Multi-task recommenders

In [1]:

```
from typing import Dict, Text
from firebase import firebase

import os
import pprint
import tempfile
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import tensorflow_recommenders as tfrs
```

Preparing the dataset

In [2]:

```
firebase = firebase.FirebaseApplication('https://thesis-bd8c8-default-rtdb.europe-west1.
firebase_ratings = firebase.get('/User_Book', None)
firebase_books = firebase.get('/Books', None)
```

In [3]:

```
ratings_df = pd.DataFrame.from_dict(firebase_ratings, orient='index')
books_df = pd.DataFrame.from_dict(firebase_books, orient='index')
```

In [4]:

```
ratings_df.head()
```

Out[4]:

	bookId	isbn	myRate	
-NWcEFgMwmdaYc-4ndml	-NWcEFeoVaEgjMwiboCA	09781565841000	5.0	3avc3TUJioP8X(
-NWcEGniRUJ0hRQmvclo	-NWcEGmQeH3eiHRZ0w7G	09781929610259	4.5	3avc3TUJioP8X(
-NWcEHtzsgGbN7rpUZ6V	-NWcEHsFfJ3yqXud1NGo	09780814326114	4.0	3avc3TUJioP8X(
-NWcEJunG4PfX-aAVxiF	-NWcEJqjxIyMQv5izHrS	09780312010447	4.5	3avc3TUJioP8X(
-NWcELFoI68JZpsfvkyF	-NWcELEEV0QR6EHobxIe	09780143036357	3.5	3avc3TUJioP8X(

In [5]:

```
len(set(books_df["isbn"]))
```

Out[5]:

709

In [6]:

```
books_df.tail()
```

Out[6]:

	author	description	documentId	genre	
-NZ0yycG_Apty0y1IjWF	J. K. Rowling	Harry Potter is lucky to reach the age of thir...		England	https://books.google.
-NZ2i9tDuY_wzMrIGvkP	Fyodor Dostoyevsky	A man must endure relentless physical and ment...		Fiction	https://books.google.
-NZ2iWjGwTf1XV4Cn7B_	Stephen Hawking	Stephen Hawking's A Brief History of Time has ...		Cosmology	https://books.google.
-NZ2ie-H6GjeHTjkHyLW	Harper Lee	At the age of eight, Scout Finch is an entrenc...		Fiction	https://books.google.
-NZ5ZECUtFhEJGuJMh_I	carte	carte		gen	

In [7]:

```
len(books_df)
```

Out[7]:

1022

In [8]:

```
#drop manually added books
books_df = books_df[books_df['rating'] != '']
```

In [9]:

```
len(books_df)
```

Out[9]:

1021

In [10]:

```
books_df = pd.DataFrame(set(books_df["isbn"]),columns=["isbn"])
```

In [11]:

```
books_df
```

Out[11]:

	isbn
0	09780851621814
1	09780393058260
2	09780835608305
3	09780751504354
4	09780743222983
...	...
703	09780805211160
704	09780813515243
705	09780761929949
706	09780374526962
707	09780375759314

708 rows × 1 columns

In [12]:

```
#transforms dataframes in datasets with tensor
ratings_dataset = tf.data.Dataset.from_tensor_slices(dict(ratings_df))
books_dataset = tf.data.Dataset.from_tensor_slices(dict(books_df))
```

In [13]:

```
ratings = ratings_dataset.map(lambda x: {
    "book_isbn": x["isbn"],
    "user_id": x["userId"],
    "user_rating": x["myRate"],
})
books = books_dataset.map(lambda x: x["isbn"])
```

In [14]:

```
ratings
```

Out[14]:

```
<_MapDataset element_spec={'book_isbn': TensorSpec(shape=(), dtype=tf.stri
ng, name=None), 'user_id': TensorSpec(shape=(), dtype=tf.string, name=Non
e), 'user_rating': TensorSpec(shape=(), dtype=tf.float64, name=None)}>
```

In [15]:

```
books
```

Out[15]:

```
<_MapDataset element_spec=TensorSpec(shape=(), dtype=tf.string, name=None)
>
```

In [16]:

```
len(books)
```

Out[16]:

```
708
```

In [17]:

```
len(ratings)
```

Out[17]:

```
1022
```

In [18]:

```
# Randomly shuffle data and split between train and test.
tf.random.set_seed(42)
shuffled = ratings.shuffle(len(ratings_df), seed=42, reshuffle_each_iteration=False)

train = shuffled.take(800)
test = shuffled.skip(800).take(210)

book_isbns = books.batch(25) #100 50
user_ids = ratings.batch(10_000).map(lambda x: x["user_id"])

unique_book_isbns = np.unique(np.concatenate(list(book_isbns)))
unique_user_ids = np.unique(np.concatenate(list(user_ids)))
```

A multi-task model

In [19]:

```
class BookModel(tfrs.models.Model):

    def __init__(self, rating_weight: float, retrieval_weight: float) -> None:
        super().__init__()

        embedding_dimension = 64 #32

        # User and book models.
        self.book_model: tf.keras.layers.Layer = tf.keras.Sequential([
          tf.keras.layers.StringLookup(
            vocabulary=unique_book_isbns, mask_token=None),
          tf.keras.layers.Embedding(len(unique_book_isbns) + 1, embedding_dimension)
        ])
        self.user_model: tf.keras.layers.Layer = 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)
        ])

        # Rating model
        self.rating_model = tf.keras.Sequential([
            tf.keras.layers.Dense(512, activation="relu"),
            tf.keras.layers.Dropout(0.3),
            tf.keras.layers.Dense(256, activation="relu"),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.Dense(128, activation="relu"),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.Dense(32, activation="relu"),
            tf.keras.layers.Dense(1),
        ])

        # The tasks.
        self.rating_task: tf.keras.layers.Layer = tfrs.tasks.Ranking(
            loss=tf.keras.losses.MeanSquaredError(),
            metrics=[tf.keras.metrics.RootMeanSquaredError()],
        )
        self.retrieval_task: tf.keras.layers.Layer = tfrs.tasks.Retrieval(
            metrics=tfrs.metrics.FactorizedTopK(
                candidates=books.batch(16).map(self.book_model) #128 32
            )
        )

        # The loss weights.
        self.rating_weight = rating_weight
        self.retrieval_weight = retrieval_weight

    def call(self, features: Dict[Text, tf.Tensor]) -> tf.Tensor:

        user_embeddings = self.user_model(features["user_id"])

        book_embeddings = self.book_model(features["book_isbn"])

        return (
            user_embeddings,
            book_embeddings,
            # We apply the multi-layered rating model to a concatentation of
            # user and book embeddings.
            self.rating_model(
                tf.concat([user_embeddings, book_embeddings], axis=1)
```

```
            ),
        )

    def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor

        ratings = features.pop("user_rating")

        user_embeddings, book_embeddings, rating_predictions = self(features)

        # We compute the loss for each task.
        rating_loss = self.rating_task(
            labels=ratings,
            predictions=rating_predictions,
        )
        retrieval_loss = self.retrieval_task(user_embeddings, book_embeddings)
        # And combine them using the loss weights.
        return (self.rating_weight * rating_loss
                + self.retrieval_weight * retrieval_loss)
```

Rating-specialized model

In [20]:

```
model = BookModel(rating_weight=1.0, retrieval_weight=0.0)
model.compile(optimizer=tf.keras.optimizers.AdamW(0.005))
```

In [21]:

```
cached_train = train.shuffle(800).batch(32).cache() #128 64
cached_test = test.batch(16).cache() #64 32
```

In [22]:

```
history_rating_model = model.fit(cached_train, epochs= 100)
metrics = model.evaluate(cached_test, return_dict=True)

print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accur
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

```
Epoch 1/100
25/25 [==============================] - 6s 70ms/step - root_mean_squar
ed_error: 1.8685 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
+00 - factorized_top_k/top_5_categorical_accuracy: 0.0050 - factorized_
top_k/top_10_categorical_accuracy: 0.0088 - factorized_top_k/top_50_cat
egorical_accuracy: 0.0700 - factorized_top_k/top_100_categorical_accura
cy: 0.1375 - loss: 3.4294 - regularization_loss: 0.0000e+00 - total_los
s: 3.4294
Epoch 2/100
25/25 [==============================] - 2s 86ms/step - root_mean_squar
ed_error: 1.1805 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
+00 - factorized_top_k/top_5_categorical_accuracy: 0.0025 - factorized_
top_k/top_10_categorical_accuracy: 0.0100 - factorized_top_k/top_50_cat
egorical_accuracy: 0.0675 - factorized_top_k/top_100_categorical_accura
cy: 0.1550 - loss: 1.3995 - regularization_loss: 0.0000e+00 - total_los
s: 1.3995
Epoch 3/100
25/25 [==============================] - 2s 78ms/step - root_mean_squar
ed_error: 1.0729 - factorized_top_k/top_1_categorical_accuracy: 0.0012
```

Retrieval-specialized model

In [23]:

```
model = BookModel(rating_weight=0.0, retrieval_weight=1.0)
model.compile(optimizer=tf.keras.optimizers.AdamW(0.005))
```

In [24]:

```
history_retrieval_model = model.fit(cached_train, epochs= 100)
metrics = model.evaluate(cached_test, return_dict=True)

print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accur
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

```
Epoch 1/100
25/25 [==============================] - 4s 68ms/step - root_mean_squar
ed_error: 3.7004 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
+00 - factorized_top_k/top_5_categorical_accuracy: 0.0025 - factorized_
top_k/top_10_categorical_accuracy: 0.0075 - factorized_top_k/top_50_cat
egorical_accuracy: 0.0725 - factorized_top_k/top_100_categorical_accura
cy: 0.1425 - loss: 110.9035 - regularization_loss: 0.0000e+00 - total_l
oss: 110.9035
Epoch 2/100
25/25 [==============================] - 2s 67ms/step - root_mean_squar
ed_error: 3.7007 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
+00 - factorized_top_k/top_5_categorical_accuracy: 0.3375 - factorized_
top_k/top_10_categorical_accuracy: 0.8988 - factorized_top_k/top_50_cat
egorical_accuracy: 0.9950 - factorized_top_k/top_100_categorical_accura
cy: 0.9987 - loss: 108.4568 - regularization_loss: 0.0000e+00 - total_l
oss: 108.4568
Epoch 3/100
25/25 [==============================] - 2s 69ms/step - root_mean_squar
ed_error: 3.7017 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
```

Joint model

In [25]:

```
model = BookModel(rating_weight=1.0, retrieval_weight=1.0)
model.compile(optimizer=tf.keras.optimizers.AdamW(0.005))
```

In [26]:

```
history_joint_model = model.fit(cached_train, epochs= 100)
metrics = model.evaluate(cached_test, return_dict=True)

print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accur
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

```
Epoch 1/100
25/25 [==============================] - 4s 73ms/step - root_mean_squar
ed_error: 1.8516 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
+00 - factorized_top_k/top_5_categorical_accuracy: 0.0037 - factorized_
top_k/top_10_categorical_accuracy: 0.0113 - factorized_top_k/top_50_cat
egorical_accuracy: 0.0662 - factorized_top_k/top_100_categorical_accura
cy: 0.1338 - loss: 114.2730 - regularization_loss: 0.0000e+00 - total_l
oss: 114.2730
Epoch 2/100
25/25 [==============================] - 2s 70ms/step - root_mean_squar
ed_error: 1.2139 - factorized_top_k/top_1_categorical_accuracy: 0.0012
- factorized_top_k/top_5_categorical_accuracy: 0.2587 - factorized_top_
k/top_10_categorical_accuracy: 0.3938 - factorized_top_k/top_50_categor
ical_accuracy: 0.6463 - factorized_top_k/top_100_categorical_accuracy:
0.7675 - loss: 111.3764 - regularization_loss: 0.0000e+00 - total_loss:
111.3764
Epoch 3/100
25/25 [==============================] - 2s 72ms/step - root_mean_squar
ed_error: 1.0910 - factorized_top_k/top_1_categorical_accuracy: 0.0000e
```

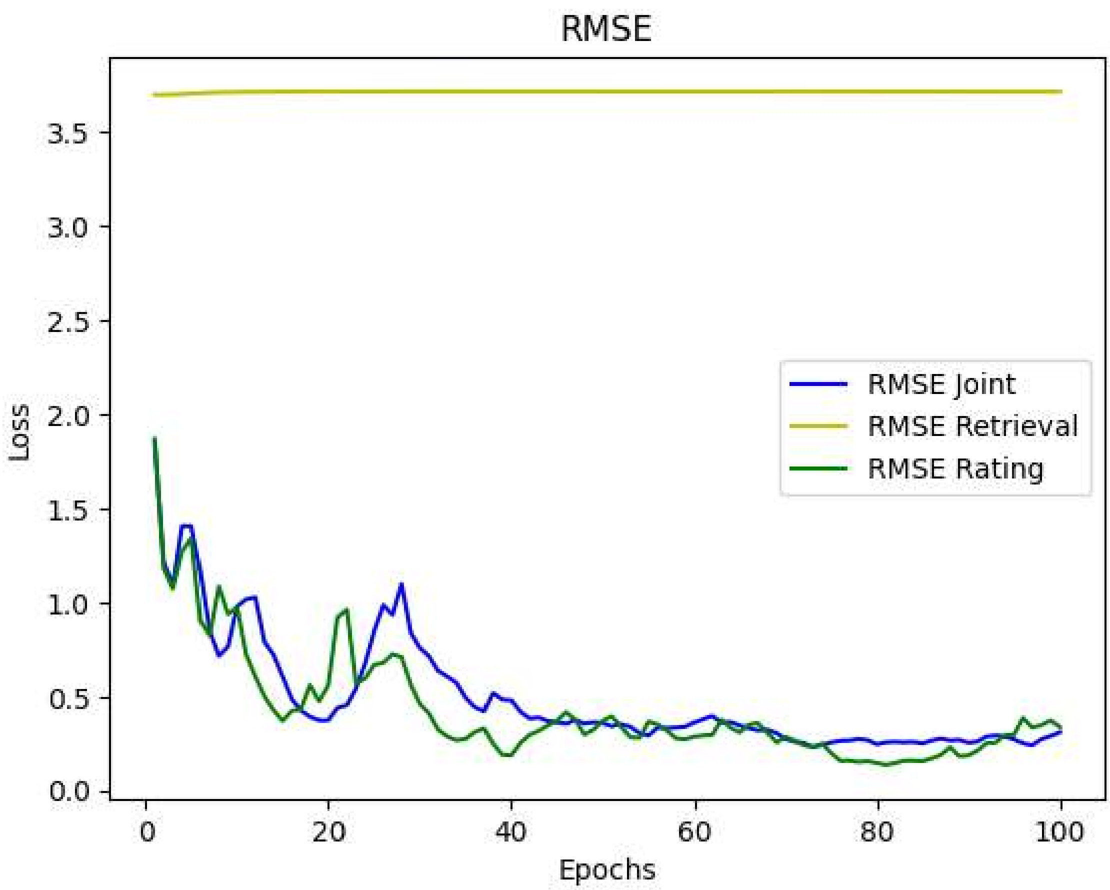
DATA

In [27]:

```
def draw_plot(name, joint, retrieval, rating):
    epochs = range(1, len(joint) + 1)

    plt.plot(epochs, joint, 'b', label= name +' Joint')
    plt.plot(epochs, retrieval, 'y', label=name + ' Retrieval')
    plt.plot(epochs, rating, 'g', label=name + ' Rating')
    plt.title(name)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

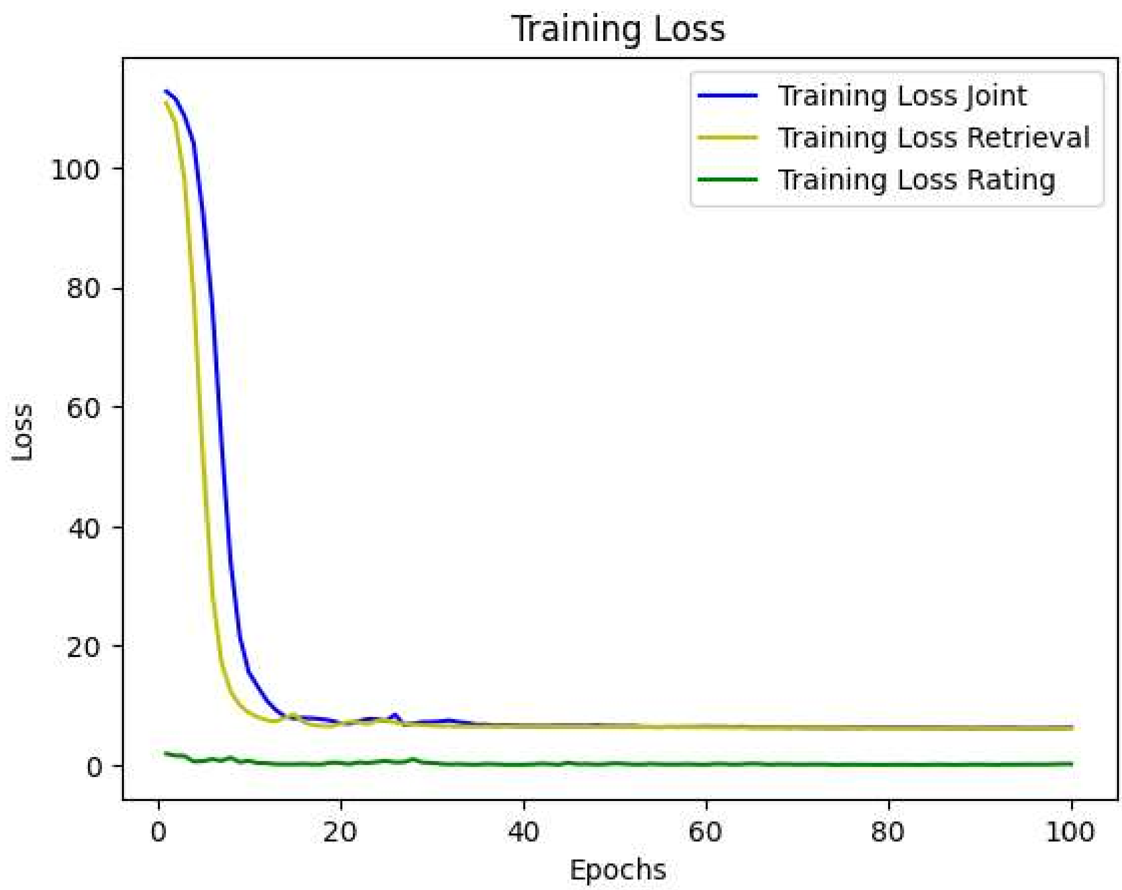
In [28]:

```
RMSE_joint = history_joint_model.history['root_mean_squared_error']
RMSE_retrieval = history_retrieval_model.history['root_mean_squared_error']
RMSE_rating = history_rating_model.history['root_mean_squared_error']

draw_plot('RMSE', RMSE_joint,RMSE_retrieval,RMSE_rating)
```



In [29]:

```
total_loss_values_joint = history_joint_model.history['total_loss']
total_loss_values_retrieval = history_retrieval_model.history['total_loss']
total_loss_values_rating = history_rating_model.history['total_loss']

draw_plot('Training Loss', total_loss_values_joint,total_loss_values_retrieval,total_los
```



Making predictions

In [30]:

```
trained_movie_embeddings, trained_user_embeddings, predicted_rating = model({
      "user_id": np.array(["3avc3TUJioP8XGD0bLK9xtV7uIG3"]),
      "book_isbn": np.array(["09781880685358"])
  })
print("Predicted rating:")
print(predicted_rating)
```

```
Predicted rating:
tf.Tensor([[0.7585014]], shape=(1, 1), dtype=float32)
```

In [31]:

```
user_id = ["3avc3TUJioP8XGD0bLK9xtV7uIG3"]
isbn_list = set(books_df["isbn"])
for isbn in isbn_list:
    trained_movie_embeddings, trained_user_embeddings, predicted_rating = model({
      "user_id": np.array(user_id),
      "book_isbn": np.array([isbn])
    })
    print(predicted_rating)
```

```
tf.Tensor([[3.6453257]], shape=(1, 1), dtype=float32)
tf.Tensor([[2.9264867]], shape=(1, 1), dtype=float32)
tf.Tensor([[2.7685647]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.0897474]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.52448]], shape=(1, 1), dtype=float32)
tf.Tensor([[3.8613882]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.6079345]], shape=(1, 1), dtype=float32)
tf.Tensor([[3.7873693]], shape=(1, 1), dtype=float32)
tf.Tensor([[0.76930666]], shape=(1, 1), dtype=float32)
tf.Tensor([[2.1546063]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.635467]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.060837]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.189184]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.3952985]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.6988673]], shape=(1, 1), dtype=float32)
tf.Tensor([[4.52448]], shape=(1, 1), dtype=float32)
tf.Tensor([[3.9922564]], shape=(1, 1), dtype=float32)
tf.Tensor([[3.2534752]], shape=(1, 1), dtype=float32)
tf.Tensor([[3.8547888]], shape=(1, 1), dtype=float32)
```

Save the model

In [32]:

```
model.retrieval_task = tfrs.tasks.Retrieval()  # Removes the metrics.
model.compile()
model.save("final_model")
```

```
WARNING:tensorflow:Skipping full serialization of Keras layer <tensorflow_
recommenders.tasks.retrieval.Retrieval object at 0x00000135DC551010>, beca
use it is not built.
INFO:tensorflow:Assets written to: final_model\assets

INFO:tensorflow:Assets written to: final_model\assets
```

In [33]:

```
# Load model
model = tf.keras.models.load_model("final_model")
```

In [34]:

```
# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model("final_model") # path to the SavedM
tflite_model = converter.convert()

# Save the model.
with open('final_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Tf Lite

In [35]:

```
interpreter = tf.lite.Interpreter(model_path="final_model.tflite")
interpreter.allocate_tensors()

#Get input details
input_details = interpreter.get_input_details()
for input_tensor in input_details:
    print("Input name:", input_tensor["name"])
    print("Input shape:", input_tensor["shape"])
    print("Input data type:", input_tensor["dtype"])
    print()
#Get output details
output_details = interpreter.get_output_details()
for output_tensor in output_details:
    print("Output name:", output_tensor["name"])
    print("Output shape:", output_tensor["shape"])
    print("Output data type:", output_tensor["dtype"])
    print()
```

```
Input name: serving_default_book_isbn:0
Input shape: [1]
Input data type: <class 'numpy.bytes_'>

Input name: serving_default_user_id:0
Input shape: [1]
Input data type: <class 'numpy.bytes_'>

Output name: StatefulPartitionedCall:0
Output shape: []
Output data type: <class 'numpy.float32'>

Output name: StatefulPartitionedCall:2
Output shape: [1 1]
Output data type: <class 'numpy.float32'>

Output name: StatefulPartitionedCall:1
Output shape: []
Output data type: <class 'numpy.float32'>
```

In [36]:

```
# Prepare the input data
input_data_isbn = np.array([b'09780143036357'], dtype=np.bytes_)
input_data_user_id = np.array(['3avc3TUJioP8XGD0bLK9xtV7uIG3'], dtype=np.bytes_)

input_details = interpreter.get_input_details()
interpreter.set_tensor(input_details[0]['index'], input_data_isbn)
interpreter.set_tensor(input_details[1]['index'], input_data_user_id)

# Run the inference
interpreter.invoke()

# Retrieve the output results
output_details = interpreter.get_output_details()

output_data_prediction = interpreter.get_tensor(output_details[0]['index'])
output_data_probabilities = interpreter.get_tensor(output_details[1]['index'])
output_data_score = interpreter.get_tensor(output_details[2]['index'])

# Process the output
#prediction = output_data_prediction.squeeze()
probabilities = output_data_probabilities.squeeze()
#score = output_data_score.squeeze()

# Print the results
#print("Prediction:", prediction)
print("Probability:", probabilities)
#print("Score:", score)
```

```
Probability: 3.3923666
```

Tensorflow recommenders

Brute Force

In [37]:

```
# Create a model that takes in raw query features, and
index = tfrs.layers.factorized_top_k.BruteForce(model.user_model)
# recommends books out of the entire books dataset.
index.index_from_dataset(
  tf.data.Dataset.zip((books.batch(100), books.batch(100).map(model.book_model)))
)

# Get recommendations.
_, isbns = index(np.array(["Pgzb07La4DUNOhYPzYXHA7CdfNi1"]))
print(f"Recommendations for user: {isbns[0, :10]}")
```

```
Recommendations for user: [b'09781857024074' b'09780375701801' b'097804650
14903' b'09780140286014'
 b'09781592289806' b'09781859843406' b'09780618257768' b'09780590428880'
 b'09780226142814' b'09780195309683']
```

Save the brute force model

In [38]:

```
index.save("final_model")
```

```
WARNING:tensorflow:Model's `__init__()` arguments contain non-serializa
ble objects. Please implement a `get_config()` method in the subclassed
Model for proper saving and loading. Defaulting to empty config.

WARNING:tensorflow:Model's `__init__()` arguments contain non-serializa
ble objects. Please implement a `get_config()` method in the subclassed
Model for proper saving and loading. Defaulting to empty config.

WARNING:tensorflow:Model's `__init__()` arguments contain non-serializa
ble objects. Please implement a `get_config()` method in the subclassed
Model for proper saving and loading. Defaulting to empty config.

WARNING:tensorflow:Model's `__init__()` arguments contain non-serializa
ble objects. Please implement a `get_config()` method in the subclassed
Model for proper saving and loading. Defaulting to empty config.

INFO:tensorflow:Assets written to: final_model\assets

INFO:tensorflow:Assets written to: final_model\assets

WARNING:tensorflow:Model's `__init__()` arguments contain non-serializa
```

In [39]:

```
# test loading
loaded = tf.saved_model.load("final_model")

# Pass a user id in, get top predicted movie titles back.
scores, isbns = loaded(["Pgzb07La4DUNOhYPzYXHA7CdfNi1"])

print(f"Recommendations: {isbns[0][:5]}")
```

```
Recommendations: [b'09781857024074' b'09780375701801' b'09780465014903'
b'09780140286014'
 b'09781592289806']
```

TFLite for the model

In [40]:

```
# Convert the model
converter = tf.lite.TFLiteConverter.from_saved_model("final_model") # path to the SavedM
tflite_model = converter.convert()

# Save the model.
with open('final_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Tf lite testing

In [41]:

```
interpreter = tf.lite.Interpreter(model_path="final_model.tflite")
interpreter.allocate_tensors()

#Get input details
input_details = interpreter.get_input_details()
for input_tensor in input_details:
    print("Input name:", input_tensor["name"])
    print("Input shape:", input_tensor["shape"])
    print("Input data type:", input_tensor["dtype"])
    print()
#Get output details
output_details = interpreter.get_output_details()
for output_tensor in output_details:
    print("Output name:", output_tensor["name"])
    print("Output shape:", output_tensor["shape"])
    print("Output data type:", output_tensor["dtype"])
    print()
```

```
Input name: serving_default_input_1:0
Input shape: [1]
Input data type: <class 'numpy.bytes_'>

Output name: StatefulPartitionedCall_1:0
Output shape: [ 1 10]
Output data type: <class 'numpy.float32'>

Output name: StatefulPartitionedCall_1:1
Output shape: [ 1 10]
Output data type: <class 'numpy.bytes_'>
```

In [42]:

```
# Prepare the input data
input_data = np.array(["zwVJUfdC0oa9hWWp9uK0hRTM71j1"], dtype=np.bytes_)

input_details = interpreter.get_input_details()
interpreter.set_tensor(input_details[0]['index'], input_data)

# Run the inference
interpreter.invoke()

# Retrieve the output results
output_details = interpreter.get_output_details()

output_data_prediction = interpreter.get_tensor(output_details[0]['index'])
output_data_classes = interpreter.get_tensor(output_details[1]['index'])

# Process the output
#prediction = output_data_prediction.squeeze()
classes = output_data_classes.squeeze().astype(str)

# Print the results
#print("Prediction:", prediction)
print("Classes:", classes)
```

```
Classes: ['09780851621814' '09780446617451' '09780253203182' '097807475736
23'
 '09780143104902' '09781841492667' '09781572244252' '09780452270848'
 '09781582406930' '09780618710539']
```

In [43]:

```
import firebase_admin
from firebase_admin import ml
from firebase_admin import credentials

firebase_admin.initialize_app(
  credentials.Certificate('thesis-bd8c8-firebase-adminsdk-fqj6e-e1d094b473.json'),
  options={
      'storageBucket': 'thesis-bd8c8.appspot.com',
  })
```

Out[43]:

```
<firebase_admin.App at 0x135ea8eb190>
```

In [44]:

```
#Upload model
```

In [45]:

```
source = ml.TFLiteGCSModelSource.from_tflite_model_file('final_model.tflite')
tflite_format = ml.TFLiteFormat(model_source=source)
model = ml.Model(display_name="final_model", model_format=tflite_format)
new_model = ml.create_model(model)
ml.publish_model(new_model.model_id)
print(new_model.model_id)
```

```
21877933
```

In [46]:

```
#Update model
```

In [47]:

```
model = ml.get_model(new_model.model_id)
source = ml.TFLiteGCSModelSource.from_tflite_model_file('final_model.tflite')
model.mode_format = ml.TFLiteFormat(model_source=source)
model.display_name = "final_model"
updated_model = ml.update_model(model)
ml.publish_model(updated_model.model_id)
```

Out[47]:

```
<firebase_admin.ml.Model at 0x135e9707210>
```

In [ ]:

```
```