**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]

))

# 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')

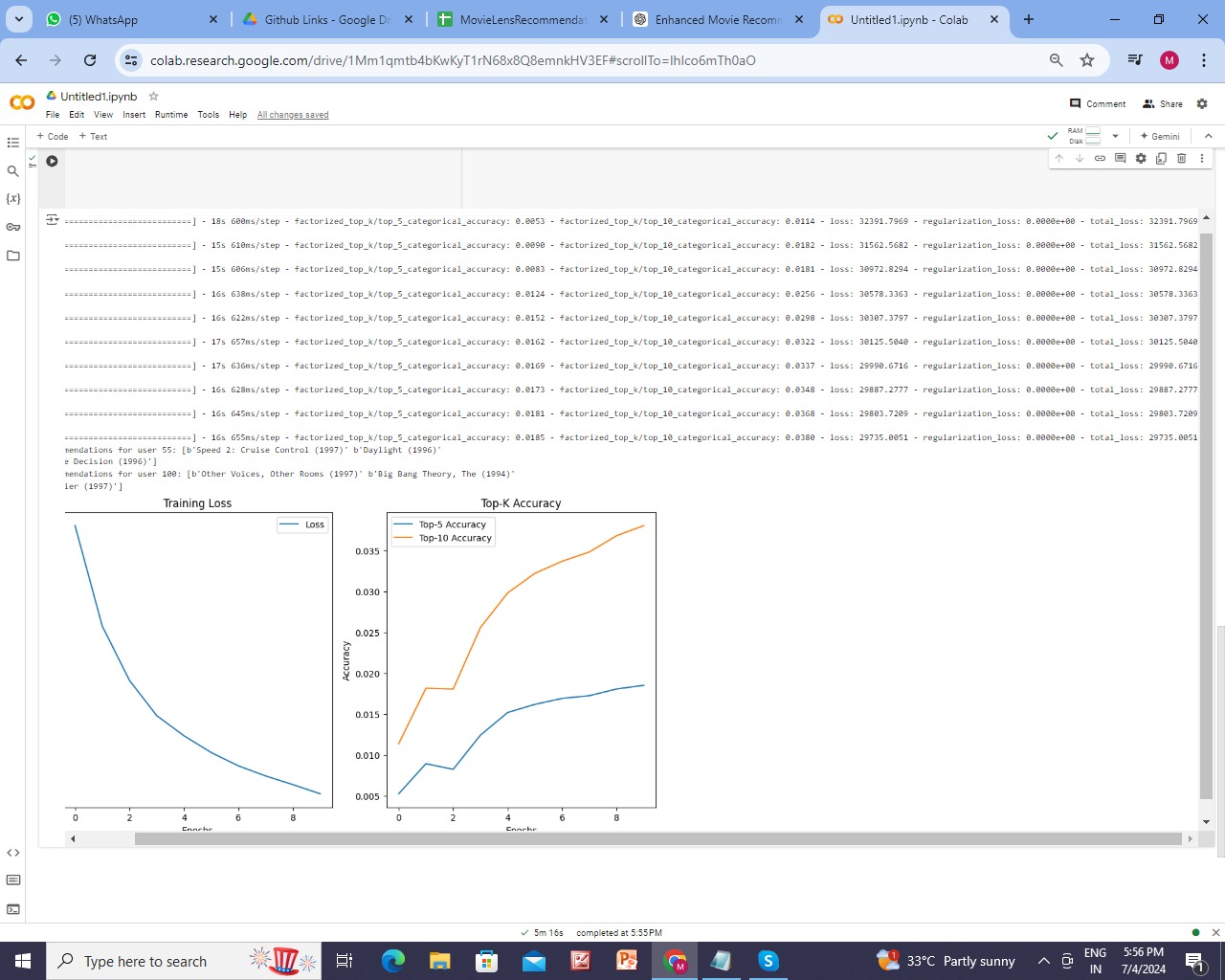
plt.xlabel('Epochs')

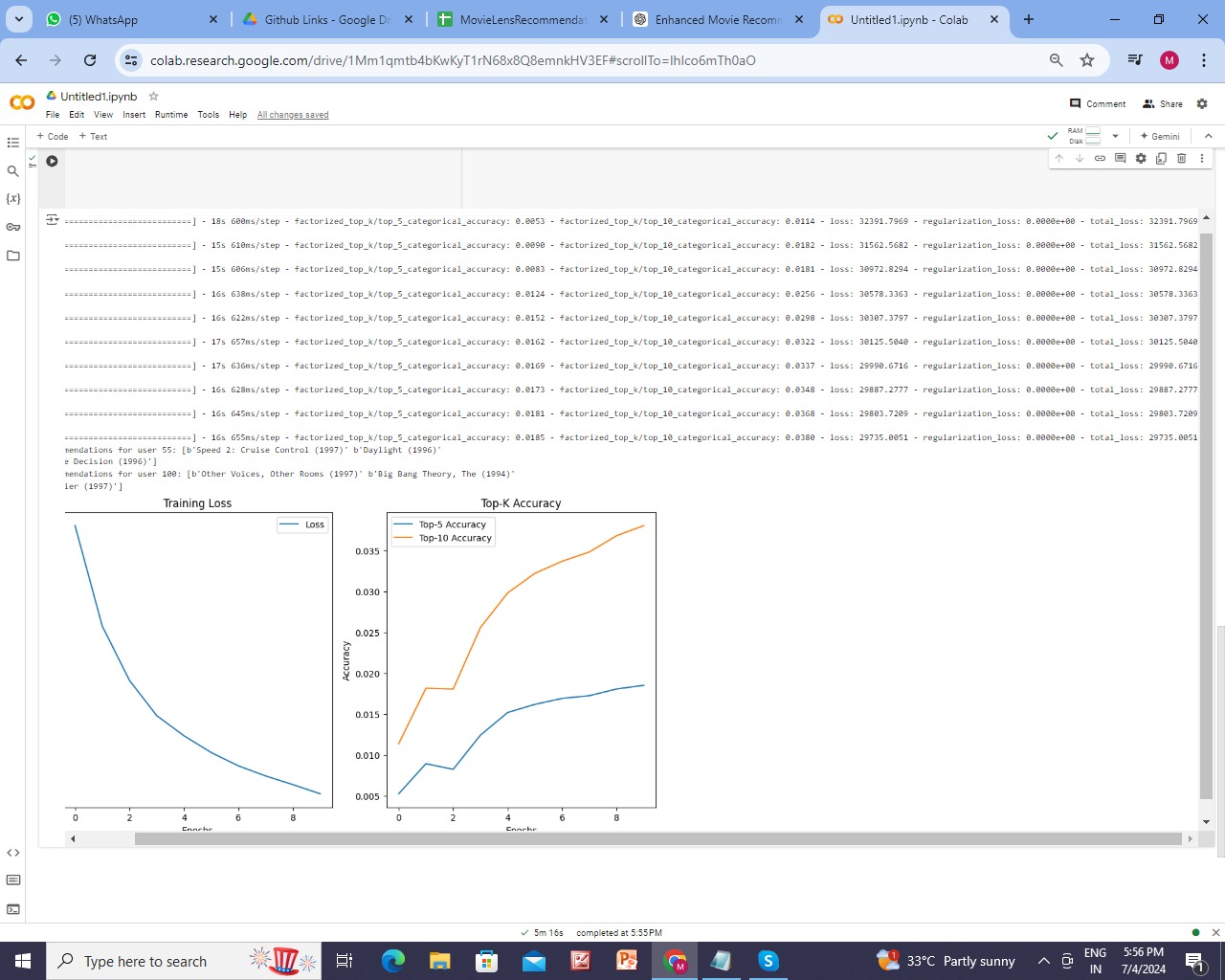
plt.ylabel('Accuracy')

plt.legend()

plt.show()

**OUTPUT**

****

****

**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')

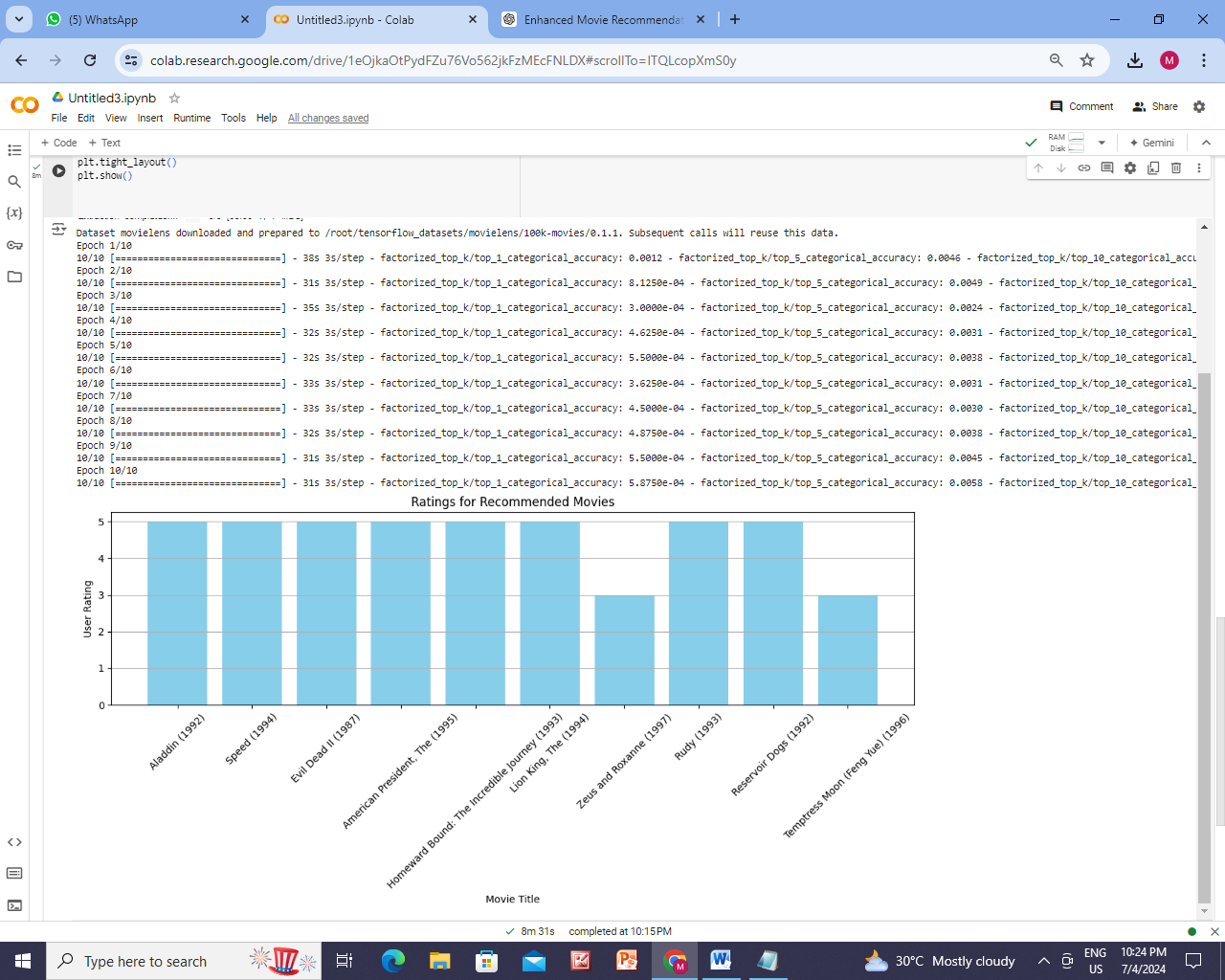
plt.xticks(rotation=45)

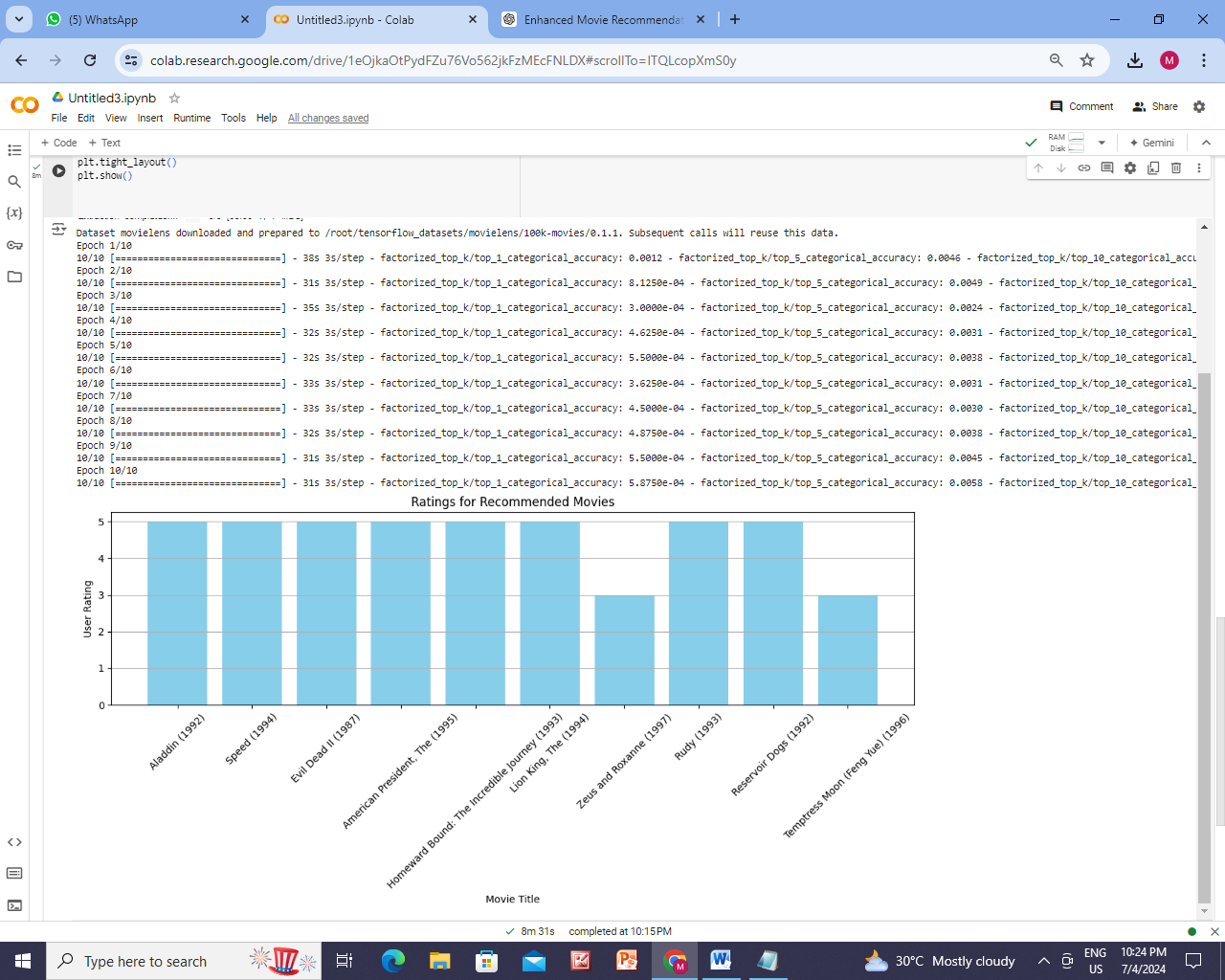
plt.grid(axis='y')

plt.tight\_layout()

plt.show()

**OUTPUT:**





**THESE ARE 3 RECOMMENDATION MODELS**