Answer 1A

As seen from the cell output below,

Sparsity = 0.017

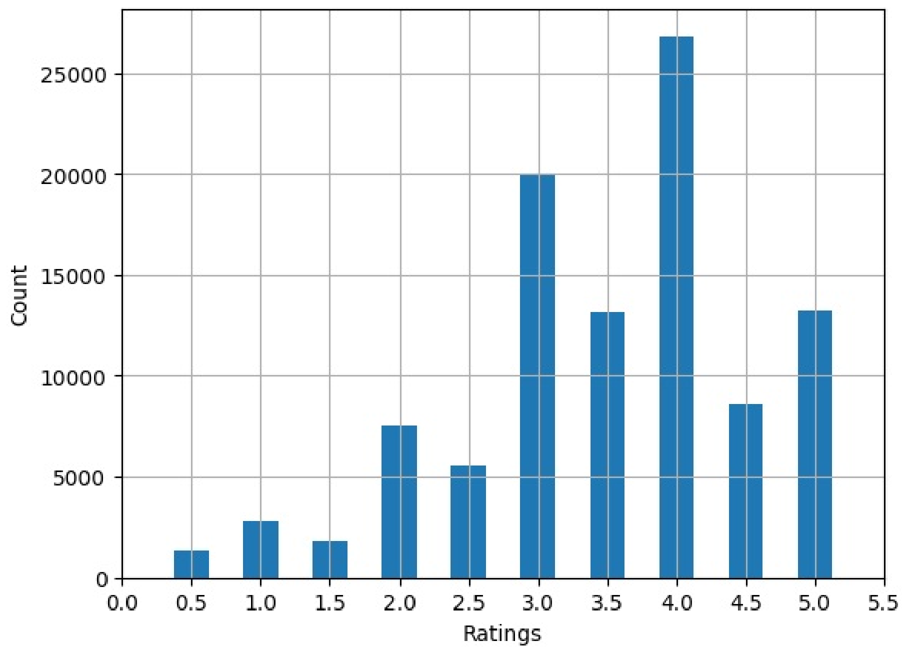
In [59]:
```
dataset_path = 'Synthetic_Movie_Lens/'
data = pd.read_csv(dataset_path+"ratings.csv",usecols=['userId','movieId','rating'])
user_ID = data['userId'].values
movie_ID = data['movieId'].values
rating = data['rating'].values
sparsity = len(rating)/(len(set(movie_ID))*len(set(user_ID)))
print('Sparsity:',sparsity)
```

```
Sparsity: 0.016999683055613623
```

Answer 1B

Histogram output is as follows below:

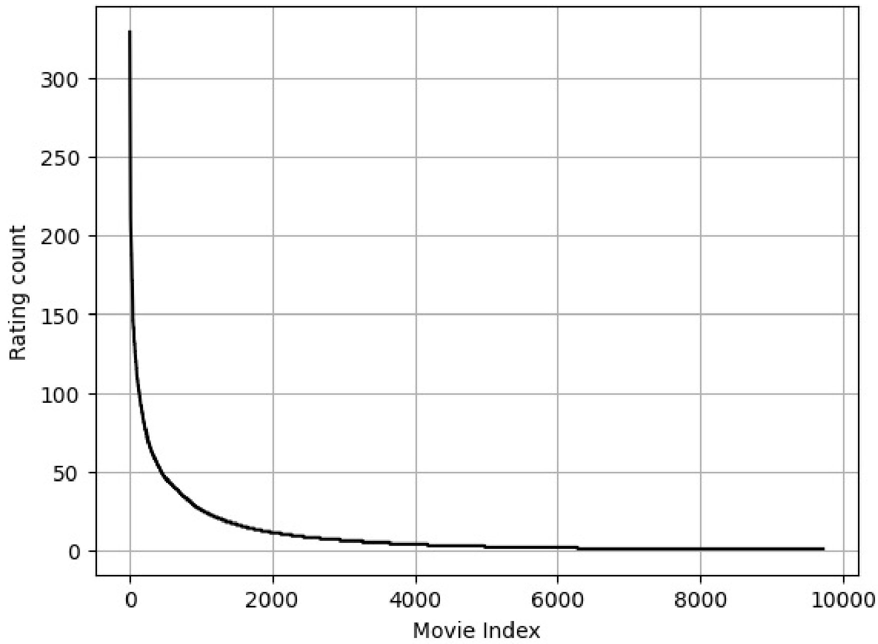
In [ ]:
```
u, inv = np.unique(rating, return_inverse=True)
plt.bar(u, np.bincount(inv), width=0.25)
locs, labels = plt.xticks()
plt.grid(linestyle='-')
plt.xticks(np.arange(0,6,0.5),rotation=0)
plt.ylabel('Count')
plt.xlabel('Ratings')
plt.show()
```



Answer 1C

Distribution of the number of ratings received among movies as in the cell output below:

In [ ]:
```
unique, counts = np.unique(movie_ID, return_counts=True)
plt.plot(range(1,len(unique)+1),counts[np.argsort(counts)[::-1]],color='black')
plt.grid(linestyle='-')
plt.ylabel('Rating count')
plt.xlabel('Movie Index')
plt.show()
```



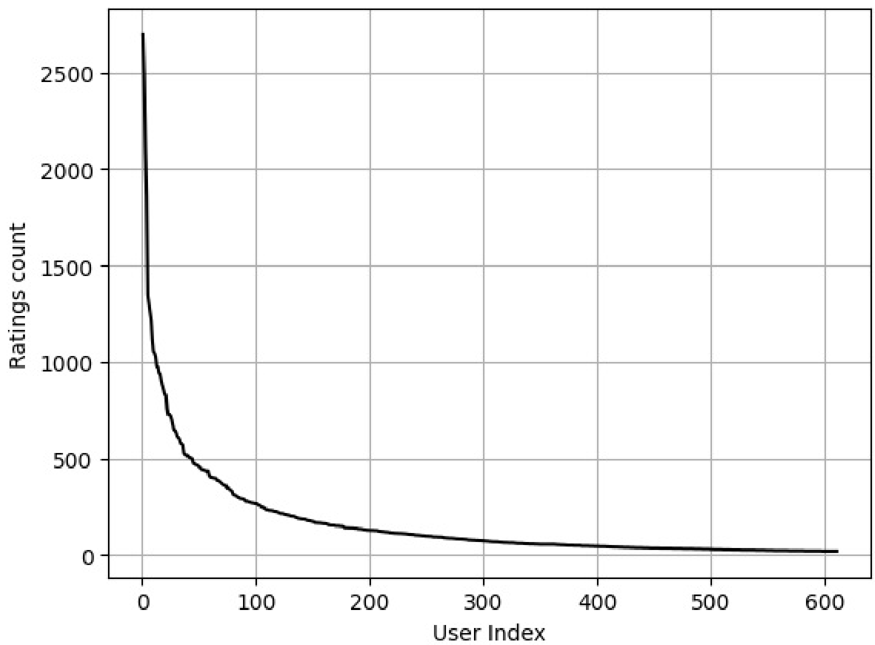
In [ ]:
```
movie_count_dict = {}
x = list(range(1,len(unique)+1))
for key in unique[np.argsort(counts)[::-1]]:
    for value in x:
        movie_count_dict[key] = value
        x.remove(value)
        break
print('Top 10 rated movies (Movie ID, Index):')
print(list(movie_count_dict.items())[0:10])
```

```
Top 10 rated movies (Movie ID, Index):
[(356, 1), (318, 2), (296, 3), (593, 4), (2571, 5), (260, 6), (480, 7), (110, 8), (589, 9), (527, 10)]
```

Answer 1D

The distribution of ratings among users is as in the cell output below:

In [ ]:
```
unique, counts = np.unique(user_ID, return_counts=True)
plt.plot(range(1,len(unique)+1),counts[np.argsort(counts)[::-1]],linestyle='-',color='black')
plt.grid(linestyle='-')
plt.ylabel('Ratings count')
plt.xlabel('User Index')
plt.show()
```



In [ ]:
```
user_count_dict = {}
x = list(range(1,len(unique)+1))
for key in unique[np.argsort(counts)[::-1]]:
    for value in x:
        user_count_dict[key] = value
```

```
        x.remove(value)
        break
print('Top 10 users who rated most number of times (User ID, Index):')
print(list(user_count_dict.items())[0:10])
```

```
Top 10 users who rated most number of times (User ID, Index):
[(414, 1), (599, 2), (474, 3), (448, 4), (274, 5), (610, 6), (68, 7), (380, 8), (606, 9), (288, 10)]
```

Answer 1E

Observations: The output cell from Answer 1C shows a consistent decrease: indicating that around 500 out of 9742 movies have garnered over 50 unique user ratings. This phenomenon elucidates the sparse nature of the ratings matrix wherein only a handful of movies have accumulated several distinct ratings.

The output cell from Answer 1D is also similar to 1C where we see that the curve is monotonically decreasing, indicating less than 50 users out of 610 providing ratings to 500 movies or more out of 9742. This again explains why the ratings matrix is sparse with a vast number of users who do not provide many number of unique ratings.

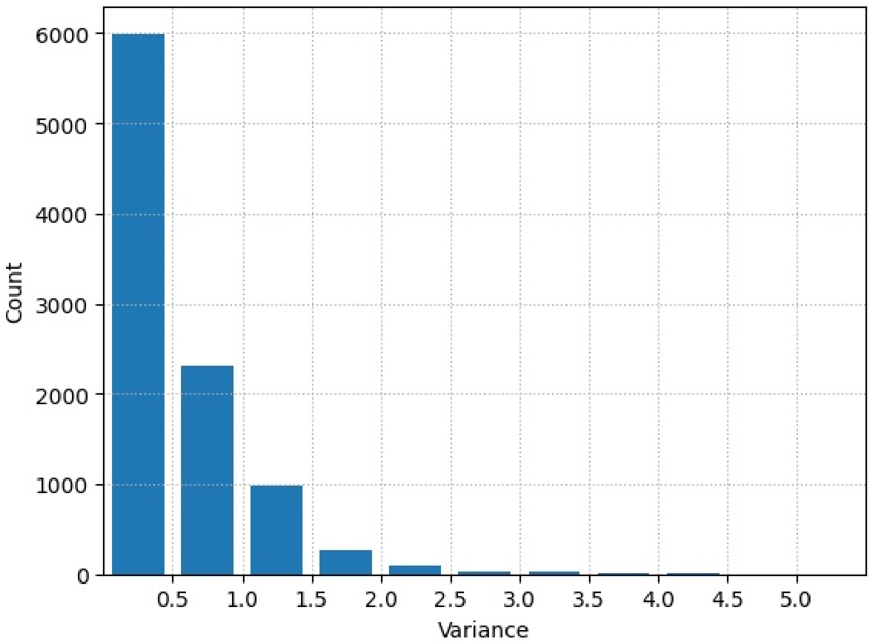
Implications: Since most of the elements in the sparse representation are not defined or 0, these elements contribute little to no information for the model being trained on the representation resulting in a model with a large number of parameters that perform poorly on those movies with a low number of ratings due to lack of sufficient ratings and overfitting on those movies with a higher number of user ratings. One could attempt to address the above issue with regularization to encourage generalization and prevent formation of ill-conditioned classifiers ,leading to a simpler model with lower number of weights.

Answer 1F

Variance histogram as below in the cell output :

In [ ]:
```
unique_movie_ID = list(set(movie_ID))
movie_ID_list = []
var_list = []
for j in range(len(unique_movie_ID)):
    indices = [i for i, x in enumerate(movie_ID) if x == unique_movie_ID[j]]
    var = np.var(np.array(rating[indices]))
    movie_ID_list.append(unique_movie_ID[j])
    var_list.append(var)
```

In [ ]:
```
plt.hist(var_list, bins=np.arange(0,5.5,0.5),rwidth=0.75)
plt.xticks(np.arange(0.5,5.5,0.5))
plt.xlim([0, 5.5])
plt.grid(linestyle=':')
plt.xlabel('Variance')
plt.ylabel('Count')
plt.show()
```



Answer 2A

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

Answer 2B

$I_u \cap I_v$ indicates the set of movies where the ratings are commonly rated by both users u and v. Many $I_u \cap I_v$ are expected to be ϕ because there will be movies rated by user 'u' but not by user 'v' and vice-versa.

Answer 3

Normalizing the raw ratings by centering them around the mean ratings of users serves to mitigate user-specific biases and characteristics in the ratings, along with reducing the impact of extreme ratings from certain users. Some users may consistently provide high or low ratings, while others may distribute their ratings across the entire spectrum. Mean centering is effective in eliminating these tendencies and outliers, making the data less noisy. This process is particularly beneficial when trying to uncover the interaction between user ratings in the prediction function, as it helps alleviate multicollinearity among predictor variables, thereby facilitating a clearer understanding of the significance of individual user ratings.

k-NN collaborative filter

In [4]:

```
# Defining the reader for surprise library
reader = Reader()
# Load the data into Surprise Dataset
ratings_dataset = Dataset.load_from_df(data[['userId', 'movieId', 'rating']], reader)
```

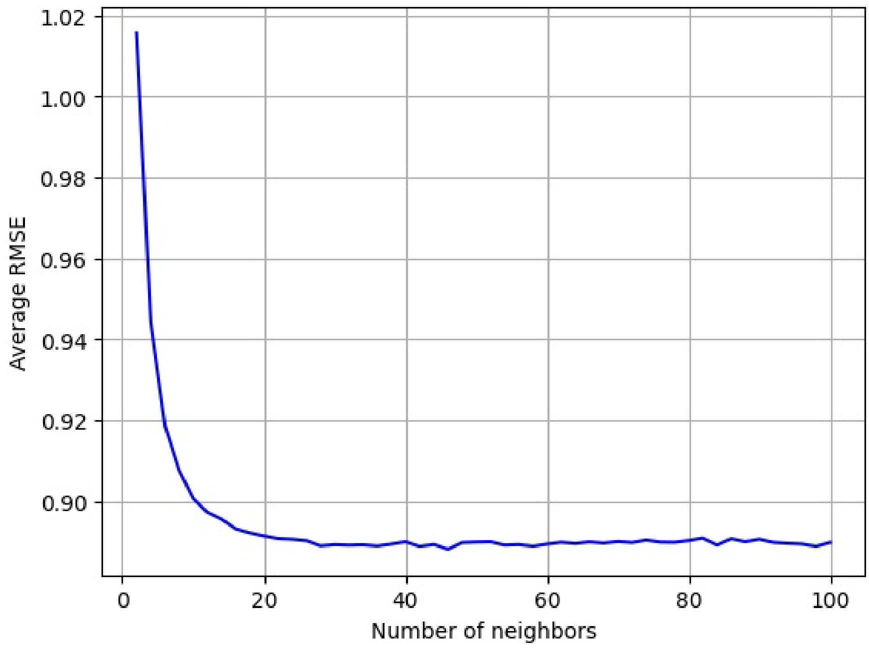
In [ ]:

```
k = np.arange(2,102,2)
rmse = []
mae = []
for item in k:
    print('Testing for k =',item)
    res = cross_validate(KNNWithMeans(k=item,sim_options={'name':'pearson'}),
                         measures=['rmse','mae'],data = ratings_dataset,cv=10,n_jobs=-1)
    rmse.append(np.mean(res['test_rmse']))
    mae.append(np.mean(res['test_mae']))
```

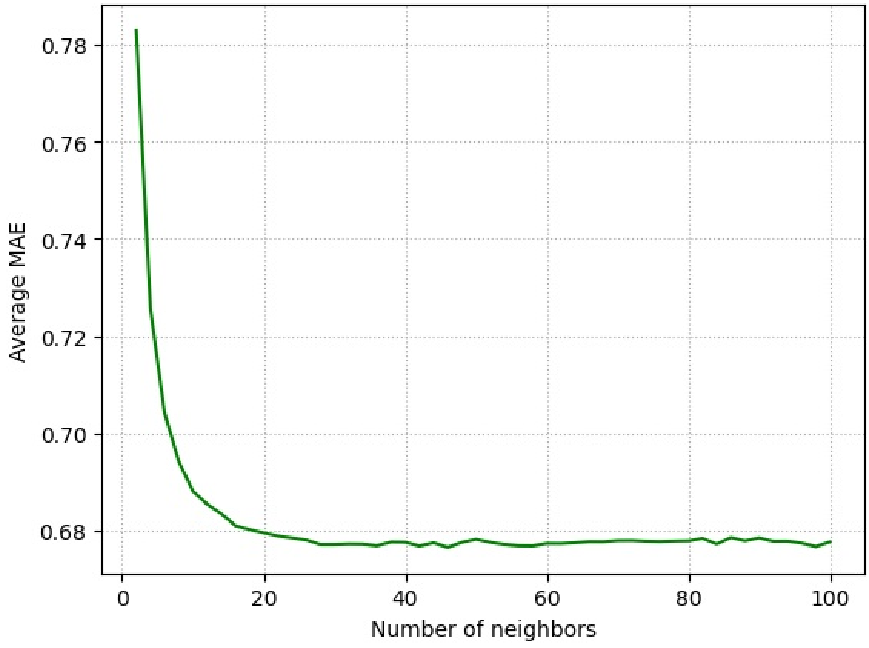
```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
Testing for k = 52
Testing for k = 54
Testing for k = 56
Testing for k = 58
Testing for k = 60
Testing for k = 62
Testing for k = 64
Testing for k = 66
Testing for k = 68
Testing for k = 70
Testing for k = 72
Testing for k = 74
Testing for k = 76
Testing for k = 78
Testing for k = 80
Testing for k = 82
Testing for k = 84
Testing for k = 86
Testing for k = 88
Testing for k = 90
Testing for k = 92
Testing for k = 94
Testing for k = 96
Testing for k = 98
Testing for k = 100
```

In [ ]:

```
plt.plot(k,rmse,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.ylabel('Average RMSE')
plt.xlabel('Number of neighbors')
plt.show()
```



In [ ]:
```
plt.plot(k,mae,linestyle='-',color='g')
plt.grid(linestyle=':')
plt.ylabel('Average MAE')
plt.xlabel('Number of neighbors')
plt.show()
```



Answer 4

The Average RMSE and MAE plots are as shown above in the cell outputs.

Answer 5

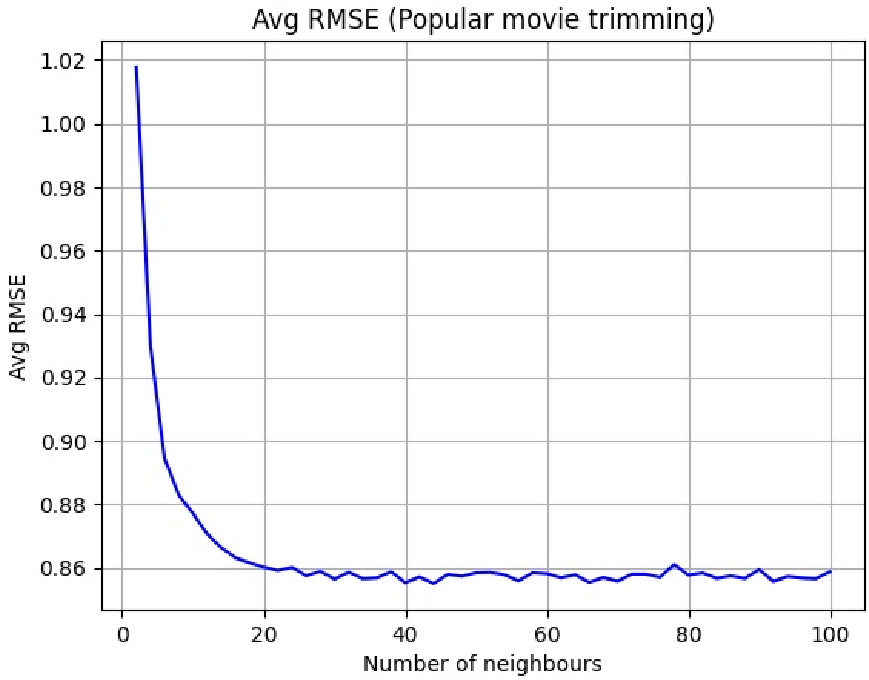
We need to figure out the smallest k value where the errors for user-based collaborative filtering stop changing. Looking at Figure 5, we can see that this happens at k = 20, and at that point, the RMSE stays slightly below 0.9, and the MAE stays near 0.68.

Popular Movie Trimming

In [ ]:
```
rmse_pop = []
kf =  KFold(n_splits=10)
for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trim_list = []
        unique, counts = np.unique([row[1] for row in testset], return_counts=True)
        for i in range(len(counts)):
            if(counts[i]<=2):
                trim_list.append(unique[i])
        trimmed_set = [j for j in testset if j[1] not in trim_list]
        res = KNNWithMeans(k=item,sim_options={'name':'pearson'},verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_pop.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
```

In [ ]:
```
plt.plot(k,rmse_pop,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.title('Avg RMSE (Popular movie trimming)')
plt.ylabel('Avg RMSE')
plt.xlabel('Number of neighbours')
plt.show()
```



In [ ]:
```
print("Minimum average RMSE (Popular movie trimming) = ", min(rmse_pop))
```

```
Minimum average RMSE (Popular movie trimming) =  0.8550024964524139
```

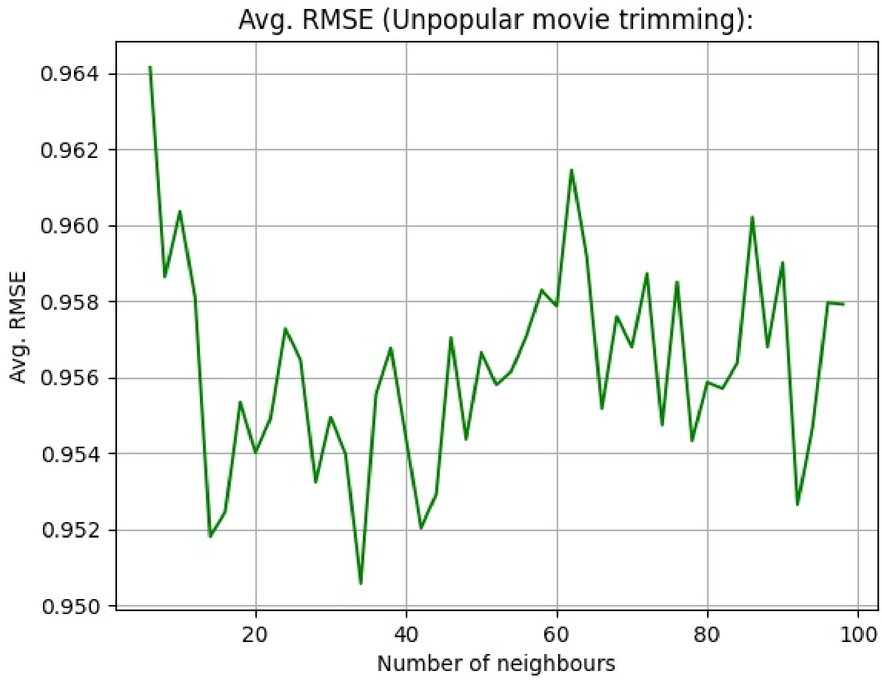
Unpopular

In [ ]:
```
rmse_unpop = []
kf =  KFold(n_splits=10)
for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trim_list = []
        unique, counts = np.unique([row[1] for row in testset], return_counts=True)
        for i in range(len(counts)):
            if(counts[i]>2):
                trim_list.append(unique[i])
        trimmed_set = [j for j in testset if j[1] not in trim_list]
        res = KNNWithMeans(k=item,sim_options={'name':'pearson'},verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_unpop.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
Testing for k = 52
Testing for k = 54
Testing for k = 56
Testing for k = 58
Testing for k = 60
Testing for k = 62
Testing for k = 64
Testing for k = 66
Testing for k = 68
Testing for k = 70
Testing for k = 72
Testing for k = 74
Testing for k = 76
Testing for k = 78
Testing for k = 80
Testing for k = 82
Testing for k = 84
Testing for k = 86
Testing for k = 88
Testing for k = 90
Testing for k = 92
Testing for k = 94
Testing for k = 96
Testing for k = 98
Testing for k = 100
```

In [ ]:

```
plt.plot(k[2:-1],rmse_unpop[2:-1],linestyle='-',color='g')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (Unpopular movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of neighbours')
plt.show()
```



In [ ]:
```
print("Minimum avg. RMSE (Unpopular movie trimming):", min(rmse_unpop))
```

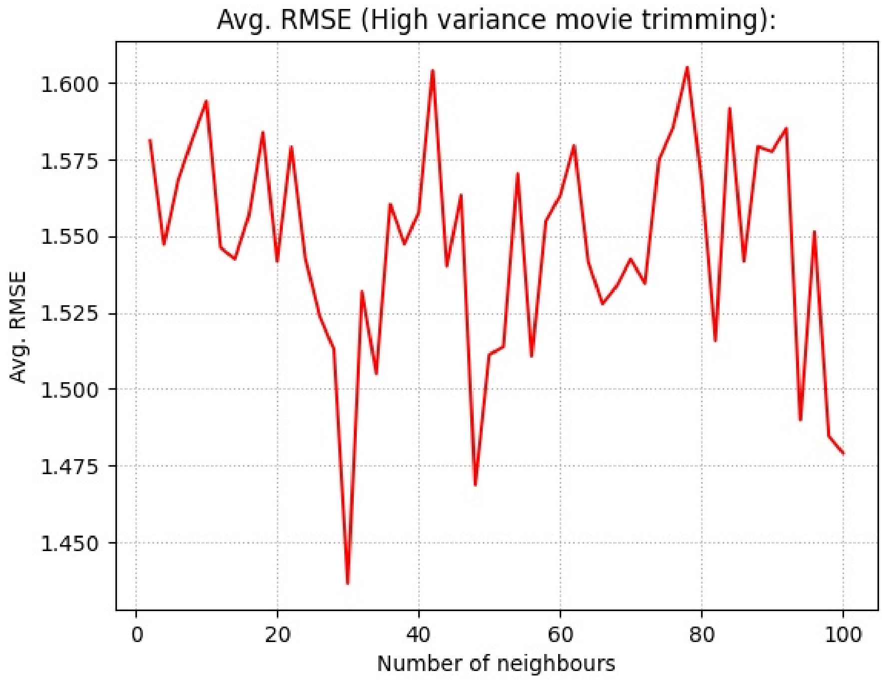
```
Minimum avg. RMSE (Unpopular movie trimming): 0.950572362148894
```

In [ ]:
```
rmse_var = []
kf =  KFold(n_splits=10)
dict_of_items = {}
for j in ratings_dataset.raw_ratings:
    if j[1] in dict_of_items.keys():
        dict_of_items[j[1]].append(j[2])
    else:
        dict_of_items[j[1]] = []
        dict_of_items[j[1]].append(j[2])

for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trimmed_set = [j for j in testset if (np.var(dict_of_items[j[1]]) >= 2 and len(dict_of_items[j[1]]) >= 
        res = KNNWithMeans(k=item,sim_options={'name':'pearson'},verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_var.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
Testing for k = 52
Testing for k = 54
Testing for k = 56
Testing for k = 58
Testing for k = 60
Testing for k = 62
Testing for k = 64
Testing for k = 66
Testing for k = 68
Testing for k = 70
Testing for k = 72
Testing for k = 74
Testing for k = 76
Testing for k = 78
Testing for k = 80
Testing for k = 82
Testing for k = 84
Testing for k = 86
Testing for k = 88
Testing for k = 90
Testing for k = 92
Testing for k = 94
Testing for k = 96
Testing for k = 98
Testing for k = 100
```

In [ ]:

```
plt.plot(k,rmse_var,linestyle='-',color='r')
plt.grid(linestyle=':')
plt.title('Avg. RMSE (High variance movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of neighbours')
plt.show()
```



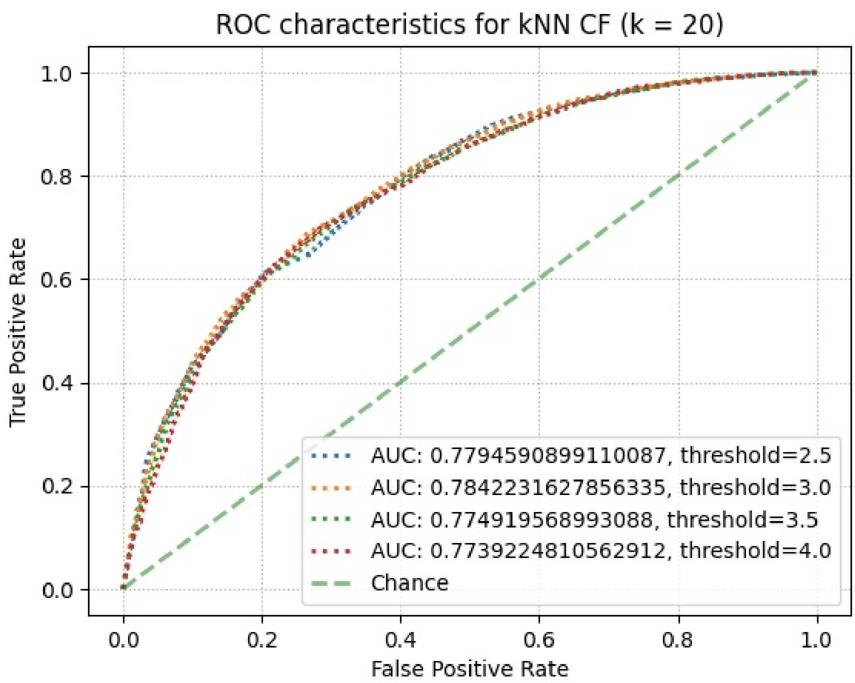
In [ ]:
```
print("Minimum avg. RMSE (High variance movie trimming):", min(rmse_var))
```

```
Minimum avg. RMSE (High variance movie trimming): 1.4366413310861565
```

ROC k-NN CF

In [ ]:
```
k = 20
thres = [2.5, 3.0, 3.5, 4.0]
trainset, testset = train_test_split(ratings_dataset, test_size=0.1)
res  = KNNWithMeans(k=k,sim_options={'name':'pearson'},verbose=False).fit(trainset).test(testset)
```

In [ ]:
```
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 characteristics for kNN CF (k = 20)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



Answer 6

• Average RMSE plots are plotted as above. Minimum avergae RMSE for each trimming type is as below:

Trimming Type	Minimum Avg. RMSE
Popular Movie Trimming	0.8550
Unpopular Movie Trimming	0.9506
High Variance Movie Trimming	1.4366

• ROC curves along with AUC values are as plotted above condensed in one plot as in cell output .

Answer 7

The optimization problem is not jointly convex for the user latent space (U) and item embedding space (V) due to the presence of numerous local minima in the gradient plane of the objective function. This arises from the fact that the matrix factorization model predicts ratings by multiplying U and V, and this approach lacks convexity properties since the objective function is permutation and rotation invariant. Given, R is the ratings matrix, the optimization problem can be solved using ALS by keeping U fixed and solving for V and vice-versa for next step. Objective formulation without regularization:

$$\min_V \sum i = 1^n \sum_{j=1}^{n} W_{ij}\left(r_{ij} - (UV^T)_{ij}\right)^2$$

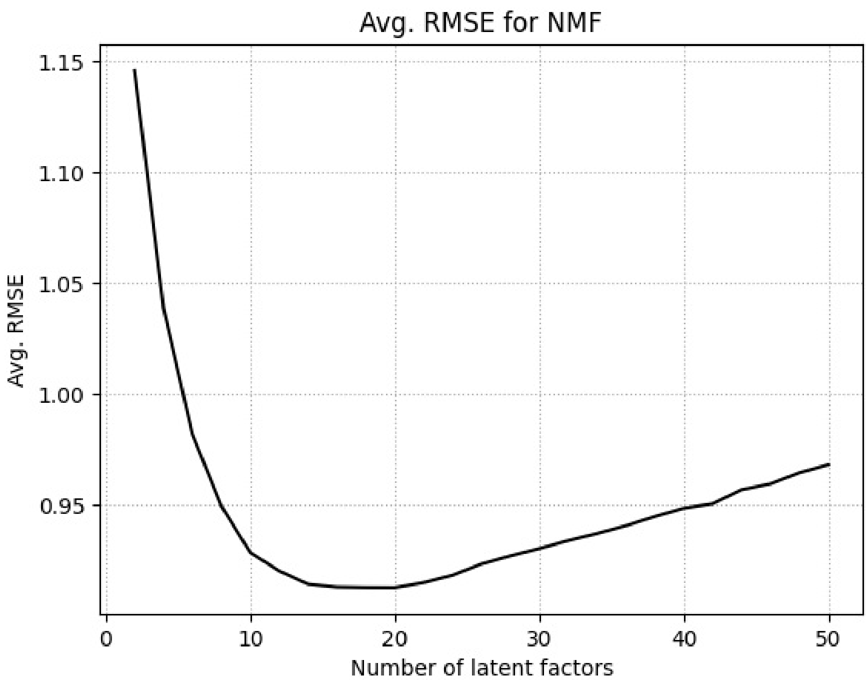
$$V = (UU^T)^{-1} UR$$

NMF

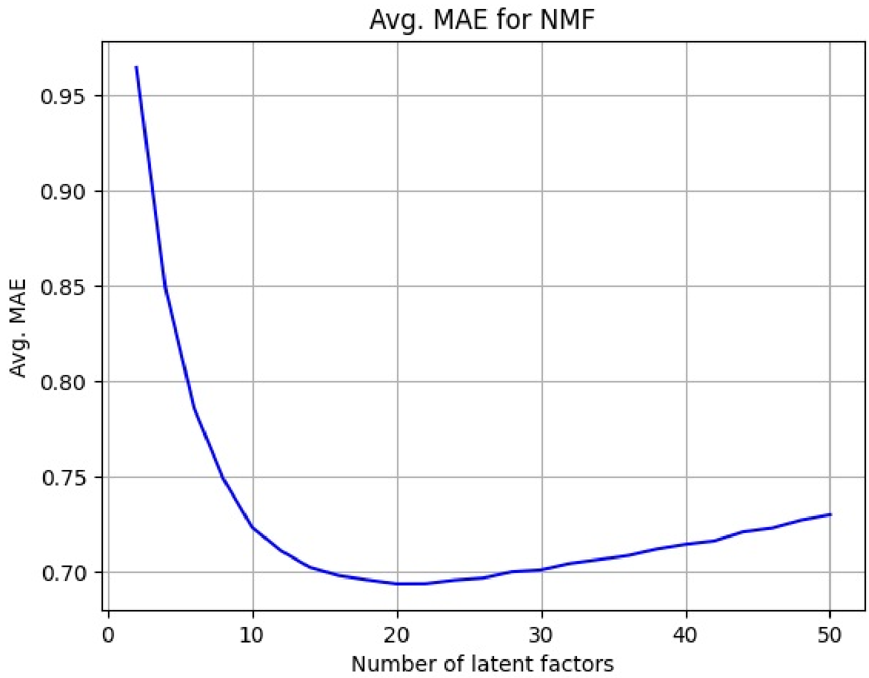
```
In [5]: k = np.arange(2,52,2)
rmse_NMF = []
mae_NMF = []
for item in k:
    print('Testing for k =',item)
    res = cross_validate(NMF(n_factors=item,n_epochs=50,verbose=False),
                         measures=['rmse','mae'],data = ratings_dataset,cv=10,n_jobs=-1)
    rmse_NMF.append(np.mean(res['test_rmse']))
    mae_NMF.append(np.mean(res['test_mae']))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [6]:
```
plt.plot(k,rmse_NMF,linestyle='-',color='black')
plt.grid(linestyle=':')
plt.title('Avg. RMSE for NMF')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [7]:
```
plt.plot(k,mae_NMF,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.title('Avg. MAE for NMF')
plt.ylabel('Avg. MAE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [8]:
```
print("Minimum avg. RMSE (NMF): %f, value of k: %d" % (min(rmse_NMF),k[[i for i, x in enumerate(rmse_NMF) if x =
print("Minimum avg. MAE (NMF): %f, value of k: %d" % (min(mae_NMF),k[[i for i, x in enumerate(mae_NMF) if x == m
```

```
Minimum avg. RMSE (NMF): 0.912769, value of k: 20
Minimum avg. MAE (NMF): 0.693552, value of k: 20
```

Answer 8A

The Average RMSE and MAE plots are as shown above.

Answer 8B

Metric	Minimum Avg. Value	Value of k
RMSE (NMF)	0.9128	20
MAE (NMF)	0.6936	20

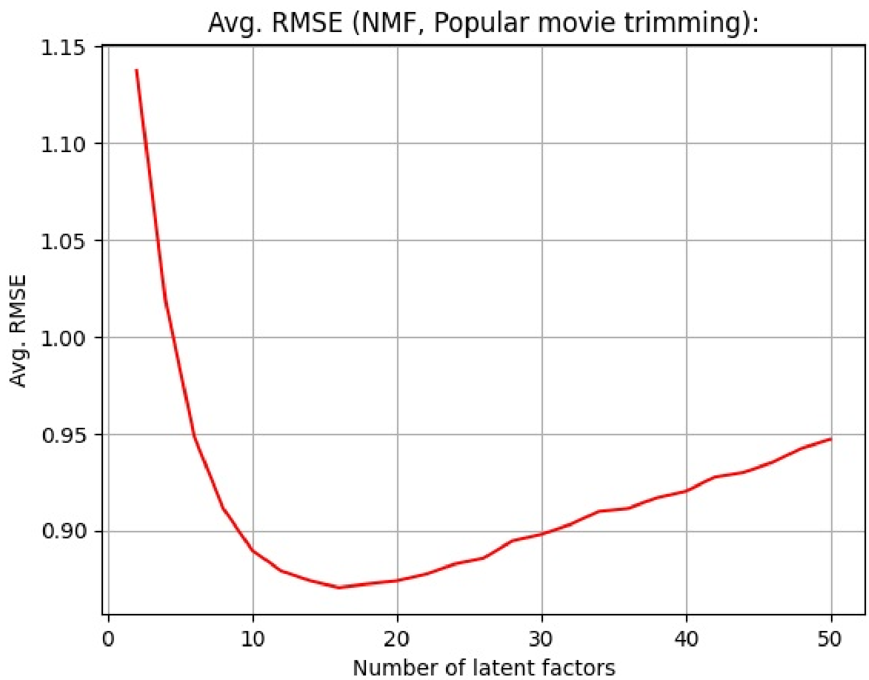
We see that in both RMSE and MAE, the value of K that yields minimum average vaue is $k = 20$. There are 19 genres in the MovieLens dataset, which is very close to the obtained optimal value of k.

Popular NMF

In [9]:
```
rmse_NMF_pop = []
kf =  KFold(n_splits=10)
for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trim_list = []
        unique, counts = np.unique([row[1] for row in testset], return_counts=True)
        for i in range(len(counts)):
            if(counts[i]<=2):
                trim_list.append(unique[i])
        trimmed_set = [j for j in testset if j[1] not in trim_list]
        res = NMF(n_factors=item,n_epochs=50,verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_NMF_pop.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [10]:
```
plt.plot(k,rmse_NMF_pop,linestyle='-',color='r')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (NMF, Popular movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [11]:
```
print("Minimum avg. RMSE (NMF, Popular movie trimming):", min(rmse_NMF_pop))
```

```
Minimum avg. RMSE (NMF, Popular movie trimming): 0.8704414533436309
```

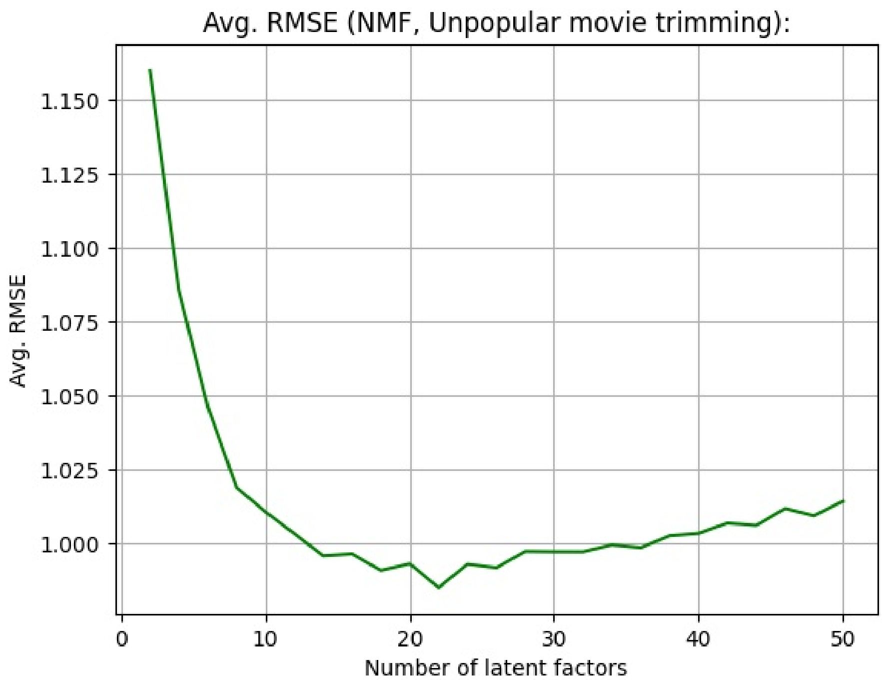
Unpopular NMF

In [12]:
```
rmse_NMF_unpop = []
kf =  KFold(n_splits=10)
for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trim_list = []
        unique, counts = np.unique([row[1] for row in testset], return_counts=True)
        for i in range(len(counts)):
            if(counts[i]>2):
                trim_list.append(unique[i])
        trimmed_set = [j for j in testset if j[1] not in trim_list]
        res = NMF(n_factors=item,n_epochs=50,verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_NMF_unpop.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [13]:

```
plt.plot(k,rmse_NMF_unpop,linestyle='-',color='g')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (NMF, Unpopular movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [14]:

```
print("Minimum avg. RMSE (NMF, Unpopular movie trimming):", min(rmse_NMF_unpop))
```

```
Minimum avg. RMSE (NMF, Unpopular movie trimming): 0.9848487484434664
```

High variance trim NMF

In [15]:

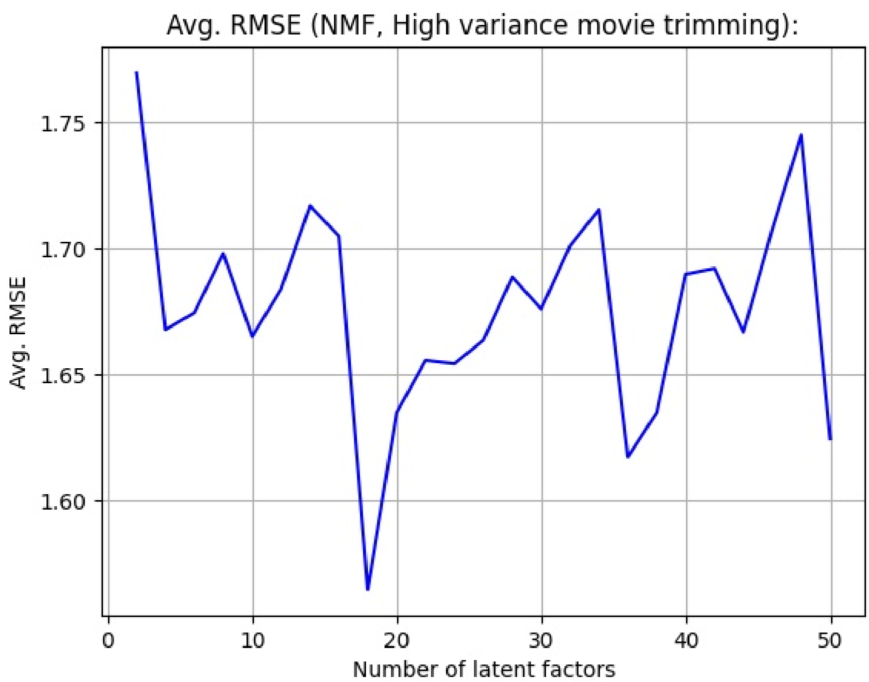
```
rmse_NMF_var = []
kf =  KFold(n_splits=10)
dict_of_items = {}
for j in ratings_dataset.raw_ratings:
    if j[1] in dict_of_items.keys():
        dict_of_items[j[1]].append(j[2])
    else:
        dict_of_items[j[1]] = []
        dict_of_items[j[1]].append(j[2])

for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trimmed_set = [j for j in testset if (np.var(dict_of_items[j[1]]) >= 2 and len(dict_of_items[j[1]]) >=
```

```
        res = NMF(n_factors=item,n_epochs=50,verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_NMF_var.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [16]:

```
plt.plot(k,rmse_NMF_var,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (NMF, High variance movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [17]:

```
print("Minimum avg. RMSE (NMF, High variance movie trimming):", min(rmse_NMF_var))
```

```
Minimum avg. RMSE (NMF, High variance movie trimming): 1.5648267743907396
```

ROC NMF

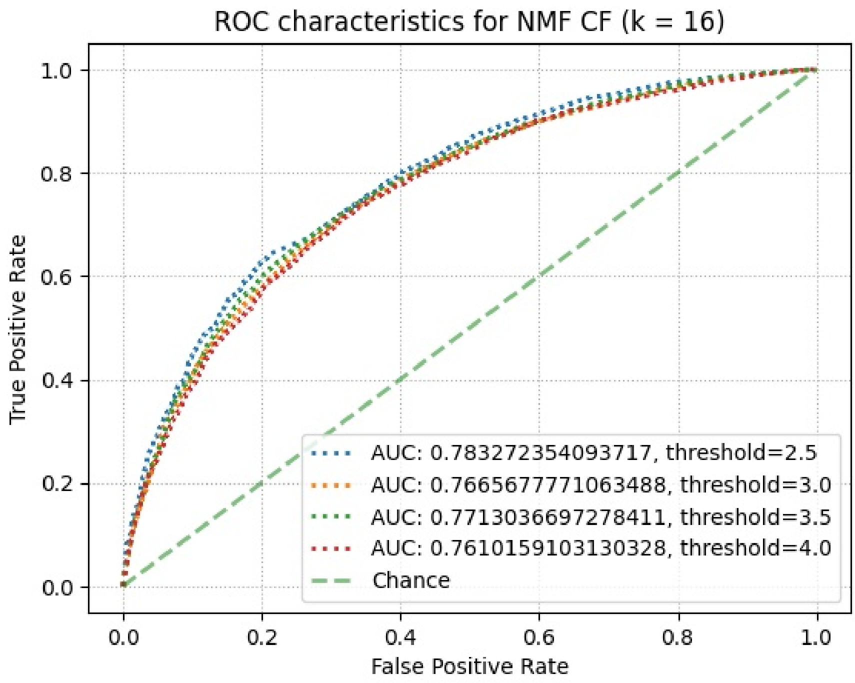
In [18]:

```
k = k[[i for i, x in enumerate(rmse_NMF) if x == min(rmse_NMF)][0]]
thres = [2.5, 3.0, 3.5, 4.0]
trainset, testset = train_test_split(ratings_dataset, test_size=0.1)
res  = NMF(n_factors=k,n_epochs=50,verbose=False).fit(trainset).test(testset)
```

In [19]:

```
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 characteristics for NMF CF (k = 16)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



Answer 8C

Popular, unpopular and high variance trimmed average RMSE and MAE plots are as shown above. Min avg RMSE (NMF) are reported below :

Trimming Type	Minimum Avg. RMSE (NMF)
Popular Movie Trimming	0.8704
Unpopular Movie Trimming	0.9848
High Variance Movie Trimming	1.5648

• ROC curve for NMF along with AUC values is as condensed into one plot as shown above

In [22]:
```
genre = pd.read_csv(dataset_path+'movies.csv',usecols=['movieId','title','genres'])
trainset, testset = train_test_split(ratings_dataset, test_size=0.1)
nmf = NMF(n_factors=20,n_epochs=50,verbose=False)
nmf.fit(trainset).test(testset)
U = nmf.pu
V = nmf.qi
```

In [27]:
```
for item in range(20):
    print('Column number of V: ',item)
    selected_col = V[:,item]
    sorted_col = np.argsort(selected_col)[::-1]
    for i in sorted_col[0:10]:
        print(genre['genres'][i])
    print('---------------------------------------------')
```

```
Column number of V:  0
Comedy
Action|Crime|Drama|War
Drama|Fantasy|Musical|Mystery|Sci-Fi
Comedy|Crime|Drama
Animation|Children|Comedy
Children|Comedy|Romance|Sci-Fi
Drama
Drama|Romance
Drama
Crime|Drama|Mystery|Romance|Thriller
---------------------------------------------
```

```
Column number of V:  1
Crime|Drama
Drama|Mystery
Action|Comedy
Comedy|War
Action|Children
Comedy|Romance
Drama
Drama
Comedy
Drama|War
---------------------------------------------
Column number of V:  2
Drama
Comedy|Horror|Thriller
Adventure|Drama|Sci-Fi
Drama
Drama
Animation|Sci-Fi
Drama
Comedy|Sci-Fi
Adventure|Fantasy|IMAX
Drama|Thriller
---------------------------------------------
Column number of V:  3
Action|Adventure|Romance
Adventure|Drama|Horror|Sci-Fi|Thriller
Comedy
Action|Comedy
Drama
Action|Crime
Drama|Mystery|Thriller
Comedy|Musical
Drama|Romance
Drama|Horror|Mystery|Thriller
---------------------------------------------
Column number of V:  4
Action|Sci-Fi
Comedy|Crime|Drama
Comedy|Drama|Romance
Comedy|Drama|Romance
Adventure|Animation|Children|Comedy|Drama|Musical|Romance
Action|Crime|Thriller
Comedy
Comedy|Drama
Action|Sci-Fi
Drama
---------------------------------------------
Column number of V:  5
Animation|Comedy
Drama|Musical|Romance
Adventure
Drama
Drama|Romance
Adventure|Children
Sci-Fi|Thriller
Romance
Adventure|Drama
Crime|Drama|Thriller
---------------------------------------------
Column number of V:  6
Comedy|Sci-Fi
Drama|Musical|Romance
Comedy
Horror|Sci-Fi
Action|Crime|Thriller
Drama
Comedy|Drama|Romance
Comedy
Comedy|Drama|Romance
Crime|Drama
---------------------------------------------
Column number of V:  7
Action|Crime|Drama
Drama
Action|Drama
Adventure|Drama|Fantasy|IMAX
Drama|Musical|Romance
Action|Adventure|Sci-Fi
Comedy|Drama
Comedy
Drama|Mystery
Horror
```

```
---------------------------------------------
Column number of V:  8
Romance|Sci-Fi
Action|Adventure|Crime|Thriller
Drama|Horror|Thriller
Drama|Horror|Thriller
Adventure|Animation|Children|Comedy
Romance|Sci-Fi|Thriller
Comedy|Romance
Children|Comedy|Drama|Fantasy
Action|Adventure|Thriller
Drama
---------------------------------------------
Column number of V:  9
Action|Adventure|Fantasy
Comedy|War
Action|Sci-Fi
Drama|War
Action|Drama|Thriller
Action|Adventure
Action|Horror|Thriller
Adventure|Children
Comedy|Horror
Action|Adventure
---------------------------------------------
Column number of V:  10
Drama
Drama|War
Crime|Thriller
Action|Adventure|Comedy
Comedy|Drama
Comedy|Drama|Romance
Drama|Romance
Action|Crime
Drama|Fantasy|Horror|Thriller
Drama
---------------------------------------------
Column number of V:  11
Drama|War
Comedy
Horror
Drama|Romance
Fantasy|Mystery|Thriller
Comedy|Horror
Comedy|Drama|Romance
Action|Drama
Drama|Mystery
Drama|Romance
---------------------------------------------
Column number of V:  12
Comedy|Drama|Romance
Comedy
Drama
Drama|Thriller
Adventure|Crime|Drama|Thriller
Comedy
Comedy|Crime
Adventure|Animation|Children|Comedy|Fantasy|Sci-Fi|IMAX
Action|Adventure|Western
Drama
---------------------------------------------
Column number of V:  13
Horror|Sci-Fi
Animation|Comedy
Action|Thriller
Action|Adventure|Comedy|Drama|Romance|War
Comedy|Fantasy|Horror|Thriller
Comedy|Drama
Comedy|Drama|Romance
Drama
Drama
Action|Drama|Thriller
---------------------------------------------
Column number of V:  14
Drama|Mystery
Horror
Animation|Sci-Fi
Comedy|Drama
Action|Adventure|Drama|Thriller|Western
Comedy|Horror|Thriller
Drama|Romance
Crime|Drama
Comedy|Romance
```

```
Crime|Drama
---------------------------------------------
Column number of V:  15
Action|Drama|War
Comedy|Romance
Western
Action|Drama
Crime|Drama
Horror
Comedy|Romance
Comedy|Romance
Drama|Thriller
Comedy
---------------------------------------------
Column number of V:  16
Comedy|Drama
Comedy|Crime
Drama
Action|Adventure|Thriller
Drama|Romance|War
Horror
Comedy|Romance
Action|Thriller
Documentary
Drama|Sci-Fi
---------------------------------------------
Column number of V:  17
Crime|Thriller
Comedy|Drama|Romance
Action|Comedy
Horror
Adventure|Romance
Action|Comedy|Crime|Drama
Adventure|Animation|Comedy|Fantasy|Musical|Romance
Comedy|Romance
Comedy
Comedy
---------------------------------------------
Column number of V:  18
Drama|Thriller
Comedy|Drama
Action|Adventure|Drama
Drama|Romance
Comedy|Crime|Drama
Animation|Comedy|Fantasy|Musical
Comedy|Fantasy|Romance
Comedy
Comedy|Horror|Musical
Action|Crime|Drama|Thriller
---------------------------------------------
Column number of V:  19
Comedy
Comedy|Romance
Comedy|Crime|Thriller
Drama|Musical
Drama|War
Action|Drama|Romance|War
Children|Comedy
Comedy|Fantasy|Horror|Musical|Thriller
Comedy|Drama
Comedy|Drama|Romance
---------------------------------------------
```

Answer 9

Looking at the genre list, it's evident that the top 10 movies are concentrated within specific genres. Each latent factor tends to cluster movies from distinct genre groups. For instance, latent factor 19 predominantly includes movies from the comedy and drama genres, latent factor 5 features movies from the drama/romance genres, and latent factor 7 encompasses movies from the action, crime, and horror genres.

MF CF

In [34]:

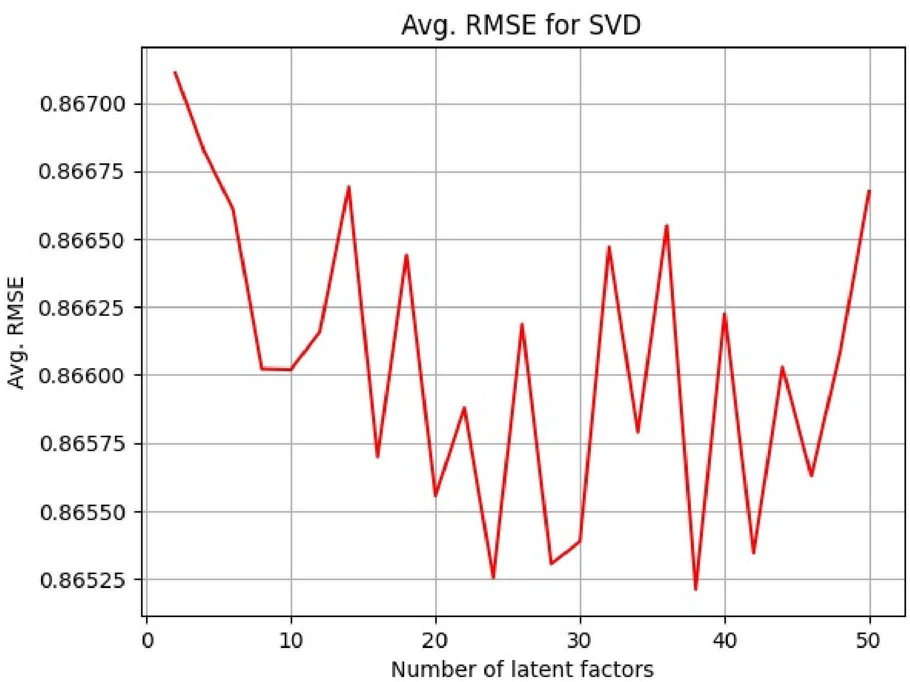
```
k = np.arange(2,52,2)
rmse_SVD = []
mae_SVD = []
for item in k:
    print('Testing for k =',item)
    res = cross_validate(SVD(n_factors=item,n_epochs=20,verbose=False),
```

```python
                          measures=['rmse','mae'],data = ratings_dataset,cv=10,n_jobs=-1)
    rmse_SVD.append(np.mean(res['test_rmse']))
    mae_SVD.append(np.mean(res['test_mae']))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

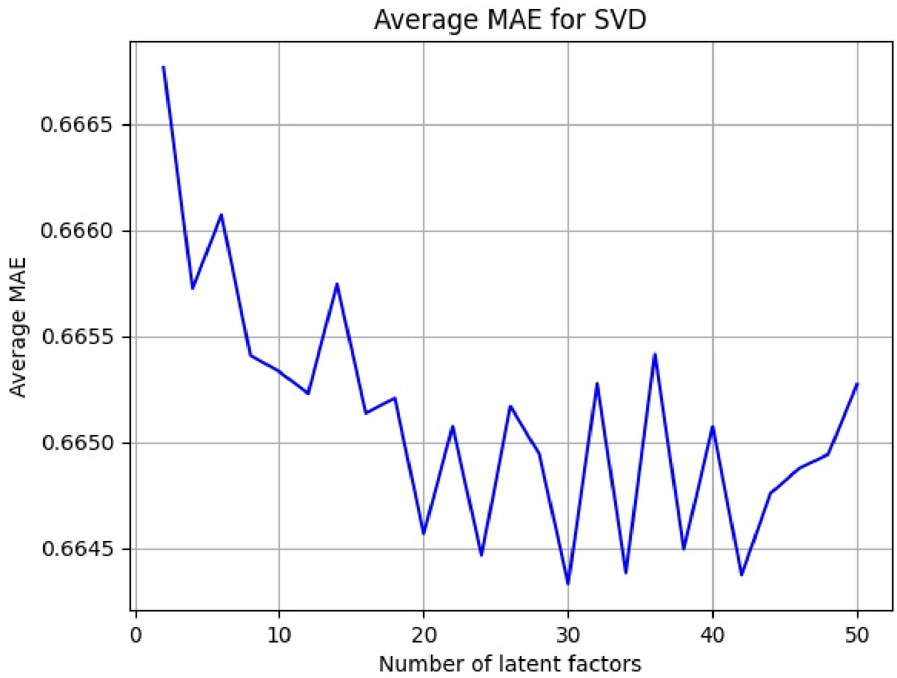
In [35]:

```python
plt.plot(k,rmse_SVD,linestyle='-',color='r')
plt.grid(linestyle='-')
plt.title('Avg. RMSE for SVD')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [36]:

```python
plt.plot(k,mae_SVD,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.title('Average MAE for SVD')
plt.ylabel('Average MAE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [37]:
```
print("Minimum avg. RMSE (SVD): %f, value of k: %d" % (min(rmse_SVD),k[[i for i, x in enumerate(rmse_SVD) if x =
print("Minimum avg. MAE (SVD): %f, value of k: %d" % (min(mae_SVD),k[[i for i, x in enumerate(mae_SVD) if x == m
```

```
Minimum avg. RMSE (SVD): 0.865211, value of k: 38
Minimum avg. MAE (SVD): 0.664331, value of k: 30
```

Answer 10A

The Average RMSE and MAE plots are as shown above.

Answer 10B

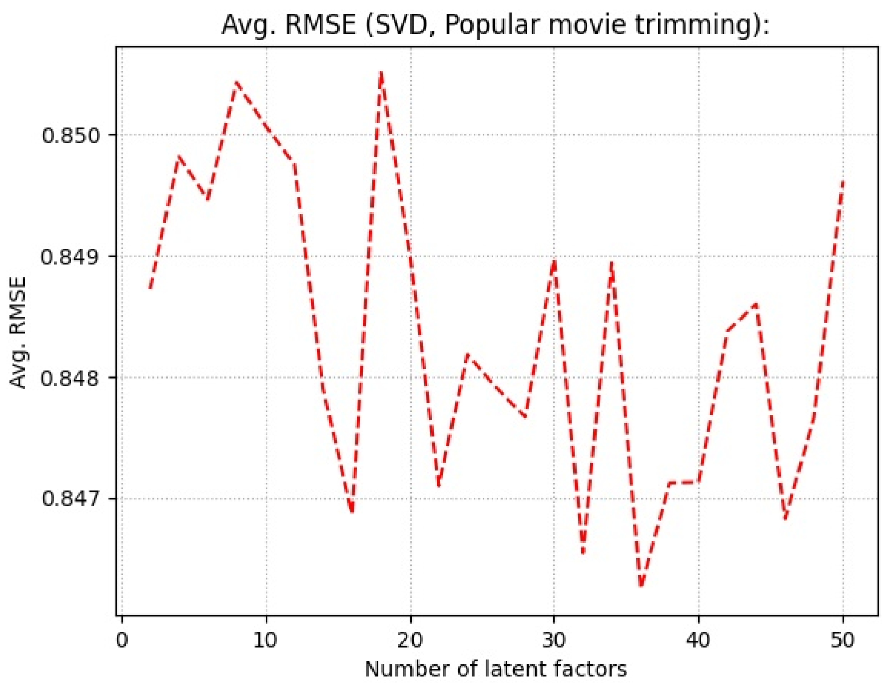
Metric	Minimum Avg. Value	Value of k
RMSE (MF)	0.8652	38
MAE (MF)	0.6643	30

We decided to stick to $k = 30$, as it is closer to the actual number of movie genres (19).

MF Popular trim

rmse_SVD_pop = [] kf = KFold(n_splits=10) for item in k: local_rmse = [] print('Testing for k =',item) for trainset, testset in kf.split(ratings_dataset): trim_list = [] unique, counts = np.unique([row[1] for row in testset], return_counts=True) for i in range(len(counts)): if(counts[i]<=2): trim_list.append(unique[i]) trimmed_set = [j for j in testset if j[1] not in trim_list] res = SVD(n_factors=item,n_epochs=20,verbose=False).fit(trainset).test(trimmed_set) local_rmse.append(accuracy.rmse(res,verbose=False)) rmse_SVD_pop.append(np.mean(local_rmse))

In [39]:
```
plt.plot(k,rmse_SVD_pop,linestyle='-',color='r')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (SVD, Popular movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [40]:
```
print("Minimum avg. RMSE (SVD, Popular movie trimming):", min(rmse_SVD_pop))
```

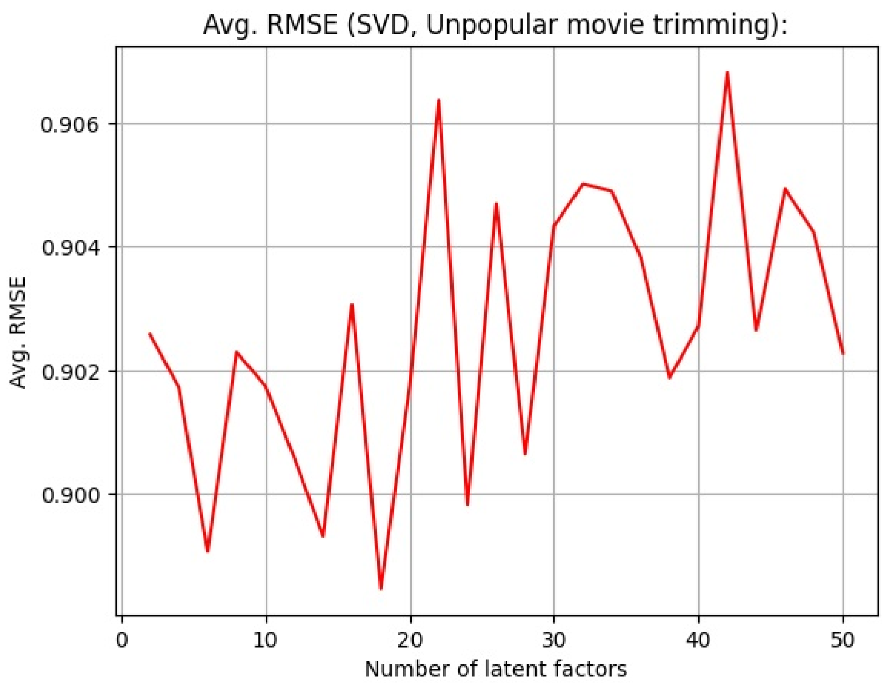
```
Minimum avg. RMSE (SVD, Popular movie trimming): 0.8462452301342986
```

MF Unpopular trim

In [41]:
```
rmse_SVD_unpop = []
kf =  KFold(n_splits=10)
for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trim_list = []
        unique, counts = np.unique([row[1] for row in testset], return_counts=True)
        for i in range(len(counts)):
            if(counts[i]>2):
                trim_list.append(unique[i])
        trimmed_set = [j for j in testset if j[1] not in trim_list]
        res = SVD(n_factors=item,n_epochs=20,verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_SVD_unpop.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [42]:
```
plt.plot(k,rmse_SVD_unpop,linestyle='-',color='r')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (SVD, Unpopular movie trimming):')
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



In [43]:
```
print("Minimum avg. RMSE (SVD, Unpopular movie trimming):", min(rmse_SVD_unpop))
```

```
Minimum avg. RMSE (SVD, Unpopular movie trimming): 0.8984724164026547
```

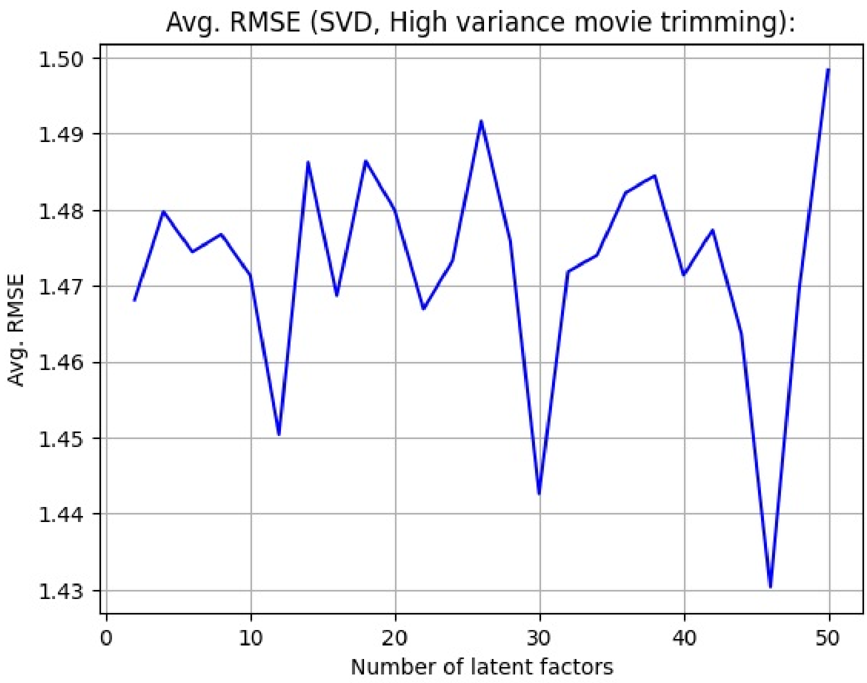
MF High variance trim

In [44]:
```
rmse_SVD_var = []
kf =  KFold(n_splits=10)
dict_of_items = {}
for j in ratings_dataset.raw_ratings:
    if j[1] in dict_of_items.keys():
        dict_of_items[j[1]].append(j[2])
    else:
        dict_of_items[j[1]] = []
        dict_of_items[j[1]].append(j[2])

for item in k:
    local_rmse = []
    print('Testing for k =',item)
    for trainset, testset in kf.split(ratings_dataset):
        trimmed_set = [j for j in testset if (np.var(dict_of_items[j[1]]) >= 2 and len(dict_of_items[j[1]]) >= !
        res = SVD(n_factors=item,n_epochs=20,verbose=False).fit(trainset).test(trimmed_set)
        local_rmse.append(accuracy.rmse(res,verbose=False))
    rmse_SVD_var.append(np.mean(local_rmse))
```

```
Testing for k = 2
Testing for k = 4
Testing for k = 6
Testing for k = 8
Testing for k = 10
Testing for k = 12
Testing for k = 14
Testing for k = 16
Testing for k = 18
Testing for k = 20
Testing for k = 22
Testing for k = 24
Testing for k = 26
Testing for k = 28
Testing for k = 30
Testing for k = 32
Testing for k = 34
Testing for k = 36
Testing for k = 38
Testing for k = 40
Testing for k = 42
Testing for k = 44
Testing for k = 46
Testing for k = 48
Testing for k = 50
```

In [45]:
```
plt.plot(k,rmse_SVD_var,linestyle='-',color='b')
plt.grid(linestyle='-')
plt.title('Avg. RMSE (SVD, High variance movie trimming):')
```

```
plt.ylabel('Avg. RMSE')
plt.xlabel('Number of latent factors')
plt.show()
```



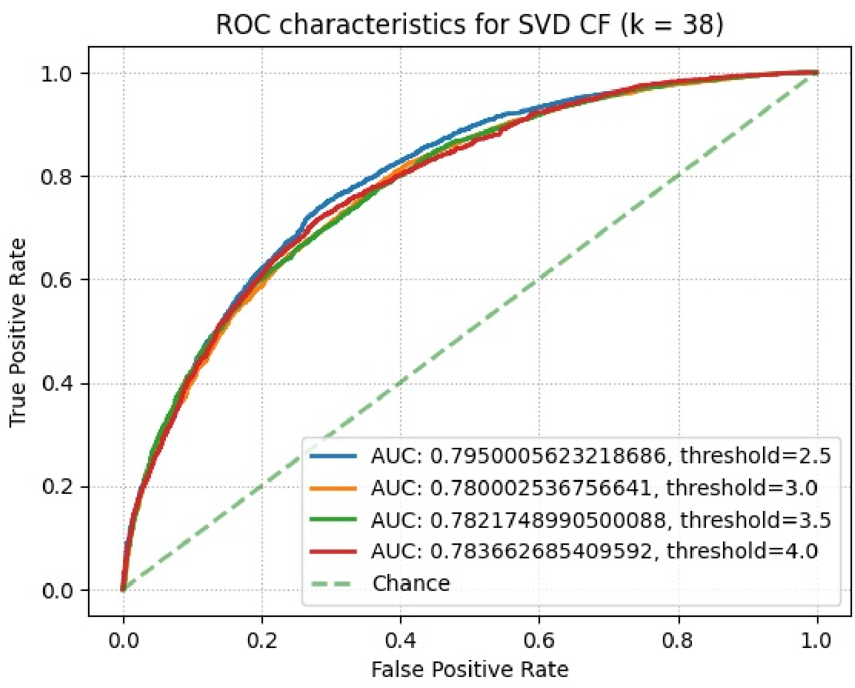
In [46]:
```
print("Minimum avg. RMSE (SVD, High variance movie trimming):", min(rmse_SVD_var))
```

```
Minimum avg. RMSE (SVD, High variance movie trimming): 1.4303842993990883
```

MF ROC

In [48]:
```
k = k[[i for i, x in enumerate(rmse_SVD) if x == min(rmse_SVD)][0]]
thres = [2.5, 3.0, 3.5, 4.0]
trainset, testset = train_test_split(ratings_dataset, test_size=0.1)
res  = SVD(n_factors=k,n_epochs=20,verbose=False).fit(trainset).test(testset)
```

In [49]:
```
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 characteristics for SVD CF (k = '+ str(k)+')')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



Answer 10C

Popular, unpopular and high variance trimmed average RMSE and MAE plots are as shown above. Min avg RMSE (MF) are reported below :

Trimming Type	Minimum Avg. RMSE (MF)
Popular Movie Trimming	0.8462
Unpopular Movie Trimming	0.8985
High Variance Movie Trimming	1.4304

• ROC curve for MF (SVD) along with AUC is as condensed into one plot as shown above

Naive collaborative filtering

In [50]:
```
user_ID_set = list(set(user_ID))
mean_ratings = []
for user in user_ID_set:
    idx = np.where(user_ID == user)
    mean_ratings.append(np.mean(rating[idx]))
```

In [51]:
```
kf = KFold(n_splits=10)
local_rmse = []
for trainset, testset in kf.split(ratings_dataset):
    res = [mean_ratings[int(row[0])-1] for row in testset]
    gt = [row[2] for row in testset]
    local_rmse.append(mean_squared_error(gt,res,squared=False))
rmse_naive = np.mean(local_rmse)
```

```
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

In [52]:

```
print('Avg. RMSE for Naive Filtering: ',rmse_naive)
```

```
Avg. RMSE for Naive Filtering:  0.9347068493123578
```

Answer 11

Average RMSE for Naive Collaborative Filtering = 0.9347

Naive CF Popular trim

In [53]:

```
local_rmse_naive_pop = []
kf =  KFold(n_splits=10)
for trainset, testset in kf.split(ratings_dataset):
    trim_list = []
    unique, counts = np.unique([row[1] for row in testset], return_counts=True)
    for i in range(len(counts)):
        if(counts[i]<=2):
            trim_list.append(unique[i])
    trimmed_set = [j for j in testset if j[1] not in trim_list]
    res = [mean_ratings[int(row[0])-1] for row in trimmed_set]
    gt = [row[2] for row in trimmed_set]
    local_rmse_naive_pop.append(mean_squared_error(gt,res,squared=False))
rmse_naive_pop = np.mean(local_rmse_naive_pop)

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

```
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
Avg. RMSE for Naive Filtering (Popular movie trimming):  0.9213387590910491
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

Naive CF Unpopular trim

In [57]:

```
local_rmse_naive_unpop = []
kf =  KFold(n_splits=10)
for trainset, testset in kf.split(ratings_dataset):
    trim_list = []
    unique, counts = np.unique([row[1] for row in testset], return_counts=True)
    for i in range(len(counts)):
        if(counts[i]>2):
            trim_list.append(unique[i])
    trimmed_set = [j for j in testset if j[1] not in trim_list]
    res = [mean_ratings[int(row[0])-1] for row in trimmed_set]
    gt = [row[2] for row in trimmed_set]
    local_rmse_naive_unpop.append(mean_squared_error(gt,res,squared=False))
rmse_naive_unpop = np.mean(local_rmse_naive_unpop)
print('Avg. RMSE for Naive Filtering (Unpopular movie trimming): ',rmse_naive_unpop)
```

```
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

```
Avg. RMSE for Naive Filtering (Unpopular movie trimming):  0.9542029107785082
```

```
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squ
ared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

Naive CF High variance trim

In [55]:

```
local_rmse_naive_var = []
kf =  KFold(n_splits=10)
dict_of_items = {}
for j in ratings_dataset.raw_ratings:
    if j[1] in dict_of_items.keys():
        dict_of_items[j[1]].append(j[2])
    else:
        dict_of_items[j[1]] = []
        dict_of_items[j[1]].append(j[2])

for trainset, testset in kf.split(ratings_dataset):
    trimmed_set = [j for j in testset if (np.var(dict_of_items[j[1]]) >= 2 and len(dict_of_items[j[1]]) >= 5)]
    res = [mean_ratings[int(row[0])-1] for row in trimmed_set]
    gt = [row[2] for row in trimmed_set]
    local_rmse_naive_var.append(mean_squared_error(gt,res,squared=False))
rmse_naive_var = np.mean(local_rmse_naive_var)

print('Avg. RMSE for Naive Filtering (High variance movie trimming): ',rmse_naive_var)
```

```
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
Avg. RMSE for Naive Filtering (High variance movie trimming):  1.4780141869117536
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
/Users/itami/Library/Python/3.9/lib/python/site-packages/sklearn/metrics/_regression.py:483: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared error, use the function'root_mean_squared_error'.
  warnings.warn(
```

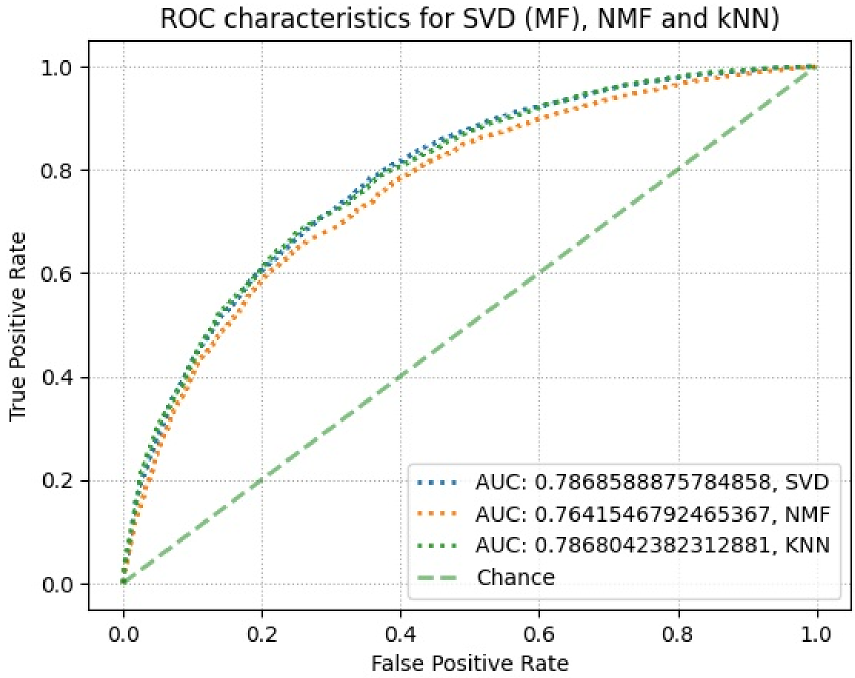
Answer 11 (continued...)

Trimming Type	Avg. RMSE for Naive Filtering
Popular Movie Trimming	0.9213
Unpopular Movie Trimming	0.9542
High Variance Movie Trimming	1.4780

Comparing most performant models

In [58]:
```python
trainset, testset = train_test_split(ratings_dataset, test_size=0.1)
res_SVD  = SVD(n_factors=22,n_epochs=20,verbose=False).fit(trainset).test(testset)
res_NMF  = NMF(n_factors=16,n_epochs=50,verbose=False).fit(trainset).test(testset)
res_KNN = KNNWithMeans(k=20,sim_options={'name':'pearson'},verbose=False).fit(trainset).test(testset)

fig, ax = plt.subplots()
thresholded_out = []
for row in res_SVD:
    if row.r_ui > 3:
        thresholded_out.append(1)
    else:
        thresholded_out.append(0)
fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row in res_SVD])
ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC: "+str(auc(fpr,tpr))+', SVD')

thresholded_out = []
for row in res_NMF:
    if row.r_ui > 3:
        thresholded_out.append(1)
    else:
        thresholded_out.append(0)
fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row in res_NMF])
ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC: "+str(auc(fpr,tpr))+', NMF')

thresholded_out = []
```

```python
for row in res_KNN:
    if row.r_ui > 3:
        thresholded_out.append(1)
    else:
        thresholded_out.append(0)
fpr, tpr, thresholds = roc_curve(thresholded_out, [row.est for row in res_KNN])
ax.plot(fpr, tpr,lw=2,linestyle=':',label="AUC: "+str(auc(fpr,tpr))+', KNN')

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 characteristics for SVD (MF), NMF and kNN)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```



Answer 12

ROC curves are as plotted in one condensed plot as above.

One can see from plot that SVD CF performs best among all the CF, followed by k-NN CF and NNMF-CF coming last. One can explain the performance as follows:

SVD>NMF Reasons:

• SVD effectively represents the higher-dimensional feature matrix without constraints on U and V, leading to a deep factorization with minimal information loss. In contrast, NMF imposes positivity constraints on U and V, resulting in fewer optimal choices.

•SVD produces embeddings with a hierarchical and geometric basis ordered by relevance, making them robust to outliers and noise. NMF, however, does not consider the geometry in the ratings matrix. Embeddings from SVD are unique and deterministic, while NMF is non-unique and stochastic, lacking guarantees of convergence to optimal U and V.

• SVD accounts for user and movie-specific bias information, normalizing them to reduce sensitivity to outliers and noise.

kNN < SVD Reasons (small difference):

• k-NN lacks separate modeling of bias information for each user or item, making it more susceptible to outliers and seldom-rated items. Direct inference on the sparse ratings matrix by k-NN results in poor prediction accuracy in high-dimensional space, exacerbating the curse of dimensionality and hindering scalability.

•Compared to latent-factor based models, k-NN is less generalizable, unable to discover semantic information and connections within the user-item ratings matrix while being sensitive to rarely rated items.

In [40]:
```
from sklearn.datasets import load_svmlight_file
from sklearn.metrics import ndcg_score
import numpy as np
import lightgbm as lgb

# Load the dataset for one fold
def load_one_fold(data_path):
    X_train, y_train, qid_train = load_svmlight_file(str(data_path + 'train.txt'), query_id=True)
    X_test, y_test, qid_test = load_svmlight_file(str(data_path + 'test.txt'), query_id=True)
    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.feature_importance(importance_type=importance_type)
```

In [22]:
```
from sklearn.datasets import load_svmlight_file
web10k_data_path = "MSLR-WEB10K/Fold1/"

# Load the Web10k dataset for one fold
X_train, y_train, qid_train = load_svmlight_file(str(web10k_data_path + 'train.txt'), query_id=True)
X_test, y_test, qid_test = load_svmlight_file(str(web10k_data_path + 'test.txt'), query_id=True)
X_vali, y_vali, qid_vali = load_svmlight_file(str(web10k_data_path + 'vali.txt'), query_id=True)

# Print the number of unique queries in total
total_unique_queries = len(set(np.concatenate((qid_train, qid_test,qid_vali))))
print(f"Total number of unique queries: {total_unique_queries}")

# Show the distribution of relevance labels
unique_labels, label_counts = np.unique(np.concatenate((y_train, y_test,y_vali)), return_counts=True)
print("\nDistribution of relevance labels:")
for label, count in zip(unique_labels, label_counts):
    print(f"Relevance label {label}: {count} samples")
```

```
Total number of unique queries: 10000

Distribution of relevance labels:
Relevance label 0.0: 624263 samples
Relevance label 1.0: 386280 samples
Relevance label 2.0: 159451 samples
Relevance label 3.0: 21317 samples
Relevance label 4.0: 8881 samples
```

Answer 13

Total number of unique queries (including vali.txt) = 10,000

Excluding vali.txt, Total number of unique queries = 8,000

Distribution of relevance labels (*including vali.txt* of Fold1):

Relevance Label	Number of Samples
0.0	624263
1.0	386280
2.0	159451
3.0	21317
4.0	8881

In [36]:

```
def train_lightgbm_model(X_train, y_train, qid_train):
    params = {
        'objective': 'lambdarank',
        'metric': 'ndcg',
        'ndcg_eval_at': [3, 5, 10],
        # Add other parameters as needed
    }

    train_data = lgb.Dataset(X_train, label=y_train, group=np.unique(qid_train, return_counts=True)[1])

    model = lgb.train(params, train_data)

    return model

# Function to evaluate model on test set and print nDCG scores for multiple k values
def evaluate_model(model, X_test, y_test, qid_test, fold_number, k_values=[3, 5, 10]):
    predictions = model.predict(X_test)
    print("#################################  EVALUATING MODEL-FOLD {}   #################################".forr

    for k in k_values:
        ndcg_at_k = compute_ndcg_all(model, X_test, y_test, qid_test, k)
        print(f"nDCG@{k}: {ndcg_at_k}")
    print("\n################################################################################################")

# Train and evaluate LightGBM models for each fold
mslr_data_path = "MSLR-WEB10K"
for fold_number in range(1, 6):
    fold_path = f"{mslr_data_path}/Fold{fold_number}/"
    print(f"\nTraining model for {fold_path}")

    # Load one fold of MSLR data
    X_train, y_train, qid_train, group_train, X_test, y_test, qid_test, group_test = load_one_fold(fold_path)

    # Train a LightGBM model
    model = train_lightgbm_model(X_train, y_train, qid_train)

    # Evaluate the model on the test set
    evaluate_model(model, X_test, y_test, qid_test,fold_number,k_values=[3, 5, 10])
```

```
Training model for MSLR-WEB10K/Fold1/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.074784 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
#################################  EVALUATING MODEL-FOLD 1  #################################
nDCG@3: 0.4564571300800643
nDCG@5: 0.4632890672260867
nDCG@10: 0.48286731451235976

################################################################################################

Training model for MSLR-WEB10K/Fold2/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.086566 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
#################################  EVALUATING MODEL-FOLD 2  #################################
nDCG@3: 0.4538895365009714
nDCG@5: 0.4573292117374164
nDCG@10: 0.4767546810011047

################################################################################################

Training model for MSLR-WEB10K/Fold3/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.077088 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
#################################  EVALUATING MODEL-FOLD 3  #################################
nDCG@3: 0.4490681494620125
nDCG@5: 0.4583480538865081
nDCG@10: 0.47589507831078093

################################################################################################

Training model for MSLR-WEB10K/Fold4/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.075023 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
#################################  EVALUATING MODEL-FOLD 4  #################################
nDCG@3: 0.461178820507814
nDCG@5: 0.4663860127875315
nDCG@10: 0.487724614983737

################################################################################################

Training model for MSLR-WEB10K/Fold5/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.079274 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
#################################  EVALUATING MODEL-FOLD 5  #################################
nDCG@3: 0.46963442883961365
nDCG@5: 0.4714315145908388
nDCG@10: 0.49035928048966515

################################################################################################
```

Answer 14

Evaluation Metric (on test set)	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
nDCG@3	0.456	0.454	0.449	0.461	0.470
nDCG@5	0.463	0.457	0.458	0.466	0.471
nDCG@10	0.483	0.477	0.476	0.488	0.490

In [41]:

```python
# Function to get the top N important features based on 'gain'
def get_top_n_features(model, n=5):
    importance_type = 'gain'
    feature_importance = get_feature_importance(model, importance_type)
    top_indices = np.argsort(feature_importance)[::-1][:n]
    return top_indices
```

```python
# Analyze and interpret results for each fold
for fold_number in range(1, 6):
    fold_path = f"{mslr_data_path}/Fold{fold_number}/"
    print(f"\nTraining model for {fold_path}")

    # Load one fold of MSLR data
    X_train, y_train, qid_train, group_train, X_test, y_test, qid_test, group_test = load_one_fold(fold_path)

    # Train a LightGBM model
    model = train_lightgbm_model(X_train, y_train, qid_train)

    # Get top 5 important features based on 'gain'
    top_features = get_top_n_features(model, n=5)

    print(f"Top 5 important features for Fold {fold_number}:")
    for i, feature_index in enumerate(top_features):
        print(f"{i + 1}. Feature {feature_index + 1}")
    print("#########################################################################")
```

```
Training model for MSLR-WEB10K/Fold1/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.079854 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
Top 5 important features for Fold 1:
1. Feature 134
2. Feature 8
3. Feature 108
4. Feature 55
5. Feature 130
##########################################################################

Training model for MSLR-WEB10K/Fold2/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.082610 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
Top 5 important features for Fold 2:
1. Feature 134
2. Feature 8
3. Feature 55
4. Feature 108
5. Feature 130
##########################################################################

Training model for MSLR-WEB10K/Fold3/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.072701 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
Top 5 important features for Fold 3:
1. Feature 134
2. Feature 55
3. Feature 108
4. Feature 130
5. Feature 8
##########################################################################

Training model for MSLR-WEB10K/Fold4/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.076135 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
Top 5 important features for Fold 4:
1. Feature 134
2. Feature 8
3. Feature 55
4. Feature 130
5. Feature 129
##########################################################################

Training model for MSLR-WEB10K/Fold5/
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.074289 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
Top 5 important features for Fold 5:
1. Feature 134
2. Feature 8
3. Feature 55
4. Feature 108
5. Feature 130
##########################################################################
```

Answer 15

Fold #	Top Feature 1	Top Feature 2	Top Feature 3	Top Feature 4	Top Feature 5
1	134	8	108	55	130
2	134	8	55	108	130
3	134	55	108	130	8
4	134	8	55	130	129
5	134	8	55	108	130

Removing top 20 and removing bottom 60

In [69]:
```python
# Function to remove top N features based on 'gain'
def remove_top_n_features(X, top_indices, n=20):
    X_csc = X.tocsc()
    indices_to_keep = np.setdiff1d(np.arange(X.shape[1]), top_indices[:n])
    return X_csc[:, indices_to_keep]

# Function to remove bottom N features based on 'gain'
def remove_bottom_n_features(X, top_indices, n=60):
    X_csc = X.tocsc()
    indices_to_keep = np.setdiff1d(np.arange(X.shape[1]), top_indices[-n:])
    return X_csc[:, indices_to_keep]

for fold_number in range(1, 6):
    fold_path = f"{mslr_data_path}/Fold{fold_number}/"
    print("****************************************************************************************************
    print(f"\nAnalyzing results for Fold {fold_number}")

    # Load one fold of MSLR data
    X_train, y_train, qid_train, group_train, X_test, y_test, qid_test, group_test = load_one_fold(fold_path)

    # Train a LightGBM model
    model = train_lightgbm_model(X_train, y_train, qid_train)

    # Get top 20 important features based on 'gain'
    top_features = get_top_n_features(model, n=20)

    # Remove top 20 features and train a new model
    X_train_removed_top = remove_top_n_features(X_train, top_features, n=20)
    X_test_removed_top = remove_top_n_features(X_test, top_features, n=20)
    print(f"REMOVING TOP 20 FEATURES for {fold_path}\n")
    print(f"-------------------------Training for {fold_path} (Removing TOP 20 features)----------------------
    new_model_top_removed = train_lightgbm_model(X_train_removed_top, y_train, qid_train)
    evaluate_model(new_model_top_removed, X_test_removed_top, y_test, qid_test,fold_number,k_values=[10])
    # Get top 60 least important features based on 'gain'
    least_features = get_top_n_features(model, n=60)

    # Remove bottom 60 features and train a new model
    X_train_removed_bottom = remove_bottom_n_features(X_train, least_features, n=60)
    X_test_removed_bottom = remove_bottom_n_features(X_test, least_features, n=60)
    print(f"REMOVING BOTTOM 60 FEATURES for {fold_path}\n")

    print(f"-------------------------Training for {fold_path}  (Removing BOTTOM 60 features)------------------
    print("\n**************************************************************************************************
    new_model_bottom_removed = train_lightgbm_model(X_train_removed_bottom, y_train, qid_train)
    evaluate_model(new_model_bottom_removed, X_test_removed_bottom, y_test, qid_test,fold_number,k_values=[10])
```

```
***********************************************************************************************************


Analyzing results for Fold 1
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.074343 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
REMOVING TOP 20 FEATURES for MSLR-WEB10K/Fold1/

-------------------------Training for MSLR-WEB10K/Fold1/ (Removing TOP 20 features)----------------------------
-:
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.075550 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
#################################  EVALUATING MODEL-FOLD 1   #################################
nDCG@10: 0.4083636029390886

#################################################################################################
REMOVING BOTTOM 60 FEATURES for MSLR-WEB10K/Fold1/

-------------------------Training for MSLR-WEB10K/Fold1/  (Removing BOTTOM 60 features)------------------------
-----:

***********************************************************************************************************
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.050178 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 13014
[LightGBM] [Info] Number of data points in the train set: 723412, number of used features: 76
```

```
#################################  EVALUATING MODEL-FOLD 1   #################################
nDCG@10: 0.3761216118110393

################################################################################################
*****************************************************************************************************************


Analyzing results for Fold 2
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.086096 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
REMOVING TOP 20 FEATURES for MSLR-WEB10K/Fold2/

--------------------------Training for MSLR-WEB10K/Fold2/ (Removing TOP 20 features)----------------------------
-:
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.071261 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 21551
[LightGBM] [Info] Number of data points in the train set: 716683, number of used features: 116
#################################  EVALUATING MODEL-FOLD 2   #################################
nDCG@10: 0.4045026694861529

################################################################################################
REMOVING BOTTOM 60 FEATURES for MSLR-WEB10K/Fold2/

--------------------------Training for MSLR-WEB10K/Fold2/  (Removing BOTTOM 60 features)------------------------
-----:

*****************************************************************************************************************
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.055455 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 12229
[LightGBM] [Info] Number of data points in the train set: 716683, number of used features: 76
#################################  EVALUATING MODEL-FOLD 2   #################################
nDCG@10: 0.3724982237661007

################################################################################################
*****************************************************************************************************************


Analyzing results for Fold 3
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.073757 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
REMOVING TOP 20 FEATURES for MSLR-WEB10K/Fold3/

--------------------------Training for MSLR-WEB10K/Fold3/ (Removing TOP 20 features)----------------------------
-:
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.064718 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
#################################  EVALUATING MODEL-FOLD 3   #################################
nDCG@10: 0.4116363812695088

################################################################################################
REMOVING BOTTOM 60 FEATURES for MSLR-WEB10K/Fold3/

--------------------------Training for MSLR-WEB10K/Fold3/  (Removing BOTTOM 60 features)------------------------
-----:

*****************************************************************************************************************
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.052060 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 12248
[LightGBM] [Info] Number of data points in the train set: 719111, number of used features: 76
#################################  EVALUATING MODEL-FOLD 3   #################################
nDCG@10: 0.3744687542130175

################################################################################################
*****************************************************************************************************************


Analyzing results for Fold 4
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.072624 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
REMOVING TOP 20 FEATURES for MSLR-WEB10K/Fold4/

-------------------------Training for MSLR-WEB10K/Fold4/ (Removing TOP 20 features)----------------------------
-:
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.069666 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
#################################  EVALUATING MODEL-FOLD 4   #################################
nDCG@10: 0.4121071637228934

###############################################################################################
REMOVING BOTTOM 60 FEATURES for MSLR-WEB10K/Fold4/

-------------------------Training for MSLR-WEB10K/Fold4/  (Removing BOTTOM 60 features)------------------------
-----:

***************************************************************************************************************
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.048760 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 12917
[LightGBM] [Info] Number of data points in the train set: 718768, number of used features: 76
#################################  EVALUATING MODEL-FOLD 4   #################################
nDCG@10: 0.3764053801895373

###############################################################################################
***************************************************************************************************************


Analyzing results for Fold 5
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.076907 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
REMOVING TOP 20 FEATURES for MSLR-WEB10K/Fold5/

-------------------------Training for MSLR-WEB10K/Fold5/ (Removing TOP 20 features)----------------------------
-:
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.085747 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
#################################  EVALUATING MODEL-FOLD 5   #################################
nDCG@10: 0.4166871494621703

###############################################################################################
REMOVING BOTTOM 60 FEATURES for MSLR-WEB10K/Fold5/

-------------------------Training for MSLR-WEB10K/Fold5/  (Removing BOTTOM 60 features)------------------------
-----:

***************************************************************************************************************
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.046780 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 13062
[LightGBM] [Info] Number of data points in the train set: 722602, number of used features: 76
#################################  EVALUATING MODEL-FOLD 5   #################################
nDCG@10: 0.3795631808598626

###############################################################################################
```

Answer 16

- Remove top 20 features:

Fold #	nDCG@10
1	0.408
2	0.405
3	0.412
4	0.412

5	0.417

We note that the performance measured by nDCG@10 shows a decrease in values compared to the original model by around 5%, indicating that these features play a role in the model's predictive capabilities. But the decrease is not much. The reasons could be probably owing to redundancy in features: the removed features might be redundant or highly correlated with other features in the dataset. In such cases, the model can still rely on the correlated information provided by the remaining features. It may also be the case that the LightGBM model may have enough capacity to compensate for the removal of a few top features. The model may have learned alternative patterns from the remaining features.

• Remove bottom 60 features:

Fold	nDCG@10
1	0.376
2	0.372
3	0.374
4	0.376
5	0.380

Removing least 60 important features leads to close to an 7 % decrement in performance. This clearly makes sense with the intuition that lower importance features don't contribute much to the model performance, with features being highly redundant. The removed features might be redundant or less informative, and their exclusion did not lead to a loss of crucial information. With respect to the model on removing top 20, it only caused a further 2% loss which is miniscule even after removing 60 feature vectors and hence we note that only the top few features are very important in a model's decision mechanism.