Course Rating Prediction using Neural Networks

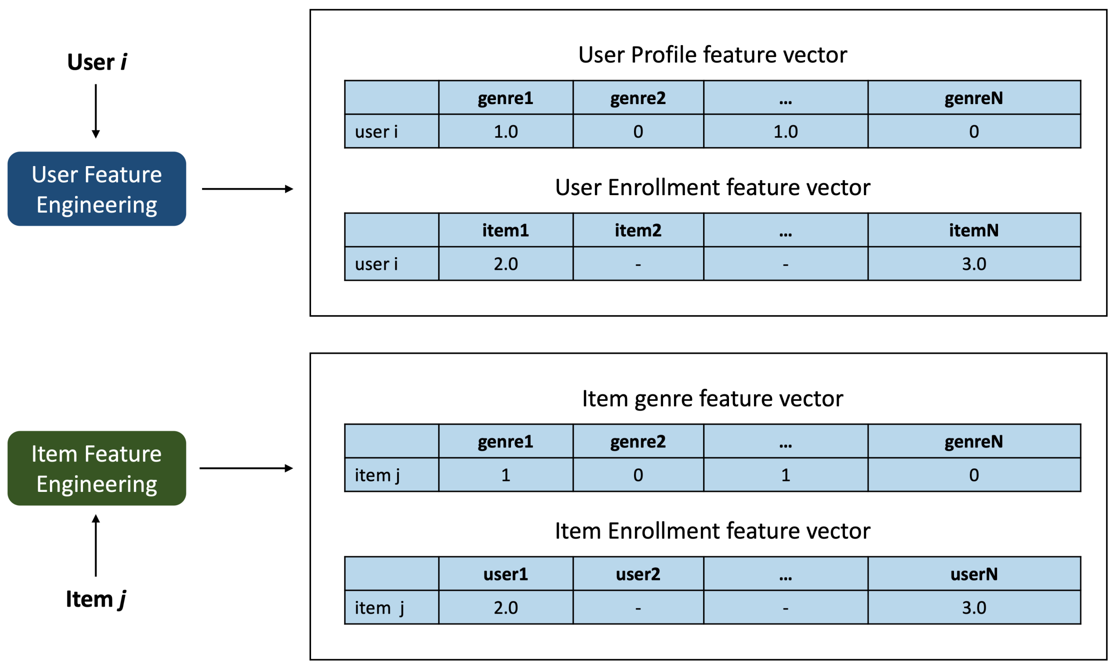
Estimated time needed: **60** minutes

In the previous labs, we have crafted several types of user and item feature vectors. For example, given a user `i`, we may build its profile feature vector and course rating feature vector, and given an item `j`, we may create its genre vector and user enrollment vectors.

With these explicit features vectors, we can perform machine learning tasks such as calculating the similarities among users or items, finding nearest neighbors, and using dot-product to estimate a rating value.

The main advantage of using these explicit features is they are highly interpretable and yield very good performance as well. The main disadvantage is we need to spend quite some effort to build and store them.

Explicit User and Item Feature Engineering



User Profile feature vector

	genre1	genre2	...	genreN
user i	1.0	0	1.0	0

User Enrollment feature vector

	item1	item2	...	itemN
user i	2.0	-	-	3.0

Item genre feature vector

	genre1	genre2	...	genreN
item j	1	0	1	0

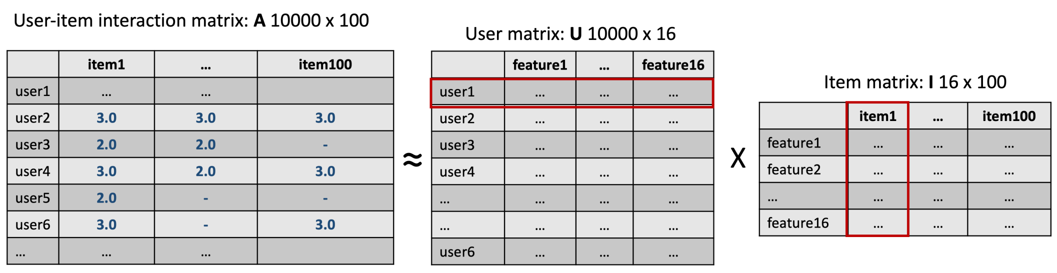
Item Enrollment feature vector

	user1	user2	...	userN
item j	2.0	-	-	3.0

Is it possible to predict a rating without building explicit feature vectors beforehand?

Yes, as you may recall, the Non-negative Matrix Factorization decomposes the user-item interaction matrix into user matrix and item matrix, which contain the latent features of users and items and you can simply dot-product them to get an estimated rating.

Non-negative Matrix Factorization



User-item interaction matrix: **A** 10000 x 100

	item1	...	item100
user1	...	...	
user2	3.0	3.0	3.0
user3	2.0	2.0	-
user4	3.0	2.0	3.0
user5	2.0	-	-
user6	3.0	-	3.0
...	...	...	

≈

User matrix: **U** 10000 x 16

	feature1	...	feature16
user1	...	...	...
user2	...	...	...
user3	...	...	...
user4	...	...	...
...	...	...	...
...	...	...	...
user6	...	...	...

X

Item matrix: **I** 16 x 100

	item1	...	item100
feature1	...	...	...
feature2	...	...	...
...	...	...	...
feature16	...	...	...

In addition to NMF, neural networks can also be used to extract the latent user and item features In fact, neural networks are very good at learning patterns from data and are widely used to extract latent features. When training neural networks, it gradually captures and stores the features within its hidden layers as weight matrices and can be extracted to represent the original data.

In this lab, you will be training neural networks to predict course ratings while simultaneously extracting users' and items' latent features.

Objectives

After completing this lab you will be able to:

- Use `tensorflow` to train neural networks to extract the user and item latent features from the hidden's layers
- Predict course ratings with trained neural networks

Prepare and setup lab environment

Install tensorflow if not installed before in your Python environment

In [1]:
```
%pip install tensorflow
```

```
Requirement already satisfied: tensorflow in /opt/conda/lib/python3.12/site-packa
ges (2.20.0)
Requirement already satisfied: absl-py>=1.0.0 in /opt/conda/lib/python3.12/site-p
ackages (from tensorflow) (2.3.1)
Requirement already satisfied: astunparse>=1.6.0 in /opt/conda/lib/python3.12/sit
e-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /opt/conda/lib/python3.12/
site-packages (from tensorflow) (25.12.19)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /opt/conda/
lib/python3.12/site-packages (from tensorflow) (0.7.0)
Requirement already satisfied: google_pasta>=0.1.1 in /opt/conda/lib/python3.12/s
ite-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in /opt/conda/lib/python3.12/site
-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt_einsum>=2.3.2 in /opt/conda/lib/python3.12/sit
e-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /opt/conda/lib/python3.12/site-packag
es (from tensorflow) (24.2)
Requirement already satisfied: protobuf>=5.28.0 in /opt/conda/lib/python3.12/site
-packages (from tensorflow) (6.33.4)
Requirement already satisfied: requests<3,>=2.21.0 in /opt/conda/lib/python3.12/s
ite-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.12/site-packa
ges (from tensorflow) (75.8.0)
Requirement already satisfied: six>=1.12.0 in /opt/conda/lib/python3.12/site-pack
ages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /opt/conda/lib/python3.12/site
-packages (from tensorflow) (3.3.0)
Requirement already satisfied: typing_extensions>=3.6.6 in /opt/conda/lib/python
3.12/site-packages (from tensorflow) (4.12.2)
Requirement already satisfied: wrapt>=1.11.0 in /opt/conda/lib/python3.12/site-pa
ckages (from tensorflow) (2.0.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /opt/conda/lib/python3.12/s
ite-packages (from tensorflow) (1.76.0)
Requirement already satisfied: tensorboard~=2.20.0 in /opt/conda/lib/python3.12/s
ite-packages (from tensorflow) (2.20.0)
Requirement already satisfied: keras>=3.10.0 in /opt/conda/lib/python3.12/site-pa
ckages (from tensorflow) (3.13.1)
Requirement already satisfied: numpy>=1.26.0 in /opt/conda/lib/python3.12/site-pa
ckages (from tensorflow) (2.4.1)
Requirement already satisfied: h5py>=3.11.0 in /opt/conda/lib/python3.12/site-pac
kages (from tensorflow) (3.15.1)
Requirement already satisfied: ml_dtypes<1.0.0,>=0.5.1 in /opt/conda/lib/python3.
12/site-packages (from tensorflow) (0.5.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /opt/conda/lib/python3.12/si
te-packages (from astunparse>=1.6.0->tensorflow) (0.45.1)
Requirement already satisfied: rich in /opt/conda/lib/python3.12/site-packages (f
rom keras>=3.10.0->tensorflow) (14.3.1)
Requirement already satisfied: namex in /opt/conda/lib/python3.12/site-packages
(from keras>=3.10.0->tensorflow) (0.1.0)
Requirement already satisfied: optree in /opt/conda/lib/python3.12/site-packages
(from keras>=3.10.0->tensorflow) (0.18.0)
Requirement already satisfied: charset_normalizer<4,>=2 in /opt/conda/lib/python
3.12/site-packages (from requests<3,>=2.21.0->tensorflow) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.12/site-pac
kages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/conda/lib/python3.12/si
te-packages (from requests<3,>=2.21.0->tensorflow) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/lib/python3.12/si
te-packages (from requests<3,>=2.21.0->tensorflow) (2024.12.14)
```

```
Requirement already satisfied: markdown>=2.6.8 in /opt/conda/lib/python3.12/site-
packages (from tensorboard~=2.20.0->tensorflow) (3.10.1)
Requirement already satisfied: pillow in /opt/conda/lib/python3.12/site-packages
(from tensorboard~=2.20.0->tensorflow) (12.1.0)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /opt/cond
a/lib/python3.12/site-packages (from tensorboard~=2.20.0->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in /opt/conda/lib/python3.12/site-
packages (from tensorboard~=2.20.0->tensorflow) (3.1.5)
Requirement already satisfied: markupsafe>=2.1.1 in /opt/conda/lib/python3.12/sit
e-packages (from werkzeug>=1.0.1->tensorboard~=2.20.0->tensorflow) (3.0.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in /opt/conda/lib/python3.1
2/site-packages (from rich->keras>=3.10.0->tensorflow) (4.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /opt/conda/lib/python3.
12/site-packages (from rich->keras>=3.10.0->tensorflow) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in /opt/conda/lib/python3.12/site-packa
ges (from markdown-it-py>=2.2.0->rich->keras>=3.10.0->tensorflow) (0.1.2)
Note: you may need to restart the kernel to use updated packages.
```

and import required libraries:

In [2]:
```
%pip install pandas
%pip install matplotlib
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
import pandas as pd
```

```
Requirement already satisfied: pandas in /opt/conda/lib/python3.12/site-packages
(3.0.0)
Requirement already satisfied: numpy>=1.26.0 in /opt/conda/lib/python3.12/site-pa
ckages (from pandas) (2.4.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/conda/lib/python3.1
2/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.12/site-package
s (from python-dateutil>=2.8.2->pandas) (1.17.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: matplotlib in /opt/conda/lib/python3.12/site-packa
ges (3.10.8)
Requirement already satisfied: contourpy>=1.0.1 in /opt/conda/lib/python3.12/site
-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.12/site-pac
kages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /opt/conda/lib/python3.12/sit
e-packages (from matplotlib) (4.61.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /opt/conda/lib/python3.12/sit
e-packages (from matplotlib) (1.4.9)
Requirement already satisfied: numpy>=1.23 in /opt/conda/lib/python3.12/site-pack
ages (from matplotlib) (2.4.1)
Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.12/site-
packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=8 in /opt/conda/lib/python3.12/site-packag
es (from matplotlib) (12.1.0)
Requirement already satisfied: pyparsing>=3 in /opt/conda/lib/python3.12/site-pac
kages (from matplotlib) (3.3.2)
Requirement already satisfied: python-dateutil>=2.7 in /opt/conda/lib/python3.12/
site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.12/site-package
s (from python-dateutil>=2.7->matplotlib) (1.17.0)
Note: you may need to restart the kernel to use updated packages.
```

```
2026-01-26 11:14:51.854178: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31]
Could not find cuda drivers on your machine, GPU will not be used.
2026-01-26 11:14:51.854659: I tensorflow/core/util/port.cc:153] oneDNN custom ope
rations are on. You may see slightly different numerical results due to floating-
point round-off errors from different computation orders. To turn them off, set t
he environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2026-01-26 11:14:51.931596: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in performa
nce-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other oper
ations, rebuild TensorFlow with the appropriate compiler flags.
2026-01-26 11:14:53.662153: I tensorflow/core/util/port.cc:153] oneDNN custom ope
rations are on. You may see slightly different numerical results due to floating-
point round-off errors from different computation orders. To turn them off, set t
he environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2026-01-26 11:14:53.662731: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31]
Could not find cuda drivers on your machine, GPU will not be used.
```

In [3]:
```
# also set a random state
rs = 123
```

Load and processing rating dataset

In [4]:
```
rating_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud
rating_df = pd.read_csv(rating_url)
rating_df.head()
```

Out[4]:

	user	item	rating
0	1889878	CC0101EN	5
1	1342067	CL0101EN	3
2	1990814	ML0120ENv3	5
3	380098	BD0211EN	5
4	779563	DS0101EN	3

This is the same rating dataset we have been using in previous lab, which contains the three main columns: `user`, `item`, and `rating`.

Next, let's figure out how many unique users and items, their total numbers will determine the sizes of one-hot encoding vectors.

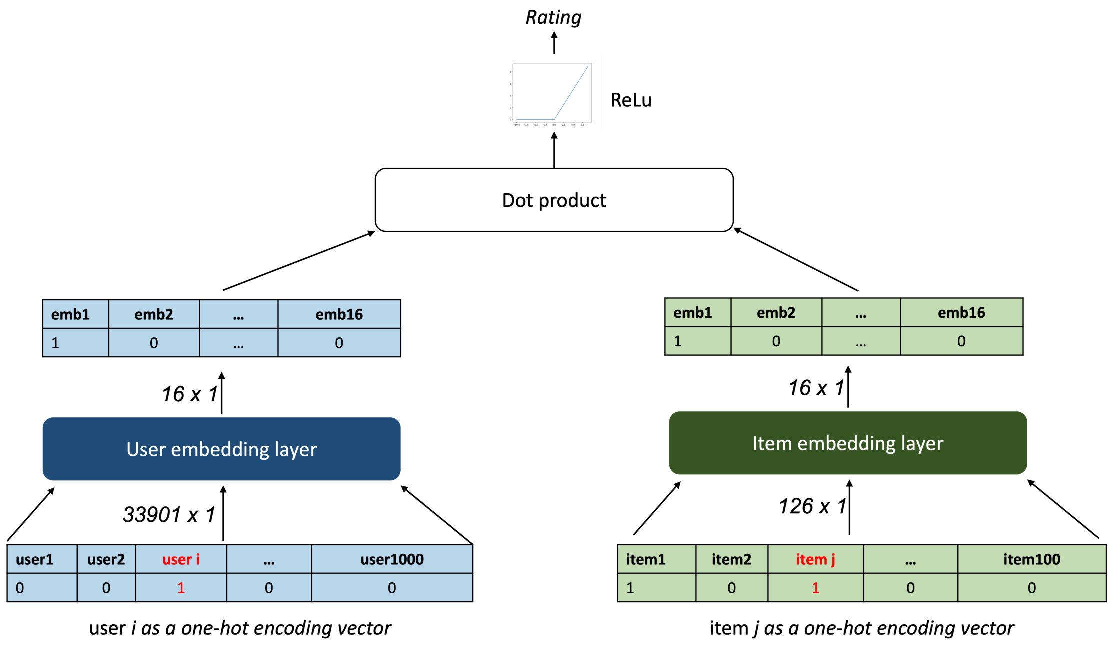
In [5]:
```
num_users = len(rating_df['user'].unique())
num_items = len(rating_df['item'].unique())
print(f"There are total `{num_users}` of users and `{num_items}` items")
```

```
There are total `33901` of users and `126` items
```

It means our each user can be represented as a `33901 x 1` one-hot vector and each item can be represented as `126 x 1` one-hot vector.

The goal is to create a neural network structure that can take the user and item one-hot vectors as inputs and outputs a rating estimation or the probability of interaction.

While training and updating the weights in the neural network, its hidden layers should be able to capture the pattern or features for each user and item. Based on this idea, we can design a simple neural network architecture like the following:



The network inputs are two one-hot encoding vectors, the blue one is for the user and the green one is for the item. Then on top of them, we added two embedding layers. Here embedding means embedding the one-hot encoding vector into a latent feature space. The embedding layer is a fully-connected layer that outputs the embedding feature vectors. For example, the user embedding layer takes `33901 x 1` one-hot vector as input and outputs a `16 x 1` embedding vector.

The embedding layer outputs two embedding vectors, which are similar to Non-negative matrix factorization. Then we could simply dot the product the user and item embedding vector to output a rating estimation.

Implementing the recommender neural network using tensorflow

This network architecture could be defined and implemented as a sub-class inheriting the `tensorflow.keras.Model` super class, let's call it `RecommenderNet()`.

In [6]:
```
class RecommenderNet(keras.Model):
    """
        Neural network model for recommendation.

        This model learns embeddings for users and items, and computes the dot p
        of the user and item embeddings to predict ratings or preferences.

        Attributes:
        - num_users (int): Number of users.
        - num_items (int): Number of items.
        - embedding_size (int): Size of embedding vectors for users and items.
    """
    def __init__(self, num_users, num_items, embedding_size=16, **kwargs):
        """
```

```
            Constructor.

            Args:
            - num_users (int): Number of users.
            - num_items (int): Number of items.
            - embedding_size (int): Size of embedding vectors for users and item
         """
        super(RecommenderNet, self).__init__(**kwargs)
        self.num_users = num_users
        self.num_items = num_items
        self.embedding_size = embedding_size

        # Define a user_embedding vector
        # Input dimension is the num_users
        # Output dimension is the embedding size
        # A name for the layer, which helps in identifying the layer within the

        self.user_embedding_layer = layers.Embedding(
            input_dim=num_users,
            output_dim=embedding_size,
            name='user_embedding_layer',
            embeddings_initializer="he_normal",
            embeddings_regularizer=keras.regularizers.l2(1e-6),
        )
        # Define a user bias layer
        # Bias is applied per user, hence output_dim is set to 1.
        self.user_bias = layers.Embedding(
            input_dim=num_users,
            output_dim=1,
            name="user_bias")

        # Define an item_embedding vector
        # Input dimension is the num_items
        # Output dimension is the embedding size
        self.item_embedding_layer = layers.Embedding(
            input_dim=num_items,
            output_dim=embedding_size,
            name='item_embedding_layer',
            embeddings_initializer="he_normal",
            embeddings_regularizer=keras.regularizers.l2(1e-6),
        )
        # Define an item bias layer
        # Bias is applied per item, hence output_dim is set to 1.
        self.item_bias = layers.Embedding(
            input_dim=num_items,
            output_dim=1,
            name="item_bias")

    def call(self, inputs):
        """
            Method called during model fitting.

            Args:
            - inputs (tf.Tensor): Input tensor containing user and item one-hot

            Returns:
            - tf.Tensor: Output tensor containing predictions.
        """
        # Compute the user embedding vector
        user_vector = self.user_embedding_layer(inputs[:, 0])
        # Compute the user bias
        user_bias = self.user_bias(inputs[:, 0])
        # Compute the item embedding vector
        item_vector = self.item_embedding_layer(inputs[:, 1])
        # Compute the item bias
        item_bias = self.item_bias(inputs[:, 1])
         # Compute dot product of user and item embeddings
        dot_user_item = tf.tensordot(user_vector, item_vector, 2)
        # Add all the components (including bias)
        x = dot_user_item + user_bias + item_bias
        # Apply ReLU activation function
        return tf.nn.relu(x)
```

TASK: Train and evaluate the RecommenderNet()

Now it's time to train and evaluate the defined `RecommenderNet()`. First, we need to process the original rating dataset a little bit by converting the actual user ids and item ids into integer indices for `tensorflow` to creating the one-hot encoding vectors.

In [7]:
```
def process_dataset(raw_data):
    """
        Preprocesses the raw dataset by encoding user and item IDs to indices.

        Args:
        - raw_data (DataFrame): Raw dataset containing user, item, and rating in

        Returns:
        - encoded_data (DataFrame): Processed dataset with user and item IDs enc
        - user_idx2id_dict (dict): Dictionary mapping user indices to original u
        - course_idx2id_dict (dict): Dictionary mapping item indices to original
    """

    encoded_data = raw_data.copy() # Make a copy of the raw dataset to avoid mod

    # Mapping user ids to indices
    user_list = encoded_data["user"].unique().tolist() # Get unique user IDs fro
    user_id2idx_dict = {x: i for i, x in enumerate(user_list)} # Create a dictio
    user_idx2id_dict = {i: x for i, x in enumerate(user_list)} # Create a dictio

    # Mapping course ids to indices
    course_list = encoded_data["item"].unique().tolist() # Get unique item (cour
    course_id2idx_dict = {x: i for i, x in enumerate(course_list)} # Create a di
    course_idx2id_dict = {i: x for i, x in enumerate(course_list)} # Create a di

    # Convert original user ids to idx
    encoded_data["user"] = encoded_data["user"].map(user_id2idx_dict)
    # Convert original course ids to idx
    encoded_data["item"] = encoded_data["item"].map(course_id2idx_dict)
    # Convert rating to int
    encoded_data["rating"] = encoded_data["rating"].values.astype("int")

    return encoded_data, user_idx2id_dict, course_idx2id_dict # Return the proce
```

In [8]:
```
# Process the raw dataset using the process_dataset function
# The function returns three values: encoded_data, user_idx2id_dict, and course_
# encoded_data: Processed dataset with user and item IDs encoded as indices
# user_idx2id_dict: Dictionary mapping user indices to original user IDs
# course_idx2id_dict: Dictionary mapping item indices to original item IDs
encoded_data, user_idx2id_dict, course_idx2id_dict = process_dataset(rating_df)
```

In [9]:
```
encoded_data.head()
```

Out[9]:

	user	item	rating
0	0	0	5
1	1	1	3
2	2	2	5
3	3	3	5
4	4	4	3

Then we can split the encoded dataset into training and testing datasets.

In [10]:
```
def generate_train_test_datasets(dataset, scale=True):
    """
        Splits the dataset into training, validation, and testing sets.

        Args:
        - dataset (DataFrame): Dataset containing user, item, and rating informa
        - scale (bool): Indicates whether to scale the ratings between 0 and 1.

       Returns:
        - x_train (array): Features for training set.
        - x_val (array): Features for validation set.
        - x_test (array): Features for testing set.
        - y_train (array): Labels for training set.
        - y_val (array): Labels for validation set.
        - y_test (array): Labels for testing set.
    """

    min_rating = min(dataset["rating"]) # Get the minimum rating from the datase
    max_rating = max(dataset["rating"]) # Get the maximum rating from the datase

    dataset = dataset.sample(frac=1, random_state=42) # Shuffle the dataset to e
    x = dataset[["user", "item"]].values # Extract features (user and item indic
    if scale:
        # Scale the ratings between 0 and 1 if scale=True
        y = dataset["rating"].apply(lambda x: (x - min_rating) / (max_rating - m
    else:
        # Otherwise, use raw ratings
        y = dataset["rating"].values

    # Assuming training on 80% of the data and testing on 10% of the data
    train_indices = int(0.8 * dataset.shape[0])
    test_indices = int(0.9 * dataset.shape[0])
    # Assigning subsets of features and labels for each set
    x_train, x_val, x_test, y_train, y_val, y_test = (
        x[:train_indices], # Training features
        x[train_indices:test_indices], # Validation features
        x[test_indices:], # Testing features
        y[:train_indices], # Training labels
        y[train_indices:test_indices], # Validation labels
        y[test_indices:], # Testing labels
    )
    return x_train, x_val, x_test, y_train, y_val, y_test # Return the training,
```

In [11]:
```
x_train, x_val, x_test, y_train, y_val, y_test = generate_train_test_datasets(en
```

If we take a look at the training input data, it is simply just a list of user indices and item indices, which is a dense format of one-hot encoding vectors.

In [12]:
```
user_indices = x_train[:, 0]
user_indices
```

Out[12]:
```
array([ 8376,  7659, 10717, ...,  3409, 28761,  4973], shape=(186644,))
```

In [13]:
```
item_indices = x_train[:, 1]
item_indices
```

Out[13]:
```
array([12, 29,  3, ..., 18, 19, 17], shape=(186644,))
```

The training output labels are a list of 0s and 1s indicating if the user has completed a course or not.

In [14]:
```
y_train
```

Out[14]:
```
array([0., 0., 0., ..., 0., 1., 0.], shape=(186644,))
```

Then we can choose a small embedding vector size to be 16 and create a `RecommenderNet()` model to be trained.

In [15]:
```
embedding_size = 16
model = RecommenderNet(num_users, num_items, embedding_size)
```

TODO: Train the RecommenderNet() model

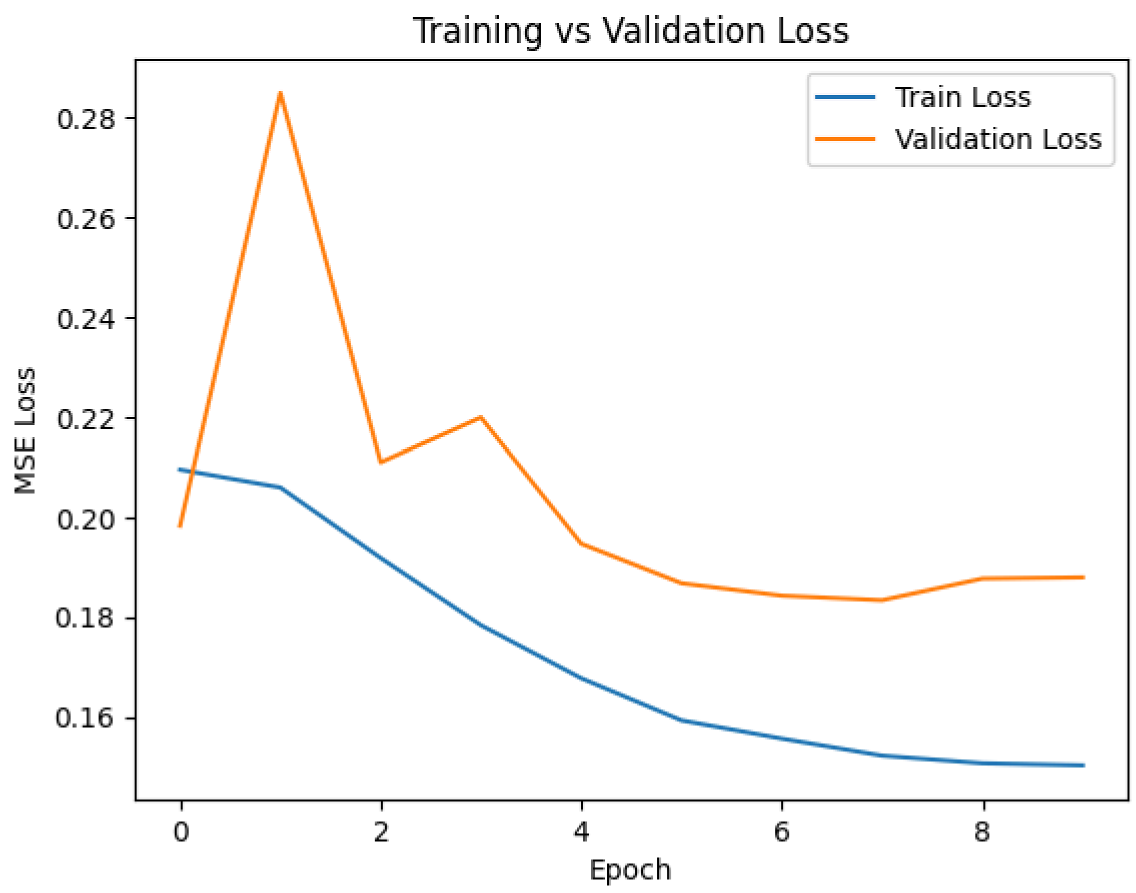
In [16]:
```
# Compile the model
model.compile(
    loss=tf.keras.losses.MeanSquaredError(),
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    metrics=[tf.keras.metrics.RootMeanSquaredError()]
)

# Train the model
history = model.fit(
    x_train,
    y_train,
    validation_data=(x_val, y_val),
    epochs=10,
    batch_size=256,
    verbose=1
)
```

```
2026-01-26 11:15:12.527464: E external/local_xla/xla/stream_executor/cuda/cuda_pl
atform.cc:51] failed call to cuInit: INTERNAL: CUDA error: Failed call to cuInit:
UNKNOWN ERROR (303)
```

```
Epoch 1/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 20s 26ms/step - loss: 0.2095 - root_mean_squared_err
or: 0.4576 - val_loss: 0.1983 - val_root_mean_squared_error: 0.4450
Epoch 2/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 25ms/step - loss: 0.2059 - root_mean_squared_err
or: 0.4534 - val_loss: 0.2850 - val_root_mean_squared_error: 0.5335
Epoch 3/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1918 - root_mean_squared_err
or: 0.4374 - val_loss: 0.2110 - val_root_mean_squared_error: 0.4587
Epoch 4/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 25ms/step - loss: 0.1783 - root_mean_squared_err
or: 0.4216 - val_loss: 0.2199 - val_root_mean_squared_error: 0.4683
Epoch 5/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1677 - root_mean_squared_err
or: 0.4087 - val_loss: 0.1947 - val_root_mean_squared_error: 0.4404
Epoch 6/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1592 - root_mean_squared_err
or: 0.3981 - val_loss: 0.1867 - val_root_mean_squared_error: 0.4313
Epoch 7/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1556 - root_mean_squared_err
or: 0.3935 - val_loss: 0.1842 - val_root_mean_squared_error: 0.4283
Epoch 8/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1522 - root_mean_squared_err
or: 0.3890 - val_loss: 0.1833 - val_root_mean_squared_error: 0.4272
Epoch 9/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1506 - root_mean_squared_err
or: 0.3870 - val_loss: 0.1877 - val_root_mean_squared_error: 0.4322
Epoch 10/10
730/730 ━━━━━━━━━━━━━━━━━━━━ 19s 26ms/step - loss: 0.1502 - root_mean_squared_err
or: 0.3864 - val_loss: 0.1879 - val_root_mean_squared_error: 0.4324
```

In [17]:

```python
plt.figure()
plt.plot(history.history["loss"], label="Train Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.legend()
plt.title("Training vs Validation Loss")
plt.show()
```



▶ Click here for Hints

TODO: Evaluate the trained model

In [18]:
```
### WRITE YOUR CODE HERE

### - call model.evaluate() to evaluate the model
test_loss, test_rmse = model.evaluate(x_test, y_test, verbose=1)

print(f"Test MSE Loss: {test_loss}")
print(f"Test RMSE: {test_rmse}")
```

```
730/730 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.2104 - root_mean_squared_erro
r: 0.4576
Test MSE Loss: 0.2103889435529709
Test RMSE: 0.4576053321361542
```

▶ Click here for Hints

Extract the user and item embedding vectors as latent feature vectors

Now, we have trained the `RecommenderNet()` model and it can predict the ratings with relatively small RMSE.

If we print the trained model then we can see its layers and their parameters/weights.

In [19]:
```
model.summary()
```

```
Model: "recommender_net"
```

Layer (type)	Output Shape	Param #
user_embedding_layer (Embedding)	(None, 16)	542,416
user_bias (Embedding)	(None, 1)	33,901
item_embedding_layer (Embedding)	(None, 16)	2,016
item_bias (Embedding)	(None, 1)	126

```
Total params: 1,735,379 (6.62 MB)
Trainable params: 578,459 (2.21 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 1,156,920 (4.41 MB)
```

In the `RecommenderNet`, the `user_embedding_layer` and `item_embedding_layer` layers contain the trained weights. Essentially, they are the latent user and item features learned by `RecommenderNet` and will be used to predict the interaction. As such, while training the neural network to predict rating, the embedding layers are simultaneously trained to extract the embedding user and item features.

We can easily get the actual weights using `model.get_layer().get_weights()` methods

In [20]:
```
# User features
user_latent_features = model.get_layer('user_embedding_layer').get_weights()[0]
print(f"User features shape: {user_latent_features.shape}")
```

```
User features shape: (33901, 16)
```

In [21]:
```
user_latent_features[0]
```

Out[21]:
```
array([ 0.01355445, -0.03457264,  0.05747632,  0.04885694,  0.01201449,
       -0.00839705,  0.02002703,  0.06986704,  0.01630105,  0.02089548,
       -0.02071207,  0.04683235,  0.00721449, -0.05498322,  0.03645994,
       -0.00731729], dtype=float32)
```

In [22]:
```
item_latent_features = model.get_layer('item_embedding_layer').get_weights()[0]
print(f"Item features shape: {item_latent_features.shape}")
```

```
Item features shape: (126, 16)
```

In [23]:
```
item_latent_features[0]
```

Out[23]:
```
array([ 2.3878640e-02, -1.3063182e-02, -1.6325232e-02,  2.9648842e-02,
        2.8971171e-03, -6.6154279e-02,  3.0644577e-02,  3.0398909e-03,
        3.2618850e-02,  2.1153687e-02,  7.5219087e-03,  2.8854717e-02,
        2.6337046e-05,  2.6389062e-03, -3.4309316e-02, -1.1618178e-02],
      dtype=float32)
```

Now, each user of the total 33901 users has been transformed into a 16 x 1 latent feature vector and each item of the total 126 has been transformed into a 16 x 1 latent feature vector.

TASK (Optional): Customize the RecommenderNet to potentially improve the model performance

The pre-defined `RecommenderNet()` is a actually very basic neural network, you are encouraged to customize it to see if model prediction performance will be improved. Here are some directions:

- Hyperparameter tuning, such as the embedding layer dimensions
- Add more hidden layers
- Try different activation functions such as `ReLu`

Summary

In this lab, you have learned and practiced predicting course ratings using neural networks. With a predefined and trained neural network, we can extract or embed users and items into latent feature spaces and further predict the interaction between a user and an item with the latent feature vectors.

Authors

Yan Luo

Other Contributors

```
toggle##


toggle|Date

toggle|-|-|-|-|

toggle|2021-10-25|1.0|Yan|Created
```