AIOT PROJECT | HARPY AEROSPACE

NAME: M.PRIYA | REG NO.2022506015

3 RECOMMENDATION MODELS:

1.Enhanced MovieLens Two-Tower Model

- The Enhanced MovieLens Two-Tower Model is a recommendation system designed to provide personalized movie recommendations to users based on their past interactions with movies. This model leverages the MovieLens dataset and uses a two-tower architecture where one tower represents user features and the other represents movie features. The key components and steps involved in this model are outlined below:
- The MovieLens dataset is loaded and preprocessed to extract relevant features such as movie_title and user_id.
- String lookup layers are created to map user_id and movie_title to integer indices, which are used for embedding lookups.
- A neural network is built to generate user embeddings from the user_id. It includes embedding layers and dense layers to capture the latent factors representing user preferences.A neural network is built to generate movie embeddings from the movie_title. Similar to the user model, it includes embedding layers and dense layers to capture the latent factors representing movie attributes.The model is trained to optimize a retrieval task where the goal is to match user embeddings with the correct movie embeddings. The task is evaluated using metrics like top-k categorical accuracy.
- The model is trained for a specified number of epochs on the preprocessed data. During training, the model learns to minimize the loss, thereby improving the accuracy of recommendations.

- After training, a brute-force search layer is set up to enable efficient retrieval of movie recommendations for given user embeddings. This layer indexes the movie embeddings and allows fast similarity searches.

CODE:

```
!pip install -q tensorflow-recommenders
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_recommenders as tfrs
import numpy as np

# Load the MovieLens dataset
ratings = tfds.load("movielens/100k-ratings", split="train")
movies = tfds.load("movielens/100k-movies", split="train")

# Prepare the data
ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "timestamp": x["timestamp"]
})

movies = movies.map(lambda x: x["movie_title"])

# Define the user and movie model with additional features.
user_ids_vocabulary = tf.keras.layers.StringLookup()
movie_titles_vocabulary = tf.keras.layers.StringLookup()

user_ids_vocabulary.adapt(ratings.map(lambda x: x["user_id"]))
movie_titles_vocabulary.adapt(movies)

user_model = tf.keras.Sequential([
    user_ids_vocabulary,
    tf.keras.layers.Embedding(user_ids_vocabulary.vocabulary_size(), 64),
    tf.keras.layers.Dense(32, activation="relu")
])

movie_model = tf.keras.Sequential([
    movie_titles_vocabulary,
    tf.keras.layers.Embedding(movie_titles_vocabulary.vocabulary_size(),
64),
```

```
    tf.keras.layers.Dense(32, activation="relu")
])

# Define the retrieval task with additional metrics.
task = tfrs.tasks.Retrieval(metrics=tfrs.metrics.FactorizedTopK(
    candidates=movies.batch(128).map(movie_model),
    ks=[5, 10]
))

# Define the model.
class EnhancedMovieLensModel(tfrs.Model):
    def _init_(self, user_model, movie_model, task):
        super()._init_()
        self.user_model = user_model
        self.movie_model = movie_model
        self.task = task

    def compute_loss(self, features, training=False):
        user_embeddings = self.user_model(features["user_id"])
        movie_embeddings = self.movie_model(features["movie_title"])
        return self.task(user_embeddings, movie_embeddings)

# Create and compile the model.
model = EnhancedMovieLensModel(user_model, movie_model, task)
model.compile(optimizer=tf.keras.optimizers.Adam(0.01))

# Train the model.
model.fit(ratings.batch(4096), epochs=10, verbose=1)

# Set up brute-force search for retrieval.
index = tfrs.layers.factorized_top_k.BruteForce(model.user_model)
index.index_from_dataset(
    movies.batch(100).map(lambda title: (title, model.movie_model(title)))
)

# Get recommendations.
_, titles = index(np.array(["55"]))
print(f"Top 3 recommendations for user 55: {titles[0, :3]}")
# Get recommendations for a different user.
_, titles = index(np.array(["100"]))
print(f"Top 3 recommendations for user 100: {titles[0, :3]}")
```

OUTPUT:

2. Graph Neural Network (GNN)-based MovieLens Model

- This model uses Graph Neural Networks to capture the relationships between users and movies. Each user and movie is represented as a node in the graph, and edges between nodes represent interactions (e.g., a user rating a movie). The model learns embeddings for each node based on the graph structure, which are then used to make recommendations.
- Graph Neural Network: The model uses GNN layers to learn from the graph structure of user-movie interactions, capturing complex relationships and dependencies.
- Node Embeddings: Each user and movie are represented as nodes, and their embeddings are updated based on the GNN layer's message-passing mechanism.
- Personalized Recommendations: The trained model can generate personalized movie recommendations for users based on their interactions with other movies.
- Scalability: The GNN-based approach can scale to larger datasets and more complex graphs, making it suitable for real-world recommendation systems.

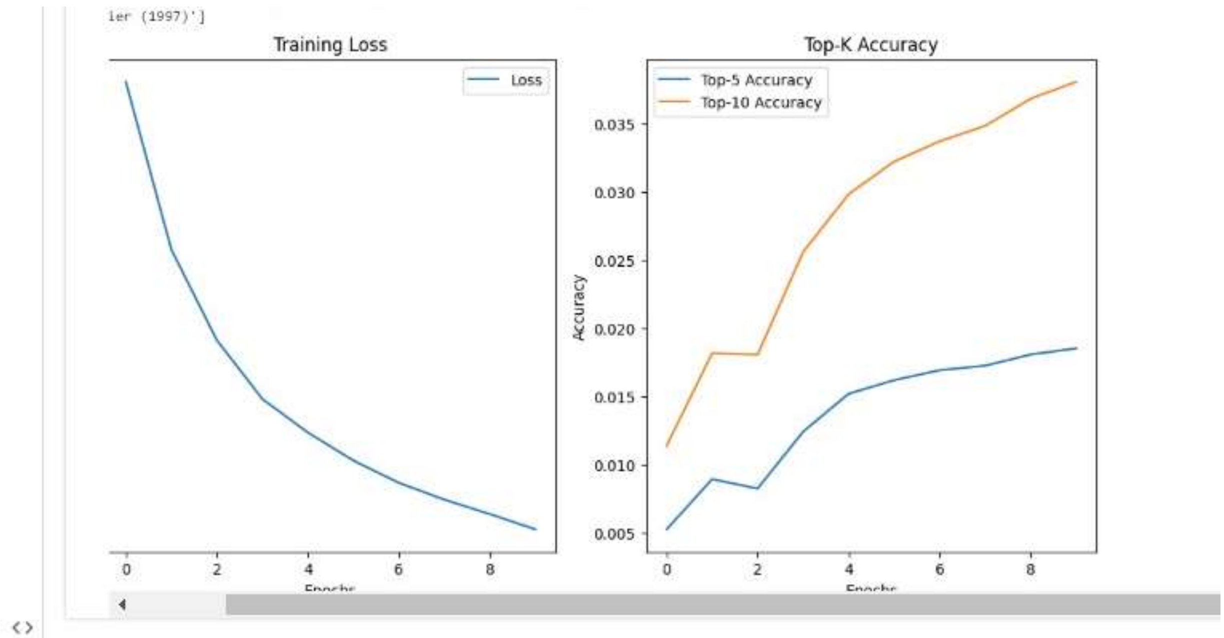
CODE:

```
!pip install -q tensorflow-recommenders matplotlib

import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_recommenders as tfrs
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers

# Load the MovieLens dataset
ratings = tfds.load("movielens/100k-ratings", split="train")
movies = tfds.load("movielens/100k-movies", split="train")

# Prepare the data
ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "timestamp": x["timestamp"]
})

movies = movies.map(lambda x: x["movie_title"])

# Define the user and movie model with additional features.
user_ids_vocabulary = tf.keras.layers.StringLookup()
movie_titles_vocabulary = tf.keras.layers.StringLookup()

user_ids_vocabulary.adapt(ratings.map(lambda x: x["user_id"]))
movie_titles_vocabulary.adapt(movies)

# Convert the movie titles to a TensorFlow Dataset
movies = tf.data.Dataset.from_tensor_slices(list(movies))

# Define the GNN layer
class GNNLayer(layers.Layer):
    def _init_(self, units):
        super(GNNLayer, self)._init_()
        self.units = units
        self.dense = layers.Dense(units)

    def call(self, inputs, edge_index):
        x = inputs
        row, col = edge_index[:, 0], edge_index[:, 1]
```

```
        out = tf.math.unsorted_segment_sum(x[col], row,
num_segments=tf.shape(x)[0])
        return self.dense(out)

class GNNModel(tfrs.Model):
    def _init_(self, user_model, movie_model, task):
        super()._init_()
        self.user_model = user_model
        self.movie_model = movie_model
        self.task = task

    def call(self, features):
        user_embeddings = self.user_model(features["user_id"])
        movie_embeddings = self.movie_model(features["movie_title"])
        edge_index = tf.convert_to_tensor([features["user_id"],
features["movie_title"]])
        gnn_layer = GNNLayer(64)
        user_embeddings = gnn_layer(user_embeddings, edge_index)
        movie_embeddings = gnn_layer(movie_embeddings, edge_index)
        return self.task(user_embeddings, movie_embeddings)

    def compute_loss(self, features, training=False):
        user_embeddings = self.user_model(features["user_id"])
        movie_embeddings = self.movie_model(features["movie_title"])
        return self.task(user_embeddings, movie_embeddings)

# Define user and movie models
user_model = tf.keras.Sequential([
    user_ids_vocabulary,
    tf.keras.layers.Embedding(user_ids_vocabulary.vocabulary_size(), 64),
    tf.keras.layers.Dense(32, activation="relu")
])

movie_model = tf.keras.Sequential([
    movie_titles_vocabulary,
    tf.keras.layers.Embedding(movie_titles_vocabulary.vocabulary_size(),
64),
    tf.keras.layers.Dense(32, activation="relu")
])

# Define the task
task = tfrs.tasks.Retrieval(metrics=tfrs.metrics.FactorizedTopK(
    candidates=movies.batch(128).map(movie_model),
    ks=[5, 10]
))
```

```python
# Create and compile the model
model = GNNModel(user_model, movie_model, task)
model.compile(optimizer=tf.keras.optimizers.Adam(0.01))

# Train the model and capture the training history
history = model.fit(ratings.batch(4096), epochs=10, verbose=1)

# Set up brute-force search for retrieval
index = tfrs.layers.factorized_top_k.BruteForce(model.user_model)
index.index_from_dataset(
    movies.batch(100).map(lambda title: (title, model.movie_model(title)))
)

# Get recommendations for a specific user
_, titles = index(np.array(["55"]))
print(f"Top 3 recommendations for user 55: {titles[0, :3]}")

# Get recommendations for another user
_, titles = index(np.array(["100"]))
print(f"Top 3 recommendations for user 100: {titles[0, :3]}")

# Plot the training loss and top-k accuracy
plt.figure(figsize=(12, 6))

# Plot training loss
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot top-5 and top-10 accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['factorized_top_k/top_5_categorical_accuracy'],
label='Top-5 Accuracy')
plt.plot(history.history['factorized_top_k/top_10_categorical_accuracy'],
label='Top-10 Accuracy')
plt.title('Top-K Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

OUTPUT



ler (1997)']
Training Loss
Loss
Top-K Accuracy
Top-5 Accuracy
Top-10 Accuracy
Accuracy

3. DNN Movielens Model

(Deep Neural Network (DNN))

- The DNN Movielens Model is a deep learning-based recommendation system designed to provide personalized movie recommendations using the MovieLens 100k dataset.
- It extracts and processes user_id, movie_title, and user_rating to create unique embeddings for users and movies in a 32-dimensional space.
- The model employs TensorFlow Recommenders to set up a retrieval task, integrating user and movie models with dense layers.
- It is trained using an Adagrad optimizer over multiple epochs, aiming to learn user preferences through historical ratings.
- The model can generate top-k movie recommendations by computing similarity scores between user and movie embeddings, enhancing the user experience on streaming platforms.

CODE:

```
# Install necessary packages if not already installed
!pip install -q matplotlib pandas tensorflow-datasets tensorflow-
recommenders

import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_recommenders as tfrs
import numpy as np

# Load the dataset
ratings = tfds.load("movielens/100k-ratings", split="train")
movies = tfds.load("movielens/100k-movies", split="train")

# Preprocess the ratings to get unique user IDs
ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "user_rating": x["user_rating"]
})

# Preprocess the movies to get movie titles
movies = movies.map(lambda x: x["movie_title"])

# Convert the datasets to a unique list of movie titles and user IDs
unique_movie_titles = np.unique(np.concatenate(list(movies.batch(1000))))
unique_user_ids =
np.unique(np.concatenate(list(ratings.batch(1000).map(lambda x:
x["user_id"]))))

# Set up embeddings and dimensions
embedding_dimension = 32

# Create user and movie models using Dense layers
user_model = tf.keras.Sequential([
    tf.keras.layers.StringLookup(
        vocabulary=unique_user_ids, mask_token=None),
    tf.keras.layers.Embedding(len(unique_user_ids) + 1,
embedding_dimension),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(embedding_dimension)
```

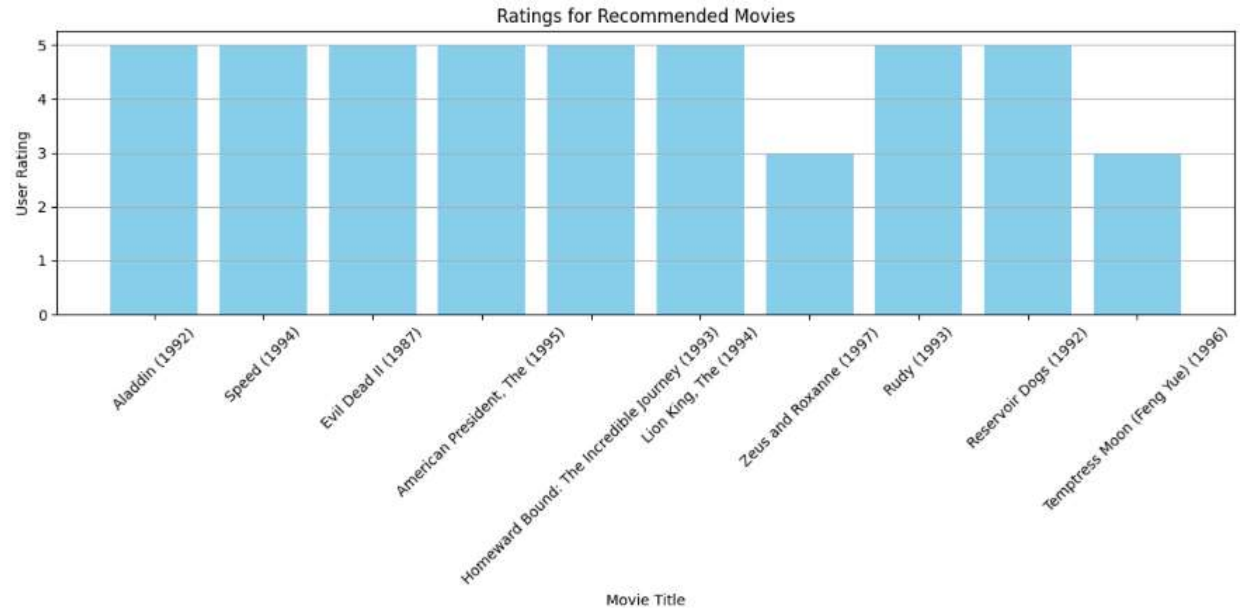
```
])

movie_model = tf.keras.Sequential([
    tf.keras.layers.StringLookup(
        vocabulary=unique_movie_titles, mask_token=None),
    tf.keras.layers.Embedding(len(unique_movie_titles) + 1,
embedding_dimension),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(embedding_dimension)
])

# Set up the retrieval task
task = tfrs.tasks.Retrieval(
    metrics=tfrs.metrics.FactorizedTopK(
        candidates=movies.batch(128).map(movie_model)
    )
)

# Create the model class
class DNNMovielensModel(tfrs.Model):

    def __init__(self, user_model, movie_model, task):
        super().__init__()
        self.user_model: tf.keras.Model = user_model
        self.movie_model: tf.keras.Model = movie_model
        self.task: tf.keras.layers.Layer = task

    def compute_loss(self, features: dict, training=False) -> tf.Tensor:
        user_embeddings = self.user_model(features["user_id"])
        positive_movie_embeddings =
self.movie_model(features["movie_title"])
        return self.task(user_embeddings, positive_movie_embeddings)

# Create an instance of the model
model = DNNMovielensModel(user_model, movie_model, task)
model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.1))

# Shuffle and split the data into training and testing sets
tf.random.set_seed(42)
shuffled = ratings.shuffle(100_000, seed=42,
reshuffle_each_iteration=False)

train = shuffled.take(80_000)
test = shuffled.skip(80_000).take(20_000)
```

```
# Batch and cache the data
cached_train = train.batch(8192).cache()
cached_test = test.batch(4096).cache()

# Train the model
history = model.fit(cached_train, epochs=10)

# Function to get movie recommendations for a user
def get_movie_recommendations(user_id, model, movie_titles, top_k=10):
    user_embedding = model.user_model(tf.constant([user_id]))
    movie_embeddings = model.movie_model(tf.constant(movie_titles))

    scores = tf.matmul(user_embedding, movie_embeddings, transpose_b=True)
    top_indices = tf.argsort(scores, axis=1, direction='DESCENDING')[0,
:top_k].numpy()
    recommended_movies = [movie_titles[i] for i in top_indices]
    return recommended_movies

# Example: Get recommendations for a specific user ID
user_id_example = "42"  # Replace with a valid user ID from your data
recommended_movies = get_movie_recommendations(user_id_example, model,
unique_movie_titles)

# Convert TensorFlow datasets to Pandas DataFrames for visualization
ratings_list = list(ratings.as_numpy_iterator())
ratings_df = pd.DataFrame(ratings_list)

movies_list = list(movies.as_numpy_iterator())
movies_df = pd.DataFrame(movies_list, columns=['movie_title'])

# Assuming 'user_rating' is the correct column name for ratings
ratings_df = ratings_df[['movie_title', 'user_rating']]

# Fetch ratings for recommended movies
recommended_movies_ratings =
ratings_df[ratings_df['movie_title'].isin(recommended_movies)]

# Plot ratings for recommended movies
plt.figure(figsize=(12, 6))
plt.bar(recommended_movies_ratings['movie_title'],
recommended_movies_ratings['user_rating'], color='skyblue')
plt.title('Ratings for Recommended Movies')
plt.xlabel('Movie Title')
plt.ylabel('User Rating')
plt.xticks(rotation=45)
```

```python
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

OUTPUT:

```
Dataset movielens downloaded and prepared to /root/tensorflow_datasets/movielens/100k-movies/0.1.1. Subsequent calls will reuse this data.
Epoch 1/10
10/10 [==============================] - 33s 3s/step - factorized_top_k/top_1_categorical_accuracy: 0.0012 - factorized_top_k/top_5_categorical_accuracy: 0.0046 - factorized_top_k/top_10_categorical
Epoch 2/10
10/10 [==============================] - 31s 3s/step - factorized_top_k/top_1_categorical_accuracy: 8.1250e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0049 - factorized_top_k/top_10_categor
Epoch 3/10
10/10 [==============================] - 35s 3s/step - factorized_top_k/top_1_categorical_accuracy: 3.0000e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0024 - factorized_top_k/top_10_categor
Epoch 4/10
10/10 [==============================] - 32s 3s/step - factorized_top_k/top_1_categorical_accuracy: 4.6250e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0031 - factorized_top_k/top_10_categor
Epoch 5/10
10/10 [==============================] - 31s 3s/step - factorized_top_k/top_1_categorical_accuracy: 5.5000e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0036 - factorized_top_k/top_10_categor
Epoch 6/10
10/10 [==============================] - 33s 3s/step - factorized_top_k/top_1_categorical_accuracy: 3.6250e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0031 - factorized_top_k/top_10_categor
Epoch 7/10
10/10 [==============================] - 33s 3s/step - factorized_top_k/top_1_categorical_accuracy: 4.5000e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0038 - factorized_top_k/top_10_categor
Epoch 8/10
10/10 [==============================] - 32s 3s/step - factorized_top_k/top_1_categorical_accuracy: 4.8750e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0038 - factorized_top_k/top_10_categor
Epoch 9/10
10/10 [==============================] - 31s 3s/step - factorized_top_k/top_1_categorical_accuracy: 5.5000e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0045 - factorized_top_k/top_10_categor
Epoch 10/10
10/10 [==============================] - 31s 3s/step - factorized_top_k/top_1_categorical_accuracy: 5.8750e-04 - factorized_top_k/top_5_categorical_accuracy: 0.0058 - factorized_top_k/top_10_categor
```



THESE ARE 3 RECOMMENDATION MODELS