Introduction

In modern recommender systems, effectively utilizing user feedback is crucial for enhancing recommendation quality and user experience. Feedback is categorized into explicit and implicit types. Explicit feedback, such as ratings and likes, directly indicates user preferences, while implicit feedback is inferred from behaviors like clicks and purchase history.

Handling these feedback types requires different approaches. Explicit feedback should not treat missing values as zero, as this may misrepresent user sentiment. Conversely, with implicit feedback, missing values can often be filled with zeros, indicating potential disinterest.

Sparsity in user ratings poses challenges for traditional algorithms like K-Nearest Neighbors (KNN). Thus, addressing sparse data is vital for building effective recommendation models. While accuracy is a key metric for evaluating recommender systems, secondary metrics like diversity (providing varied recommendations) and serendipity (offering unexpected yet relevant suggestions) also enhance user engagement and satisfaction.

This project will develop a hybrid recommender system using a meta-level approach, combining content-based and collaborative filtering. Content-based filtering helps mitigate the cold-start problem by leveraging item features to make initial recommendations. By extracting movie details from IMDb, the system will group users with similar preferences based on content features and then apply collaborative filtering to make predictions.

This customized approach, termed “collaboration via content,” enhances traditional collaborative filtering by incorporating content data to identify similar users. While effective for recommending movies within specific genres, this strategy can also diversify recommendations in industries aiming to encourage users to explore new products, particularly during business expansions.

In [1]:
```
import warnings

# Suppress all warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np
import re
from sklearn.model_selection import train_test_split
# from surprise import Reader, Dataset, SVD, evaluate
```

In [2]:
```
movies_columns = ['movieId', 'title', 'genres']
ratings_columns = ['userId','movieId', 'rating', 'timestamp']

movies_df = pd.read_csv('/Users/top/Documents/GitHub/data_science_proj
ratings_df = pd.read_csv('/Users/top/Documents/GitHub/data_science_pro
```

TLDR

1. Content-Based Filtering: This approach utilizes item characteristics to provide recommendations, effectively addressing the cold-start problem by suggesting similar items based on user preferences, even with limited interaction history.
2. Complex Models and Overfitting: While sophisticated models can be appealing, they risk overfitting, leading to poor generalization on unseen data. Balancing model complexity with regularization techniques and validation is essential for robust performance.
3. Matrix Factorization and Gradient Descent: These techniques enhance collaborative filtering by learning user-item interactions without filling missing values with zeros, which can introduce bias. Gradient descent optimizes latent factors, resulting in more accurate and personalized recommendations.

Preprocessing

To handle bias in ratings, where some users might give consistently high or low ratings, I will use centered ratings. This means adjusting each user's rating by subtracting their average rating.

In [3]:
```
ratings_df['centered_rating'] = ratings_df['rating'] - ratings_df.grou
```

In [4]:
```
genre=[]

for num in range(0, len(movies_df)):
    key = movies_df.iloc[num]['title']
    value = movies_df.iloc[num]['genres'].split('|')
    genre.append(value)

movies_df['genres'] = genre
```

In [5]:
```
p = re.compile(r"(?:\((\d{4})\))?\s*$")
years=[]
for movies in movies_df['title']:
     m = p.search(movies)
     year = m.group(1)
     years.append(year)
movies_df['year']=years
```

In [6]:
```
movies_name=[]
raw=[]

for movies in movies_df['title']:
     m = p.search(movies)
     year = m.group(0)
     new=re.split(year, movies)
     raw.append(new)
for i in range(len(raw)):
    movies_name.append(raw[i][0][:-2].title())

movies_df['title'] = movies_name
```

In [7]:
```
movies_df['features'] = (
    movies_df['title'] +
    ' ' + movies_df['genres'].apply(lambda x: ' '.join(x)) +
    ' ' + movies_df['year'].astype(str)
)
```

Content-based filtering

In [8]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

In [9]:
```

common_movies = ratings_df['movieId'].unique()
filtered_movies_df = movies_df[movies_df['movieId'].isin(common_movies

tfid = TfidfVectorizer(stop_words='english')
matrix = tfid.fit_transform(filtered_movies_df['features'])
cosine_sim = cosine_similarity(matrix, matrix)
```

In [10]:
```
ratings_matrix = ratings_df.pivot(index='userId', columns='movieId', v
num_users = ratings_matrix.shape[0]
num_items = ratings_matrix.shape[1]
```

In [42]:
```python
def get_similar_users(cosine_sim, user_index, num_similar=5):
    """Get the top similar users based on cosine similarity."""
    similar_indices = cosine_sim[user_index].argsort()[-num_similar-1:
    return similar_indices
```

Collaborative filtering

In [12]:
```python
class MatrixFactorizationWithBias:
    def __init__(self, R, k, alpha, lambda_reg, num_epochs, tolerance=
        self.R = R
        self.num_users, self.num_items = R.shape
        self.k = k
        self.alpha = alpha
        self.lambda_reg = lambda_reg
        self.num_epochs = num_epochs
        self.tolerance = tolerance


        # Uniform initialization
        self.U = np.random.uniform(low=-0.01, high=0.01, size=(self.nu
        self.V = np.random.uniform(low=-0.01, high=0.01, size=(self.nu

    def train(self):
        previous_mse = float('inf')
        for epoch in range(self.num_epochs):
            self.gradient_descent()
            mse = self.compute_error()
            print(f"Epoch: {epoch + 1}, MSE: {mse:.4f}")

            if abs(previous_mse - mse) < self.tolerance:
                print("Convergence reached.")
                break
            previous_mse = mse

    def gradient_descent(self):
        S = np.argwhere(~np.isnan(self.R))
        for i, j in S:
            prediction = self.predict(i, j)
            eij = self.R[i, j] - prediction

            if np.isnan(eij):
                print(f"NaN error found for user {i}, item {j}. Predic
                continue

            for q in range(self.k):
                update_U = self.alpha * (eij * self.V[j, q] - self.lam
                update_V = self.alpha * (eij * self.U[i, q] - self.lam

                if np.isnan(update_U) or np.isnan(update_V):
                    print(f"NaN update for user {i}, item {j}, latent
```

```
                    continue

                self.U[i, q] += update_U
                self.V[j, q] += update_V

    def predict(self, i, j):
        return np.dot(self.U[i], self.V[j])

    def compute_error(self):
        xs, ys = np.argwhere(~np.isnan(self.R)).T
        predicted = np.array([self.predict(i, j) for i, j in zip(xs, y
        observed_ratings = self.R[xs, ys]

        if np.any(np.isnan(predicted)) or np.any(np.isnan(observed_rat
            print("NaN found in predictions or observed ratings.")

        error = np.sum((observed_ratings - predicted) ** 2)
        return np.sqrt(error / len(xs)) if len(xs) > 0 else float('nan
```

In [13]:
```
def create_ratings_matrix(df):
    return df.pivot(index='userId', columns='movieId', values='centere
```

The structure of a rating matrix differs from traditional machine learning tasks, where you typically have clear dependent (target) and independent (feature) variables. In a rating matrix, there isn't a straightforward distinction between these two. Instead, the matrix contains user-item interactions (like movie ratings), where each entry represents a user's rating for an item. This makes it challenging to directly apply typical machine learning techniques.

To evaluate the model's performance, a common approach used in recommender systems is the hold-out method. In this method, a portion of the ratings is "hidden" or set aside as a test set, while the remaining ratings are used for training the model. The hidden ratings are then used to check how well the model can predict unseen data, allowing for a more realistic assessment of its accuracy in making recommendations.

In [14]:
```
def split_data_holdout_entry_based(ratings_matrix, test_fraction=0.2):
    R = ratings_matrix.values.copy()

    observed_indices = np.array(np.where(~np.isnan(R))).T
    np.random.shuffle(observed_indices)

    num_test = int(len(observed_indices) * test_fraction)

    test_indices = observed_indices[:num_test]
    train_indices = observed_indices[num_test:]

    R_train = R.copy()
    R_test = np.full(R.shape, np.nan)

    for idx in test_indices:
        i, j = idx
        R_test[i, j] = R[i, j]
        R_train[i, j] = np.nan

    print("Original Data Size:", R.shape)
    print("Training Set Size:", np.sum(~np.isnan(R_train)))
    print("Test Set Size:", np.sum(~np.isnan(R_test)))

    return R_train, R_test, test_indices
```

to double check the split data by hold out method

In [15]:
```
pivot_rating_df = create_ratings_matrix(ratings_df)
R_train, R_test, test_indices = split_data_holdout_entry_based(pivot_r
```

```
Original Data Size: (69878, 10677)
Training Set Size: 8000044
Test Set Size: 2000010
```

In [16]:
```
train_df = pd.DataFrame(R_train, index=pivot_rating_df.index, columns=
test_df = pd.DataFrame(R_test, index=pivot_rating_df.index, columns=pi

print("Original Data Size:", ratings_df.shape)
print("Training Set Size:", train_df.count().sum())
print("Test Set Size:", test_df.count().sum())


overlap = np.where(~np.isnan(R_train) & ~np.isnan(R_test))
print("Number of overlapping entries:", len(overlap[0]))

train_ratings = train_df.stack().reset_index()
test_ratings = test_df.stack().reset_index()

test_in_train = test_ratings[test_ratings[['userId', 'movieId']].apply
print("Test ratings found in training set:", len(test_in_train))

print("Coverage in Training Set:", len(train_ratings) / (len(ratings_d
print("Coverage in Test Set:", len(test_ratings) / (len(ratings_df) *
```

```
Original Data Size: (10000054, 5)
Training Set Size: 8000044
Test Set Size: 2000010
Number of overlapping entries: 0
Test ratings found in training set: 0
Coverage in Training Set: 0.007492742156032294 %
Coverage in Test Set: 0.001873184602420455 %
```

In the model, latent factors (k), learning rate (alpha), and Regularization (lambda) are the hypeparameters in the model.
Small k, high lambda, small alpha = High bias, low variance (underfitting).
Large k, low lambda, large alpha = Low bias, high variance (overfitting).

In [17]:
```
def calculate_rmse(predictions, actuals):
    mse = np.mean((predictions - actuals) ** 2)
    return np.sqrt(mse)
```

In [18]:

```
def hyperparameter_tuning(ratings_matrix, param_grid, test_fraction=0.

    R_train, R_test, test_indices = split_data_holdout_entry_based(rat

    best_rmse = float('inf')
    best_params = {}

    for k in param_grid['k']:
        for alpha in param_grid['alpha']:
            for lambda_reg in param_grid['lambda_reg']:
                print(f"Testing k={k}, alpha={alpha}, lambda_reg={lamb

                mf_model = MatrixFactorizationWithBias(R_train, k=k, a
                mf_model.train()

                test_predictions = []
                for i, j in test_indices:
                    if not np.isnan(R_test[i, j]):
                        prediction = mf_model.predict(i, j)
                        test_predictions.append(prediction)

                actuals = R_test[test_indices[:, 0], test_indices[:, 1

                if len(test_predictions) > 0:
                    rmse = calculate_rmse(np.array(test_predictions),
                else:
                    print("No predictions were made.")
                    rmse = float('nan')

                print(f"Validation RMSE: {rmse:.4f}")

                if rmse < best_rmse:
                    best_rmse = rmse
                    best_params = {'k': k, 'alpha': alpha, 'lambda_reg

    print("Best Parameters:")
    print(best_params)
    print(f"Best Validation RMSE: {best_rmse:.4f}")

    return best_params, best_rmse
```

The relationship between k (latent factors), alpha (learning rate), and lambda_reg (regularization) is simple:

k (Latent Factors): Increasing k improves the model's accuracy by capturing more patterns, but the benefits decrease as k gets larger, making the model slower to train.

alpha (Learning Rate): A higher alpha helps the model learn faster but can cause instability at the start. A moderate alpha balances speed and stability.

lambda_reg (Regularization): Regularization prevents the model from overfitting by limiting complexity. A steady value of lambda_reg keeps the model stable across different settings of k and alpha.

Together, k controls complexity, alpha affects learning speed, and lambda_reg ensures stability. Finding the right balance leads to faster training and better predictions.

In [20]:
```
# Define the hyperparameter grid
param_grid = {
    'k': [10, 20, 30, 50],  # Smaller dimensions for faster training
    'alpha': [0.001, 0.005, 0.009],  # Higher learning rate to speed u
    'lambda_reg': [0.01]  # Smaller regularization to limit the impact
}

best_params, best_rmse = hyperparameter_tuning(ratings_matrix, param_g
```

```
Epoch: 8, MSE: 0.8283
Epoch: 9, MSE: 0.8151
Epoch: 10, MSE: 0.8037
Validation RMSE: 0.8344
Testing k=50, alpha=0.009, lambda_reg=0.01
Epoch: 1, MSE: 3.5905
Epoch: 2, MSE: 1.2306
Epoch: 3, MSE: 0.9477
Epoch: 4, MSE: 0.8817
Epoch: 5, MSE: 0.8459
Epoch: 6, MSE: 0.8214
Epoch: 7, MSE: 0.8016
Epoch: 8, MSE: 0.7842
Epoch: 9, MSE: 0.7681
Epoch: 10, MSE: 0.7526
Validation RMSE: 0.8148
Best Parameters:
{'k': 50, 'alpha': 0.009, 'lambda_reg': 0.01}
Best Validation RMSE: 0.8148
```

In this experiment, different values of k (the number of latent factors), alpha (learning rate), and lambda_reg (regularization strength) are tested to observe how they affect the error (measured by Mean Squared Error, MSE) and model performance (Validation RMSE).

Learning rate (alpha): As the learning rate increases (from 0.001 to 0.009), the model converges more quickly, meaning the error decreases faster. However, if the learning rate is too high (e.g., 0.009), the initial MSE starts higher, but the model still achieves a lower final error.

Number of latent factors (k): Increasing k (from 10 to 50) slightly improves performance, as shown by lower Validation RMSE, but the improvement becomes smaller at higher values of k.

Stability of errors: The MSE tends to decrease steadily over epochs, regardless of the parameter combination. For higher values of k and alpha, the final Validation RMSE improves, meaning better prediction accuracy.

Overall, alpha = 0.009 and k = 30 or 50 seem to provide the best balance between fast convergence and low error. Regularization (lambda_reg = 0.01) is kept constant and appears to help maintain stability without overfitting.

Collaborative filtering alone is already performing well (as indicated by a low MSE), adding content-based filtering may not significantly improve performance. If the content-based model is more complex or trained on the same data, it may lead to overfitting.

In [67]:
```
class MetaHybridRecommender:
    def __init__(self, ratings_matrix, cosine_sim, k, alpha, lambda_re
        self.ratings_matrix = ratings_matrix
        self.cosine_sim = cosine_sim
        self.cf_model = MatrixFactorizationWithBias(ratings_matrix, k,

    def train(self):
        self.cf_model.train()

    def recommend_all_users(self, num_recommendations=5):

        cf_predictions = self.cf_model.predict_all_users()

        all_recommendations = []
        for user_id in range(self.ratings_matrix.shape[0]):

            content_predictions = self.get_content_based_recommendatic

            combined_predictions = self.combine_predictions(cf_predict

            recommended_indices = np.argsort(combined_predictions)[::-
            all_recommendations.append(recommended_indices)

        return all_recommendations
```

```
    def get_content_based_recommendations(self, user_id):

        user_ratings = self.ratings_matrix[user_id]
        rated_indices = np.argwhere(~np.isnan(user_ratings)).flatten()


        weighted_scores = np.zeros(self.ratings_matrix.shape[1])
        for idx in rated_indices:
            sim_scores = self.cosine_sim[idx]
            weighted_scores += sim_scores * user_ratings[idx]


        weighted_scores /= len(rated_indices) if len(rated_indices) >
        return weighted_scores

    def combine_predictions(self, cf_scores, content_scores, cf_weight
        return (cf_weight * cf_scores) + (content_weight * content_sco

    def predict_all_users(self):

        return np.array([self.cf_model.predict(i, j) for i in range(se

    def compute_mse_rmse(self):

        predicted_ratings = self.predict_all_users()

        actual_ratings = self.ratings_matrix.flatten()
        predicted_ratings = predicted_ratings.flatten()

        mse = np.mean((actual_ratings - predicted_ratings) ** 2)


        rmse = np.sqrt(mse)

        return mse, rmse
```

Due to the limitation for computational resouces and data resources, it leads tp overfitting when combining with content-based recommender system.