svd = svd.fit(train)
      svd = svd.test(trimTest)
      currRMSE.append(accuracy.rmse(svd))
    RMSEMFpop.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSEMFpop))
```

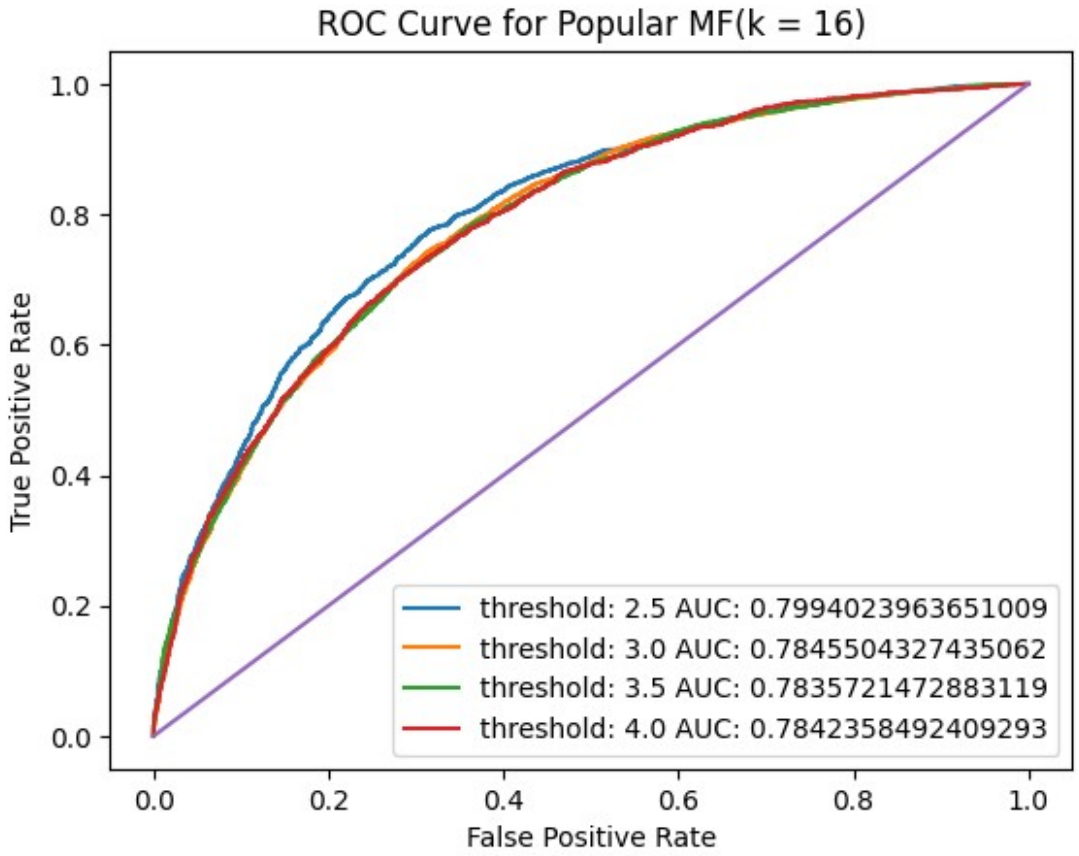
```
Minimum Average RMSE: 0.857810686086436
```

```
plt.plot(k,RMSEMFpop)
plt.title('Average RMSE for MF and Popular movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



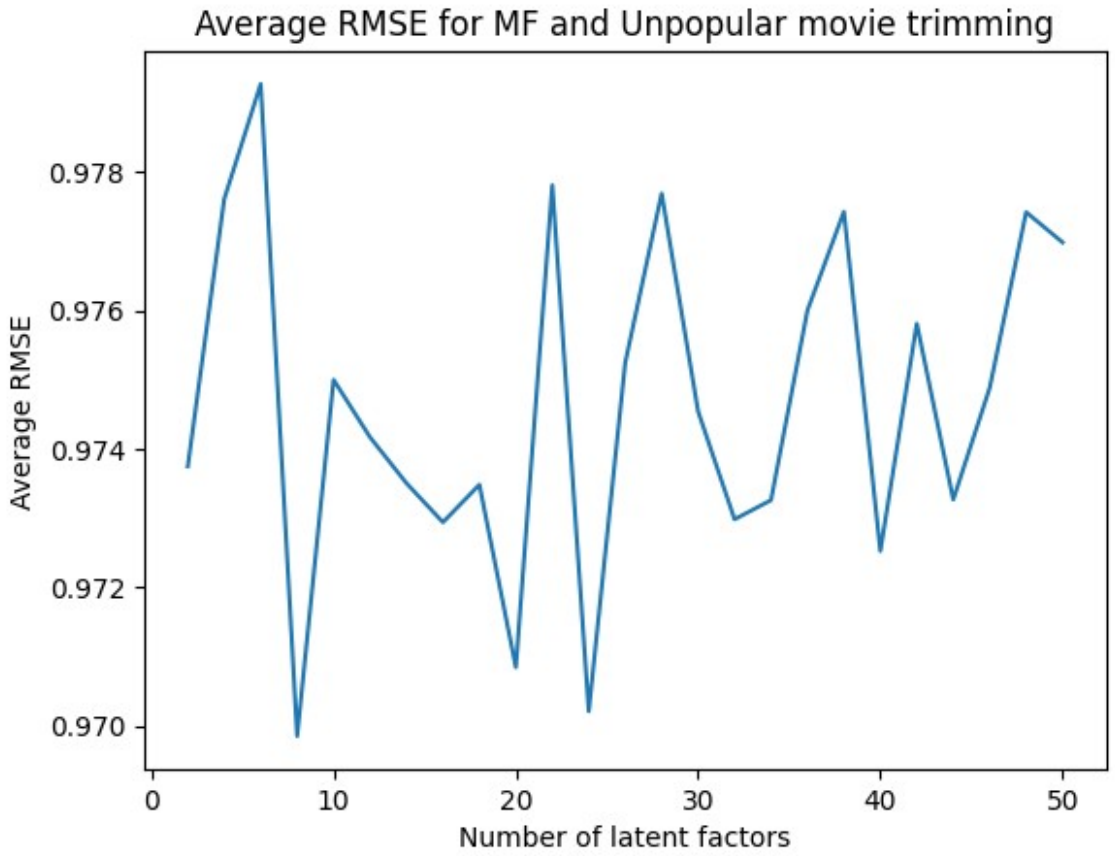

```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_pop = list(zip(k,RMSEMFpop))
min_rmse_pop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
svd = SVD(n_factors=min_rmse_pop[0])
svd = svd.fit(train)
svd = svd.test(test)
fig, ax = plt.subplots()
for threshold in threshold_vals:
    predicted = []
    for row in svd:
        if row.r_ui <= threshold:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in svd]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(threshold) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for Popular MF(k = ' + str(min_rmse_pop[0]) + ')'
plt.title(title)
```

```
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



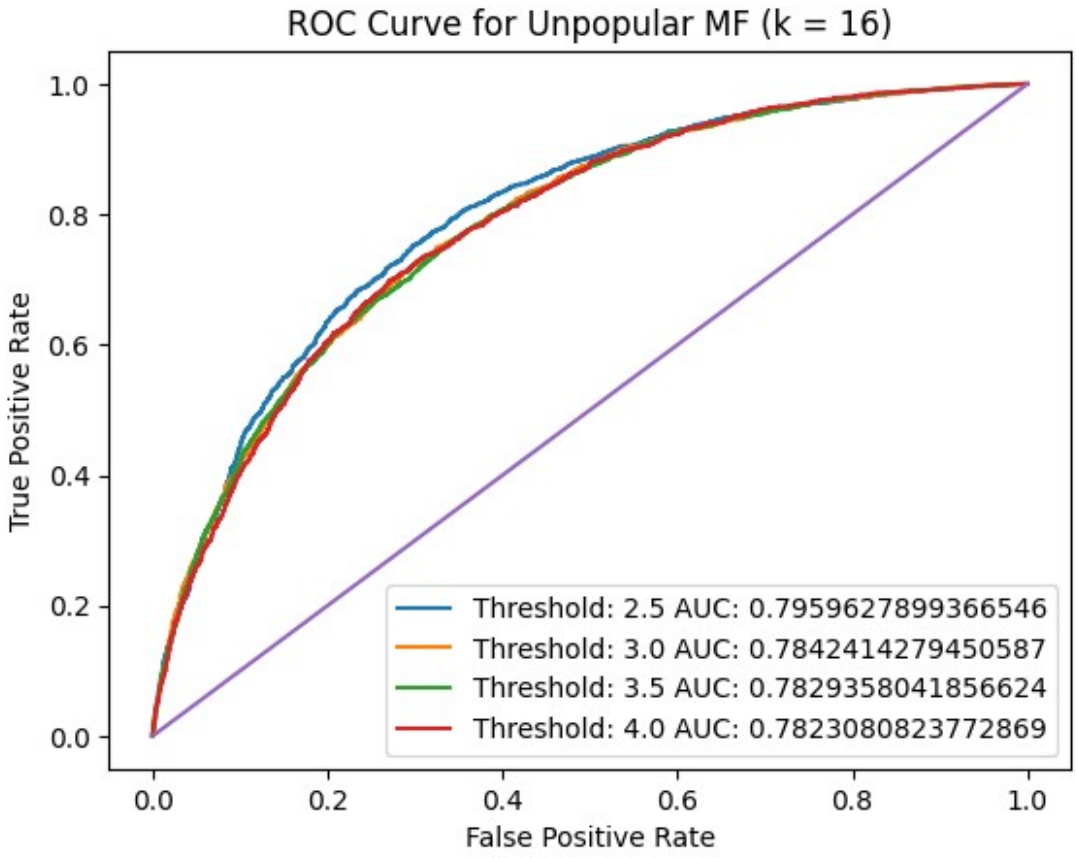

```
#Unpopular
RMSEMFunpop = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
      trimTest = [row for row in test if len(tracker[row[1]]) <= 2]
      svd = SVD(n_factors=idx)
      svd = svd.fit(train)
      svd = svd.test(trimTest)
      currRMSE.append(accuracy.rmse(svd))
    RMSEMFunpop.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSEMFunpop))

Minimum Average RMSE: 0.9698486551499339
```

```
plt.plot(k,RMSEMFunpop)
plt.title('Average RMSE for MF and Unpopular movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```




```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_unpop = list(zip(k,RMSEMFunpop))
min_rmse_unpop = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
svd = SVD(n_factors=min_rmse_unpop[0])
svd = svd.fit(train)
svd = svd.test(test)
fig, ax = plt.subplots()
for threshold in threshold_vals:
    predicted = []
    for row in svd:
        if row.r_ui <= threshold:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in svd]
```

```
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(threshold) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for Unpopular MF (k = ' + str(min_rmse_unpop[0]) +
')'
plt.title(title)
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

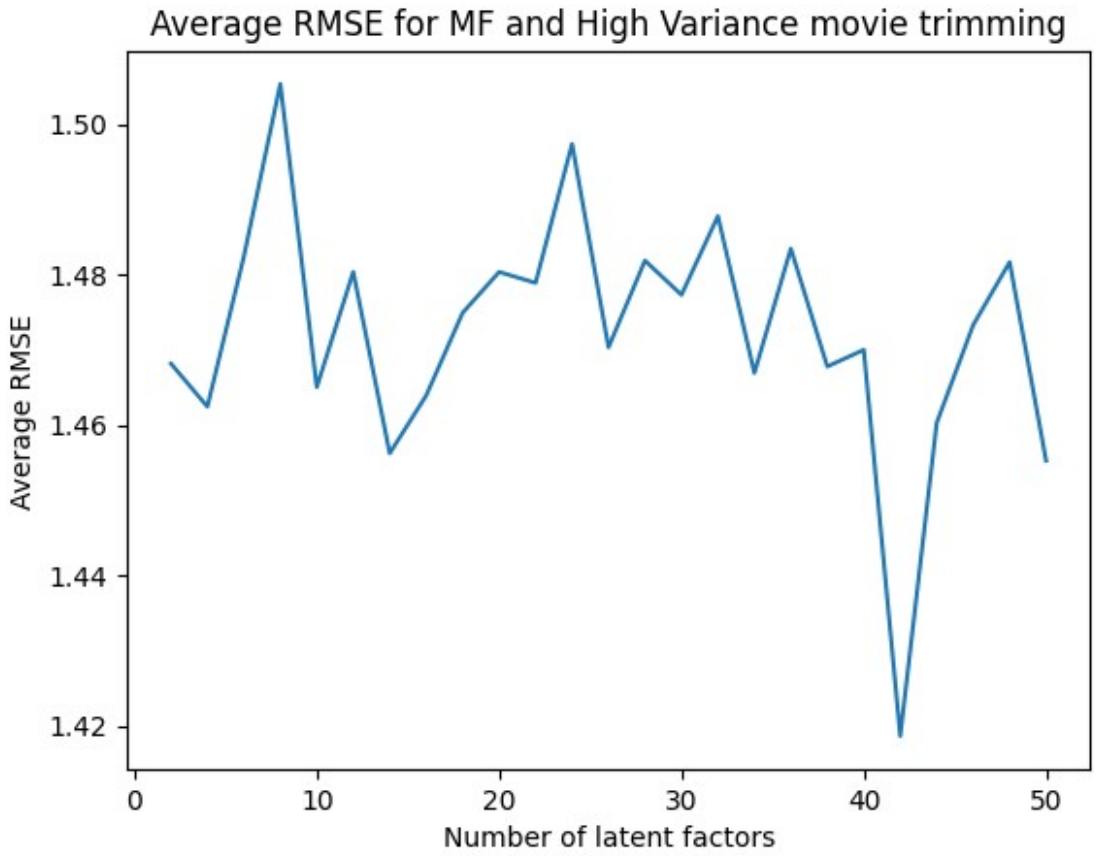



```
#High Variance
RMSEMFhivar = []
kf =  KFold(n_splits=10)
for idx in k:
    currRMSE = []
    print(idx)
    for train, test in kf.split(data):
        trimTest = [row for row in test if (np.var(tracker[row[1]]) >=
2 and len(tracker[row[1]]) >= 5)]
```

```
        svd = SVD(n_factors=idx)
        svd = svd.fit(train)
        svd = svd.test(trimTest)
        currRMSE.append(accuracy.rmse(svd,verbose=False))
    RMSEMFhivar.append(np.mean(currRMSE))

print("Minimum Average RMSE:", min(RMSEMFhivar))
```

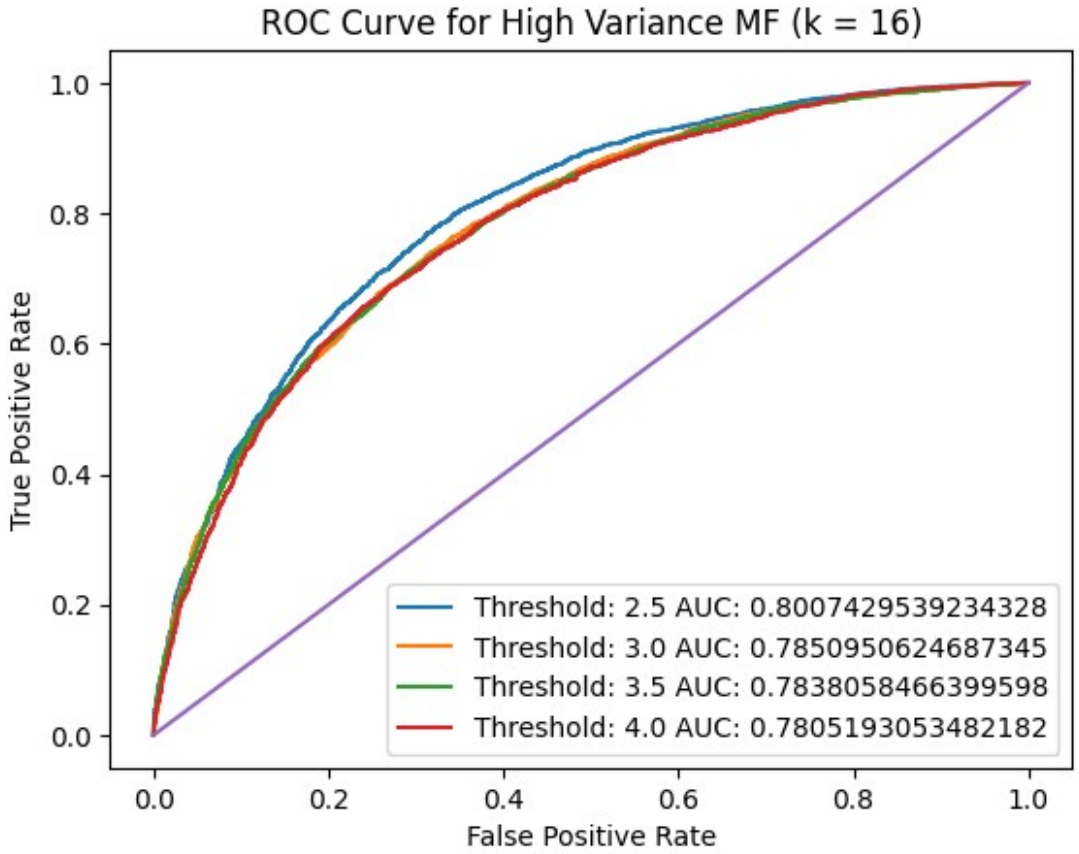
```
Minimum Average RMSE: 1.4186729308157928
```

```
plt.plot(k,RMSEMFhivar)
plt.title('Average RMSE for MF and High Variance movie trimming')
plt.ylabel('Average RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```




```
threshold_vals = [2.5, 3.0, 3.5, 4.0]
comb_rmse_hivar = list(zip(k,RMSEMFhivar))
min_rmse_hivar = min(comb_rmse, key = lambda t: t[1])
train, test = train_test_split(data, test_size=0.1)
svd = SVD(n_factors=min_rmse_hivar[0])
svd = svd.fit(train)
svd = svd.test(test)
```

```
fig, ax = plt.subplots()
for threshold in threshold_vals:
    predicted = []
    for row in svd:
        if row.r_ui <= threshold:
            predicted.append(0)
        else:
            predicted.append(1)
    actual = [row.est for row in svd]
    fpr, tpr, thresholds = roc_curve(predicted, actual)
    tmpLabel = 'Threshold: ' + str(threshold) + " AUC: " +
str(auc(fpr,tpr))
    ax.plot(fpr, tpr, label=tmpLabel)
ax.plot([0, 1], [0, 1])
title = 'ROC Curve for High Variance MF (k = ' +
str(min_rmse_hivar[0]) + ')'
plt.title(title)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```



###Question *11* **Design a naive collaborative filter to predict the ratings of the movies in the original dataset and evaluate it's performance using 10-fold cross validation. Compute the average RMSE by averaging the RMSE across all 10 folds. Report the average RMSE.**

```
from sklearn.metrics import mean_squared_error
from surprise.model_selection import KFold
tracker = {}
for row in data.raw_ratings:
  if row[0] not in tracker:
    tracker[row[0]] = []
  tracker[row[0]].append(row[2])

user_tracker = {}
for idx in tracker:
  user_tracker[idx] = np.mean(tracker[idx])

rmse = []
kf = KFold(n_splits=10)
for trainset, testset in kf.split(data):
  predicted = [user_tracker[i[0]] for i in testset]
  actual = [float(idx[2]) for idx in testset]
  rmse += [np.sqrt(mean_squared_error(actual,predicted))]
avg_rmse = np.mean(rmse)

print("Average RMSE for naive collaborative filter: ", avg_rmse)
```

```
Average RMSE for naive collaborative filter:  0.934690951896582
```

Performance on dataset subsets: For each of Popular, Unpopular and High-Variance test subsets -

- Design a naive collaborative filter for each trimmed set and evaluate its performance using 10-fold cross validation.

- Compute the average RMSE by averaging the RMSE across all 10 folds. Report the average RMSE

```
#Popular
tracker1 = {}
for row in data.raw_ratings:
  if row[1] not in tracker1:
    tracker1[row[1]] = []
  tracker1[row[1]].append(row[2])

rmse = []
kf = KFold(n_splits=10)
for trainset, testset in kf.split(data):
  popular = [row for row in testset if len(tracker1[row[1]]) > 2]
  predicted = [user_tracker[i[0]] for i in popular]
  actual = [idx[2] for idx in popular]
```

```
  rmse += [np.sqrt(mean_squared_error(actual,predicted))]
averageRmsePop = np.mean(rmse)

print('Avg. RMSE for Naive Filtering (Popular movie trimming):
',averageRmsePop)
```

```
Avg. RMSE for Naive Filtering (Popular movie trimming):
0.9322666805847106
```

```
#Unpopular
rmse = []
kf = KFold(n_splits=10)
for trainset, testset in kf.split(data):
  popular = [row for row in testset if len(tracker1[row[1]]) <= 2]
  predicted = [user_tracker[i[0]] for i in popular]
  actual = [idx[2] for idx in popular]
  rmse += [np.sqrt(mean_squared_error(actual,predicted))]
averageRmseUnPop = np.mean(rmse)

print('Avg. RMSE for Naive Filtering (Unpopular movie trimming):
',averageRmseUnPop)
```

```
Avg. RMSE for Naive Filtering (Unpopular movie trimming):
0.9708290953335542
```

```
#High Variance
rmse = []
kf = KFold(n_splits=10)
for trainset, testset in kf.split(data):
  hivar = [row for row in testset if (len(tracker1[row[1]]) >= 5 and
np.var(tracker1[row[1]]) >= 2)]
  predicted = [user_tracker[i[0]] for i in hivar]
  actual = [idx[2] for idx in hivar]
  rmse += [np.sqrt(mean_squared_error(actual,predicted))]
averageRmseHiVar = np.mean(rmse)

print('Avg. RMSE for Naive Filtering (Unpopular movie trimming):
',averageRmseHiVar)
```

```
Avg. RMSE for Naive Filtering (Unpopular movie trimming):
1.4659303344188763
```

###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.

When threshold = 3, we can see that MF seems to perform the best at predicting the ratings of movies as it has a smoother curve and largest area under the curve. KNN seems to perform the next best and NMF seems to perform the worst.

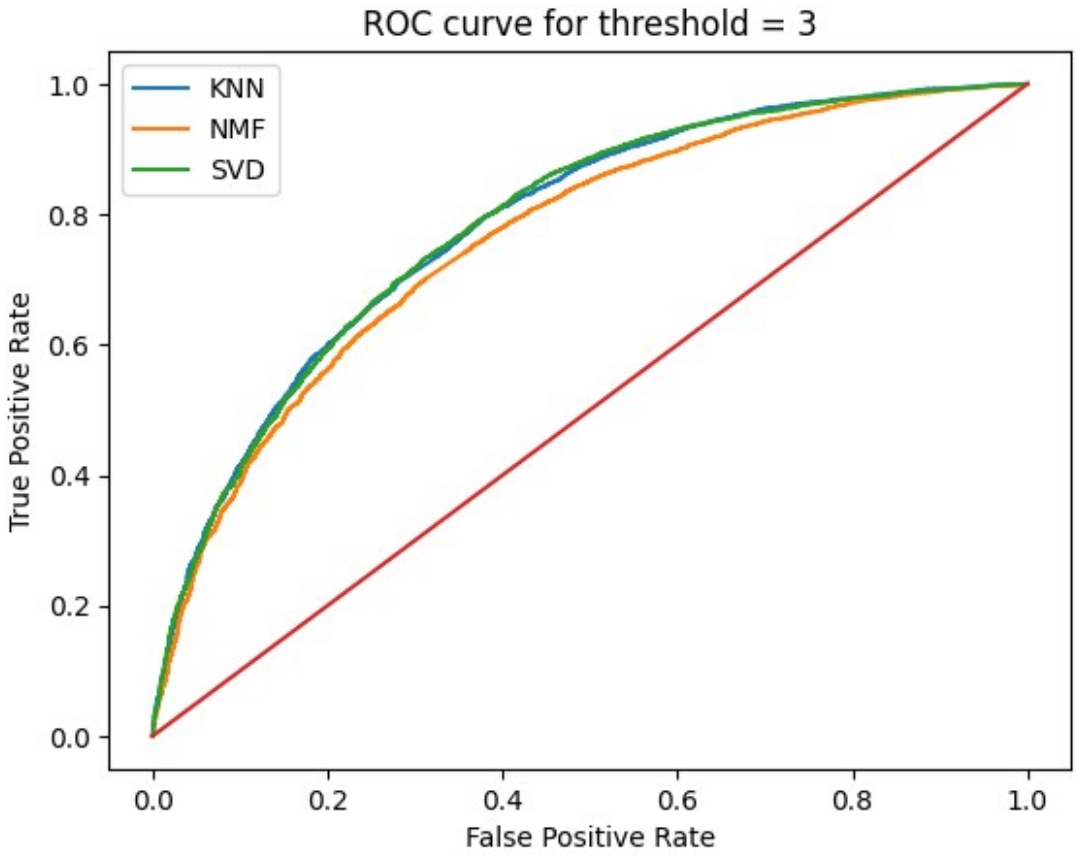
```
train, test = train_test_split(data, test_size=.1)
threshold = 3
plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

```
<Figure size 640x480 with 0 Axes>
```

```
#KNN
knn = KNNWithMeans(k=20,sim_options={'name':'pearson'},verbose=False)
knn = knn.fit(train)
knn = knn.test(test)
actual = []
for i in knn:
  if i.r_ui <= threshold:
      actual.append(0)
  else:
      actual.append(1)
scores = [idx.est for idx in knn]
fpr_knn, tpr_knn, thresholds = roc_curve(actual, scores)

nmf = NMF(n_factors=20)
nmf = nmf.fit(train)
nmf = nmf.test(test)
actual = []
for i in nmf:
    if i.r_ui <= threshold:
      actual.append(0)
    else:
      actual.append(1)

scores = [i.est for i in nmf]
fpr_nmf, tpr_nmf, thresholds = roc_curve(actual, scores)

svd = SVD(n_factors=16)
svd = svd.fit(train)
svd = svd.test(test)
actual = []
for i in svd:
    if i.r_ui <= threshold:
      actual.append(0)
    else:
      actual.append(1)

scores = [i.est for i in svd]
fpr_svd, tpr_svd, thresholds = roc_curve(actual, scores)

plt.plot(fpr_knn, tpr_knn,label="KNN")
plt.plot(fpr_nmf, tpr_nmf, label='NMF')
plt.plot(fpr_svd, tpr_svd, label='SVD')
plt.plot([0, 1], [0, 1])
```

```
plt.legend()
plt.ylabel('True Positive Rate');
plt.xlabel('False Positive Rate');
plt.title('ROC curve for threshold = 3')
plt.show()
```



Project 3

Kuei-Tzu Hu 206300553

Sreya Muppalla 505675909

Christina Lee 406299676

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Ranking

Question 13

QUESTION 13: Data Understanding and Preprocessing:

• Use the provided helper code for loading and pre-processing Web10k data.

```
!pip install lightgbm
```

```
Requirement already satisfied: lightgbm in c:\users\myura\anaconda3\
lib\site-packages (4.3.0)
Requirement already satisfied: numpy in c:\users\myura\anaconda3\lib\
site-packages (from lightgbm) (1.24.3)
Requirement already satisfied: scipy in c:\users\myura\anaconda3\lib\
site-packages (from lightgbm) (1.11.1)
```

```
from sklearn.datasets import load_svmlight_file
from sklearn.metrics import ndcg_score
import numpy as np

# Load the dataset for one fold
def load_one_fole(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)
    y_train = y_train.astype(int)
    y_test = y_test.astype(int)
    _, group_train = np.unique(qid_train, return_counts=True)
    _, group_test = np.unique(qid_test, return_counts=True)
    return X_train, y_train, qid_train, group_train, X_test, y_test,
qid_test, group_test

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)

datapath1 = "./MSLR-WEB10K/Fold1/"
datapath2 = "./MSLR-WEB10K/Fold2/"
datapath3 = "./MSLR-WEB10K/Fold3/"
datapath4 = "./MSLR-WEB10K/Fold4/"
datapath5 = "./MSLR-WEB10K/Fold5/"

X1_train, y1_train, qid1_train, group1_train, X1_test, y1_test,
qid1_test, group1_test = load_one_fole(datapath1)
X2_train, y2_train, qid2_train, group2_train, X2_test, y2_test,
qid2_test, group2_test = load_one_fole(datapath2)
X3_train, y3_train, qid3_train, group3_train, X3_test, y3_test,
qid3_test, group3_test = load_one_fole(datapath3)
X4_train, y4_train, qid4_train, group4_train, X4_test, y4_test,
qid4_test, group4_test = load_one_fole(datapath4)
X5_train, y5_train, qid5_train, group5_train, X5_test, y5_test,
qid5_test, group5_test = load_one_fole(datapath5)

X_train = [X1_train, X2_train, X3_train, X4_train, X5_train]
y_train = [y1_train, y2_train, y3_train, y4_train, y5_train]
qid_train = [qid1_train, qid2_train, qid3_train, qid4_train,
qid5_train]
group_train = [group1_train, group2_train, group3_train, group4_train,
```

```
group5_train]

X_test = [X1_test, X2_test, X3_test, X4_test, X5_test]
y_test = [y1_test, y2_test, y3_test, y4_test, y5_test]
qid_test = [qid1_test, qid2_test, qid3_test, qid4_test, qid5_test]
group_test = [group1_test, group2_test, group3_test, group4_test,
group5_test]

for i in np.arange(len(X_train)) :
    print(X_train[i].shape)
```

```
(723412, 136)
(716683, 136)
(719111, 136)
(718768, 136)
(722602, 136)
```

• Print out the number of unique queries in total and show distribution of relevance labels.

- 6000 unique queries for each folders

```
# number of unique queries

for i in np.arange(len(group_train)) :
    print(group_train[i].shape)
```

```
(6000,)
(6000,)
(6000,)
(6000,)
(6000,)
```

```
y1, count1 = np.unique(y1_train, return_counts=True)
y2, count2 = np.unique(y2_train, return_counts=True)
y3, count3 = np.unique(y3_train, return_counts=True)
y4, count4 = np.unique(y4_train, return_counts=True)
y5, count5 = np.unique(y5_train, return_counts=True)

count = np.stack((count1, count2, count3, count4, count5))

count[0][1]
```

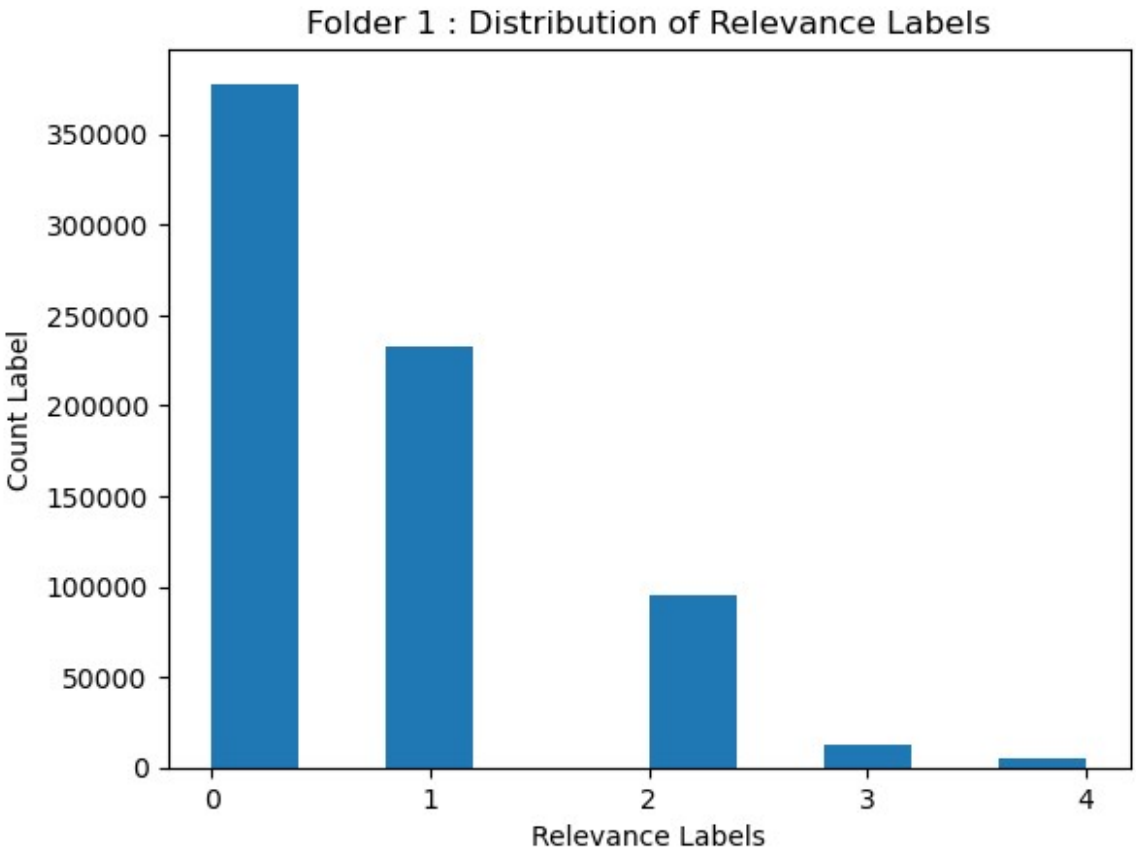
```
232569
```

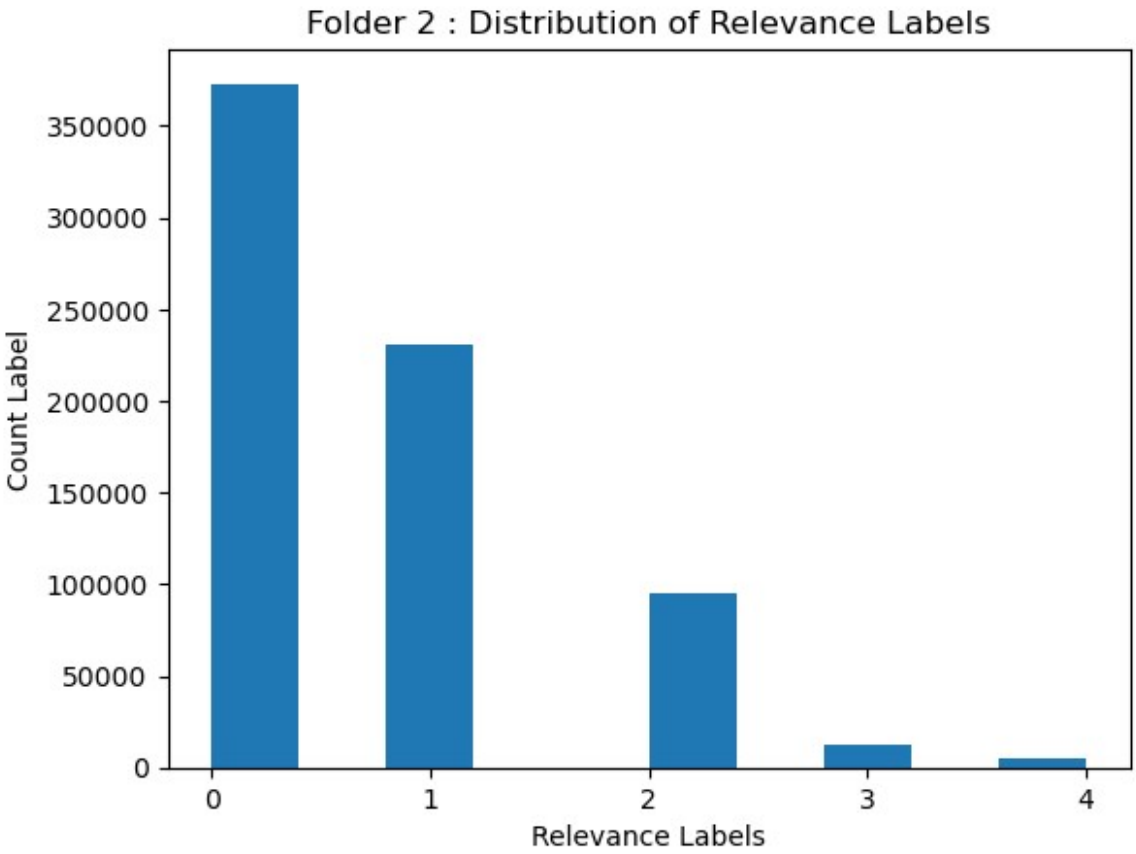
```
from tabulate import tabulate

col_names = ["0", "1", "2", "3", "4"]

data = [["Fold 1 ", count[0][0], count[0][1], count[0][2], count[0]
[3], count[0][4]],
        ["Fold 2 ", count[1][0], count[1][1], count[1][2], count[1]
[3], count[1][4]],
```

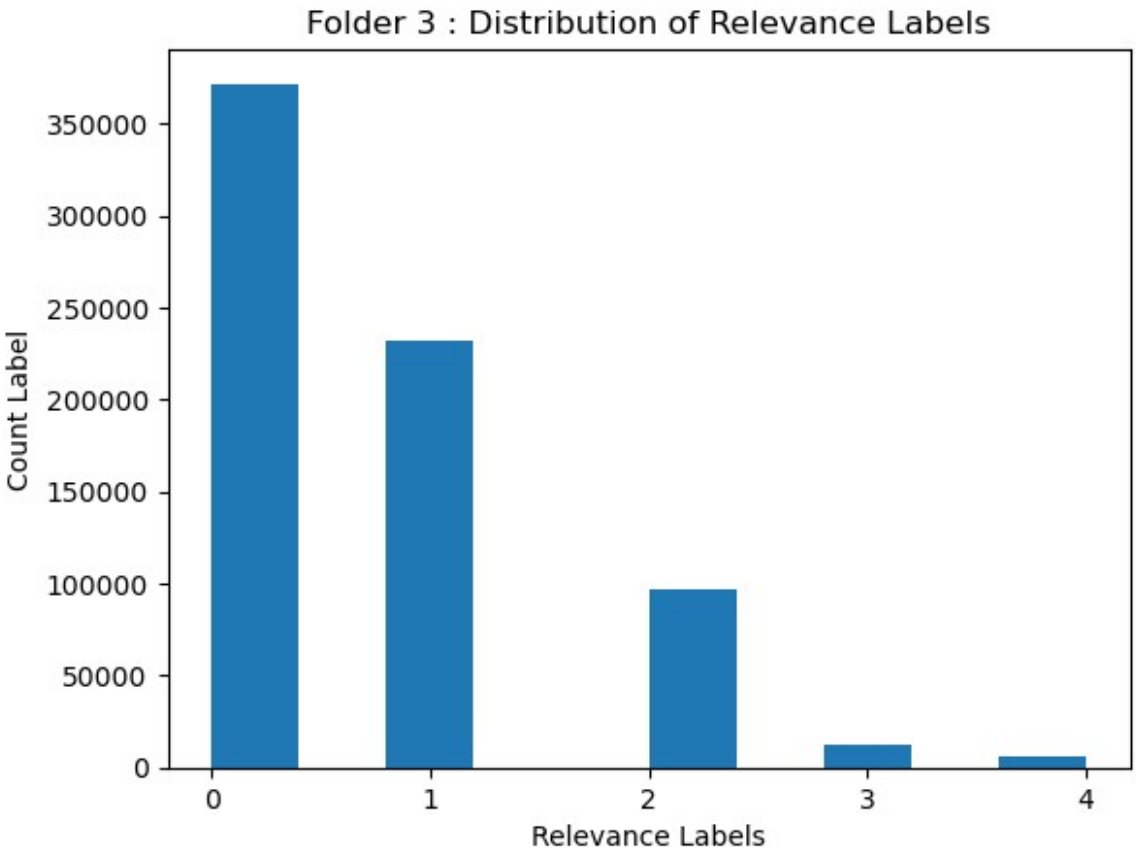
```
        ["Fold 3 ", count[2][0], count[2][1], count[2][2], count[2]
[3], count[2][4]],
        ["Fold 4 ", count[3][0], count[3][1], count[3][2], count[3]
[3], count[3][4]],
        ["Fold 5 ", count[4][0], count[4][1], count[4][2], count[4]
[3], count[4][4]]]

print(tabulate(data, headers=col_names))
```

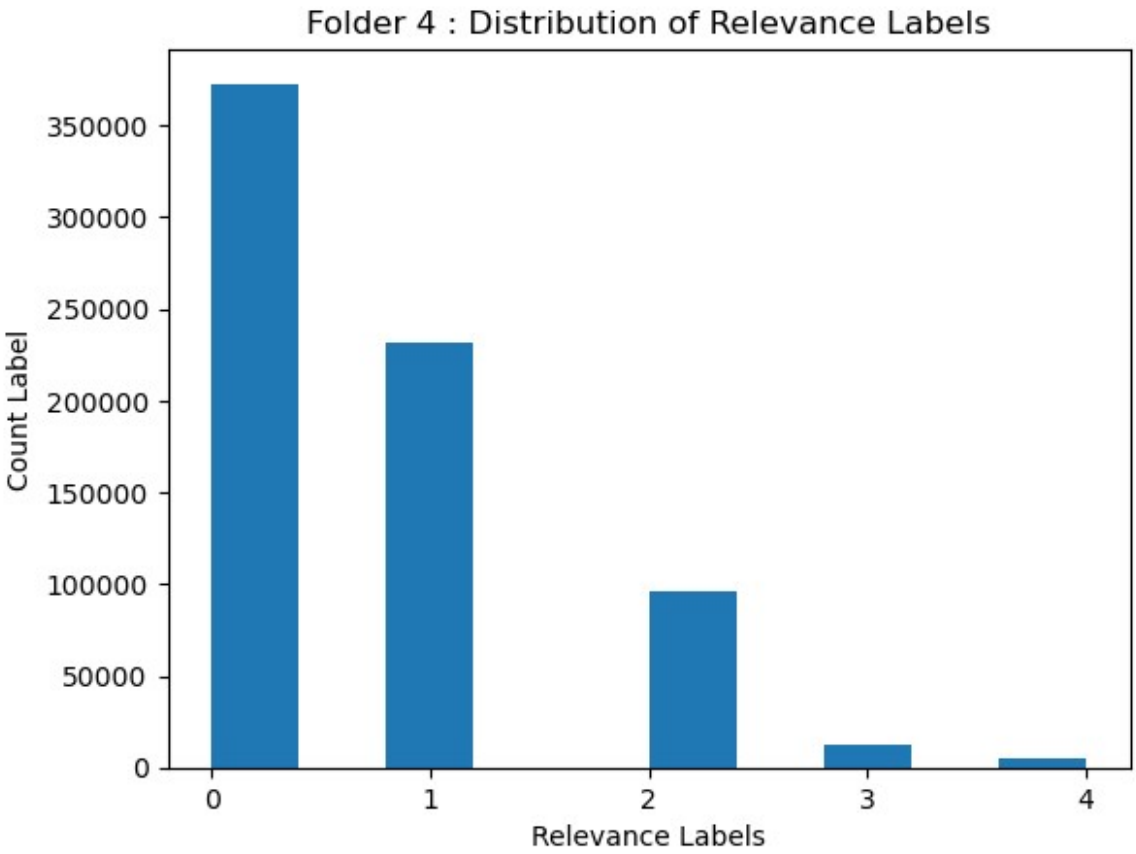
```
             0       1      2      3     4
------  ------  ------  -----  -----  ----
Fold 1  377957  232569  95082  12658  5146
Fold 2  373029  230368  95117  12814  5355
Fold 3  371725  232302  96663  12903  5518
Fold 4  372756  231727  96244  12712  5329
Fold 5  377322  231874  95247  12864  5295
```

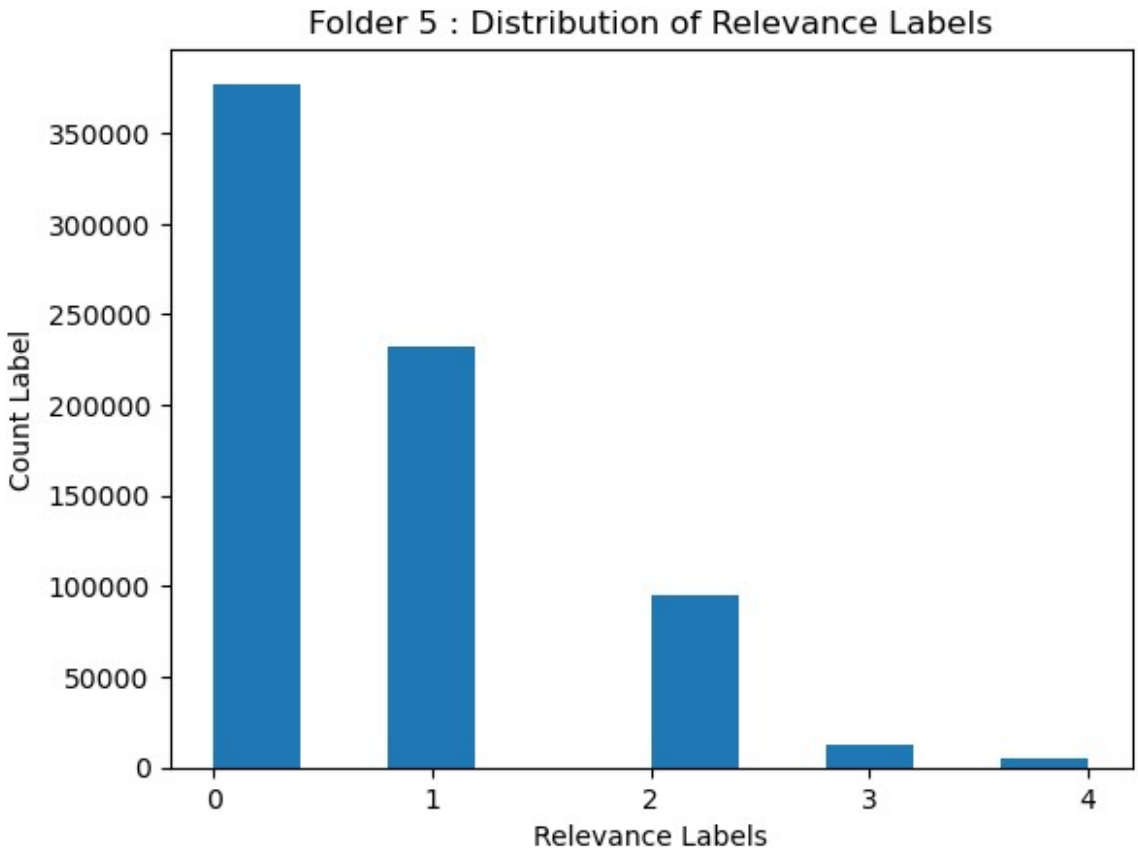
```
y_train = [y1_train, y2_train, y3_train, y4_train, y5_train]

for i in np.arange(5) :
  plt.figure()
  plt.hist(y_train[i])
  plt.xticks(np.arange(0, 4.5, 1))
  plt.xlabel("Relevance Labels");
  plt.ylabel("Count Label");
  plt.title(f"Folder {i + 1} : Distribution of Relevance Labels")
```


Folder 1 : Distribution of Relevance Labels
Count Label
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
Relevance Labels


Folder 2 : Distribution of Relevance Labels
Count Label
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
Relevance Labels


Folder 3 : Distribution of Relevance Labels
Count Label
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
Relevance Labels


Folder 4 : Distribution of Relevance Labels
Count Label
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
Relevance Labels



Question 14

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.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.456457	0.45389	0.449068	0.461179	0.469634
nDCG@5	0.463289	0.457329	0.458348	0.466386	0.471432
nDCG@10	0.482867	0.476755	0.475895	0.487725	0.490359

```
import lightgbm as lgb
gbm1 = lgb.LGBMRanker()
gbm2 = lgb.LGBMRanker()
gbm3 = lgb.LGBMRanker()
gbm4 = lgb.LGBMRanker()
gbm5 = lgb.LGBMRanker()

fold1 = gbm1.fit(X1_train, y1_train, group=group1_train)
fold2 = gbm2.fit(X2_train, y2_train, group=group2_train)
```

```
fold3 = gbm3.fit(X3_train, y3_train, group=group3_train)
fold4 = gbm4.fit(X4_train, y4_train, group=group4_train)
fold5 = gbm5.fit(X5_train, y5_train, group=group5_train)
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.469313 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 25637
[LightGBM] [Info] Number of data points in the train set: 723412,
number of used features: 136
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.558580 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 25623
[LightGBM] [Info] Number of data points in the train set: 716683,
number of used features: 136
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.391367 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 25659
[LightGBM] [Info] Number of data points in the train set: 719111,
number of used features: 136
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.460278 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 25631
[LightGBM] [Info] Number of data points in the train set: 718768,
number of used features: 136
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.393762 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 25501
[LightGBM] [Info] Number of data points in the train set: 722602,
number of used features: 136
```

```
folds = [fold1, fold2, fold3, fold4, fold5]
X_test = [X1_test, X2_test, X3_test, X4_test, X5_test]
y_test = [y1_test, y2_test, y3_test, y4_test, y5_test]
qid_test = [qid1_test, qid2_test, qid3_test, qid4_test, qid5_test]

ndcg3 = []
ndcg5 = []
ndcg10 = []

for i in np.arange(5) :
```

```
    n3 = compute_ndcg_all(folds[i], X_test[i], y_test[i], qid_test[i],
k=3)
    n5 = compute_ndcg_all(folds[i], X_test[i], y_test[i], qid_test[i],
k=5)
    n10 = compute_ndcg_all(folds[i], X_test[i], y_test[i],
qid_test[i], k=10)
    ndcg3 = np.append(ndcg3, n3)
    ndcg5 = np.append(ndcg5, n5)
    ndcg10 = np.append(ndcg10, n10)

col_names = ["Fold 1", "Fold 2", "Fold 3", "Fold 4", "Fold 5"]

data = [["nDCG@3", ndcg3[0], ndcg3[1], ndcg3[2], ndcg3[3], ndcg3[4]],
        ["nDCG@5", ndcg5[0], ndcg5[1], ndcg5[2], ndcg5[3], ndcg5[4]],
        ["nDCG@10", ndcg10[0], ndcg10[1], ndcg10[2], ndcg10[3],
ndcg10[4]]]

print(tabulate(data, headers=col_names))
```

```
           Fold 1    Fold 2    Fold 3    Fold 4    Fold 5
-------  --------  --------  --------  --------  --------
nDCG@3   0.456457  0.45389   0.449068  0.461179  0.469634
nDCG@5   0.463289  0.457329  0.458348  0.466386  0.471432
nDCG@10  0.482867  0.476755  0.475895  0.487725  0.490359
```

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.
Please use model.booster .feature importance(importance type='gain') as demonstrated here for retrieving importance score per feature. You can also find helper code in the provided notebook.

```
# Fold 1

importance_df1 = (
    pd.DataFrame({
        'feature_name': fold1.feature_name_,
        'importance_gain': get_feature_importance(fold1,
importance_type='gain'),
        'importance_split': get_feature_importance(fold1,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df1.head())
```

```
  feature_name  importance_gain  importance_split
0   Column_133     23856.702951                92
```

```
1     Column_7      4248.546391                13
2   Column_107      4135.244450               116
3    Column_54      4078.463216                29
4   Column_129      3635.037024               146
```

```
# Fold 2

importance_df2 = (
    pd.DataFrame({
        'feature_name': fold2.feature_name_,
        'importance_gain': get_feature_importance(fold2,
importance_type='gain'),
        'importance_split': get_feature_importance(fold2,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df2.head())
```

```
  feature_name  importance_gain  importance_split
0   Column_133     23578.908250                82
1     Column_7      5157.964912                18
2    Column_54      4386.669757                44
3   Column_107      4094.012172               105
4   Column_129      4035.070673               151
```

```
# Fold 3

importance_df3 = (
    pd.DataFrame({
        'feature_name': fold3.feature_name_,
        'importance_gain': get_feature_importance(fold3,
importance_type='gain'),
        'importance_split': get_feature_importance(fold3,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df3.head())
```

```
  feature_name  importance_gain  importance_split
0   Column_133     23218.075441                80
1    Column_54      4991.303372                35
2   Column_107      4226.807395                93
3   Column_129      4059.752514               157
4     Column_7      3691.792320                11
```

```
# Fold 4
```

```
importance_df4 = (
    pd.DataFrame({
        'feature_name': fold4.feature_name_,
        'importance_gain': get_feature_importance(fold4,
importance_type='gain'),
        'importance_split': get_feature_importance(fold4,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df4.head())
```

```
  feature_name  importance_gain  importance_split
0   Column_133     23796.899673               78
1     Column_7      4622.622978               15
2    Column_54      3883.481706               20
3   Column_129      3356.846980              158
4   Column_128      3207.575537              113
```

```
# Fold 5

importance_df5 = (
    pd.DataFrame({
        'feature_name': fold5.feature_name_,
        'importance_gain': get_feature_importance(fold5,
importance_type='gain'),
        'importance_split': get_feature_importance(fold5,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df5.head())
```

```
  feature_name  importance_gain  importance_split
0   Column_133     23540.942354               92
1     Column_7      4794.945172               12
2    Column_54      4079.608554               27
3   Column_107      3514.835752               87
4   Column_129      3209.058444              146
```

QUESTION 16: Experiments with Subset of Features:

For each of the five provided folds:

- 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 queryurl 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.

Original Model nDCG :

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.456457	0.45389	0.449068	0.461179	0.469634
nDCG@5	0.463289	0.457329	0.458348	0.466386	0.471432
nDCG@10	0.482867	0.476755	0.475895	0.487725	0.490359

Drop 20 most important feature Model nDCG :

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.454253	0.45729	0.44979	0.460635	0.470186
nDCG@5	0.462657	0.460267	0.45864	0.46734	0.473352
nDCG@10	0.481971	0.477253	0.477436	0.488881	0.490817

By removing the important 20 features for ranking the data, I expected for the DCG score to have a signoficant decrease. The outcome aligns with the expectations, as comparing to the original nDCG, the score decreased with every k and every folder by about 0.07.

```
feature1 = np.zeros(136)
feature2 = np.zeros(136)
feature3 = np.zeros(136)
feature4 = np.zeros(136)
feature5 = np.zeros(136)

for i in np.arange(136) :
    feature1[i] = ''.join(x for x in importance_df1.feature_name[i] if
x.isdigit())
    feature2[i] = ''.join(x for x in importance_df2.feature_name[i] if
x.isdigit())
    feature3[i] = ''.join(x for x in importance_df3.feature_name[i] if
x.isdigit())
    feature4[i] = ''.join(x for x in importance_df4.feature_name[i] if
x.isdigit())
    feature5[i] = ''.join(x for x in importance_df5.feature_name[i] if
x.isdigit())

features = [feature1, feature2, feature3, feature4, feature5]

print(feature1[2])
```

```
107.0
```

```
def dropcols_fancy(M, idx_to_drop):
    idx_to_drop = np.unique(idx_to_drop)
    keep = ~np.in1d(np.arange(M.shape[1]), idx_to_drop,
```

```
assume_unique=True)
    return M[:, np.where(keep)[0]]

ind_drop20 = [features[0][0:20], features[1][0:20], features[2][0:20],
features[3][0:20], features[4][0:20]]
print(ind_drop20)
```

```
[array([133.,   7., 107.,  54., 129., 128., 134.,  64., 126.,  14.,
132.,
        13., 122., 125., 108., 130.,  29., 127., 109.,  48.]),
array([133.,   7.,  54., 107., 129., 128., 132., 126.,  13., 134.,
10.,
       130., 125., 108.,  64.,  14., 122.,  48., 127.,  29.]),
array([133.,  54., 107., 129.,   7., 128.,  13.,  14., 134., 126.,
132.,
       130., 125., 114.,  29., 108., 127., 109.,  52.,  48.]),
array([133.,   7.,  54., 129., 128., 107.,  13., 134.,  64.,  14.,
132.,
       130., 126., 125., 108.,  29.,  47., 114.,  48., 127.]),
array([133.,   7.,  54., 107., 129., 128., 134., 126.,  13., 132.,
130.,
        48., 122.,  14.,  64., 125., 127., 108., 100.,  10.])]
```

```
X_train_drop20 = []
X_test_drop20 = []

for i in np.arange(5) :
    X_train_drop20 = np.append(X_train_drop20,
dropcols_fancy(X_train[i], ind_drop20[i]))
    X_test_drop20 = np.append(X_test_drop20, dropcols_fancy(X_test[i],
ind_drop20[i]))

print(X_train_drop20[4].shape)
print(X_train[4].shape)
```

```
(722602, 116)
(722602, 136)
```

```
print(X_train_drop20)
```

```
[<723412x116 sparse matrix of type '<class 'numpy.float64'>'
     with 83915792 stored elements in Compressed Sparse Row format>
 <716683x116 sparse matrix of type '<class 'numpy.float64'>'
     with 83135228 stored elements in Compressed Sparse Row format>
 <719111x116 sparse matrix of type '<class 'numpy.float64'>'
     with 83416876 stored elements in Compressed Sparse Row format>
 <718768x116 sparse matrix of type '<class 'numpy.float64'>'
     with 83377088 stored elements in Compressed Sparse Row format>
 <722602x116 sparse matrix of type '<class 'numpy.float64'>'
     with 83821832 stored elements in Compressed Sparse Row format>]
```

```
gbm1_drop20 = lgb.LGBMRanker()
gbm2_drop20 = lgb.LGBMRanker()
gbm3_drop20 = lgb.LGBMRanker()
gbm4_drop20 = lgb.LGBMRanker()
gbm5_drop20 = lgb.LGBMRanker()

fold1_drop20 = gbm1_drop20.fit(X_train_drop20[0], y1_train,
group=group1_train)
fold2_drop20 = gbm2_drop20.fit(X_train_drop20[1], y2_train,
group=group2_train)
fold3_drop20 = gbm3_drop20.fit(X_train_drop20[2], y3_train,
group=group3_train)
fold4_drop20 = gbm4_drop20.fit(X_train_drop20[3], y4_train,
group=group4_train)
fold5_drop20 = gbm5_drop20.fit(X_train_drop20[4], y5_train,
group=group5_train)

folds_drop20 = [fold1_drop20, fold2_drop20, fold3_drop20,
fold4_drop20, fold5_drop20]
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.648338 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 21582
[LightGBM] [Info] Number of data points in the train set: 723412,
number of used features: 116
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead
of testing was 1.054795 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 21551
[LightGBM] [Info] Number of data points in the train set: 716683,
number of used features: 116
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.381820 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 21720
[LightGBM] [Info] Number of data points in the train set: 719111,
number of used features: 116
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.393714 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 21670
[LightGBM] [Info] Number of data points in the train set: 718768,
number of used features: 116
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.464923 seconds.
You can set `force_row_wise=true` to remove the overhead.
```

```
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 21348
[LightGBM] [Info] Number of data points in the train set: 722602,
number of used features: 116
```

```
ndcg3_drop20 = []
ndcg5_drop20 = []
ndcg10_drop20 = []

for i in np.arange(5) :
    n3_d20 = compute_ndcg_all(folds_drop20[i], X_test_drop20[i],
y_test[i], qid_test[i], k=3)
    n5_d20 = compute_ndcg_all(folds_drop20[i], X_test_drop20[i],
y_test[i], qid_test[i], k=5)
    n10_d20 = compute_ndcg_all(folds_drop20[i], X_test_drop20[i],
y_test[i], qid_test[i], k=10)
    ndcg3_drop20 = np.append(ndcg3_drop20, n3_d20)
    ndcg5_drop20 = np.append(ndcg5_drop20, n5_d20)
    ndcg10_drop20 = np.append(ndcg10_drop20, n10_d20)

col_names = ["Fold 1", "Fold 2", "Fold 3", "Fold 4", "Fold 5"]

data = [["nDCG@3", ndcg3_drop20[0], ndcg3_drop20[1], ndcg3_drop20[2],
ndcg3_drop20[3], ndcg3_drop20[4]],
        ["nDCG@5", ndcg5_drop20[0], ndcg5_drop20[1], ndcg5_drop20[2],
ndcg5_drop20[3], ndcg5_drop20[4]],
        ["nDCG@10", ndcg10_drop20[0], ndcg10_drop20[1],
ndcg10_drop20[2], ndcg10_drop20[3], ndcg10_drop20[4]]]

print(tabulate(data, headers=col_names))
```

```
           Fold 1    Fold 2    Fold 3    Fold 4    Fold 5
-------  --------  --------  --------  --------  --------
nDCG@3   0.379675  0.373945  0.382383  0.381977  0.384283
nDCG@5   0.38503   0.381954  0.389996  0.39281   0.392168
nDCG@10  0.408364  0.404503  0.411636  0.412107  0.416687
```

```
importance_df1_drop20 = (
    pd.DataFrame({
        'feature_name': fold1_drop20.feature_name_,
        'importance_gain': get_feature_importance(fold1_drop20,
importance_type='gain'),
        'importance_split': get_feature_importance(fold1_drop20,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df1_drop20.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_47      7130.373842                26
1   Column_115      5221.429379               136
2     Column_9      3150.081209               184
3    Column_57      2859.952607                37
4    Column_52      2340.531604               115
```

```
importance_df2_drop20 = (
    pd.DataFrame({
        'feature_name': fold2_drop20.feature_name_,
        'importance_gain': get_feature_importance(fold2_drop20,
importance_type='gain'),
        'importance_split': get_feature_importance(fold2_drop20,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df2_drop20.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_46      7395.156866                24
1   Column_115      5048.871017               138
2    Column_56      2996.123232                39
3   Column_114      2575.246776               184
4    Column_51      2424.622715               109
```

```
importance_df3_drop20 = (
    pd.DataFrame({
        'feature_name': fold3_drop20.feature_name_,
        'importance_gain': get_feature_importance(fold3_drop20,
importance_type='gain'),
        'importance_split': get_feature_importance(fold3_drop20,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df3_drop20.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_24      5193.621002                 6
1   Column_115      4975.877589               147
2     Column_9      3713.671880               181
3    Column_56      3524.709081                46
4    Column_43      2704.906328               113
```

```
importance_df4_drop20 = (
    pd.DataFrame({
        'feature_name': fold4_drop20.feature_name_,
```

```
        'importance_gain': get_feature_importance(fold4_drop20,
importance_type='gain'),
        'importance_split': get_feature_importance(fold4_drop20,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df4_drop20.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_46      8002.362762                30
1   Column_115      4566.683576               133
2     Column_9      3429.345109               202
3    Column_51      2785.960996               110
4    Column_56      2668.623970                44
```

```
importance_df5_drop20 = (
    pd.DataFrame({
        'feature_name': fold5_drop20.feature_name_,
        'importance_gain': get_feature_importance(fold5_drop20,
importance_type='gain'),
        'importance_split': get_feature_importance(fold5_drop20,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df5_drop20.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_47      7827.671422                18
1   Column_115      4843.975987               127
2    Column_57      3432.321164                50
3    Column_43      2622.286555               114
4   Column_114      2455.116591               188
```

- 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.

Original Model nDCG :

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.456457	0.45389	0.449068	0.461179	0.469634
nDCG@5	0.463289	0.457329	0.458348	0.466386	0.471432
nDCG@10	0.482867	0.476755	0.475895	0.487725	0.490359

Drop 60 least important features Model nDCG :

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.454253	0.45729	0.44979	0.460635	0.470186
nDCG@5	0.462657	0.460267	0.45864	0.46734	0.473352
nDCG@10	0.481971	0.477253	0.477436	0.488881	0.490817

By removing the least important 60 features, I expected the nDCG score to decrease, but by a smaller degree than removing the top important 20 features. The least important features might not have much importance for training, but as almost half of the features are dropped I expected it to have a reasonable affect. As in result, the nDCG score remained almost the same, moslty very slightly decreasing, but in some cases, rather increasing. The reason of the consistance of the nDCG scores is, as shown below, that even though there are many features dropped, they have very low importance gain compared to the more important values. Therefore thay have very little affect to the ranking model. The slight increase of some of the scores could be caused by eliminating irrelevant features, enabling to focus on the more important features.

```
print(importance_df1[0:20])
```

```
   feature_name  importance_gain  importance_split
0    Column_133     23856.702951                92
1      Column_7      4248.546391                13
2    Column_107      4135.244450               116
3     Column_54      4078.463216                29
4    Column_129      3635.037024               146
5    Column_128      3141.889132               149
6    Column_134      2928.917522                94
7     Column_64      2473.033697                36
8    Column_126      2460.174453                80
9     Column_14      2398.556180                66
10   Column_132      2235.702615               114
11    Column_13      2173.701421                95
12   Column_122      1707.601828                52
13   Column_125      1586.174248               102
14   Column_108      1533.270157                61
15   Column_130      1530.053324               122
16    Column_29      1244.337857                37
17   Column_127      1143.260724                83
18   Column_109      1031.612578                56
19    Column_48       987.355606                23
```

```
print(importance_df1[76:136])
```

	feature_name	importance_gain	importance_split
76	Column_62	80.436729	13
77	Column_35	80.088470	14
78	Column_76	77.891471	6
79	Column_89	77.751060	14
80	Column_30	76.663699	11
81	Column_55	74.273940	15
82	Column_79	68.735949	12
83	Column_81	67.386480	4
84	Column_50	65.964510	12
85	Column_87	65.475030	11
86	Column_20	62.900810	14
87	Column_102	62.823020	12
88	Column_93	59.304261	7
89	Column_40	59.196060	10
90	Column_57	56.855900	7
91	Column_25	55.490171	10
92	Column_34	55.162021	5
93	Column_67	54.072680	10
94	Column_45	53.760900	8
95	Column_75	53.547240	5
96	Column_69	52.827990	12
97	Column_119	52.731190	7
98	Column_113	52.103120	9
99	Column_110	43.336620	7
100	Column_101	40.393650	5
101	Column_84	40.188040	8
102	Column_38	39.378199	2
103	Column_77	32.972811	6
104	Column_1	29.837270	3
105	Column_78	27.164711	4
106	Column_94	26.704240	5
107	Column_103	25.130560	3
108	Column_8	24.854520	2
109	Column_9	22.847670	3
110	Column_90	20.716510	4
111	Column_68	19.123650	4
112	Column_37	18.903010	3
113	Column_61	15.429110	2
114	Column_27	8.969240	1
115	Column_41	7.181500	2
116	Column_66	6.699100	1
117	Column_51	5.437930	1
118	Column_3	5.431610	1
119	Column_32	4.664720	1
120	Column_4	3.927180	1
121	Column_22	2.533040	1
122	Column_23	0.000000	0
123	Column_26	0.000000	0
124	Column_28	0.000000	0

```
125    Column_31          0.000000                   0
126    Column_99          0.000000                   0
127    Column_33          0.000000                   0
128    Column_98          0.000000                   0
129    Column_43          0.000000                   0
130    Column_56          0.000000                   0
131    Column_91          0.000000                   0
132    Column_95          0.000000                   0
133    Column_96          0.000000                   0
134    Column_97          0.000000                   0
135     Column_0          0.000000                   0
```

```
ind_drop60 = [features[0][76:136], features[1][76:136], features[2]
[76:136], features[3][76:136], features[4][76:136]]
print(ind_drop60)
```

```
[array([ 62.,  35.,  76.,  89.,  30.,  55.,  79.,  81.,  50.,  87.,
20.,
       102.,  93.,  40.,  57.,  25.,  34.,  67.,  45.,  75.,  69.,
119.,
       113., 110., 101.,  84.,  38.,  77.,   1.,  78.,  94., 103.,
8.,
         9.,  90.,  68.,  37.,  61.,  27.,  41.,  66.,  51.,   3.,
32.,
         4.,  22.,  23.,  26.,  28.,  31.,  99.,  33.,  98.,  43.,
56.,
        91.,  95.,  96.,  97.,   0.]), array([102.,  93., 124.,   0.,
120.,  51.,  22.,  55.,   9., 116.,  80.,
        30.,  57.,  82.,  34.,  85.,  94.,  77., 111.,  39.,  37.,
42.,
        36.,  90.,  78.,  67.,  86.,  38.,  35.,  92.,   1., 103.,
91.,
        76.,   6.,  81.,  40.,  68.,   3.,  43.,   4.,  27.,  66.,
61.,
        32.,  23.,  97.,  99.,  95.,   8., 101.,  21.,  33.,  96.,
26.,
        41.,  28.,  56.,  31.,  98.]), array([ 67.,   8.,  36., 102.,
93.,  45.,  70.,  20., 116.,   6.,  79.,
        57.,  81.,  77.,  30.,  53.,  60.,  34.,  35.,  94.,  39.,
25.,
        55., 111.,  51.,  85.,  76.,   9.,  90.,  68.,  89.,  22.,
78.,
        21.,  38.,  91.,   1., 103.,  37.,   4., 101.,  41.,  23.,
56.,
        43.,  61.,   0.,  96.,  33.,  32.,  31.,  95.,  27.,  26.,
66.,
        97.,  98.,  99.,   3.,  28.]), array([ 87.,  30., 120.,  78.,
77.,  84.,  51.,  42.,  92.,  76.,  55.,
        58.,  68., 103.,  60., 111., 102.,  86.,  93.,  57.,  39.,
79.,
```

```
        67.,   0.,  34.,  90.,   2.,  94.,  56., 101.,  35.,  81.,
25.,
        80.,  91.,  32.,   9.,  22.,   4.,  21.,  43.,  41.,  36.,
3.,
        37.,  33.,  27.,  23.,  66.,  61.,  99.,   8.,  95.,  98.,
26.,
        31.,  97.,  96.,  38.,  28.]), array([ 30.,  44.,  40., 120.,
75.,  20.,  55.,  78., 110.,  34.,  89.,
         9.,  84.,  80.,  60.,  25.,  93., 115.,   1., 111.,   0.,
22.,
        50., 103.,  67.,  85.,  94.,  68.,  90.,  61., 101.,  91.,
38.,
        32.,  51.,  56.,  43.,  35.,  81.,  76.,   8.,  33.,  66.,
37.,
        23.,  36.,  21.,  26.,   4.,   3.,  99.,  95.,  27.,  28.,
31.,
        41.,  98.,  96.,  86.,  97.])]

X_train_drop60 = []
X_test_drop60 = []

for i in np.arange(5) :
    X_train_drop60 = np.append(X_train_drop60,
dropcols_fancy(X_train[i], ind_drop60[i]))
    X_test_drop60 = np.append(X_test_drop60, dropcols_fancy(X_test[i],
ind_drop60[i]))

print(X_train_drop60[4].shape)
print(X_train[4].shape)

(722602, 76)
(722602, 136)

gbm1_drop60 = lgb.LGBMRanker()
gbm2_drop60 = lgb.LGBMRanker()
gbm3_drop60 = lgb.LGBMRanker()
gbm4_drop60 = lgb.LGBMRanker()
gbm5_drop60 = lgb.LGBMRanker()

fold1_drop60 = gbm1_drop60.fit(X_train_drop60[0], y1_train,
group=group1_train)
fold2_drop60 = gbm2_drop60.fit(X_train_drop60[1], y2_train,
group=group2_train)
fold3_drop60 = gbm3_drop60.fit(X_train_drop60[2], y3_train,
group=group3_train)
fold4_drop60 = gbm4_drop60.fit(X_train_drop60[3], y4_train,
group=group4_train)
fold5_drop60 = gbm5_drop60.fit(X_train_drop60[4], y5_train,
group=group5_train)
```

```
folds_drop60 = [fold1_drop60, fold2_drop60, fold3_drop60,
fold4_drop60, fold5_drop60]
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.303759 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 16271
[LightGBM] [Info] Number of data points in the train set: 723412,
number of used features: 76
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead
of testing was 1.091896 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 16780
[LightGBM] [Info] Number of data points in the train set: 716683,
number of used features: 76
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.177854 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 17029
[LightGBM] [Info] Number of data points in the train set: 719111,
number of used features: 76
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead
of testing was 0.830182 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 16794
[LightGBM] [Info] Number of data points in the train set: 718768,
number of used features: 76
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.282994 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 16338
[LightGBM] [Info] Number of data points in the train set: 722602,
number of used features: 76
```

```
ndcg3_drop60 = []
ndcg5_drop60 = []
ndcg10_drop60 = []

for i in np.arange(5) :
    n3_d60 = compute_ndcg_all(folds_drop60[i], X_test_drop60[i],
y_test[i], qid_test[i], k=3)
    n5_d60 = compute_ndcg_all(folds_drop60[i], X_test_drop60[i],
y_test[i], qid_test[i], k=5)
    n10_d60 = compute_ndcg_all(folds_drop60[i], X_test_drop60[i],
y_test[i], qid_test[i], k=10)
    ndcg3_drop60 = np.append(ndcg3_drop60, n3_d60)
```

```
    ndcg5_drop60 = np.append(ndcg5_drop60, n5_d60)
    ndcg10_drop60 = np.append(ndcg10_drop60, n10_d60)

col_names = ["Fold 1", "Fold 2", "Fold 3", "Fold 4", "Fold 5"]

data = [["nDCG@3", ndcg3_drop60[0], ndcg3_drop60[1], ndcg3_drop60[2],
ndcg3_drop60[3], ndcg3_drop60[4]],
        ["nDCG@5", ndcg5_drop60[0], ndcg5_drop60[1], ndcg5_drop60[2],
ndcg5_drop60[3], ndcg5_drop60[4]],
        ["nDCG@10", ndcg10_drop60[0], ndcg10_drop60[1],
ndcg10_drop60[2], ndcg10_drop60[3], ndcg10_drop60[4]]]

print(tabulate(data, headers=col_names))
```

```
           Fold 1    Fold 2    Fold 3    Fold 4    Fold 5
-------  --------  --------  --------  --------  --------
nDCG@3   0.454253  0.45729   0.44979   0.460635  0.470186
nDCG@5   0.462657  0.460267  0.45864   0.46734   0.473352
nDCG@10  0.481971  0.477253  0.477436  0.488881  0.490817
```

```
importance_df1_drop60 = (
    pd.DataFrame({
        'feature_name': fold1_drop60.feature_name_,
        'importance_gain': get_feature_importance(fold1_drop60,
importance_type='gain'),
        'importance_split': get_feature_importance(fold1_drop60,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df1_drop60.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_73     23856.270343                90
1    Column_27      4358.591591                33
2    Column_50      4186.807418               124
3     Column_3      4161.406613                24
4    Column_69      3666.158772               150
```

```
importance_df2_drop60 = (
    pd.DataFrame({
        'feature_name': fold2_drop60.feature_name_,
        'importance_gain': get_feature_importance(fold2_drop60,
importance_type='gain'),
        'importance_split': get_feature_importance(fold2_drop60,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
```

```
)
print(importance_df2_drop60.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_73     23712.889327                82
1     Column_2      5248.989300                19
2    Column_51      4299.243018               110
3    Column_26      4290.079625                42
4    Column_69      4153.481874               157
```

```
importance_df3_drop60 = (
    pd.DataFrame({
        'feature_name': fold3_drop60.feature_name_,
        'importance_gain': get_feature_importance(fold3_drop60,
importance_type='gain'),
        'importance_split': get_feature_importance(fold3_drop60,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df3_drop60.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_73     23353.006551                75
1    Column_24      4878.790909                24
2    Column_49      4219.279321                91
3    Column_69      3950.991465               169
4     Column_2      3709.648280                17
```

```
importance_df4_drop60 = (
    pd.DataFrame({
        'feature_name': fold4_drop60.feature_name_,
        'importance_gain': get_feature_importance(fold4_drop60,
importance_type='gain'),
        'importance_split': get_feature_importance(fold4_drop60,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df4_drop60.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_73     23852.530037                78
1     Column_3      4690.577678                14
2    Column_27      4277.049753                24
3    Column_49      3377.314892                99
4    Column_68      3337.080847               124
```

```
importance_df5_drop60 = (
    pd.DataFrame({
        'feature_name': fold5_drop60.feature_name_,
        'importance_gain': get_feature_importance(fold5_drop60,
importance_type='gain'),
        'importance_split': get_feature_importance(fold5_drop60,
importance_type='split'),
    })
    .sort_values('importance_gain', ascending=False)
    .reset_index(drop=True)
)
print(importance_df5_drop60.head())
```

```
  feature_name  importance_gain  importance_split
0    Column_73     23388.157174                88
1     Column_3      4828.335942                13
2    Column_25      4173.481593                38
3    Column_51      3655.949524                99
4    Column_69      3321.905646               151
```