In [1]:
```
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

In [2]:
```
movies = pd.read_csv("ml-latest/movies.csv")

movies.head(10)
```

Out[2]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

In [3]:
```
ratings = pd.read_csv("ml-latest/ratings.csv")
ratings.head(5)
```

Out[3]:

	userId	movieId	rating	timestamp
0	1	1	4.0	1225734739
1	1	110	4.0	1225865086
2	1	158	4.0	1225733503
3	1	260	4.5	1225735204
4	1	356	5.0	1225735119

1.1. **Adapt** the content-based implementation by using the 33 million ratings dataset instead to generate top 10 recommendations using cosine-similarity for a specific user (Use the genre and decade features to compute a user vector by taking the average of the feature vectors for the movies the user has watched, weighted by their rating. Then, use that user vector to calculate similarity between the user vector and each of the movie feature vectors the user did not rate).

In [4]:
```
movies['genres'] = movies['genres'].apply(lambda x: x.split("|"))
movies.head()
```

Out[4]:

	movieId	title	genres
0	1	Toy Story (1995)	[Adventure, Animation, Children, Comedy, Fantasy]
1	2	Jumanji (1995)	[Adventure, Children, Fantasy]

2	3	Grumpier Old Men (1995)	[Comedy, Romance]
3	4	Waiting to Exhale (1995)	[Comedy, Drama, Romance]
4	5	Father of the Bride Part II (1995)	[Comedy]

In [5]:
```
from collections import Counter

genres_counts = Counter(g for genres in movies['genres'] for g in genres)
print(f"There are {len(genres_counts)} genre labels.")
genres_counts
```

Out[5]:
```
There are 20 genre labels.
Counter({'Drama': 33681,
         'Comedy': 22830,
         'Thriller': 11675,
         'Romance': 10172,
         'Action': 9563,
         'Documentary': 9283,
         'Horror': 8570,
         '(no genres listed)': 7060,
         'Crime': 6917,
         'Adventure': 5349,
         'Sci-Fi': 4850,
         'Animation': 4579,
         'Children': 4367,
         'Mystery': 3972,
         'Fantasy': 3821,
         'War': 2301,
         'Western': 1690,
         'Musical': 1059,
         'Film-Noir': 354,
         'IMAX': 195})
```

In [6]:
```
movies = movies[movies['genres']!='(no genres listed)']

del genres_counts['(no genres listed)']
```

In [7]:
```
print("The 5 most common genres: \n", genres_counts.most_common(5))
```

```
The 5 most common genres:
 [('Drama', 33681), ('Comedy', 22830), ('Thriller', 11675), ('Romance', 10172), ('Actio
n', 9563)]
```

In [8]:
```
import re

def extract_year_from_title(title):
    match = re.search(r'\((\d{4})\)', title)
    if match:
        return int(match.group(1))
    return None

title = "Toy Story (1995)"
year = extract_year_from_title(title)
print(f"Year of release: {year}")
print(type(year))

movies['year'] = movies['title'].apply(extract_year_from_title)
movies.head()
```

```
Year of release: 1995
<class 'int'>
```

Out[8]:

	movieId	title	genres	year
0	1	Toy Story (1995)	[Adventure, Animation, Children, Comedy, Fantasy]	1995.0

1	2	Jumanji (1995)	[Adventure, Children, Fantasy]	1995.0
2	3	Grumpier Old Men (1995)	[Comedy, Romance]	1995.0
3	4	Waiting to Exhale (1995)	[Comedy, Drama, Romance]	1995.0
4	5	Father of the Bride Part II (1995)	[Comedy]	1995.0

In [9]:
```
movies['year'].nunique()
```

Out[9]:
```
142
```

In [10]:
```
print(f"Original number of movies: {movies['movieId'].nunique()}")
```

```
Original number of movies: 86537
```

In [11]:
```
movies = movies[~movies['year'].isnull()]
print(f"Number of movies after removing null years: {movies['movieId'].nunique()}")
```

```
Number of movies after removing null years: 85919
```

In [12]:
```
x = 1995

def get_decade(year):
    year = str(year)
    decade_prefix = year[0:3] # get first 3 digits of year
    decade = f'{decade_prefix}0' # append 0 at the end
    return int(decade)

get_decade(x)
```

Out[12]:
```
1990
```

In [13]:
```
def round_down(year):
    return year - (year%10)

round_down(x)
```

Out[13]:
```
1990
```

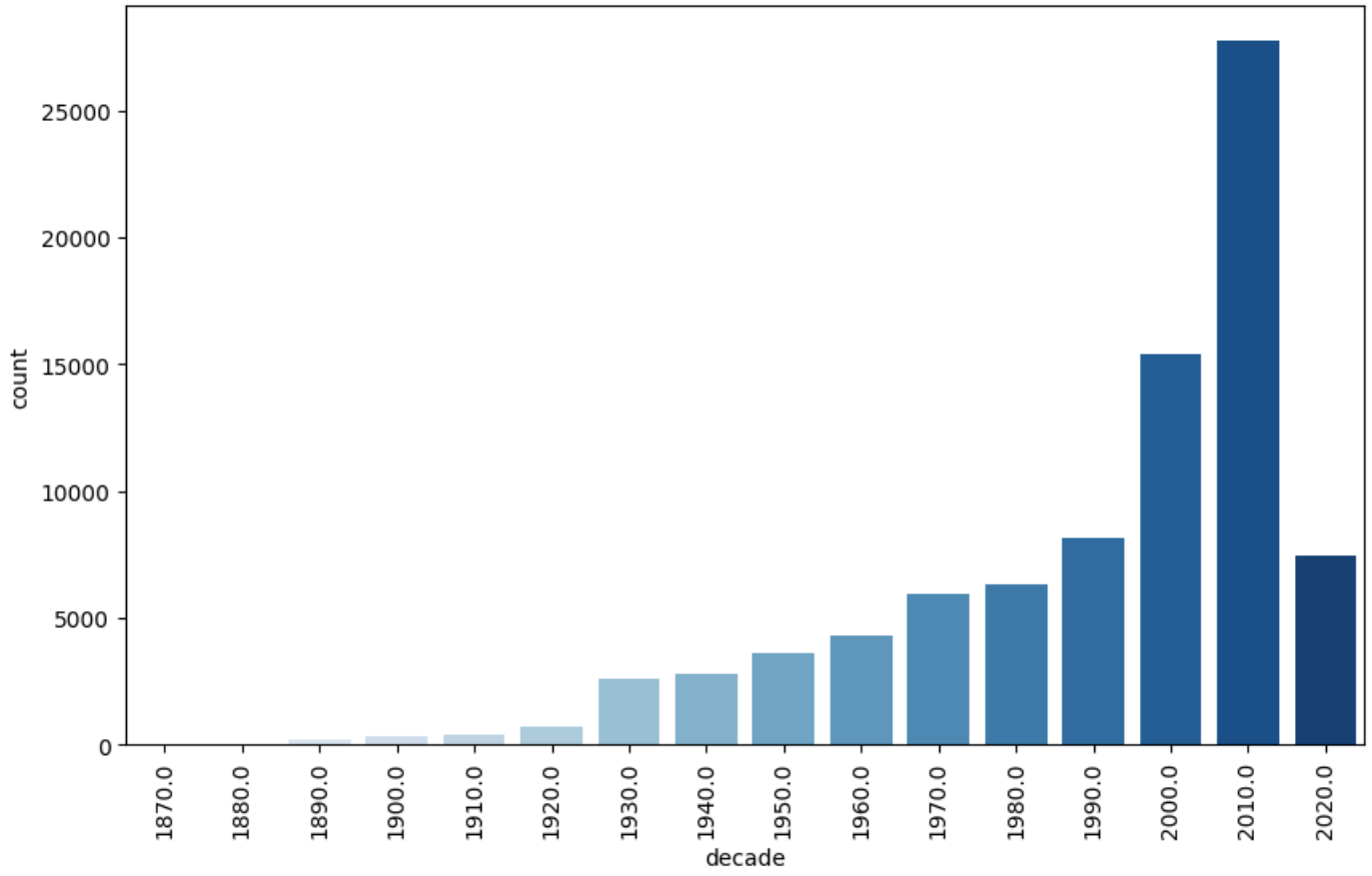
In [14]:
```
movies['decade'] = movies['year'].apply(round_down)
```

In [15]:
```
print(movies['decade'].unique())
```

```
[1990. 1970. 1980. 1960. 1930. 1940. 1950. 1920. 1910. 2000. 1900. 2010.
 1890. 1880. 1870. 2020.]
```

In [16]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10,6))
sns.countplot(data=movies, x='decade', palette='Blues', order=sorted(movies['decade'].un
plt.xticks(rotation=90)
plt.show()
```



In [17]:
```
genres = list(genres_counts.keys())

for g in genres:
    movies[g] = movies['genres'].transform(lambda x: int(g in x))
```

In [18]:
```
movies[genres].head()
```

Out[18]:

	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Crime	Thriller	Horror	M
0	1	1	1	1	1	0	0	0	0	0	0	
1	1	0	1	0	1	0	0	0	0	0	0	
2	0	0	0	1	0	1	0	0	0	0	0	
3	0	0	0	1	0	1	1	0	0	0	0	
4	0	0	0	1	0	0	0	0	0	0	0	

In [19]:
```
movie_decades = pd.get_dummies(movies['decade'])
movie_decades = movie_decades.astype(int)
print(movie_decades.head())
```

```
   1870.0  1880.0  1890.0  1900.0  1910.0  1920.0  1930.0  1940.0  1950.0  \
0       0       0       0       0       0       0       0       0       0
1       0       0       0       0       0       0       0       0       0
2       0       0       0       0       0       0       0       0       0
3       0       0       0       0       0       0       0       0       0
4       0       0       0       0       0       0       0       0       0

   1960.0  1970.0  1980.0  1990.0  2000.0  2010.0  2020.0
0       0       0       0       1       0       0       0
1       0       0       0       1       0       0       0
2       0       0       0       1       0       0       0
3       0       0       0       1       0       0       0
4       0       0       0       1       0       0       0
```

In [20]:
```
movie_decades = pd.get_dummies(movies['decade'])
movie_decades.head()
```

Out[20]:

	1870.0	1880.0	1890.0	1900.0	1910.0	1920.0	1930.0	1940.0	1950.0	1960.0	1970.0	1980.0	199
0	False	False	False	False	False	False	False	False	False	False	False	False	
1	False	False	False	False	False	False	False	False	False	False	False	False	
2	False	False	False	False	False	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	False	False	False	False	False	

In [21]:
```
movie_features = pd.concat([movies[genres], movie_decades], axis=1)
movie_features.head()
```

Out[21]:

	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Crime	Thriller	...	1930
0	1	1	1	1	1	0	0	0	0	0	...	Fals
1	1	0	1	0	1	0	0	0	0	0	...	Fals
2	0	0	0	1	0	1	0	0	0	0	...	Fals
3	0	0	0	1	0	1	1	0	0	0	...	Fals
4	0	0	0	1	0	0	0	0	0	0	...	Fals

5 rows × 35 columns

In [22]:
```
# Convert the boolean decade to int
movie_decades = movie_decades * 1
movie_features = pd.concat([movies[genres], movie_decades], axis=1)
print(movie_features.head())
```

```
   Adventure  Animation  Children  Comedy  Fantasy  Romance  Drama  Action
0          1          1         1       1        1        0      0       0  \
1          1          0         1       0        1        0      0       0
2          0          0         0       1        0        1      0       0
3          0          0         0       1        0        1      1       0
4          0          0         0       1        0        0      0       0

   Crime  Thriller  ...  1930.0  1940.0  1950.0  1960.0  1970.0  1980.0
0      0         0  ...       0       0       0       0       0       0  \
1      0         0  ...       0       0       0       0       0       0
2      0         0  ...       0       0       0       0       0       0
3      0         0  ...       0       0       0       0       0       0
4      0         0  ...       0       0       0       0       0       0

   1990.0  2000.0  2010.0  2020.0
0       1       0       0       0
1       1       0       0       0
2       1       0       0       0
3       1       0       0       0
4       1       0       0       0

[5 rows x 35 columns]
```

In [23]:
```
#from sklearn.metrics.pairwise import cosine_similarity

#cosine_sim = cosine_similarity(movie_features, movie_features)
#print(f"Dimensions of our movie features cosine similarity matrix: {cosine_sim.shape}")
```

1.1. **Adapt** the content-based implementation by using the 33 million ratings dataset instead to generate top 10 recommendations using cosine-similarity for a specific user (Use the genre and decade features to compute a user vector by taking the average of the feature vectors for the movies the user has watched, weighted by their rating. Then, use that user vector to calculate similarity between the user vector and each of the movie feature vectors the user did not rate).

In [24]:
```
def compute_user_vector(user_id, ratings, movie_features):
    user_movies = ratings[ratings['userId'] == user_id]
    user_vector = np.zeros(movie_features.shape[1])

    for index, row in user_movies.iterrows():
        movie_id = row['movieId']
        rating = row['rating']
        movie_vector = movie_features.loc[movie_id].values
        user_vector += rating * movie_vector

    user_vector /= len(user_movies)

    return user_vector
```

In [25]:
```
from sklearn.metrics.pairwise import cosine_similarity

def get_recommendations_for_user(user_id, ratings, movie_features, n=10):
    user_vector = compute_user_vector(user_id, ratings, movie_features)
    user_vector = user_vector.reshape(1, -1)

    movies_rated_by_user = ratings[ratings['userId'] == user_id]['movieId'].tolist()
    movies_not_rated_by_user = movie_features[~movie_features.index.isin(movies_rated_by

    cosine_sim_scores = cosine_similarity(user_vector, movies_not_rated_by_user)
    cosine_sim_scores = cosine_sim_scores[0]

    recommended_movie_ids = movies_not_rated_by_user.index[np.argsort(-cosine_sim_scores

    return recommended_movie_ids
```

In [26]:
```
user_id = 123  # Replace with any user
recommended_movie_ids = get_recommendations_for_user(user_id, ratings, movie_features)
print(movies['title'].loc[recommended_movie_ids])
```

```
2505                                    SLC Punk! (1998)
304      Three Colors: White (Trzy kolory: Bialy) (1994)
48483              Confessions of a Sorority Girl (1994)
35571                                   Hotel Room (1993)
300                                      Roommates (1995)
49112                 Weapons of Mass Distraction (1997)
17041            God's Comedy (A Comédia de Deus) (1995)
11677                                        Uranus (1990)
40972                    Tales of a Golden Geisha (1990)
9088                                     Breast Men (1997)
Name: title, dtype: object
```

1.2. **Train** the collaborative filtering system and select optimal hyperparameters on training dataset (80\%) using 5-fold cross validation. Use the RSME metric to evaluate your system in each split. (You can use the Surprise library for this)

In [27]:
```
from surprise import Dataset, Reader
from surprise import SVD
from surprise.model_selection import cross_validate, train_test_split
```

In [28]:
```
small_ratings = ratings.sample(frac=0.01, random_state=42) #had to take a fraction of th
```

```
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(ratings[['userId', 'movieId', 'rating']], reader)
```

In [29]:
```
trainset, testset = train_test_split(data, test_size=0.2)
```

In [ ]:
```
algo = SVD()

# Run 5-fold cross-validation and print
cross_validate(algo, data, measures=['RMSE'], cv=5, verbose=True)
```

In [ ]:
```
from surprise.model_selection import GridSearchCV

param_grid = {
    'n_epochs': [5, 10, 20],
    'lr_all': [0.002, 0.005, 0.01],
    'reg_all': [0.2, 0.4, 0.6]
}

gs = GridSearchCV(SVD, param_grid, measures=['rmse'], cv=5)
gs.fit(data)

# Best RMSE score
print(gs.best_score['rmse'])

# Best hyperparameters
print(gs.best_params['rmse'])
```

In [ ]:
```
# Use best parameters from grid search
algo = SVD(n_epochs=gs.best_params['rmse']['n_epochs'],
           lr_all=gs.best_params['rmse']['lr_all'],
           reg_all=gs.best_params['rmse']['reg_all'])

algo.fit(trainset)
predictions = algo.test(testset)

from surprise import accuracy
accuracy.rmse(predictions)
```

1.3. **Implement** the MAE and RMSE functions from scratch to compute the two types of error in the following (input parameters are two vectors): True ratings of a user: `[2,3,4,2,1,1,1,2,3,4,5,6]` Predicted ratings of a user: `[2.1,3.5,2,1,1.5,1.3,0.8,1.5,1.1,4.5,5,6.1]`

In [ ]:
```
def MAE(y_true, y_pred):
    assert len(y_true) == len(y_pred), "Both vectors should be of same length"
    N = len(y_true)
    mae = sum(abs(y_true[i] - y_pred[i]) for i in range(N)) / N
    return mae
```

In [ ]:
```
def RMSE(y_true, y_pred):
    assert len(y_true) == len(y_pred), "Both vectors should be of same length"
    N = len(y_true)
    rmse = sum((y_true[i] - y_pred[i])**2 for i in range(N))
    rmse = (rmse / N)**0.5
    return rmse
```

In [ ]:
```
y_true = [2,3,4,2,1,1,1,2,3,4,5,6]
y_pred = [2.1,3.5,2,1,1.5,1.3,0.8,1.5,1.1,4.5,5,6.1]

mae = MAE(y_true, y_pred)
rmse = RMSE(y_true, y_pred)
```

```
print(f"MAE: {mae}")
print(f"RMSE: {rmse}")
```

Task 2: Content-Based Recommendation

Implement a new recommendation algorithm that uses the MovieLens Tag Genome to recommend movies that are similar to a user's rated movies. The Tag Genome comes with the ML-33M data set, in the genome-scores.csv file. This file contains three columns: movie ID, tag ID, and relevance (the meaning of the tag IDs is in genome-tags.csv). It is complete, in that it has relevance scores for every movie for a large number of tags, although it does not cover every movie in the MovieLens data. You can think of this file as defining a vector for every movie: an approximately 1100-dimensional vector describing the movie in terms of Tag Genome tags.

Objectives

2.1. Write an algorithm by:

- Computing a user tag vector by taking the average of the tag vectors for the movies the user has watched, weighted by their rating (that is, the user's value for a tag is the average of their movies' value for that tag. You can do this relatively efficiently with NumPy vectorized operations.).
- Select a user and get the top 10 recommendations that have not been rated by that user before. This is done by computing the Pearson correlation between the user tag vector and each of the movie's tag vector not rated by the user.
- Suppose that the content based filtering of Task 1 is an ideal state-of-the-art system where it always outputs an ideal relevant top 10 recommendation movies. How do these recommendations of this task compare to the content-based filtering done in Task 1 for the same user you selected? Implement and compute the nDCG@10 from scratch of the ranking of that user. Example:

  Ideal system ranking for user i (fixed): `[1,1,1,1,1,1,1,1,1,1]`
  An example of your algorithm output ranking for user i: `[0,1,0,1,1,0,1,0,1,1]`

  where 1 in the ideal ranking indicates that the item is relevant, and 1 in your algorithm indicates that the item is relevant because it is found in the rankings of the ideal system regardless of the position of the item in the ranking@10.

In [ ]:

In [ ]: