**Movie Recommendation System**

pip install -q tensorflow-recommenders  
pip install -q --upgrade tensorflow-datasets  
pip install -q scann

import os  
import pprint  
import tempfile  
  
from typing import Dict, Text  
  
import numpy as np  
import tensorflow as tf  
import tensorflow\_datasets as tfds

import tensorflow\_recommenders as tfrs

# Ratings data.  
ratings = tfds.load("movielens/100k-ratings", split="train")  
# Features of all the available movies.  
movies = tfds.load("movielens/100k-movies", split="train")

for x in ratings.take(1).as\_numpy\_iterator():  
  pprint.pprint(x)

for x in movies.take(1).as\_numpy\_iterator():  
  pprint.pprint(x)

ratings = ratings.map(lambda x: {  
    "movie\_title": x["movie\_title"],  
    "user\_id": x["user\_id"],  
})  
movies = movies.map(lambda x: x["movie\_title"])

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)

movie\_titles = movies.batch(1\_000)  
user\_ids = ratings.batch(1\_000\_000).map(lambda x: x["user\_id"])  
  
unique\_movie\_titles = np.unique(np.concatenate(list(movie\_titles)))  
unique\_user\_ids = np.unique(np.concatenate(list(user\_ids)))  
  
unique\_movie\_titles[:10]

embedding\_dimension = 32

user\_model = tf.keras.Sequential([  
  tf.keras.layers.StringLookup(  
      vocabulary=unique\_user\_ids, mask\_token=None),  
  # We add an additional embedding to account for unknown tokens.  
  tf.keras.layers.Embedding(len(unique\_user\_ids) + 1, 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)  
])

metrics = tfrs.metrics.FactorizedTopK(  
  candidates=movies.batch(128).map(movie\_model)  
)

task = tfrs.tasks.Retrieval(  
  metrics=metrics  
)

class MovielensModel(tfrs.Model):  
  
  def \_\_init\_\_(self, user\_model, movie\_model):  
    super().\_\_init\_\_()  
    self.movie\_model: tf.keras.Model = movie\_model  
    self.user\_model: tf.keras.Model = user\_model  
    self.task: tf.keras.layers.Layer = task  
  
  def compute\_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:  
    # We pick out the user features and pass them into the user model.  
    user\_embeddings = self.user\_model(features["user\_id"])  
    # And pick out the movie features and pass them into the movie model,  
    # getting embeddings back.  
    positive\_movie\_embeddings = self.movie\_model(features["movie\_title"])  
  
    # The task computes the loss and the metrics.  
    return self.task(user\_embeddings, positive\_movie\_embeddings)

class NoBaseClassMovielensModel(tf.keras.Model):  
  
  def \_\_init\_\_(self, user\_model, movie\_model):  
    super().\_\_init\_\_()  
    self.movie\_model: tf.keras.Model = movie\_model  
    self.user\_model: tf.keras.Model = user\_model  
    self.task: tf.keras.layers.Layer = task  
  
  def train\_step(self, features: Dict[Text, tf.Tensor]) -> tf.Tensor:  
  
    # Set up a gradient tape to record gradients.  
    with tf.GradientTape() as tape:  
  
      # Loss computation.  
      user\_embeddings = self.user\_model(features["user\_id"])  
      positive\_movie\_embeddings = self.movie\_model(features["movie\_title"])  
      loss = self.task(user\_embeddings, positive\_movie\_embeddings)  
  
      # Handle regularization losses as well.  
      regularization\_loss = sum(self.losses)  
  
      total\_loss = loss + regularization\_loss  
  
    gradients = tape.gradient(total\_loss, self.trainable\_variables)  
    self.optimizer.apply\_gradients(zip(gradients, self.trainable\_variables))  
  
    metrics = {metric.name: metric.result() for metric in self.metrics}  
    metrics["loss"] = loss  
    metrics["regularization\_loss"] = regularization\_loss  
    metrics["total\_loss"] = total\_loss  
  
    return metrics  
  
  def test\_step(self, features: Dict[Text, tf.Tensor]) -> tf.Tensor:  
  
    # Loss computation.  
    user\_embeddings = self.user\_model(features["user\_id"])  
    positive\_movie\_embeddings = self.movie\_model(features["movie\_title"])  
    loss = self.task(user\_embeddings, positive\_movie\_embeddings)  
  
    # Handle regularization losses as well.  
    regularization\_loss = sum(self.losses)  
  
    total\_loss = loss + regularization\_loss  
  
  metrics = {metric.name: metric.result() for metric in self.metrics}  
  metrics["loss"] = loss  
  metrics["regularization\_loss"] = regularization\_loss  
 metrics["total\_loss"] = total\_loss

return metrics

model = MovielensModel(user\_model, movie\_model)  
model.compile(optimizer=tf.keras.optimizers.Adagrad(learning\_rate=0.1))

cached\_train = train.shuffle(100\_000).batch(8192).cache()  
cached\_test = test.batch(4096).cache()

model.fit(cached\_train, epochs=3)

model.evaluate(cached\_test, return\_dict=True)

# Create a model that takes in raw query features, and  
index = tfrs.layers.factorized\_top\_k.BruteForce(model.user\_model)  
# recommends movies out of the entire movies dataset.  
index.index\_from\_dataset(  
  tf.data.Dataset.zip((movies.batch(100), movies.batch(100).map(model.movie\_model)))  
)  
  
# Get recommendations.  
\_, titles = index(tf.constant(["42"]))  
print(f"Recommendations for user 42: {titles[0, :3]}")

# Export the query model.  
with tempfile.TemporaryDirectory() as tmp:  
  path = os.path.join(tmp, "model")  
  
  # Save the index.  
  tf.saved\_model.save(index, path)  
  
  # Load it back; can also be done in TensorFlow Serving.  
  loaded = tf.saved\_model.load(path)  
  
  # Pass a user id in, get top predicted movie titles back.  
  scores, titles = loaded(["42"])  
  
  print(f"Recommendations: {titles[0][:3]}")

scann\_index = tfrs.layers.factorized\_top\_k.ScaNN(model.user\_model)  
scann\_index.index\_from\_dataset(  
  tf.data.Dataset.zip((movies.batch(100), movies.batch(100).map(model.movie\_model)))  
)

# Get recommendations.  
\_, titles = scann\_index(tf.constant(["42"]))  
print(f"Recommendations for user 42: {titles[0, :3]}")

# Export the query model.  
with tempfile.TemporaryDirectory() as tmp:  
  path = os.path.join(tmp, "model")  
  
  # Save the index.  
  tf.saved\_model.save(  
      scann\_index,  
      path,  
      options=tf.saved\_model.SaveOptions(namespace\_whitelist=["Scann"])  
  )  
  
  # Load it back; can also be done in TensorFlow Serving.  
  loaded = tf.saved\_model.load(path)  
  
  # Pass a user id in, get top predicted movie titles back.  
  scores, titles = loaded(["42"])  
  
  print(f"Recommendations: {titles[0][:3]}")