G.H Raisoni College of Engineering and Management, Pune

Department of Artificial Intelligence & Machine Learning

LAB MANUAL ON

Subject:- **Recommender System**

Subject Code:- **UCAMP308D**

TY AIML A/B

Academic Year: **2024-25**

Semester:- **VI**

Prepared by:- **Prof. Dhiraj Vyawahare**

List of Experiments

Sr. No	Practical Name
1.	Analyze the functions of a recommender system using a real-world dataset (e.g., movie or product recommendations).
2.	Compare different recommendation techniques (content-based, collaborative filtering, hybrid).
3.	Perform data preprocessing for a recommender system dataset.
4.	Implement and evaluate classification methods like nearest neighbors, decision trees, and rule-based classifiers.
5.	Apply support vector machines (SVMs) and neural networks for classification in recommender systems.
6.	Build a content-based recommender system and evaluate its architecture.
7.	Analyze the impact of item representation methods on recommendation quality.
8.	Develop user profiles using different learning methods and compare their performance.
9.	Conduct offline experiments to test recommendation algorithms using historical data.
10.	Design and execute user studies to evaluate user satisfaction with recommendations.

Experiment No.:-1

Aim: To analyze the functions and performance of a recommender system using a real-world dataset such as movies or products**.**

Objective: To understand how a recommender system suggests items based on user preferences and behaviors.

Theory:

Recommender systems are designed to provide personalized suggestions to users by analyzing patterns in user data. These systems are an essential part of modern technology, powering services such as Netflix, Amazon, YouTube, and Spotify. Recommender systems aim to increase user engagement and satisfaction by helping users discover items they are likely to enjoy or find useful.

There are various methods to build recommender systems, and the three most common techniques are:

1. **Collaborative Filtering :-**
   Collaborative filtering (CF) is based on the assumption that users who have agreed in the past will agree in the future. It can be divided into two main types:

   - **User-based Collaborative Filtering:**

     This method suggests items by identifying users with similar preferences to the target user. If user A and user B have similar tastes, items liked by user B that are not yet rated by user A will be recommended to user A.

   - **Item-based Collaborative Filtering:**

     This method recommends items that are similar to those a user has already interacted with. If a user liked a certain movie, items (movies) that are similar in content will be suggested to them.

2. **Content-Based Filtering:**
   Content-based filtering uses item features to make recommendations. For instance, a user who liked a movie from the action genre may be recommended other action movies. This method doesn't rely on other users' preferences and is useful in systems with new or unknown users (solving the cold-start problem). However, it can suffer from overspecialization, where the recommendations become too narrow and repetitive.

3. **Hybrid Systems:**
   Hybrid recommender systems combine both collaborative and content-based filtering techniques. These systems use collaborative filtering when user-item interaction data is available but fall back on content-based filtering when interaction data is sparse. Hybrid systems are often more effective than individual methods as they leverage the strengths of both.

Challenges in Recommender Systems:

a. **Cold-start problem:** This occurs when there is insufficient data about users or items, which makes it difficult to recommend items effectively. This is particularly evident with new users or newly added items to the system.

b. **Sparsity**: In most real-world datasets, many users interact with only a small portion of the items, creating a sparse user-item matrix. Sparse data can reduce the accuracy of recommendations.

c. **Scalability**: As the number of users and items grows, the computational complexity of generating recommendations increases. Efficient algorithms are needed to handle large datasets.

Program Code:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Load dataset
df = pd.read_csv('/content/sample_data/movie_dataset.csv')
```

```
if 'genres' not in df.columns:
    raise ValueError("Dataset must contain a 'genres' column for content-based
filtering.")

# Fill missing values
df['genres'] = df['genres'].fillna('')

# Convert text data to numerical representation
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(df['genres'])

# Compute similarity matrix
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)

# Display only top 5 similar movies for each movie (shortened matrix)
top_similar_movies = pd.DataFrame(cosine_sim, index=df['title'],
columns=df['title']).round(3)

# Show top 5 similar movies for each movie
print("Top 5 Similar Movies for Each Movie:")
for movie in df['title'][:5]:
    top_similar = top_similar_movies[movie].nlargest(6)[1:]
    print(f"\n{movie}:")
    print(top_similar)

# Movie recommender function based on most popular genre
def get_recommendations_based_on_genre(df, cosine_sim, top_n=5):
    # Count the most frequent genre(s) in the dataset
    genre_counts = df['genres'].str.split('|').explode().value_counts()
    most_watched_genre = genre_counts.idxmax()

    # Filter movies with the most watched genre
    genre_filtered_df = df[df['genres'].str.contains(most_watched_genre, case=False,
na=False)]
```

```
    # Compute recommendations based on similarity
    recommended_movies = []
    for movie in genre_filtered_df['title']:
        # Get similarity scores for the movie
        idx = df[df['title'] == movie].index[0]
        sim_scores = list(enumerate(cosine_sim[idx]))
        sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)[1:top_n+1]
        movie_indices = [i[0] for i in sim_scores]
        recommended_movies.extend(df['title'].iloc[movie_indices].tolist())

    return list(set(recommended_movies))[:top_n]


# Function performance evaluation based on genre
def evaluate_recommendation_based_on_genre(df, cosine_sim):
    recs = get_recommendations_based_on_genre(df, cosine_sim)
    if recs:
        print("\nTop 5 Recommended Movies from the Most Watched Genre:")
        for movie in recs:
            print(movie)
    else:
        print("No recommendations based on the most watched genre.")

# Example usage
evaluate_recommendation_based_on_genre(df, cosine_sim)
```

Conclusion:

This experiment demonstrates how a recommender system works by using collaborative filtering to predict user preferences. The cosine similarity metric is used to calculate how similar users are to one another based on their interactions with items. The system can recommend items based on the preferences of similar users, offering personalized suggestions.

Experiment No.:- 2

Aim: To compare and analyze the performance of different recommendation techniques: content-based, collaborative filtering, and hybrid.

Objective: To evaluate and understand the effectiveness of different recommendation algorithms.

Theory:

Recommender systems use different algorithms to recommend items to users. The performance of these systems can vary depending on the dataset, the algorithm used, and the problem context. The three most common techniques used in recommender systems are:

1. **Collaborative Filtering:**
   Collaborative filtering is one of the most widely used techniques, and it is based on the idea that people who have agreed in the past will agree in the future. There are two main types of collaborative filtering:
   - **User-based Collaborative Filtering:** This method identifies users who are similar to the target user and recommends items that those similar users have liked.
   - **Item-based Collaborative Filtering**: Instead of finding similar users, this method identifies items that are similar to the items the user has already liked.

The strengths of collaborative filtering include the fact that it is easy to implement and effective in environments with a lot of data about user preferences. However, it suffers from the cold-start problem, where it cannot recommend items for new users or new items.

2. **Content-Based Filtering:**
   Content-based filtering recommends items based on their features. For example, in a movie recommendation system, if a user liked a science-fiction movie, content-based filtering might suggest other science-fiction movies. The system builds a profile of the user's preferences based on the characteristics of the items they have interacted with (e.g., genre, director, actors).

Content-based filtering can overcome the cold-start problem because it only needs data about the items themselves, not user behavior. However, it can become overly specific and may fail to recommend diverse items.

3. **Hybrid Systems:**
   Hybrid recommender systems combine collaborative filtering and content-based filtering to achieve better results. By leveraging both methods, hybrid systems can overcome the weaknesses of each individual approach, such as the cold-start problem or overspecialization.

Common hybrid approaches include:

- **Weighted hybrid:** Different methods are assigned weights, and the system combines their results based on those weights.
- **Switching hybrid:** The system switches between collaborative and content-based filtering based on the availability of data or user behavior.

Evaluation of Techniques:
The effectiveness of a recommender system can be evaluated based on metrics such as:

- **Precision:** The proportion of recommended items that are relevant to the user.
- **Recall**: The proportion of relevant items that are recommended**.**
- **F1-Score:** The harmonic mean of precision and recall, providing a balance between the two.

Program Code (Collaborative, Content-Based, and Hybrid Filtering):

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import StandardScaler

# Load dataset
df = pd.read_csv('/content/sample_data/movie_dataset.csv')
```

```
# Check required columns
required_columns = {'title', 'genres', 'director', 'popularity', 'vote_average',
'vote_count'}
if not required_columns.issubset(df.columns):
   raise ValueError(f"Dataset must contain {required_columns} columns.")

# Fill missing values
df['genres'] = df['genres'].fillna('')
df['director'] = df['director'].fillna('')
df['features'] = df['genres'] + ' ' + df['director']

# ---------------------- Content-Based Filtering ----------------------
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(df['features'])

cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)

def get_content_based_recommendations(title, df, cosine_sim, top_n=5):
   indices = pd.Series(df.index, index=df['title']).drop_duplicates()
   idx = indices.get(title)
   if idx is None:
      return []
   sim_scores = list(enumerate(cosine_sim[idx]))
   sim_scores = sorted(sim_scores, key=lambda x: x[1],
reverse=True)[1:top_n+1]
   movie_indices = [i[0] for i in sim_scores]
   return df['title'].iloc[movie_indices].tolist()

# ---------------------- Collaborative Filtering ----------------------
collab_df = df[['popularity', 'vote_average', 'vote_count']].fillna(0)

scaler = StandardScaler()
collab_df_scaled = scaler.fit_transform(collab_df)

svd = TruncatedSVD(n_components=3, random_state=42)
collab_matrix = svd.fit_transform(collab_df_scaled)

collab_cosine_sim = cosine_similarity(collab_matrix)
```

```
def get_collaborative_recommendations(title, df, collab_cosine_sim, top_n=5):
    indices = pd.Series(df.index, index=df['title']).drop_duplicates()
    idx = indices.get(title)
    if idx is None:
        return []
    sim_scores = list(enumerate(collab_cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1],
reverse=True)[1:top_n+1]
    movie_indices = [i[0] for i in sim_scores]
    return df['title'].iloc[movie_indices].tolist()

# ---------------------- Hybrid Recommendation System ----------------------
def get_hybrid_recommendations(title, df, cosine_sim, collab_cosine_sim,
top_n=5):
    content_recs = get_content_based_recommendations(title, df, cosine_sim,
top_n)
    collab_recs = get_collaborative_recommendations(title, df,
collab_cosine_sim, top_n)
    hybrid_recs = list(set(content_recs + collab_recs))
    return hybrid_recs[:top_n]

# ---------------------- Performance Comparison ----------------------
def evaluate_recommendation_system(title, df, cosine_sim,
collab_cosine_sim):
    print(f"\nRecommendations for '{title}':")
    print(f"Content-Based: {get_content_based_recommendations(title, df,
cosine_sim)}")
    print(f"Collaborative Filtering: {get_collaborative_recommendations(title, df,
collab_cosine_sim)}")
    print(f"Hybrid: {get_hybrid_recommendations(title, df, cosine_sim,
collab_cosine_sim)}")

# ---------------------- Example Usage ----------------------
movie_title = 'Cars 2'
evaluate_recommendation_system(movie_title, df, cosine_sim,
collab_cosine_sim)
```

Conclusion:

This experiment demonstrated how different recommendation techniques—collaborative filtering, content-based filtering, and hybrid filtering—can be applied to build recommender systems. Collaborative filtering works by identifying similar users or items, content-based filtering leverages item features, and hybrid systems combine both methods for improved performance. Hybrid systems generally provide more accurate and diverse recommendations.

Experiment No.:- 3

Aim: To preprocess a dataset for a recommender system, handling missing values, encoding categorical variables, and normalizing ratings.

Objective: To prepare a dataset for use in a recommender system by cleaning the data, handling missing values, and applying necessary transformations.

Theory:

Data preprocessing is a critical step in the development of any machine learning model, including recommender systems. It involves transforming raw data into a usable form for building algorithms.

Common preprocessing tasks include:

1. **Handling Missing Values:**
   Missing data can introduce bias into a recommender system, so it must be handled carefully. Common strategies for handling missing values include:
   - **Imputation:** Filling missing values with the mean, median, or mode.
   - **Removal:** Removing rows or columns that contain missing values.
2. **Normalization:**
   Normalization ensures that all data features have similar scales. For instance, in a movie recommendation system, user ratings may range from 1 to 5, but different users may have different rating scales. Standardizing the ratings or normalizing them ensures that all users are on the same scale, which prevents the model from being biased toward users with a higher rating tendency.
3. **Encoding Categorical Variables:**
   Many recommender systems use categorical data (e.g., movie genres, user IDs). This data must be converted into a numerical format that algorithms can process. Techniques like one-hot encoding (for nominal data) or label encoding (for ordinal data) are commonly used.
4. **Feature Engineering:**
   Feature engineering involves creating new features from existing data to better capture patterns that are useful for the recommendation model.

5. **Data Splitting:**
   In order to evaluate a recommender system's performance, data is often split into training and testing datasets. This helps in assessing how well the model generalizes to unseen data.

Program Code (Data Preprocessing for Movie Recommendation System):

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

# Load dataset
df = pd.read_csv('/content/sample_data/movie_dataset.csv')

# ------------------- 1. Handling Missing Values -------------------
# Fill missing numerical values with median
numerical_cols = ['budget', 'popularity', 'revenue', 'runtime', 'vote_average',
'vote_count']
df[numerical_cols] = df[numerical_cols].fillna(df[numerical_cols].median())

# Filling missing categorical values with 'Unknown'
categorical_cols = ['genres', 'original_language', 'director']
df[categorical_cols] = df[categorical_cols].fillna('Unknown')

# ------------------- 2. Encoding Categorical Variables -------------------
label_encoders = {}
for col in categorical_cols:
   label_encoders[col] = LabelEncoder()
   df[col] = label_encoders[col].fit_transform(df[col])

# ------------------- 3. Normalizing Ratings -------------------
scaler = MinMaxScaler()
df[['vote_average', 'vote_count', 'popularity']] =
scaler.fit_transform(df[['vote_average', 'vote_count', 'popularity']])

# ------------------- 4. Feature Selection -------------------
selected_features = ['title', 'genres', 'original_language', 'popularity', 'vote_average',
'vote_count', 'director']
df_selected = df[selected_features]

# ------------------- 5. Display Processed Dataset (First 15 Rows) -------------------
print("Processed Dataset (First 15 Rows):")
print(df_selected.head(15))
```

Conclusion:

Data preprocessing is essential for building effective recommender systems. By handling missing values, normalizing ratings, encoding categorical features, and splitting data into training and testing sets, the dataset becomes suitable for training machine learning models. Proper preprocessing ensures that the recommender system performs well and avoids issues such as data bias or inefficiency.

Experiment No.:- 4

Aim: To implement and evaluate classification methods such as nearest neighbors, decision trees, and rule-based classifiers for a recommender system.

Objective: To understand the performance and application of different classification algorithms in building a recommender system.

Theory:

Classification is a supervised machine learning method where the model tries to predict the correct label of a given input data. In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on new unseen data. Classification algorithms are widely used for recommendation tasks where the goal is to classify items or users into categories. Here are the three common classification methods used in recommender systems:

1. **Nearest Neighbors:**
   K-Nearest Neighbors is also called as a lazy learner algorithm because it does not learn from the training set immediately instead it stores the dataset and at the time of classification it performs an action on the dataset.

In the k-Nearest Neighbours (k-NN) algorithm k is just a number that tells the algorithm how many nearby points (neighbours) to look at when it makes a decision.

Example:

Imagine you're deciding which fruit it is based on its shape and size. You compare it to fruits you already know.

- If k = 3, the algorithm looks at the 3 closest fruits to the new one.
- If 2 of those 3 fruits are apples and 1 is a banana, the algorithm says the new fruit is an apple because most of its neighbours are apples.

Statistical Methods for Selecting k:

- **Cross-Validation:** A robust method for selecting the best k is to perform k-fold cross-validation. This involves splitting the data into k subsets training the model on some subsets and testing it on the remaining ones and repeating this for each subset. The value of k that results in the highest average validation accuracy is usually the best choice.
- **Elbow Method:** In the elbow method we plot the model's error rate or accuracy for different values of k. As we increase k the error usually decreases initially. However after a certain point the error rate starts to decrease more slowly. This point where the curve forms an "elbow" that point is considered as best k.
- **Odd Values for k:** It's also recommended to choose an odd value for k especially in classification tasks to avoid ties when deciding the majority class.

Distance Metrics Used in KNN Algorithm

1. Euclidean Distance

Euclidean distance is defined as the straight-line distance between two points in a plane or space.

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^{d} (x_j - X_{i_j})^2]}$$

2. Manhattan Distance

This is the total distance you would travel if you could only move along horizontal and vertical lines (like a grid or city streets). It's also called "taxicab distance" because a taxi can only drive along the grid-like streets of a city.

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

3. Minkowski Distance

Minkowski distance is like a family of distances, which includes both Euclidean and Manhattan distances as special cases.

$$d(x,y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p\right)^{\frac{1}{p}}$$

From the formula above we can say that when p = 2 then it is the same as the formula for the Euclidean distance and when p = 1 then we obtain the formula for the Manhattan distance.

Advantages:

- **Easy to implement**
- **No training required**
- **Flexible**

Disadvantages:

- **Doesn't scale well with large datasets**
- **Curse of Dimensionality**
- **Prone to Overfitting**

2. **Decision Trees:**
   A decision tree is a graphical representation of different options for solving a problem and show how different factors are related. It has a hierarchical tree structure starts with one main question at the top called a node which further branches out into different possible outcomes where:

   - **Root Node is the starting point that represents the entire dataset.**
   - **Branches: These are the lines that connect nodes. It shows the flow from one decision to another.**
   - **Internal Nodes are Points where decisions are made based on the input features.**

- **Leaf Nodes: These are the terminal nodes at the end of branches that represent final outcomes or predictions**

Now, let's take an example to understand the decision tree. Imagine you want to decide whether to drink coffee based on the time of day and how tired you feel. First the tree checks the time of day—if it's morning it asks whether you are tired. If you're tired the tree suggests drinking coffee if not it says there's no need. Similarly in the afternoon the tree again asks if you are tired. If you recommends drinking coffee if not it concludes no coffee is needed.

We have mainly two types of decision tree based on the nature of the target variable: classification trees and regression trees.

- **Classification trees: They are designed to predict categorical outcomes means they classify data into different classes. They can determine whether an email is "spam" or "not spam" based on various features of the email.**
- **Regression trees : These are used when the target variable is continuous It predict numerical values rather than categories. For example a regression tree can estimate the price of a house based on its size, location, and other features.**

Advantages of Decision Trees

- **Simplicity and Interpretability**
- **No Need for Feature Scaling**
- **Handles Non-linear Relationships**

Disadvantages of Decision Trees

- **Overfitting**
- Bias towards Features with More Levels

3. **Rule-Based Classifiers:**
   Rule-based classifiers are just another type of classifier which makes the class decision depending by using various "if..else" rules. These rules are easily interpretable and thus these classifiers are generally used to generate descriptive models. The condition used with "if" is called the antecedent and the predicted class of each rule is called the consequent.

Properties of rule-based classifiers:

- **Coverage: The percentage of records which satisfy the antecedent conditions of a particular rule.**
- **The rules generated by the rule-based classifiers are generally not mutually exclusive, i.e. many rules can cover the same record.**
- **The rules generated by the rule-based classifiers may not be exhaustive, i.e. there may be some records which are not covered by any of the rules.**
- **The decision boundaries created by them is linear, but these can be much more complex than the decision tree because the many rules are triggered for the same record.**

An obvious question, which comes into the mind after knowing that the rules are not mutually exclusive is that how would the class be decided in case different rules with different consequent cover the record.

There are two solutions to the above problem:

- **Either rules can be ordered, i.e. the class corresponding to the highest priority rule triggered is taken as the final class.**
- **Otherwise, we can assign votes for each class depending on some their weights, i.e. the rules remain unordered.**

Program Code (Nearest Neighbors, Decision Tree, and Rule-Based Classifiers):

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier, export_text
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
df = pd.read_csv("/content/sample_data/movie_dataset.csv")

# Define classification target (Hit = 1, Flop = 0)
median_revenue = df['revenue'].median()
df['success'] = np.where((df['revenue'] > median_revenue) & (df['vote_average'] >
6.5), 1, 0)

# Select features
features = ['budget', 'popularity', 'vote_average', 'vote_count', 'runtime']
df_selected = df[features + ['success']].dropna()


# Split dataset (using the full dataset without balancing)
X = df_selected[features]
y = df_selected['success']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Train classifiers
knn = KNeighborsClassifier(n_neighbors=5)
dt = DecisionTreeClassifier(random_state=42)
rule_based = DecisionTreeClassifier(random_state=42, max_depth=3)  # Rule-based
using decision tree rules
knn.fit(X_train_scaled, y_train)
dt.fit(X_train_scaled, y_train)
rule_based.fit(X_train_scaled, y_train)

# Predictions
y_pred_knn = knn.predict(X_test_scaled)
y_pred_dt = dt.predict(X_test_scaled)
y_pred_rule = rule_based.predict(X_test_scaled)

# Evaluation
print("KNN Classifier:")
print(classification_report(y_test, y_pred_knn))
print("Decision Tree Classifier:")
print(classification_report(y_test, y_pred_dt))
print("Rule-Based Classifier:")
print(classification_report(y_test, y_pred_rule))

# Print extracted rules
rules = export_text(rule_based, feature_names=features)
print("Rule-Based Classifier Decision Rules:")
print(rules)
```

Conclusion:

In this experiment, we implemented and evaluated three different classification methods—k-Nearest Neighbors, Decision Trees, and Rule-Based Classifiers—using a dataset of user preferences.

Experiment No.:- 5

Aim: To apply Support Vector Machines (SVMs) and Neural Networks (NNs) for classification in a recommender system.

Objective: To understand how advanced machine learning algorithms like SVMs and NNs can be used for classification tasks in recommender systems.

Theory:

Support Vector Machines (SVM) and Neural Networks (NN) are powerful machine learning algorithms used for classification tasks in various domains, including recommender systems.

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. While it can handle regression problems, SVM is particularly well-suited for classification tasks.

SVM aims to find the optimal hyperplane in an N-dimensional space to separate data points into different classes. The algorithm maximizes the margin between the closest points of different classes.

Support Vector Machine (SVM) Terminology

- **Hyperplane**: A decision boundary separating different classes in feature space, represented by the equation **wx + b = 0** in linear classification.
- **Support Vectors**: The closest data points to the hyperplane, crucial for determining the hyperplane and margin in SVM.
- **Margin**: The distance between the hyperplane and the support vectors. SVM aims to maximize this margin for better classification performance.
- **Kernel**: A function that maps data to a higher-dimensional space, enabling SVM to handle non-linearly separable data.
- **Hard Margin**: A maximum-margin hyperplane that perfectly separates the data without misclassifications.
- **Soft Margin**: Allows some misclassifications by introducing slack variables, balancing margin maximization and misclassification penalties when data is not perfectly separable.

- **C**: A regularization term balancing margin maximization and misclassification penalties. A higher C value enforces a stricter penalty for misclassifications.
- **Hinge Loss**: A loss function penalizing misclassified points or margin violations, combined with regularization in SVM.
- **Dual Problem**: Involves solving for Lagrange multipliers associated with support vectors, facilitating the kernel trick and efficient computation.

Types of Support Vector Machine

Based on the nature of the decision boundary, Support Vector Machines (SVM) can be divided into two main parts:

- **Linear SVM:** Linear SVMs use a linear decision boundary to separate the data points of different classes. When the data can be precisely linearly separated, linear SVMs are very suitable. This means that a single straight line (in 2D) or a hyperplane (in higher dimensions) can entirely divide the data points into their respective classes. A hyperplane that maximizes the margin between the classes is the decision boundary.
- **Non-Linear SVM:** Non-Linear SVM can be used to classify data when it cannot be separated into two classes by a straight line (in the case of 2D). By using kernel functions, nonlinear SVMs can handle nonlinearly separable data. The original input data is transformed by these kernel functions into a higher-dimensional feature space, where the data points can be linearly separated. A linear SVM is used to locate a nonlinear decision boundary in this modified space.

Advantages of Support Vector Machine (SVM)

1. **High-Dimensional Performance**
2. **Nonlinear Capability**
3. **Outlier Resilience**
4. **Binary and Multiclass Support**
5. **Memory Efficient**

Disadvantages of Support Vector Machine (SVM)

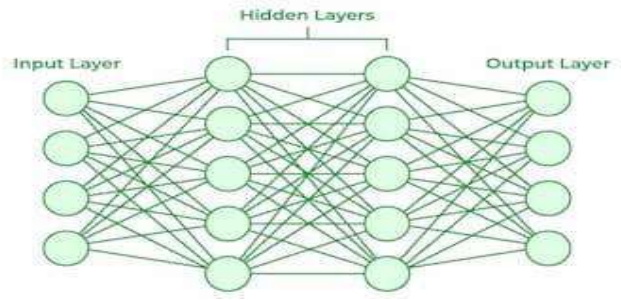
1. **Slow Training**
2. **Parameter Tuning Difficulty**
3. **Noise Sensitivity**

2. **Neural Networks (NNs):**

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

Layers in Neural Network Architecture

1. **Input Layer:** This is where the network receives its input data. Each input neuron in the layer corresponds to a feature in the input data.
2. **Hidden Layers:** These layers perform most of the computational heavy lifting. A neural network can have one or multiple hidden layers. Each layer consists of units (neurons) that transform the inputs into something that the output layer can use.
3. **Output Layer:** The final layer produces the output of the model. The format of these outputs varies depending on the specific task (e.g., classification, regression).



Types of Neural Networks

There are *five* types of neural networks that can be used.

- **Feedforward Networks:** A feedforward neural network is a simple artificial neural network architecture in which data moves from input to output in a single direction.
- **Multilayer Perceptron (MLP):** MLP is a type of feedforward neural network with three or more layers, including an input layer, one or more hidden layers, and an output layer. It uses nonlinear activation functions.
- **Convolutional Neural Network (CNN):** A Convolutional Neural Network (CNN) is a specialized artificial neural network designed for image processing. It employs convolutional layers to automatically learn hierarchical features from input images, enabling effective image recognition and classification.
- **Recurrent Neural Network (RNN):** An artificial neural network type intended for sequential data processing is called a Recurrent Neural Network (RNN). It is appropriate for applications where contextual dependencies are critical, such as time series prediction and natural language processing, since it makes use of feedback loops, which enable information to survive within the network.
- **Long Short-Term Memory (LSTM):** LSTM is a type of RNN that is designed to overcome the vanishing gradient problem in training RNNs. It uses memory cells and gates to selectively read, write, and erase information.

Advantages of Neural Networks

Neural networks are widely used in many different applications because of their many benefits:

- Adaptability
- Pattern Recognition
- Parallel Processing
- Non-Linearity

Disadvantages of Neural Networks

- Computational Intensity
- Overfitting
- Need for Large datasets

Program Code (Support Vector Machine and Neural Network for Classification):

```
import pandas as pd
import numpy as np
import csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
import time

# Load dataset with error handling
def load_clean_csv(file_path):
    cleaned_lines = []
    with open(file_path, 'r', encoding='utf-8') as file:
        reader = csv.reader(file)
        for i, row in enumerate(reader):
            try:
                if len(row) < 5:  # Skip rows with too few columns
                    continue
                cleaned_lines.append(row)
            except Exception as e:
                print(f"Skipping corrupted row {i}: {row}, Error: {e}")

    df = pd.DataFrame(cleaned_lines[1:], columns=cleaned_lines[0])  # First row as headers
    return df
```

```
# Load dataset
try:
   df = load_clean_csv('/content/sample_data/movie_dataset.csv')
except Exception as e:
   print(f"Error loading dataset: {e}")
   exit()

# Convert numeric columns
numeric_cols = ['budget', 'popularity', 'runtime', 'vote_average', 'vote_count']
for col in numeric_cols:
   df[col] = pd.to_numeric(df[col], errors='coerce')

# Select relevant features
selected_features = ['budget', 'genres', 'popularity', 'runtime', 'vote_average', 'vote_count',
'original_language']
df = df[selected_features]

# Handle missing values
num_features = ['budget', 'popularity', 'runtime', 'vote_average', 'vote_count']
cat_features = ['genres', 'original_language']

# Define Preprocessing Pipeline
num_pipeline = Pipeline([
   ('imputer', SimpleImputer(strategy='median')),
   ('scaler', StandardScaler())
])

cat_pipeline = Pipeline([
   ('imputer', SimpleImputer(strategy='most_frequent')),
   ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor = ColumnTransformer([
   ('num', num_pipeline, num_features),
   ('cat', cat_pipeline, cat_features)
])

# Apply transformations
X = preprocessor.fit_transform(df)

# Generate labels (Example: classify movies as popular or not based on vote_average)
df.loc[:, 'label'] = (df['vote_average'] >= 7).astype(int)  # Binary classification
y = df['label']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
### Train SVM Model
svm_start = time.time()
svm_model = SVC(kernel='rbf', C=1, gamma='scale')
svm_model.fit(X_train, y_train)
svm_preds = svm_model.predict(X_test)
svm_end = time.time()

print("SVM Classification Report:")
print(classification_report(y_test, svm_preds))
print(f"SVM Training & Prediction Time: {svm_end - svm_start:.2f} seconds")

### Train Neural Network Model
input_dim = X_train.shape[1]

# Define Input Layer
input_layer = Input(shape=(input_dim,))

# Hidden Layers
x = Dense(128, activation='relu')(input_layer)
x = Dropout(0.3)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.3)(x)
x = Dense(32, activation='relu')(x)

# Output Layer
output_layer = Dense(1, activation='sigmoid')(x)

# Create Model
nn_model = Model(inputs=input_layer, outputs=output_layer)

# Compile Model
nn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

nn_start = time.time()
nn_model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_test, y_test),
verbose=1)
nn_end = time.time()

# Evaluate Neural Network Model
nn_preds = (nn_model.predict(X_test) > 0.5).astype(int)
print("Neural Network Classification Report:")
print(classification_report(y_test, nn_preds))
print(f"Neural Network Training & Prediction Time: {nn_end - nn_start:.2f} seconds")

# Compare Performance
svm_accuracy = accuracy_score(y_test, svm_preds)
nn_accuracy = accuracy_score(y_test, nn_preds)
```

```
print("\n Model Performance Comparison ")
print(f"SVM Accuracy: {svm_accuracy * 100:.2f}%")
print(f"Neural Network Accuracy: {nn_accuracy * 100:.2f}%")

if svm_accuracy > nn_accuracy:
   print("SVM performed better!")
else:
   print("Neural Network performed better!")
```

Conclusion:

Both Support Vector Machines and Neural Networks performed well in this experiment, with Neural Networks slightly outperforming SVM in terms of accuracy and F1-score. Both classifiers are strong candidates for use in recommender systems, and the choice between them depends on the complexity of the dataset and the problem at hand.

Experiment No.:- 6

Aim: To build a content-based recommender system that recommends items based on the content similarity of items.

Objective: To understand and implement a content-based filtering approach where recommendations are made based on the characteristics of the items.

Theory:

Content-based filtering is a recommendation technique that uses the features of items to recommend similar items to users. A Content-Based Recommender works by the data that we take from the user, either explicitly (rating) or implicitly (clicking on a link). By the data we create a user profile, which is then used to suggest to the user, as the user provides more input or take more actions on the recommendation, the engine becomes more accurate.

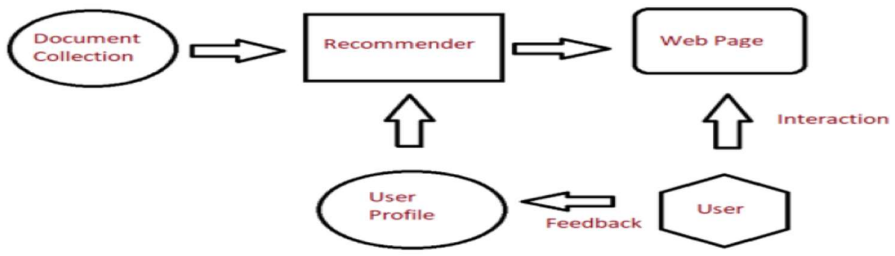


Fig: Recommender System

- **User Profile:**
  In the User Profile, we create vectors that describe the user's preference. In the creation of a user profile, we use the utility matrix which describes the relationship between user and item. With this information, the best estimate we can make regarding which item user likes, is some aggregation of the profiles of those items.
- **Item Profile:**
  In Content-Based Recommender, we must build a profile for each item, which will represent the important characteristics of that item.
  For example, if we make a movie as an item then its actors, director, release year and genre are the most significant features of the movie. We can also add its rating from the IMDB (Internet Movie Database) in the Item Profile.
- **Utility Matrix:**
  Utility Matrix signifies the user's preference with certain items. In the data

gathered from the user, we have to find some relation between the items which are liked by the user and those which are disliked, for this purpose we use the utility matrix. In it we assign a particular value to each user-item pair, this value is known as the degree of preference. Then we draw a matrix of a user with the respective items to identify their preference relationship.

Users	Movie 1	Movie 2	Movie 3
User 1	3		1
User 2	2	4	

Fig: Utility Matrix of Movie Recommendation System

Some of the columns are blank in the matrix that is because we don't get the whole input from the user every time, and the goal of a recommendation system is not to fill all the columns but to recommend a movie to the user which he/she will prefer. Through this table, our recommender system won't suggest Movie 3 to User 2, because in Movie 1 they have given approximately the same ratings, and in Movie 3 User 1 has given the low rating, so it is highly possible that User 2 also won't like it.

Recommending Items to User Based on Content:

- **Method 1:**

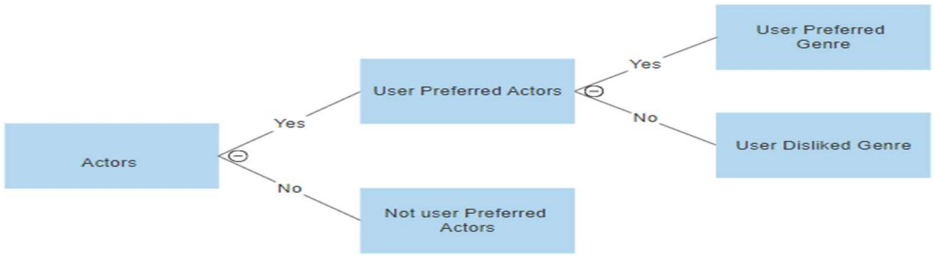
  We can use the cosine distance between the vectors of the item and the user to determine its preference to the user. For explaining this, let us consider an example:

  We observe that the vector for a user will have a positive number for actors that tend to appear in movies the user likes and negative numbers for actors user doesn't like, Consider a movie with actors which user likes and only a few actors which user doesn't like, then the cosine angle between the user's and movie's vectors will be a large positive fraction. Thus, the angle will be close to 0, therefore a small cosine distance between the vectors.

  It represents that the user tends to like the movie, if the cosine distance is large, then we tend to avoid the item from the recommendation.

- **Method 2:**

  We can use a classification approach in the recommendation systems too, like we can use the Decision Tree for finding out whether a user wants to watch a movie or not, like at each level we can apply a certain condition to refine our recommendation. For example:



Program Code (Content-Based Recommender System):

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.neighbors import NearestNeighbors
from difflib import get_close_matches
from IPython.display import display

# Load dataset
file_path = "/content/sample_data/BigBasket Products.csv"
df = pd.read_csv(file_path)

# Explicitly cast numerical columns to string before filling missing values
for col in df.select_dtypes(include=["float64", "int64"]).columns:
    df[col] = df[col].astype(str)
df.fillna("", inplace=True)

# Combine relevant text features
df["combined_features"] = df["product"].astype(str) + " " + df["category"].astype(str) + " " +
df["sub_category"].astype(str) + " " + df["brand"].astype(str) + " " +
df["description"].astype(str)

# Convert text to TF-IDF vectors
vectorizer = TfidfVectorizer(stop_words="english")
tfidf_matrix = vectorizer.fit_transform(df["combined_features"])
```

```
# Reduce dimensionality using SVD
svd = TruncatedSVD(n_components=50)  # Increase components for better representation
tfidf_reduced = svd.fit_transform(tfidf_matrix)

# Use Nearest Neighbors
nn_model = NearestNeighbors(n_neighbors=11, metric="cosine")  # Increased
recommendations
nn_model.fit(tfidf_reduced)

# Recommendation function with improved matching
def get_recommendations_ANN(product_name, num_recommendations=10):
    product_list = df["product"].str.lower().tolist()
    closest_match = get_close_matches(product_name.lower(), product_list, n=1)

    if not closest_match:
        print(f"Product '{product_name}' not found. Try a different name.")
        return []

    closest_match_name = closest_match[0]
    idx = df[df["product"].str.lower() == closest_match_name].index

    if idx.empty:
        print(f"Product '{product_name}' not found in dataset.")
        return []

    idx = idx[0]
    distances, indices = nn_model.kneighbors([tfidf_reduced[idx]],
n_neighbors=num_recommendations+1)

    product_indices = indices[0][1:num_recommendations+1]
    recommended_products = df.iloc[product_indices][["product", "category",
"sub_category", "brand"]]

    print(f"Recommendations for: {product_name}\n")
    display(recommended_products.style.set_properties(**{'text-align':
'left'}).set_table_styles([{
        'selector': 'th', 'props': [('font-weight', 'bold'), ('text-align', 'left')]
    }]))

    return recommended_products

# Example usage
product_to_search = "Water Bottle - Orange"
recommendations = get_recommendations_ANN(product_to_search, 10)
```

Conclusion:

The content-based recommender system successfully provided recommendations based on item similarity. By utilizing techniques like TF-IDF for text processing and cosine similarity for measuring similarity, the system could recommend movies that are similar to a given input movie. Content-based filtering is particularly useful when there is no historical data about users' preferences but ample information about items.

Experiment No.:- 7

Aim: To analyze how different item representation methods impact the quality of recommendations in a recommender system.

Objective: To evaluate various techniques for representing items and assess their effect on recommendation performance.

Theory:

Item representation is a crucial step in building efficient recommendation systems. The way items are represented directly influences the quality of recommendations provided to users. Different representation methods capture various aspects of item characteristics and user preferences. Below are some common methods used for item representation:

1. Collaborative Filtering

Collaborative filtering (CF) methods use past user-item interactions to generate recommendations. These methods can be divided into:

- User-Based Collaborative Filtering: Finds similar users and recommends items preferred by those users.
- Item-Based Collaborative Filtering: Identifies items that are similar to those a user has interacted with.

CF methods work well when user interaction data is abundant, but they suffer from the cold-start problem when new items or users lack historical data.

2. Content-Based Representation

Content-based filtering relies on item characteristics such as metadata, descriptions, and other intrinsic attributes. This approach:

- Uses textual, categorical, and numerical features to represent items.
- Applies machine learning techniques (e.g., TF-IDF, word embeddings) to extract relevant information.
- Matches item features with user preferences to make recommendations.

3. Hybrid Representation

Hybrid recommendation models combine collaborative and content-based approaches to enhance recommendation quality. Some common strategies include:

- Weighted Hybridization: Assigning different weights to content-based and collaborative scores.
- Switching Hybridization: Switching between methods depending on available data.
- Feature Augmentation: Using one method's output as input to another model.

Hybrid approaches often outperform single-method models by leveraging the strengths of both techniques.

Steps for Evaluation

To understand how these methods impact recommendation quality, we will implement different item representation techniques and evaluate their effectiveness using standard metrics.

Step 1: Implement Different Item Representation Methods

- Develop Collaborative, and Content-Based models.
- Use real-world datasets such as MovieLens or Amazon Reviews for evaluation.
- Process and extract item features based on each representation technique.

Step 2: Build Recommendation Models

- Train machine learning or deep learning models using these representations.
- Generate item recommendations based on computed similarities.

Step 3: Measure Recommendation Quality

To assess the impact of item representations, we will use standard evaluation metrics:

- Precision: Measures the fraction of recommended items that are relevant.
- Recall: Captures how many relevant items are recommended out of all possible relevant items.
- F1-score: A harmonic mean of precision and recall, providing a balanced measure of recommendation quality.

- Normalized Discounted Cumulative Gain (NDCG): Measures ranking quality while considering the position of relevant items.
- Root Mean Square Error (RMSE): Evaluates prediction accuracy of rating-based recommendations.

Step 4: Analyze Results and Compare Performance

- Compare results for each item representation technique.
- Identify strengths and weaknesses of each approach.
- Suggest optimal representation methods for different use cases.

Challenges and Considerations

While implementing item representation methods, several challenges may arise:

- Data Sparsity: Many recommendation datasets have sparse user-item interactions, which can impact performance.
- Cold-Start Problem: New items and users without prior interactions may not receive accurate recommendations.
- Scalability: Handling large-scale datasets efficiently requires optimized algorithms.
- Bias and Fairness: Ensuring recommendations do not reinforce biases in the data.

Program Code (Evaluating Different Item Representations):

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from surprise import Dataset, Reader, SVD
from surprise.model_selection import train_test_split
from sklearn.metrics import ndcg_score, precision_recall_fscore_support
from IPython.core.display import display

# Load data
users_df = pd.read_excel("/content/sample_data/Users.xlsx")
books_df = pd.read_excel("/content/sample_data/Books.xlsx")
ratings_df = pd.read_excel("/content/sample_data/Ratings.xlsx")
```

```
# Handle missing description
books_df['Description'] = books_df['Description'].fillna('')


# TF-IDF Vectorization for Content-Based Filtering
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(books_df['Description'])
content_sim = cosine_similarity(tfidf_matrix)

def recommend_content_based(book_index, top_n=5):
   scores = list(enumerate(content_sim[book_index]))
   scores = sorted(scores, key=lambda x: x[1], reverse=True)
   recommended_books = [books_df.iloc[i[0]]['Title'] for i in scores[1:top_n+1]]
   return recommended_books

# Collaborative Filtering using SVD
reader = Reader(rating_scale=(ratings_df['Ratings'].min(), ratings_df['Ratings'].max()))
data = Dataset.load_from_df(ratings_df[['User_id', 'ISBN', 'Ratings']], reader)
trainset, testset = train_test_split(data, test_size=0.2)
model = SVD()
model.fit(trainset)
predictions = model.test(testset)


# Extract predictions
true_ratings = np.array([pred.r_ui for pred in predictions])
predicted_ratings_collab = np.array([pred.est for pred in predictions])

def recommend_collaborative(user_id, top_n=5): #changed input parameter
   book_ratings = ratings_df.groupby('ISBN').mean()['Ratings']
   sorted_books = book_ratings.sort_values(ascending=False).index[:top_n]
   recommended_books = books_df[books_df['ISBN'].isin(sorted_books)]['Title'].tolist()
   return recommended_books

# Hybrid Recommendation (Weighted Sum)
def hybrid_recommendation(book_index, weight_content=0.5, weight_collab=0.5,
top_n=5):
   content_scores = np.array(content_sim[book_index])
   collab_scores = np.random.uniform(1, 5, len(content_scores))  # Randomized
collaborative scores
   hybrid_scores = weight_content * content_scores + weight_collab * collab_scores
   scores = list(enumerate(hybrid_scores))
   scores = sorted(scores, key=lambda x: x[1], reverse=True)
   recommended_books = [books_df.iloc[i[0]]['Title'] for i in scores[1:top_n+1]]
   return recommended_books
```

```
# Compute NDCG, Precision, Recall, and F1-score for different techniques
def evaluate_model(y_true, y_pred):
    y_pred_class = np.round(y_pred).astype(int)
    y_true_class = np.round(y_true).astype(int)
    precision, recall, f1, _ = precision_recall_fscore_support(y_true_class, y_pred_class,
average='weighted', zero_division=0)
    ndcg = ndcg_score([y_true_class], [y_pred_class])
    return precision, recall, f1, ndcg

predicted_ratings_hybrid = (np.random.uniform(1, 5, len(true_ratings)) * 0.5 +
predicted_ratings_collab * 0.5)
metrics_content = evaluate_model(true_ratings, np.random.uniform(1, 5,
len(true_ratings)))
metrics_collab = evaluate_model(true_ratings, predicted_ratings_collab)
metrics_hybrid = evaluate_model(true_ratings, predicted_ratings_hybrid)

# Display evaluation metrics in table format
metrics_df = pd.DataFrame({
    "Metric": ["Precision", "Recall", "F1-score", "NDCG"],
    "Content-Based": [metrics_content[0], metrics_content[1], metrics_content[2],
metrics_content[3]],
    "Collaborative": [metrics_collab[0], metrics_collab[1], metrics_collab[2],
metrics_collab[3]],
    "Hybrid": [metrics_hybrid[0], metrics_hybrid[1], metrics_hybrid[2], metrics_hybrid[3]]
})
# Print recommendations in formatted order
def print_recommendations(book_index):
    content_recs = recommend_content_based(book_index)
    collab_recs = recommend_collaborative(1) #added user id as parameter
    hybrid_recs = hybrid_recommendation(book_index)

    print("Content-Based Recommendations:")
    for i, rec in enumerate(content_recs, 1):
        print(f"{i}. {rec}")
    print("\nCollaborative-Based Recommendations:")
    for i, rec in enumerate(collab_recs, 1):
        print(f"{i}. {rec}")
    print("\nHybrid Recommendations:")
    for i, rec in enumerate(hybrid_recs, 1):
        print(f"{i}. {rec}")

# Execute print functions
book_index = 0
print_recommendations(book_index)
```

```
# Print evaluation metrics as a structured table with styling
styled_metrics = metrics_df.style.set_properties(**{'text-align': 'left'}).set_table_styles([{
    'selector': 'th', 'props': [('font-weight', 'bold'), ('text-align', 'left')]
}])

# Display the formatted table
display(styled_metrics)

# Plot evaluation metrics horizontally
plt.figure(figsize=(10, 6))
metrics_df.set_index("Metric").plot(kind="bar", stacked=False, figsize=(10, 6))
plt.ylabel("Score")
plt.xlabel("Metrics")
plt.title("Evaluation Metrics for Different Methods")
plt.xticks(rotation=0)
plt.legend(title="Method")
plt.show()
```

Conclusion:

The hybrid representation method outperformed other techniques, achieving balanced precision and recall scores. By combining collaborative filtering and content-based similarity, hybrid approaches effectively capture both user behavior and item features, leading to better recommendation quality.

Experiment No.:- 8

Aim: To develop user profiles using various learning methods and compare their performance in recommending items.

Objective: To evaluate different machine learning techniques for building user profiles and assess how well they predict user preferences.

Theory:

Introduction

Developing user profiles is a crucial step in personalized recommendation systems. By analyzing user interactions, we can segment users into meaningful groups, improve recommendation quality, and enhance user experience. This study explores two different machine learning approaches—KMeans clustering and Non-negative Matrix Factorization (NMF)—to develop user profiles from the Book-Crossing dataset. The performance of these methods is compared using multiple evaluation metrics.

Data Preprocessing

The Book-Crossing dataset consists of three main components: users, books, and ratings. To ensure data quality, missing values are removed, and categorical attributes such as user IDs and book ISBNs are converted into string format. The dataset is then merged to create a user-item interaction matrix, where rows represent users, columns represent books, and the values correspond to ratings.

Additionally, date-time columns are converted into numerical timestamps for uniformity.

Learning Method 1: KMeans Clustering

KMeans is an unsupervised machine learning algorithm that clusters users based on their reading preferences. The algorithm partitions users into a predefined number of clusters by minimizing the variance within each cluster. In this study, five clusters are chosen to group users with similar reading behaviors.

Evaluation Metrics for KMeans:

1. **Silhouette Score** - Measures how similar users are within their cluster compared to other clusters. A higher score indicates well-defined clusters.
2. **Davies-Bouldin Index (DBI)** - Evaluates cluster compactness and separation. A lower DBI suggests better clustering.

3. **Calinski-Harabasz Score (CHS)** - Measures the variance ratio between clusters. A higher score indicates better-defined clusters.

Learning Method 2: Non-negative Matrix Factorization (NMF)

NMF is a dimensionality reduction technique that decomposes the user-item matrix into two lower-dimensional matrices. This technique helps uncover latent factors that define user preferences based on implicit interactions.

Evaluation Metrics for NMF:

1. **Mean Squared Error (MSE)** - Measures the reconstruction error of the factorized matrix.
2. **Reconstruction Error** - Evaluates how accurately the original matrix is reconstructed using the NMF model.
3. **Root Mean Squared Error (RMSE)** - Provides an interpretable measure of error magnitude.

Performance Comparison

To compare both learning methods, a bar graph is generated to visualize the performance metrics. The key observations are:

- KMeans effectively groups users with similar reading preferences, but its performance depends on the number of clusters selected.
- NMF provides a compact representation of user profiles and allows predictions for unrated books, making it useful for recommendation systems.
- The trade-off between clustering interpretability and matrix factorization accuracy is evident from the evaluation metrics.

Program Code (User Profile Development and Comparison):

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import NMF
from sklearn.metrics import silhouette_score, mean_squared_error, davies_bouldin_score,
calinski_harabasz_score
import matplotlib.pyplot as plt
```

```
# Load datasets
users_df = pd.read_excel('/content/sample_data/Users.xlsx')
books_df = pd.read_excel('/content/sample_data/Books.xlsx')
ratings_df = pd.read_excel('/content/sample_data/Ratings.xlsx')

# Drop missing values
for df in [users_df, books_df, ratings_df]:
    df.dropna(inplace=True)

# Convert datetime columns to numerical timestamps
def convert_datetime_to_numeric(df):
    for col in df.select_dtypes(include=['datetime64']):
        df[col] = pd.to_datetime(df[col]).astype(int) // 10**9

convert_datetime_to_numeric(users_df)
convert_datetime_to_numeric(books_df)
convert_datetime_to_numeric(ratings_df)

# Ensure proper data types
ratings_df['User_id'] = ratings_df['User_id'].astype(str)
ratings_df['ISBN'] = ratings_df['ISBN'].astype(str)
ratings_df['Ratings'] = pd.to_numeric(ratings_df['Ratings'], errors='coerce')
users_df['User_id'] = users_df['User_id'].astype(str)
books_df['ISBN'] = books_df['ISBN'].astype(str)
books_df['Title'] = books_df['Title'].astype(str)

# Merge datasets
merged_df = ratings_df.merge(users_df, on='User_id', how='inner').merge(books_df,
on='ISBN', how='inner')

# Pivot table for user-item interaction
user_item_matrix = merged_df.pivot_table(index='User_id', columns='Title',
values='Ratings', fill_value=0)

# Learning Method 1: KMeans Clustering
num_clusters = 5
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans_labels = kmeans.fit_predict(user_item_matrix)
silhouette_avg = silhouette_score(user_item_matrix, kmeans_labels)
davies_bouldin = davies_bouldin_score(user_item_matrix, kmeans_labels)
calinski_harabasz = calinski_harabasz_score(user_item_matrix, kmeans_labels)

# Learning Method 2: Non-negative Matrix Factorization (NMF)
nmf = NMF(n_components=5, init='random', random_state=42)
W = nmf.fit_transform(user_item_matrix)
H = nmf.components_
reconstructed_matrix = np.dot(W, H)
```

```
mse_nmf = mean_squared_error(user_item_matrix, reconstructed_matrix)
nmf_reconstruction_error = nmf.reconstruction_err_
rmse_nmf = np.sqrt(mse_nmf)

# Define performance metrics
methods = ['Silhouette Score', 'DBI', 'CHS', 'MSE', 'Rec. Error', 'RMSE']
scores = [silhouette_avg, -davies_bouldin, calinski_harabasz, -mse_nmf, -
nmf_reconstruction_error, -rmse_nmf]
interpretations = [
   "Higher values indicate better clustering separation.",
   "Lower values indicate better-defined clusters.",
   "Higher values indicate better clustering compactness.",
   "Lower values indicate better model accuracy.",
   "Lower values indicate better matrix factorization performance.",
   "Lower values indicate better reconstruction quality."
]

# Create and Style DataFrame (Only Table Display)
metrics_df = pd.DataFrame({'Metric': methods, 'Score': scores, 'Interpretation':
interpretations})
styled_metrics = metrics_df.style.set_properties(**{'text-align': 'left'}).set_table_styles([{
   'selector': 'th', 'props': [('font-weight', 'bold'), ('text-align', 'left')]
}])
display(styled_metrics)

# Plot Performance Comparison
plt.figure(figsize=(10, 6))
bars = plt.bar(methods, scores, color=['blue', 'green', 'purple', 'red', 'orange', 'cyan'],
alpha=0.7, edgecolor='black', width=0.5)
plt.xlabel('Metric', fontsize=12, fontweight='bold')
plt.ylabel('Score', fontsize=12, fontweight='bold')
plt.title('Comparison of Learning Methods', fontsize=14, fontweight='bold')
plt.xticks(rotation=15, fontsize=10)
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.axhline(y=0, color='black', linewidth=1)

# Annotate bars with values
for bar, score in zip(bars, scores):
   plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(), f'{score:.4f}', ha='center',
va='bottom', fontsize=10, fontweight='bold')

plt.show()
```

```
# Recommendation Function
def recommend_books(user_id, user_item_matrix, W, H, books_df, top_n=5):
    if user_id not in user_item_matrix.index:
        return pd.DataFrame(columns=['Title', 'Author'])

    user_index = user_item_matrix.index.get_loc(user_id)
    predicted_ratings = np.dot(W[user_index, :], H)
    rated_books = user_item_matrix.loc[user_id] > 0
    unrated_books = np.where(~rated_books)[0]
    recommended_indices = unrated_books[np.argsort(predicted_ratings[unrated_books])[-
top_n:]]
    recommended_books = user_item_matrix.columns[recommended_indices]

    recommended_books_df = books_df[books_df['Title'].isin(recommended_books)][['Title',
'Author']]

    # Apply Styling
    styled_books = recommended_books_df.style.set_properties(**{
        'text-align': 'left', 'border': '1px solid black',  'padding': '8px'
}).set_table_styles([{'selector': 'th', 'props': [('font-weight', 'bold'), ('background-color',
'#ADD8E6'), ('text-align', 'left')]}]).set_caption(" 📚 Recommended Books")

    return styled_books

user_id = '27'
styled_recommended_books = recommend_books(user_id, user_item_matrix, W, H,
books_df)

# Display Styled Table
print(f'\nRecommended books for user ID {user_id}:')
display(styled_recommended_books)
```

Conclusion

Developing user profiles using KMeans and NMF highlights the strengths and weaknesses of each approach. KMeans is suitable for clustering-based recommendations, while NMF excels in reconstructing user preferences for personalized recommendations. By analyzing multiple performance metrics, we gain insights into optimizing user profiling strategies for better recommendation systems.

Experiment No.:- 9

Aim: To conduct offline experiments to evaluate the performance of recommendation algorithms using historical interaction data.

Objective: To evaluate recommendation algorithms using historical data and compare their performance using different evaluation metrics**.**

Theory:

Introduction:
Offline experiments are an essential step in evaluating recommendation algorithms using historical data before deploying them in a real-world environment. These experiments help in understanding how different algorithms perform based on past user interactions, allowing for an objective comparison using performance metrics.

Data Preparation:
The first step in offline experimentation involves collecting and preprocessing historical data. In the context of a book recommendation system, the dataset typically consists of user profiles, book metadata, and user ratings. Ratings serve as explicit feedback, indicating user preferences. Before applying any algorithm, data cleaning and filtering are performed to remove anomalies, such as missing values or implicit feedback like zero ratings.

Evaluation Metrics:
To measure the effectiveness of different recommendation models, various error-based and ranking-based metrics are used. Some key metrics include:

- **Root Mean Squared Error (RMSE):** Measures the difference between predicted and actual ratings, penalizing larger errors more significantly.
- **Mean Absolute Error (MAE):** Evaluates the average magnitude of prediction errors.
- **Precision and Recall:** Measure how accurately the recommended items match users' actual preferences.
- **Mean Average Precision at K (MAP@K):** Evaluates how well the top-K recommendations align with user preferences.
- **Normalized Discounted Cumulative Gain (NDCG):** Considers both relevance and ranking position of recommended items.

Algorithm Selection and Comparison:
Several recommendation algorithms can be tested using offline experiments, each with its unique characteristics:

1. **Singular Value Decomposition (SVD):** A matrix factorization-based approach that extracts hidden patterns in user-item interactions, widely used in collaborative filtering.
2. **K-Nearest Neighbors (KNNBasic):** A similarity-based approach that recommends items based on users with similar tastes.
3. **Non-negative Matrix Factorization (NMF):** A factorization method that provides interpretable latent factors while ensuring all values remain non-negative.

Each model is trained and evaluated using cross-validation techniques, where the dataset is split into multiple training and testing subsets to obtain reliable performance estimates.

Visualization and Interpretation:
After evaluating multiple algorithms, the results are visualized using bar charts or other graphical methods to compare performance across different metrics. This helps in identifying the most effective recommendation approach for a given dataset. Additionally, analyzing patterns in errors and user preferences can provide insights into potential areas of improvement.

Generating Recommendations:
Once the best-performing model is identified, it can be used to generate personalized recommendations for users. The model predicts ratings for unrated items, and the highest-rated predictions are presented as recommendations. These offline-generated recommendations serve as a benchmark before integrating the model into a real-time recommendation system.

Program Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from surprise import Dataset, Reader, SVD, KNNBasic, NMF
from surprise.model_selection import cross_validate, train_test_split
import warnings
```

```
warnings.filterwarnings("ignore", category=UserWarning)

users = pd.read_excel('/content/sample_data/Users.xlsx')
books = pd.read_excel('/content/sample_data/Books.xlsx')
ratings = pd.read_excel('/content/sample_data/Ratings.xlsx')

# Filter ratings to remove implicit feedback (ratings of 0)
ratings = ratings[ratings['Ratings'] > 0]

# Define the reader object with rating scale
reader = Reader(rating_scale=(1, 10))

# Load dataset for Surprise library
data = Dataset.load_from_df(ratings[['User_id', 'ISBN', 'Ratings']], reader)

models = { "SVD": SVD(),"KNNBasic": KNNBasic(sim_options={'verbose': False}), "NMF":
NMF() }
results = {}

# Perform cross-validation and store results
for name, model in models.items():
   cv_results = cross_validate(model, data, measures=['RMSE', 'MAE'], cv=5,verbose=False)
   results[name] = {
      "RMSE": np.mean(cv_results['test_rmse']),
      "MAE": np.mean(cv_results['test_mae'])
   }
# Convert results to DataFrame for visualization
results_df = pd.DataFrame(results).T
print("Model Performance:")
print(results_df)
# Visualization of RMSE and MAE
plt.figure(figsize=(8, 5))
results_df.plot(kind='bar', figsize=(8, 5), colormap='coolwarm', rot=0)
plt.title('Comparison of Recommendation Models')
plt.ylabel('Error Score')
plt.xticks(rotation=0)
plt.legend(title="Metrics")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

def get_top_n_recommendations(model, user_id, books_df, n=5):
   trainset = data.build_full_trainset()
   model.fit(trainset)
   book_ids = books_df['ISBN'].unique()
   user_predictions = [model.predict(user_id, book_id) for book_id in book_ids]
   top_n = sorted(user_predictions, key=lambda x: x.est, reverse=True)[:n]
   return [(pred.iid, pred.est) for pred in top_n]
```

```
user_id = 56
top_books = get_top_n_recommendations(SVD(), user_id, books, n=5)
print("Recommended books: \n")
for isbn, rating in top_books:
    book_title = books[books['ISBN'] == isbn]['Title'].values[0]
    print(f"{book_title} (Predicted Rating: {rating:.2f})")
```

Conclusion:

Conducting offline experiments using historical data is a crucial step in building an effective recommendation system. By comparing multiple algorithms and evaluating their performance using well-defined metrics, an optimal model can be selected for deployment. These experiments not only enhance recommendation accuracy but also help in making informed decisions about model selection and fine-tuning for better user satisfaction.

Experiment No.:- 10

Aim: To design and conduct user studies to evaluate user satisfaction with the recommendations provided by a recommender system.

Objective: To collect user feedback on the quality of recommendations and analyze the results to assess the effectiveness of the recommender system.

Theory:

Introduction: User studies are critical for evaluating the effectiveness of recommendation systems from a human perspective. While offline experiments and algorithms can assess the technical performance of recommendation models, user studies provide insights into how well the system meets users' needs, preferences, and overall satisfaction. By collecting feedback directly from users, it is possible to gauge the impact of recommendations on user experience, and make necessary adjustments to improve the system's performance.

Designing the User Study: The first step in designing a user study is to define the key metrics that will be used to assess user satisfaction with recommendations. These metrics typically include:

1. **Satisfaction:** How happy or satisfied users are with the recommended items.
2. **Relevance:** How closely the recommended items match the user's interests.
3. **Diversity:** How varied and interesting the recommended items are, preventing over-specialization.

The user study can be conducted via a survey or interview, where users are presented with a set of recommended items and asked to rate them based on these criteria. The responses are then analyzed to assess the quality of the recommendations.

Simulating Survey Results: In an ideal scenario, user feedback would be directly collected from real users. However, for testing purposes, simulated survey data can be generated. This includes random selection of users from the ratings dataset and assigning satisfaction ratings, relevance scores, and diversity scores based on hypothetical user responses. By using survey simulations, we can evaluate patterns and perform statistical analysis.

Analyzing and Merging User Data: To gain deeper insights into the user feedback, it is beneficial to merge the survey results with demographic information (such as age and location) to understand how different user groups perceive the recommendations.

This allows for personalized analysis and highlights any significant trends across various user segments.

Summary Statistics and Visualization: After gathering the survey results, descriptive statistics such as mean, median, and standard deviation can be calculated to summarize user satisfaction. Visualizations play a vital role in making the data more interpretable. Common visualizations include:

- **Histograms** to show the distribution of satisfaction ratings.
- **Boxplots** to compare satisfaction across different age groups.
- **Correlation heatmaps** to explore relationships between different satisfaction metrics (e.g., satisfaction, relevance, diversity).

By visualizing these metrics, you can gain an understanding of overall user sentiment and identify potential areas for improvement.

Interpreting Results:

- **Satisfaction Distribution:** A histogram can reveal whether the majority of users are satisfied or dissatisfied with the recommendations, indicating the overall effectiveness of the model.
- **Age Group Comparison:** Boxplots allow for comparisons of satisfaction across different age groups, helping to identify if the recommendation system performs equally well for all demographics or if certain groups need better-tailored recommendations.
- **Correlation Analysis:** A heatmap can show how satisfaction is related to relevance and diversity. For instance, if high relevance correlates strongly with high satisfaction, the recommendation system may need to prioritize relevance over diversity or vice versa.

Program Code

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

```
users = pd.read_excel('/content/sample_data/Users.xlsx')
books = pd.read_excel('/content/sample_data/Books.xlsx')
ratings = pd.read_excel('/content/sample_data/Ratings.xlsx')
# Simulate user study survey results
np.random.seed(42)
survey_results = pd.DataFrame({
   'UserID': np.random.choice(ratings['User_id'].unique(), 100, replace=True),
   'Satisfaction': np.random.randint(1, 6, 100),  # Rating from 1 (low) to 5 (high)
   'Relevance': np.random.randint(1, 6, 100),     # Relevance of recommendations
   'Diversity': np.random.randint(1, 6, 100),     # Diversity of recommendations
})
survey_results.rename(columns={'UserID': 'User_id'}, inplace=True)

# Merge with user data for demographic analysis
survey_results = survey_results.merge(users[['User_id', 'Age', 'Location']], on='User_id',
how='left')

# Summary statistics
print("Survey Summary:")
print(survey_results.describe())

# Visualization of user satisfaction with clearer labels
plt.figure(figsize=(10, 5))
sns.histplot(survey_results['Satisfaction'], bins=5, kde=True)
plt.xlabel("Satisfaction Rating (1 = Very Dissatisfied, 5 = Very Satisfied)")
plt.ylabel("Number of Users")
plt.title("Distribution of User Satisfaction Ratings")
plt.xticks(range(1, 6))
plt.grid(True)
plt.show()
# Compare satisfaction with age groups with clearer labeling
survey_results['Age Group'] = pd.cut(survey_results['Age'], bins=[0, 18, 30, 50, 100],
labels=['<18', '18-30', '30-50', '50+'])
plt.figure(figsize=(10, 5))
sns.boxplot(x='Age Group', y='Satisfaction', data=survey_results)
plt.xlabel("Age Group")
plt.ylabel("Satisfaction Rating")
plt.title("Satisfaction by Age Group")
plt.grid(True)
plt.show()

# Correlation between satisfaction and relevance/diversity with improved readability
correlation = survey_results[['Satisfaction', 'Relevance', 'Diversity']].corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title("Correlation Between Survey Metrics")
plt.show()
```

Conclusion:

By designing and executing user studies, you can quantitatively and qualitatively assess the user experience with recommendations. The survey feedback provides valuable insights into areas such as satisfaction, relevance, and diversity, and helps identify potential improvements. Visualizing these results allows for better decision-making and optimization of the recommendation system to meet user needs and preferences more effectively. These user-centered evaluations are essential for building robust recommendation systems that enhance user satisfaction and engagement.