```
# Worksheet 17

Name: Shangyuan Chen
UID: U58846351

### Topics

- Recommender Systems

### Recommender Systems

In the example in class of recommending movies to users we used
rating as a measure of similarity between users and movies and
predicted rating for a user is a proxy for how highly a movie
recommended. So the higher the predicted rating for a user, the
recommendation it would be.

a) Consider a streaming platform that only has "like" or "dislik
rating). Describe how you would build a recommender system in th
```

Worksheet 17

Name: Shangyuan Chen UID: U58846351

Topics

- Recommender Systems

Recommender Systems

In the example in class of recommending movies to users we used the movie rating as a measure of similarity between users and movies and thus the predicted rating for a user is a proxy for how highly a movie should be recommended. So the higher the predicted rating for a user, the higher a recommendation it would be.

a) Consider a streaming platform that only has "like" or "dislike" (no 1-5 rating). Describe how you would build a recommender system in this case.

1. **Data Collection**

- **Binary Feedback**: Collect data where users have liked or disliked various movies. This data forms the basis of understanding user preferences.
- **User and Item Features**: Gather additional data about users (e.g., demographics, viewing habits) and movies (e.g., genre, director, release year).

2. **Choosing a Model Type**

You can use either a collaborative filtering approach, a content-based approach, or a hybrid approach:

- **Collaborative Filtering (CF)**:
  - **User-Based CF**: Predict whether a user will like a movie based on how similar users have liked or disliked it. Similarity between users can be calculated using metrics like Jaccard similarity, which is well-suited for binary data.
  - **Item-Based CF**: Recommend movies that are similar to other movies the user has liked. Similarity between items can be calculated based on the pattern of likes and dislikes across all users.
- **Content-Based Filtering**:
  - Use the features of the movies (such as genres or directors) that a user has liked to recommend similar movies. Machine learning models can predict likes/dislikes based on these features.
- **Hybrid Approaches**:
  - Combine both collaborative and content-based methods to leverage the strengths of both, potentially improving recommendation quality and accuracy.

3. **Similarity Metrics for Binary Data**

- **Jaccard Similarity**: Particularly useful for binary data. It measures the similarity between two sets (e.g., the sets of movies liked by two users) and is defined as the size of the intersection divided by the size of the union of the sample sets.
- **Cosine Similarity**: Though typically used for continuous data, it can be adapted for binary data by treating likes as 1s and dislikes as 0s.

4. **Machine Learning Models**

- **Logistic Regression**: Given features of a user and a movie, predict the likelihood of a like or dislike.
- **Neural Networks**: More complex models that can learn non-linear relationships between user and movie features and the outcome (like/dislike).
- **Decision Trees or Random Forests**: These can handle categorical data well and can be used to model the likelihood of a user liking a movie based on past behavior and movie features.

5. **Evaluation**

- **Accuracy**: Since the recommendations are binary, traditional classification metrics such as accuracy, precision, recall, and F1-score are appropriate.
- **A/B Testing**: Real-world testing where one group of users receives recommendations from the new model and another group receives recommendations from the old model or no recommendations. This helps in understanding the effectiveness of the recommender

system in actual usage.

6. **Implementation Considerations**

- **Scalability**: Ensure that the system can handle a large number of users and items efficiently.
- **Cold Start Problem**: For new users or movies with no data, use content-based filtering or demographic information to make initial recommendations.
- **Feedback Loop**: Implement mechanisms to continuously learn from the users' reactions to the recommendations to improve the system over time.

Using this approach, you can build a recommender system that is tailored to the binary nature of user feedback on a streaming platform, providing personalized recommendations effectively.

b) Describe 3 challenges of building a recommender system

Building a recommender system involves tackling several challenges that can affect the accuracy, efficiency, and user satisfaction of the system. Here are three significant challenges:

1. **Cold Start Problem**

The cold start problem refers to the difficulty in making accurate recommendations for new users or new items that have little to no historical interaction data. This is a common challenge in recommender systems, particularly in scenarios where new content is frequently added or there is high user turnover.

- **New Users**: Without historical data, the system struggles to determine the preferences of new users. Generic recommendations might not meet their specific interests, potentially leading to a poor user experience.
- **New Items**: Similarly, new items lack user interaction data, making it difficult to recommend them accurately. This can hinder the exposure of potentially popular new content.

Solution Strategies:

- **Demographic Information**: Using user demographic data (age, location, etc.) to make initial recommendations based on similar profiles.
- **Content-Based Filtering**: Recommending items based on their characteristics rather than user interactions, which is particularly useful for new items.
- **Hybrid Models**: Combining multiple recommendation strategies to mitigate the lack of data.

2. **Scalability and Performance**

As the number of users and items grows, the data matrix used in many recommender systems becomes extremely large, which can lead to scalability issues. Managing and processing this vast amount of data efficiently while still providing real-time or near-real-time recommendations can be technically challenging.

- **Large Datasets**: Handling large-scale data requires significant computational resources, especially for model training and real-time querying.
- **Latency**: Providing timely recommendations is crucial, especially in dynamic environments like online retail or streaming services where user preferences and availabilities change rapidly.

Solution Strategies:

- **Matrix Factorization Techniques**: Such as Singular Value Decomposition (SVD) or Alternating Least Squares (ALS), which reduce the dimensionality of the recommendation problem.
- **Distributed Computing**: Leveraging platforms like Apache Spark to handle large datasets and complex computations across a distributed system.
- **Approximation Algorithms**: Using techniques like locality-sensitive hashing (LSH) to approximate similarity calculations, reducing computation time without drastically compromising quality.

3. **Diversity and Serendipity**

A recommender system that continually suggests items too similar to those the user has already experienced can lead to a dull user experience. Ensuring diversity (offering a wide range of options) and serendipity (unexpected but relevant recommendations) is essential to keep users engaged and satisfied.

- **Over-specialization**: Systems might become too niche, constantly recommending items too similar to those a user has previously liked.
- **User Exploration**: Encouraging users to explore new genres or categories without alienating them can be challenging.

Solution Strategies:

- **Exploration vs. Exploitation**: Implementing algorithms that balance between exploiting known user preferences and exploring new territories to enhance user discovery.
- **Diverse Recommendation Algorithms**: Incorporating mechanisms to intentionally diversify the recommendations, such as adding randomization or prioritizing under-recommended items.

- **Multi-criteria Recommendations**: Using additional dimensions of data, not just past interactions, to enrich the recommendation context and outcome.

These challenges highlight the complexities of designing and implementing a robust recommender system. Addressing them effectively requires a thoughtful combination of technology, algorithmic creativity, and an understanding of user behavior.

c) Why is SVD not an option for collaborative filtering?

Challenge of Missing Data

In collaborative filtering, the data matrix (usually representing users as rows and items as columns) is typically very sparse, meaning most of the entries are missing. This sparsity arises because any given user interacts with only a small subset of available items (e.g., rates or purchases them), leaving most of the matrix entries as unknown.

Key Issue with Traditional SVD:

- **Complete Data Requirement**: Traditional SVD requires a complete matrix with no missing entries to perform the decomposition. It decomposes a matrix (A) into three matrices (U), (\Sigma), and (V^T) such that (A = U \Sigma V^T), where (U) and (V) contain the left and right singular vectors, and (\Sigma) contains the singular values.
- **Handling Missing Entries**: In the context of collaborative filtering, the missing entries are not zero but are genuinely unknown, which means they can't simply be treated as zero (doing so would heavily bias the model towards assuming non-interaction as negative feedback).

Computational Practicality

- **Imputation before SVD**: One way to use traditional SVD in such scenarios is to fill in the missing entries with some form of imputation (e.g., overall mean, user/item mean ratings). However, this approach can introduce significant biases and distort the latent structures in the data, potentially leading to misleading recommendations.
- **Computational Cost**: Even if imputation were a viable solution, the computational expense of applying SVD on very large matrices (which becomes even larger after imputation) can be prohibitive in real-world systems.

Better Alternatives

Given these issues with traditional SVD:

- **Probabilistic and Incremental Approaches**: More robust methods such as Probabilistic Matrix Factorization or algorithms designed for sparse data, like SVD++, are generally preferred. These methods are specifically tailored to handle missing data by optimizing only over the known entries.
- **Alternating Least Squares (ALS)**: In methods like ALS, the optimization alternates between fixing user factors and solving for item factors (and vice versa), which efficiently handles the sparsity by breaking down the problem.

Conclusion

Traditional SVD isn't typically used directly in collaborative filtering mainly because it doesn't naturally accommodate the sparse, incomplete nature of user-item matrices in these scenarios. The requirement to handle missing data appropriately without introducing substantial bias or computational inefficiencies leads to the preference for specialized adaptations or entirely different approaches in practical recommendation systems.

d) Use the code below to train a recommender system on a dataset of amazon movies

```
# Note: requires py3.10
import findspark
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, confusion_matrix

from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext
from pyspark.ml.recommendation import ALS
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

findspark.init()
conf = SparkConf()
conf.set("spark.executor.memory","28g")
conf.set("spark.driver.memory", "28g")
conf.set("spark.driver.cores", "8")
sc = SparkContext.getOrCreate(conf)
spark = SparkSession.builder.getOrCreate()
```

```
init_df = pd.read_csv("./train.csv").dropna()
init_df['UserId_fact'] = init_df['UserId'].astype('category').cat.codes
init_df['ProductId_fact'] = init_df['ProductId'].astype('category').cat.codes

# Split training set into training and testing set
X_train_processed, X_test_processed, Y_train, Y_test = train_test_split(
                init_df.drop(['Score'], axis=1),
                init_df['Score'],
                test_size=1/4.0,
                random_state=0
        )

X_train_processed['Score'] = Y_train
df = spark.createDataFrame(X_train_processed[['UserId_fact', 'ProductId_fact', 'Score']])
als = ALS(
        userCol="UserId_fact",
        itemCol="ProductId_fact",
        ratingCol="Score",
        coldStartStrategy="drop",
        nonnegative=True,
        rank=100
)
# param_grid = ParamGridBuilder().addGrid(
                # als.rank, [10, 50]).addGrid(
                # als.regParam, [.1]).addGrid(
                # # als.maxIter, [10]).build()
# evaluator = RegressionEvaluator(
                # metricName="rmse",
                # labelCol="Score",
                # # predictionCol="prediction")
# cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid, evaluator=evaluator, numFolds=3, parallelism = 6)
# cv_fit = cv.fit(df)
# rec_sys = cv_fit.bestModel

rec_sys = als.fit(df)
# rec_sys.save('rec_sys.obj') # so we don't have to re-train it
rec = rec_sys.transform(spark.createDataFrame(X_test_processed[['UserId_fact', 'ProductId_fact']])).toPandas()
X_test_processed['Score'] = rec['prediction'].values.reshape(-1, 1)

print("Kaggle RMSE = ", mean_squared_error(X_test_processed['Score'], Y_test, squared=False))

cm = confusion_matrix(Y_test, X_test_processed['Score'], normalize='true')
sns.heatmap(cm, annot=True)
plt.title('Confusion matrix of the classifier')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```