Q4 – TP3

State of the Art

Growing tendencies in recommender systems, according to our literature review, include conversational recommender systems, which establish systems that talk to the users and convert the feedback, using Natural Language Processing techniques [3][4], and recommender systems for the health care domain [9]. Trending social topics include the fairness of recommender systems [2][6], as well as digital nudging [5].

Several different Artificial Intelligence approaches have been surveyed by [8], like deep neural networks, transfer learning, active learning and evolutionary algorithms. We can see that deep learning is indeed a popular topic in recommender systems, being cited by another research, like [10], [11] and [12]. As for the techniques, we can see that transformers are a very popular method in new works, like in [7], [8] and [13].

Therefore, we have decided to use a Transformer model for our project. Our code is based on the code provided by the paper "A Transformer-based recommendation system", by Khalid Salama, which is heavily inspired by the model presented in [1].

References

[1] Chen, Qiwei, et al. "Behavior sequence transformer for e-commerce recommendation in alibaba." Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data. 2019.

[2] Deldjoo, Yashar, Alejandro Bellogin, and Tommaso Di Noia. "Explaining recommender systems fairness and accuracy through the lens of data characteristics." Information Processing & Management 58.5 (2021): 102662.

[3] Gao, Chongming, et al. "Advances and challenges in conversational recommender systems: A survey." AI Open 2 (2021): 100-126.

[4] Jannach, Dietmar, et al. "A survey on conversational recommender systems." ACM Computing Surveys (CSUR) 54.5 (2021): 1-36.

[5] Jesse, Mathias, and Dietmar Jannach. "Digital nudging with recommender systems: Survey and future directions." Computers in Human Behavior Reports 3 (2021): 100052.

[6] Li, Yunqi, Yingqiang Ge, and Yongfeng Zhang. "Tutorial on Fairness of Machine Learning in Recommender Systems." Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021.

[7] Liu, Davide, George Philippe Farajalla, and Alexandre Boulenger. "Transformer-based Banking Products Recommender System." Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021.

[8] McDonnell, Serena, et al. "Cyberbert: A deep dynamic-state session-based recommender system for cyber threat recognition." 2021 IEEE Aerospace Conference (50100). IEEE, 2021.

[9] Tran, Thi Ngoc Trang, et al. "Recommender systems in the healthcare domain: state-of-the-art and research issues." Journal of Intelligent Information Systems 57.1 (2021): 171-201.

[10] Shambour, Qusai. "A deep learning based algorithm for multi-criteria recommender systems." Knowledge-Based Systems 211 (2021): 106545.

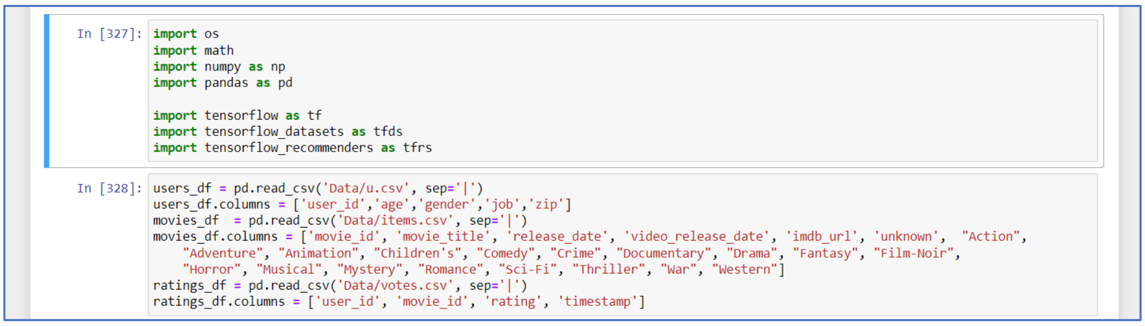
[11] Shin, Kyuyong, et al. "One4all user representation for recommender systems in e-commerce." arXiv preprint arXiv:2106.00573 (2021).

[12] Wang, Shoujin, et al. "Graph learning based recommender systems: A review." arXiv preprint arXiv:2105.06339 (2021).

[13] Xia, Lianghao, et al. "Knowledge-enhanced hierarchical graph transformer network for multi-behavior recommendation." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. No. 5. 2021.

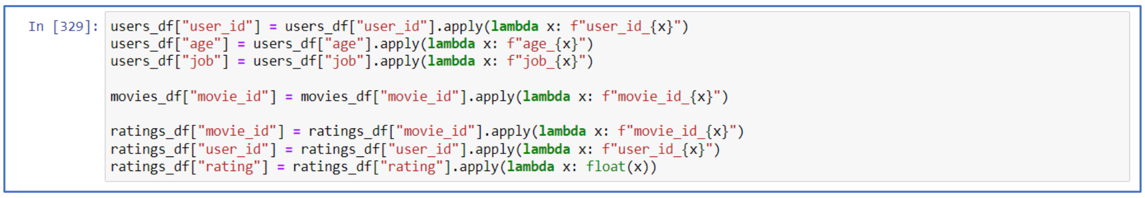
Our Model

Our model is based on the code given by Khalid Salama, inspired by the model presented in [1]. The full code will be attached to this document. The first step is to load the libraries and the CSVs:



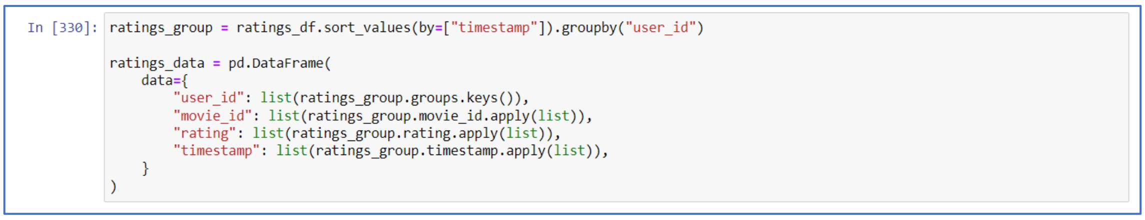
```
In [327]: import os
          import math
          import numpy as np
          import pandas as pd

          import tensorflow as tf
          import tensorflow_datasets as tfds
          import tensorflow_recommenders as tfrs

In [328]: users_df = pd.read_csv('Data/u.csv', sep='|')
          users_df.columns = ['user_id','age','gender','job','zip']
          movies_df  = pd.read_csv('Data/items.csv', sep='|')
          movies_df.columns = ['movie_id', 'movie_title', 'release_date', 'video_release_date', 'imdb_url', 'unknown',  "Action",
              "Adventure", "Animation", "Children's", "Comedy", "Crime", "Documentary", "Drama", "Fantasy", "Film-Noir",
              "Horror", "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller", "War", "Western"]
          ratings_df = pd.read_csv('Data/votes.csv', sep='|')
          ratings_df.columns = ['user_id', 'movie_id', 'rating', 'timestamp']
```

Then, we substitute the values for a whole string, containing the item type and the item id:



```
In [329]: users_df["user_id"] = users_df["user_id"].apply(lambda x: f"user_id_{x}")
          users_df["age"] = users_df["age"].apply(lambda x: f"age_{x}")
          users_df["job"] = users_df["job"].apply(lambda x: f"job_{x}")

          movies_df["movie_id"] = movies_df["movie_id"].apply(lambda x: f"movie_id_{x}")

          ratings_df["movie_id"] = ratings_df["movie_id"].apply(lambda x: f"movie_id_{x}")
          ratings_df["user_id"] = ratings_df["user_id"].apply(lambda x: f"user_id_{x}")
          ratings_df["rating"] = ratings_df["rating"].apply(lambda x: float(x))
```

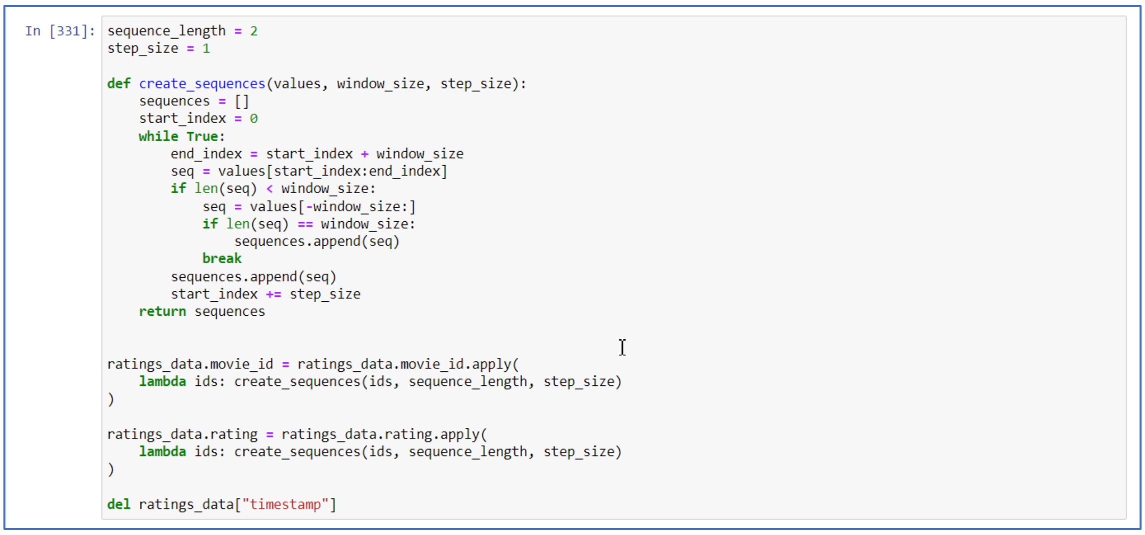
Then, we create a new dataframe, ordering the items by timestamp and grouping the items by their user ids.



```
In [330]: ratings_group = ratings_df.sort_values(by=["timestamp"]).groupby("user_id")

          ratings_data = pd.DataFrame(
              data={
                  "user_id": list(ratings_group.groups.keys()),
                  "movie_id": list(ratings_group.movie_id.apply(list)),
                  "rating": list(ratings_group.rating.apply(list)),
                  "timestamp": list(ratings_group.timestamp.apply(list)),
              }
          )
```

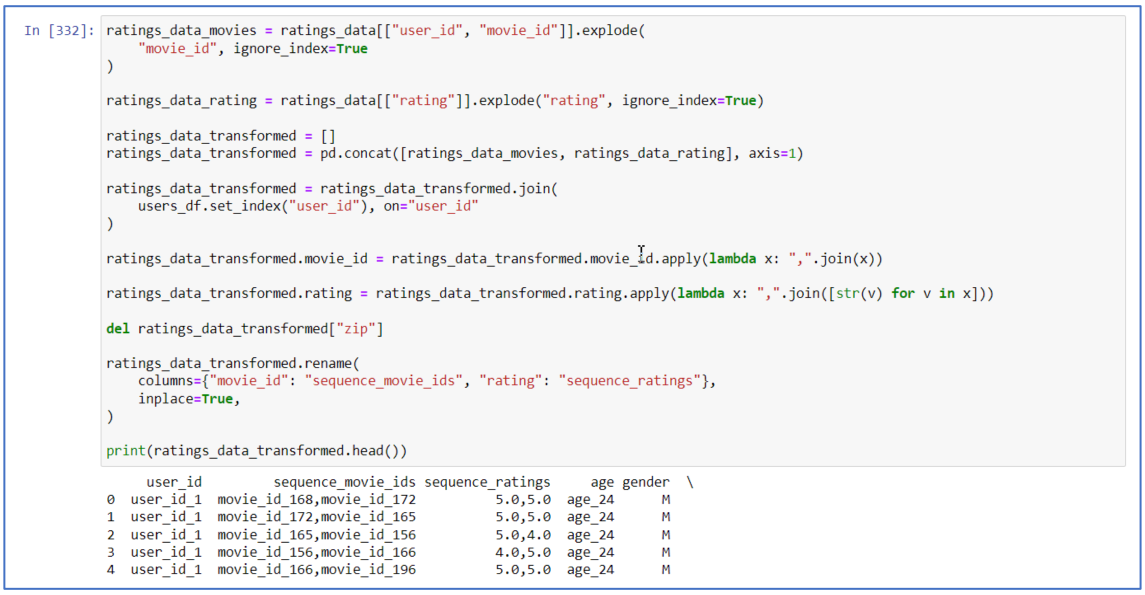
After that, we have the sequence creator. In order to process the items in our Behavior Sequence Transformer, we have to concatenate many items in a single sequence. The step size defines the speed rate of each step. In our tests, we can see that the model behaves better with even sequence numbers, and with low step sizes. For the example below, we are going to work with sequence length 2, and step size 1.

After that, since we no longer need the timestamps, they are deleted from the dataset.



```
In [331]: sequence_length = 2
          step_size = 1

          def create_sequences(values, window_size, step_size):
              sequences = []
              start_index = 0
              while True:
                  end_index = start_index + window_size
                  seq = values[start_index:end_index]
                  if len(seq) < window_size:
                      seq = values[-window_size:]
                      if len(seq) == window_size:
                          sequences.append(seq)
                      break
                  sequences.append(seq)
                  start_index += step_size
              return sequences


          ratings_data.movie_id = ratings_data.movie_id.apply(
              lambda ids: create_sequences(ids, sequence_length, step_size)
          )

          ratings_data.rating = ratings_data.rating.apply(
              lambda ids: create_sequences(ids, sequence_length, step_size)
          )

          del ratings_data["timestamp"]
```

Then we create a new transformed dataset, containing the sequences of movie ids and ratings.



```
In [332]: ratings_data_movies = ratings_data[["user_id", "movie_id"]].explode(
              "movie_id", ignore_index=True
          )

          ratings_data_rating = ratings_data[["rating"]].explode("rating", ignore_index=True)

          ratings_data_transformed = []
          ratings_data_transformed = pd.concat([ratings_data_movies, ratings_data_rating], axis=1)

          ratings_data_transformed = ratings_data_transformed.join(
              users_df.set_index("user_id"), on="user_id"
          )

          ratings_data_transformed.movie_id = ratings_data_transformed.movie_id.apply(lambda x: ",".join(x))

          ratings_data_transformed.rating = ratings_data_transformed.rating.apply(lambda x: ",".join([str(v) for v in x]))

          del ratings_data_transformed["zip"]

          ratings_data_transformed.rename(
              columns={"movie_id": "sequence_movie_ids", "rating": "sequence_ratings"},
              inplace=True,
          )

          print(ratings_data_transformed.head())
```

```
     user_id         sequence_movie_ids sequence_ratings     age gender  \
0  user_id_1  movie_id_168,movie_id_172          5.0,5.0  age_24      M   
1  user_id_1  movie_id_172,movie_id_165          5.0,5.0  age_24      M   
2  user_id_1  movie_id_165,movie_id_156          5.0,4.0  age_24      M   
3  user_id_1  movie_id_156,movie_id_166          4.0,5.0  age_24      M   
4  user_id_1  movie_id_166,movie_id_196          5.0,5.0  age_24      M   
```

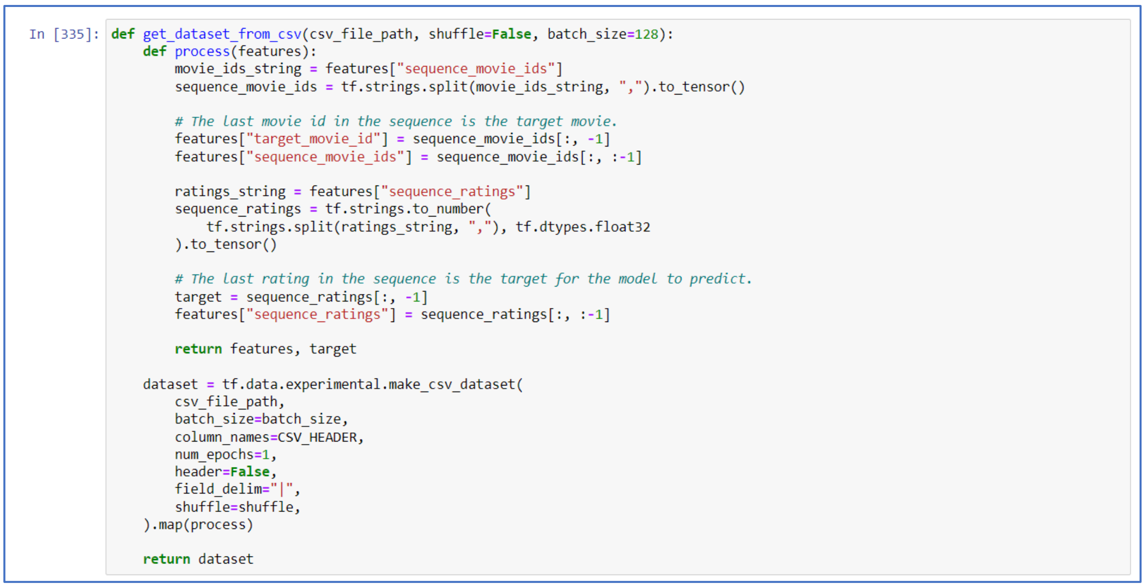
Then, we split the dataset into 80/20 for training and validation, as requested. Some lists and a dictionary are used for the metadata.

In our code, we have commented this first cell, so that the method would use the same sets for training and validation in every model compilation, so that it would be easier to compare the effectiveness of the model. The cell can be uncommented to see its functionality.

In [333]:
```
#random_selection = np.random.rand(len(ratings_data_transformed.index)) <= 0.80
#train_data = ratings_data_transformed[random_selection]
#test_data = ratings_data_transformed[~random_selection]

#train_data.to_csv("train_data_100K-v2.csv", index=False, sep="|", header=False)
#test_data.to_csv("test_data_100K-v2.csv", index=False, sep="|", header=False)
```

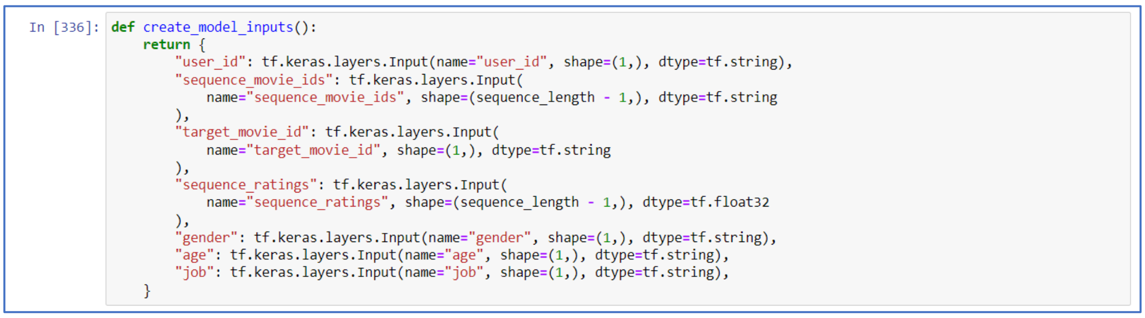
In [334]:
```
CSV_HEADER = list(ratings_data_transformed.columns)

CATEGORICAL_FEATURES_WITH_VOCABULARY = {
    "user_id": list(users_df.user_id.unique()),
    "movie_id": list(movies_df.movie_id.unique()),
    "gender": list(users_df.gender.unique()),
    "age": list(users_df.age.unique()),
    "job": list(users_df.job.unique()),
}

USER_FEATURES = ["gender", "age", "job"]

MOVIE_FEATURES = ["genre"]
```

Then we have a method to read the dataset from the saved CSV.



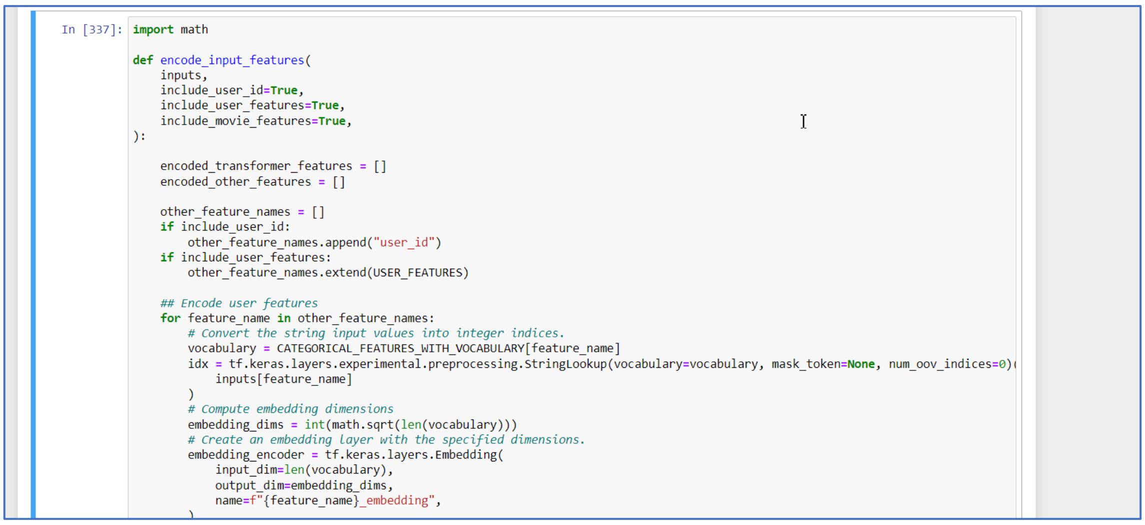
```
In [335]: def get_dataset_from_csv(csv_file_path, shuffle=False, batch_size=128):
              def process(features):
                  movie_ids_string = features["sequence_movie_ids"]
                  sequence_movie_ids = tf.strings.split(movie_ids_string, ",").to_tensor()

                  # The last movie id in the sequence is the target movie.
                  features["target_movie_id"] = sequence_movie_ids[:, -1]
                  features["sequence_movie_ids"] = sequence_movie_ids[:, :-1]

                  ratings_string = features["sequence_ratings"]
                  sequence_ratings = tf.strings.to_number(
                      tf.strings.split(ratings_string, ","), tf.dtypes.float32
                  ).to_tensor()

                  # The last rating in the sequence is the target for the model to predict.
                  target = sequence_ratings[:, -1]
                  features["sequence_ratings"] = sequence_ratings[:, :-1]

                  return features, target

              dataset = tf.data.experimental.make_csv_dataset(
                  csv_file_path,
                  batch_size=batch_size,
                  column_names=CSV_HEADER,
                  num_epochs=1,
                  header=False,
                  field_delim="|",
                  shuffle=shuffle,
              ).map(process)

              return dataset
```

Now we have a method to create the inputs for the neural network:

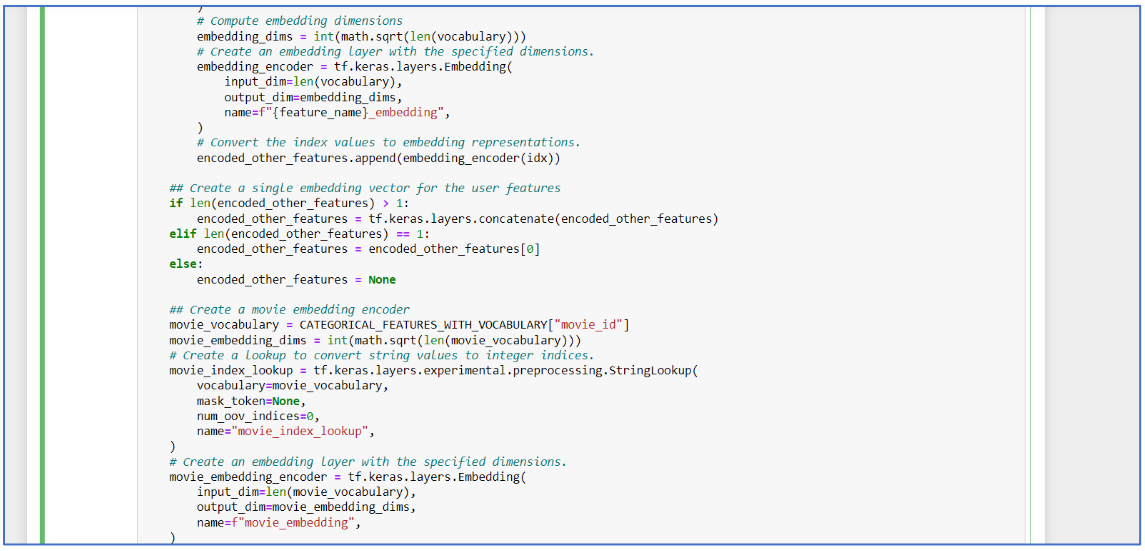


```
In [336]: def create_model_inputs():
              return {
                  "user_id": tf.keras.layers.Input(name="user_id", shape=(1,), dtype=tf.string),
                  "sequence_movie_ids": tf.keras.layers.Input(
                      name="sequence_movie_ids", shape=(sequence_length - 1,), dtype=tf.string
                  ),
                  "target_movie_id": tf.keras.layers.Input(
                      name="target_movie_id", shape=(1,), dtype=tf.string
                  ),
                  "sequence_ratings": tf.keras.layers.Input(
                      name="sequence_ratings", shape=(sequence_length - 1,), dtype=tf.float32
                  ),
                  "gender": tf.keras.layers.Input(name="gender", shape=(1,), dtype=tf.string),
                  "age": tf.keras.layers.Input(name="age", shape=(1,), dtype=tf.string),
                  "job": tf.keras.layers.Input(name="job", shape=(1,), dtype=tf.string),
              }
```

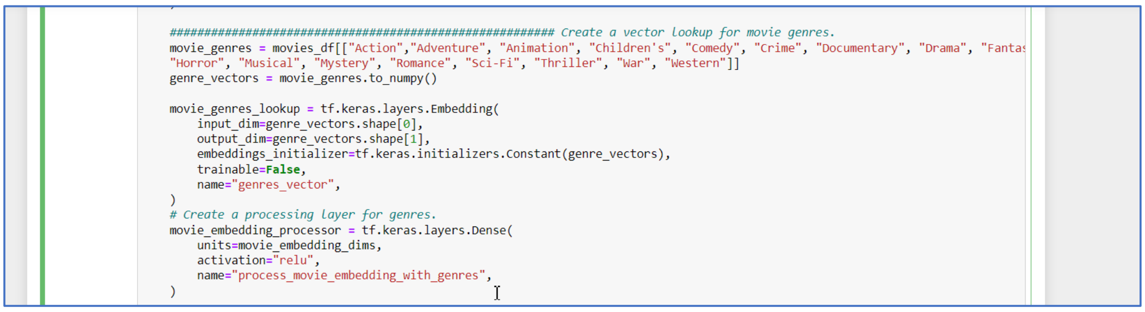
Next, we have a method to create the embeddings for the inputs, using keras.layers. StringLookup and keras.layers.Embedding



```
In [337]: import math

          def encode_input_features(
              inputs,
              include_user_id=True,
              include_user_features=True,
              include_movie_features=True,
          ):

              encoded_transformer_features = []
              encoded_other_features = []

              other_feature_names = []
              if include_user_id:
                  other_feature_names.append("user_id")
              if include_user_features:
                  other_feature_names.extend(USER_FEATURES)

              ## Encode user features
              for feature_name in other_feature_names:
                  # Convert the string input values into integer indices.
                  vocabulary = CATEGORICAL_FEATURES_WITH_VOCABULARY[feature_name]
                  idx = tf.keras.layers.experimental.preprocessing.StringLookup(vocabulary=vocabulary, mask_token=None, num_oov_indices=0)(
                      inputs[feature_name]
                  )
                  # Compute embedding dimensions
                  embedding_dims = int(math.sqrt(len(vocabulary)))
                  # Create an embedding layer with the specified dimensions.
                  embedding_encoder = tf.keras.layers.Embedding(
                      input_dim=len(vocabulary),
                      output_dim=embedding_dims,
                      name=f"{feature_name}_embedding",
                  )
```

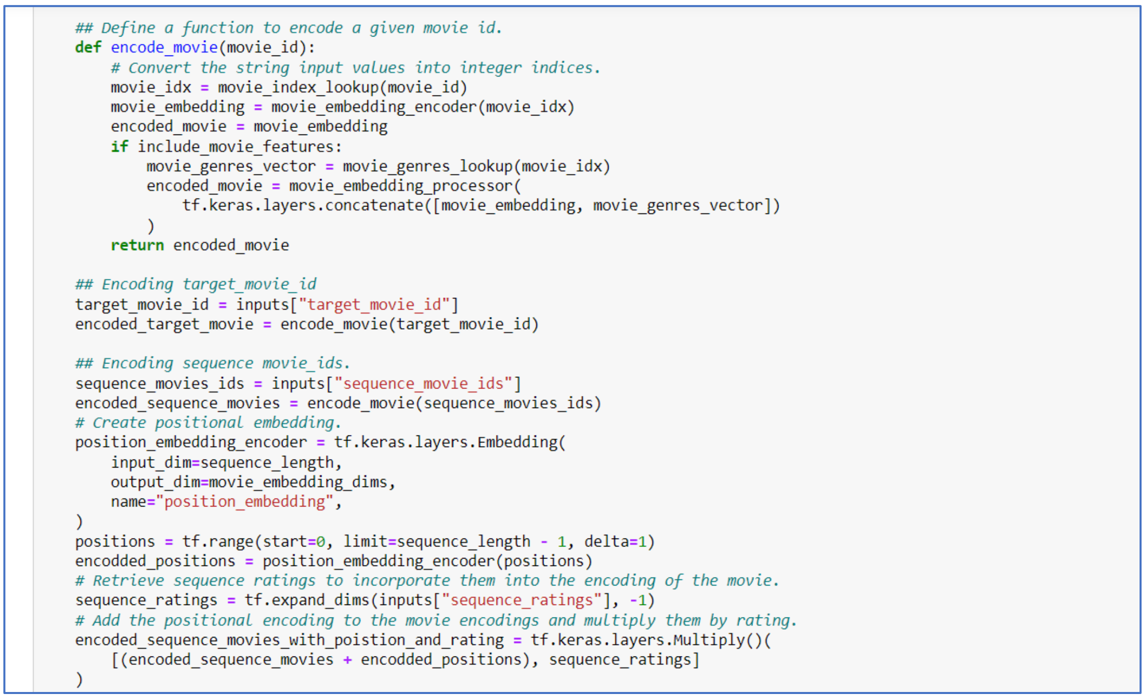


```
        # Compute embedding dimensions
        embedding_dims = int(math.sqrt(len(vocabulary)))
        # Create an embedding layer with the specified dimensions.
        embedding_encoder = tf.keras.layers.Embedding(
            input_dim=len(vocabulary),
            output_dim=embedding_dims,
            name=f"{feature_name}_embedding",
        )
        # Convert the index values to embedding representations.
        encoded_other_features.append(embedding_encoder(idx))

    ## Create a single embedding vector for the user features
    if len(encoded_other_features) > 1:
        encoded_other_features = tf.keras.layers.concatenate(encoded_other_features)
    elif len(encoded_other_features) == 1:
        encoded_other_features = encoded_other_features[0]
    else:
        encoded_other_features = None

    ## Create a movie embedding encoder
    movie_vocabulary = CATEGORICAL_FEATURES_WITH_VOCABULARY["movie_id"]
    movie_embedding_dims = int(math.sqrt(len(movie_vocabulary)))
    # Create a lookup to convert string values to integer indices.
    movie_index_lookup = tf.keras.layers.experimental.preprocessing.StringLookup(
        vocabulary=movie_vocabulary,
        mask_token=None,
        num_oov_indices=0,
        name="movie_index_lookup",
    )
    # Create an embedding layer with the specified dimensions.
    movie_embedding_encoder = tf.keras.layers.Embedding(
        input_dim=len(movie_vocabulary),
        output_dim=movie_embedding_dims,
        name=f"movie_embedding",
    )
```

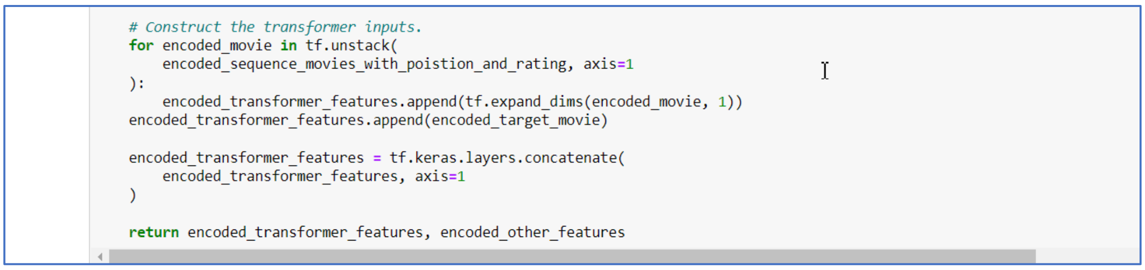


```
    ######################################################### Create a vector lookup for movie genres.
    movie_genres = movies_df[["Action","Adventure", "Animation", "Children's", "Comedy", "Crime", "Documentary", "Drama", "Fantas
    "Horror", "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller", "War", "Western"]]
    genre_vectors = movie_genres.to_numpy()

    movie_genres_lookup = tf.keras.layers.Embedding(
        input_dim=genre_vectors.shape[0],
        output_dim=genre_vectors.shape[1],
        embeddings_initializer=tf.keras.initializers.Constant(genre_vectors),
        trainable=False,
        name="genres_vector",
    )
    # Create a processing layer for genres.
    movie_embedding_processor = tf.keras.layers.Dense(
        units=movie_embedding_dims,
        activation="relu",
        name="process_movie_embedding_with_genres",
    )
```

Then, we have a method to encode the movie id, respecting the sequence that we have created in the previous methods.



```
    ## Define a function to encode a given movie id.
    def encode_movie(movie_id):
        # Convert the string input values into integer indices.
        movie_idx = movie_index_lookup(movie_id)
        movie_embedding = movie_embedding_encoder(movie_idx)
        encoded_movie = movie_embedding
        if include_movie_features:
            movie_genres_vector = movie_genres_lookup(movie_idx)
            encoded_movie = movie_embedding_processor(
                tf.keras.layers.concatenate([movie_embedding, movie_genres_vector])
            )
        return encoded_movie

    ## Encoding target_movie_id
    target_movie_id = inputs["target_movie_id"]
    encoded_target_movie = encode_movie(target_movie_id)

    ## Encoding sequence movie_ids.
    sequence_movies_ids = inputs["sequence_movie_ids"]
    encoded_sequence_movies = encode_movie(sequence_movies_ids)
    # Create positional embedding.
    position_embedding_encoder = tf.keras.layers.Embedding(
        input_dim=sequence_length,
        output_dim=movie_embedding_dims,
        name="position_embedding",
    )
    positions = tf.range(start=0, limit=sequence_length - 1, delta=1)
    encodded_positions = position_embedding_encoder(positions)
    # Retrieve sequence ratings to incorporate them into the encoding of the movie.
    sequence_ratings = tf.expand_dims(inputs["sequence_ratings"], -1)
    # Add the positional encoding to the movie encodings and multiply them by rating.
    encoded_sequence_movies_with_poistion_and_rating = tf.keras.layers.Multiply()(
        [(encoded_sequence_movies + encodded_positions), sequence_ratings]
    )
```



```
# Construct the transformer inputs.
for encoded_movie in tf.unstack(
    encoded_sequence_movies_with_poistion_and_rating, axis=1
):
    encoded_transformer_features.append(tf.expand_dims(encoded_movie, 1))
encoded_transformer_features.append(encoded_target_movie)

encoded_transformer_features = tf.keras.layers.concatenate(
    encoded_transformer_features, axis=1
)

return encoded_transformer_features, encoded_other_features
```

Now, for the model. Here, we describe that we want to use user id, user and movie features, and in the list “hidden_units”, we establish the number of dense layers, as well as the number of neurons of each layer.

In our model, after several tests, we have found that using more layers will less neurons usually work best than less layers with more neurons. The original paper, [1], uses a 3-layer architecture, with 1024, 512 and 256 neurons, respectively. Those are the configurations of the original paper. The “embedding size” is our sequence length, and “transformer block” is our step size. The other parameters have the same name.

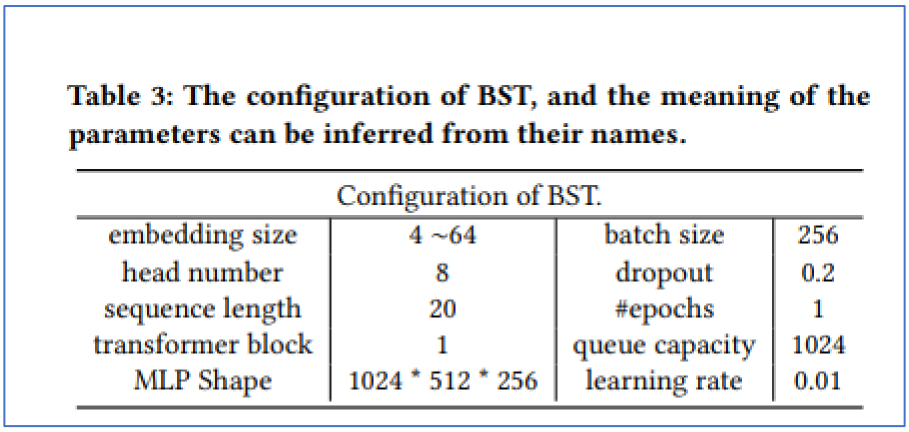
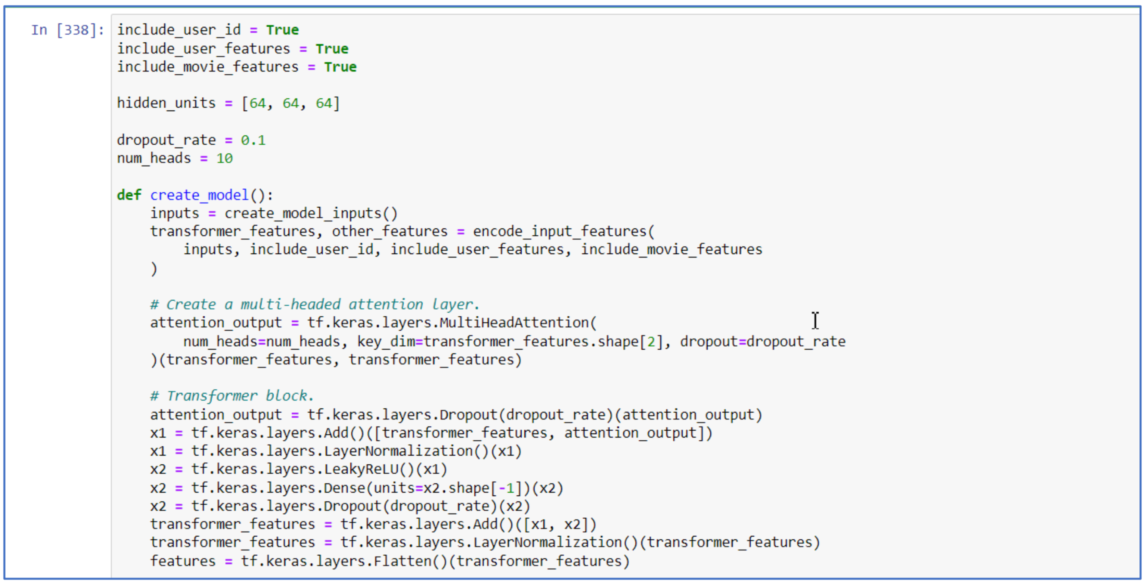


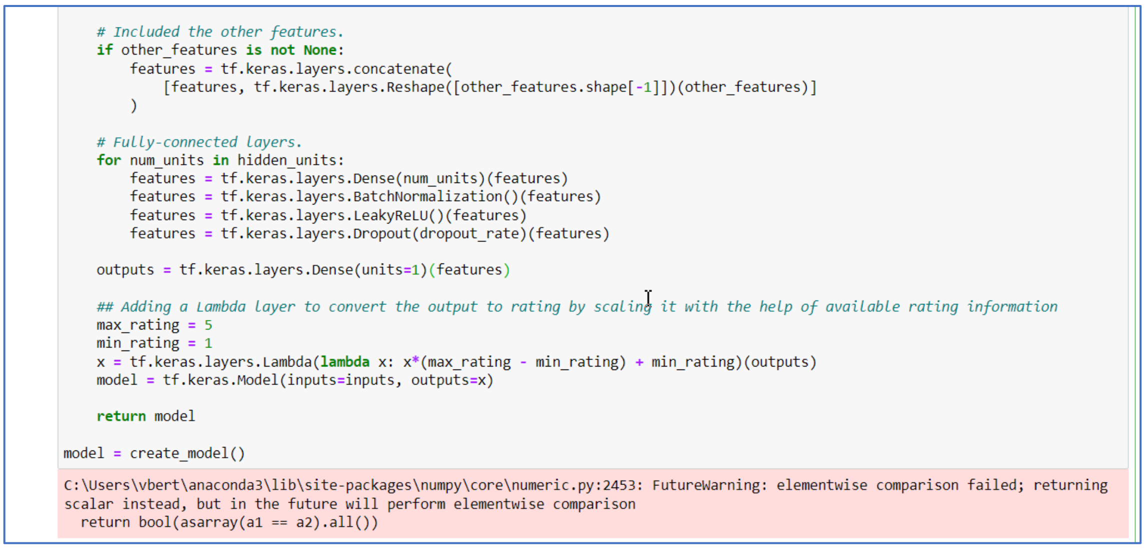
Table 3: The configuration of BST, and the meaning of the parameters can be inferred from their names.

Configuration of BST.			
embedding size	4 ~64	batch size	256
head number	8	dropout	0.2
sequence length	20	#epochs	1
transformer block	1	queue capacity	1024
MLP Shape	1024 * 512 * 256	learning rate	0.01

And for the code:



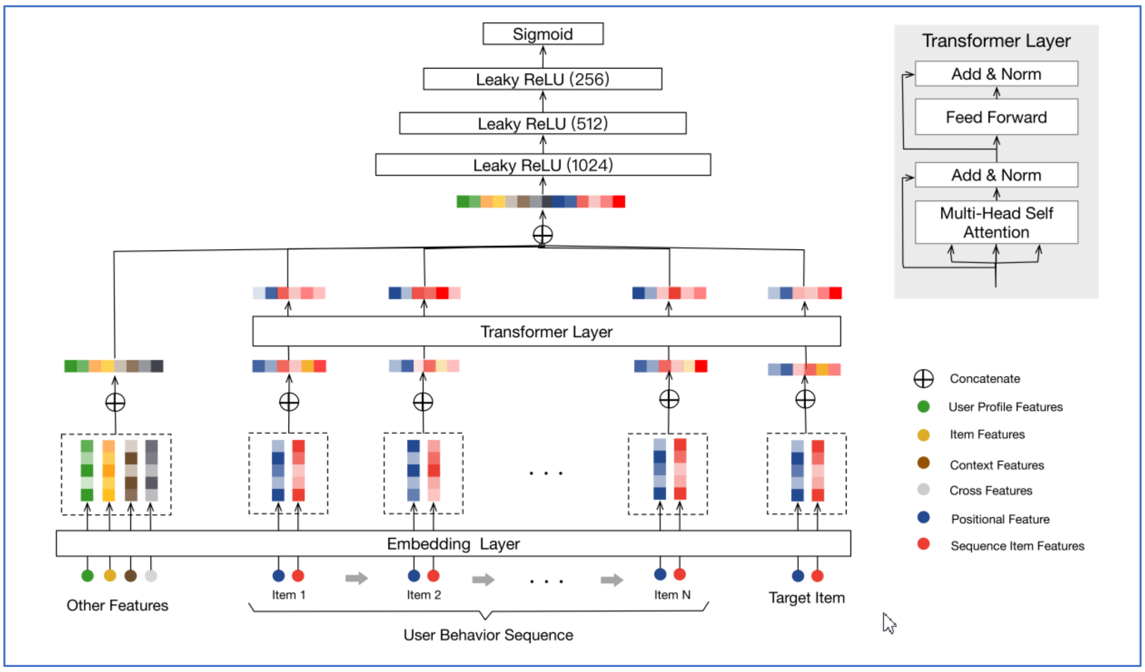
```
In [338]: include_user_id = True
          include_user_features = True
          include_movie_features = True

          hidden_units = [64, 64, 64]

          dropout_rate = 0.1
          num_heads = 10

          def create_model():
              inputs = create_model_inputs()
              transformer_features, other_features = encode_input_features(
                  inputs, include_user_id, include_user_features, include_movie_features
              )

              # Create a multi-headed attention layer.
              attention_output = tf.keras.layers.MultiHeadAttention(
                  num_heads=num_heads, key_dim=transformer_features.shape[2], dropout=dropout_rate
              )(transformer_features, transformer_features)

              # Transformer block.
              attention_output = tf.keras.layers.Dropout(dropout_rate)(attention_output)
              x1 = tf.keras.layers.Add()([transformer_features, attention_output])
              x1 = tf.keras.layers.LayerNormalization()(x1)
              x2 = tf.keras.layers.LeakyReLU()(x1)
              x2 = tf.keras.layers.Dense(units=x2.shape[-1])(x2)
              x2 = tf.keras.layers.Dropout(dropout_rate)(x2)
              transformer_features = tf.keras.layers.Add()([x1, x2])
              transformer_features = tf.keras.layers.LayerNormalization()(transformer_features)
              features = tf.keras.layers.Flatten()(transformer_features)
```



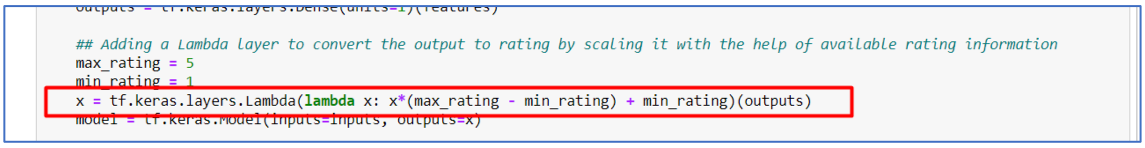
```
    # Included the other features.
    if other_features is not None:
        features = tf.keras.layers.concatenate(
            [features, tf.keras.layers.Reshape([other_features.shape[-1]])(other_features)]
        )

    # Fully-connected layers.
    for num_units in hidden_units:
        features = tf.keras.layers.Dense(num_units)(features)
        features = tf.keras.layers.BatchNormalization()(features)
        features = tf.keras.layers.LeakyReLU()(features)
        features = tf.keras.layers.Dropout(dropout_rate)(features)

    outputs = tf.keras.layers.Dense(units=1)(features)

    ## Adding a Lambda layer to convert the output to rating by scaling it with the help of available rating information
    max_rating = 5
    min_rating = 1
    x = tf.keras.layers.Lambda(lambda x: x*(max_rating - min_rating) + min_rating)(outputs)
    model = tf.keras.Model(inputs=inputs, outputs=x)

    return model

model = create_model()
```

```
C:\Users\vbert\anaconda3\lib\site-packages\numpy\core\numeric.py:2453: FutureWarning: elementwise comparison failed; returning
scalar instead, but in the future will perform elementwise comparison
  return bool(asarray(a1 == a2).all())
```

The method is described below, by the picture extracted from [1].



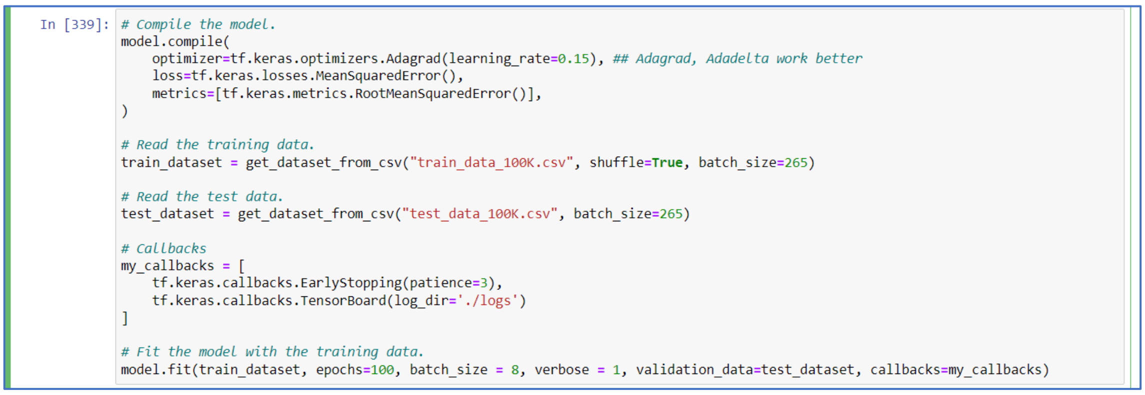
A significant change from the original method is that we apply a normalization based on the min and max ratings.



```
    outputs = tf.keras.layers.Dense(units=1)(features)

    ## Adding a Lambda layer to convert the output to rating by scaling it with the help of available rating information
    max_rating = 5
    min_rating = 1
    x = tf.keras.layers.Lambda(lambda x: x*(max_rating - min_rating) + min_rating)(outputs)
    model = tf.keras.Model(inputs=inputs, outputs=x)
```

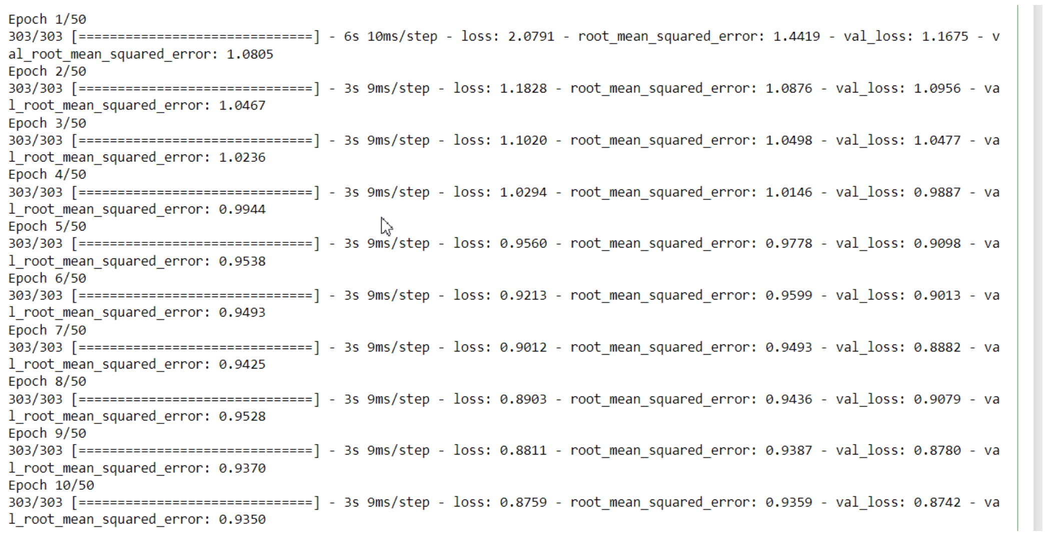
And for the model compilation, we have found the a higher learning rate, usually around 0.1, works better than a small one. The optimizer has also been changed to either Adagrad or Adadelta, that work better than Adam for our model.

The callbacks used were the same as the ones from the other questions of the TP.

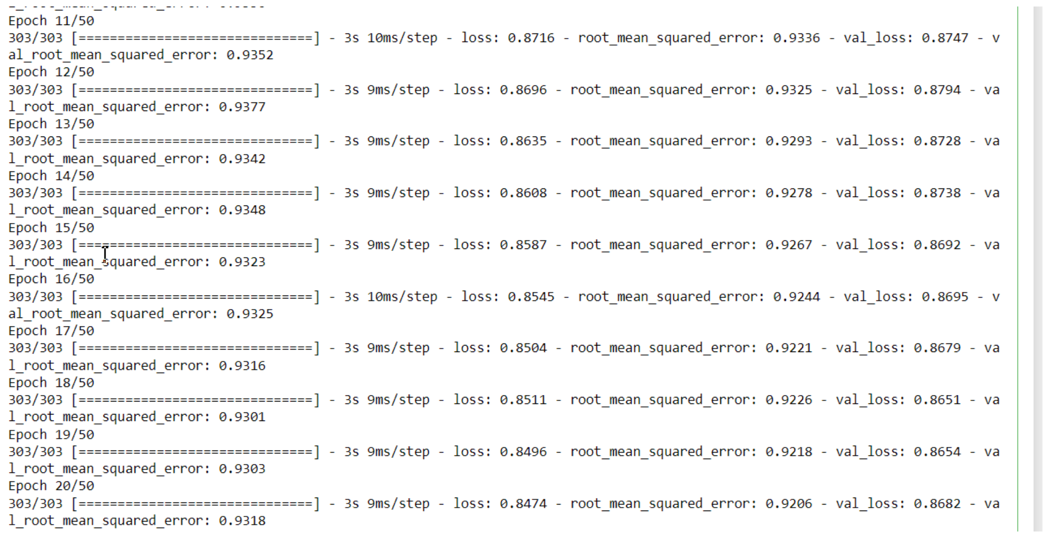


```
In [339]: # Compile the model.
          model.compile(
              optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.15), ## Adagrad, Adadelta work better
              loss=tf.keras.losses.MeanSquaredError(),
              metrics=[tf.keras.metrics.RootMeanSquaredError()],
          )

          # Read the training data.
          train_dataset = get_dataset_from_csv("train_data_100K.csv", shuffle=True, batch_size=265)

          # Read the test data.
          test_dataset = get_dataset_from_csv("test_data_100K.csv", batch_size=265)

          # Callbacks
          my_callbacks = [
              tf.keras.callbacks.EarlyStopping(patience=3),
              tf.keras.callbacks.TensorBoard(log_dir='./logs')
          ]

          # Fit the model with the training data.
          model.fit(train_dataset, epochs=100, batch_size = 8, verbose = 1, validation_data=test_dataset, callbacks=my_callbacks)
```

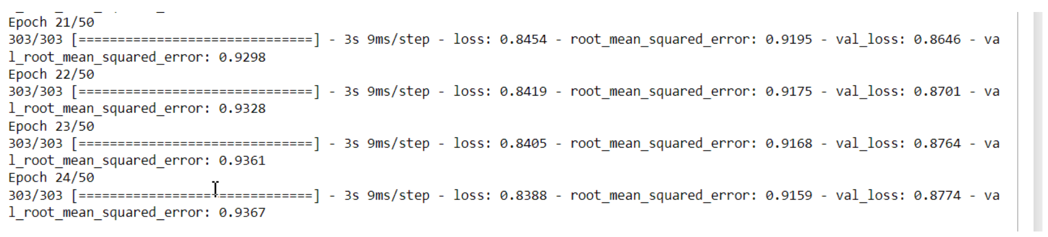
Here are our executions:



```
Epoch 1/50
303/303 [==============================] - 6s 10ms/step - loss: 2.0791 - root_mean_squared_error: 1.4419 - val_loss: 1.1675 - v
al_root_mean_squared_error: 1.0805
Epoch 2/50
303/303 [==============================] - 3s 9ms/step - loss: 1.1828 - root_mean_squared_error: 1.0876 - val_loss: 1.0956 - va
l_root_mean_squared_error: 1.0467
Epoch 3/50
303/303 [==============================] - 3s 9ms/step - loss: 1.1020 - root_mean_squared_error: 1.0498 - val_loss: 1.0477 - va
l_root_mean_squared_error: 1.0236
Epoch 4/50
303/303 [==============================] - 3s 9ms/step - loss: 1.0294 - root_mean_squared_error: 1.0146 - val_loss: 0.9887 - va
l_root_mean_squared_error: 0.9944
Epoch 5/50
303/303 [==============================] - 3s 9ms/step - loss: 0.9560 - root_mean_squared_error: 0.9778 - val_loss: 0.9098 - va
l_root_mean_squared_error: 0.9538
Epoch 6/50
303/303 [==============================] - 3s 9ms/step - loss: 0.9213 - root_mean_squared_error: 0.9599 - val_loss: 0.9013 - va
l_root_mean_squared_error: 0.9493
Epoch 7/50
303/303 [==============================] - 3s 9ms/step - loss: 0.9012 - root_mean_squared_error: 0.9493 - val_loss: 0.8882 - va
l_root_mean_squared_error: 0.9425
Epoch 8/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8903 - root_mean_squared_error: 0.9436 - val_loss: 0.9079 - va
l_root_mean_squared_error: 0.9528
Epoch 9/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8811 - root_mean_squared_error: 0.9387 - val_loss: 0.8780 - va
l_root_mean_squared_error: 0.9370
Epoch 10/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8759 - root_mean_squared_error: 0.9359 - val_loss: 0.8742 - va
l_root_mean_squared_error: 0.9350
```



```
Epoch 11/50
303/303 [==============================] - 3s 10ms/step - loss: 0.8716 - root_mean_squared_error: 0.9336 - val_loss: 0.8747 - v
al_root_mean_squared_error: 0.9352
Epoch 12/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8696 - root_mean_squared_error: 0.9325 - val_loss: 0.8794 - va
l_root_mean_squared_error: 0.9377
Epoch 13/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8635 - root_mean_squared_error: 0.9293 - val_loss: 0.8728 - va
l_root_mean_squared_error: 0.9342
Epoch 14/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8608 - root_mean_squared_error: 0.9278 - val_loss: 0.8738 - va
l_root_mean_squared_error: 0.9348
Epoch 15/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8587 - root_mean_squared_error: 0.9267 - val_loss: 0.8692 - va
l_root_mean_squared_error: 0.9323
Epoch 16/50
303/303 [==============================] - 3s 10ms/step - loss: 0.8545 - root_mean_squared_error: 0.9244 - val_loss: 0.8695 - v
al_root_mean_squared_error: 0.9325
Epoch 17/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8504 - root_mean_squared_error: 0.9221 - val_loss: 0.8679 - va
l_root_mean_squared_error: 0.9316
Epoch 18/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8511 - root_mean_squared_error: 0.9226 - val_loss: 0.8651 - va
l_root_mean_squared_error: 0.9301
Epoch 19/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8496 - root_mean_squared_error: 0.9218 - val_loss: 0.8654 - va
l_root_mean_squared_error: 0.9303
Epoch 20/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8474 - root_mean_squared_error: 0.9206 - val_loss: 0.8682 - va
l_root_mean_squared_error: 0.9318
```



```
Epoch 21/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8454 - root_mean_squared_error: 0.9195 - val_loss: 0.8646 - va
l_root_mean_squared_error: 0.9298
Epoch 22/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8419 - root_mean_squared_error: 0.9175 - val_loss: 0.8701 - va
l_root_mean_squared_error: 0.9328
Epoch 23/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8405 - root_mean_squared_error: 0.9168 - val_loss: 0.8764 - va
l_root_mean_squared_error: 0.9361
Epoch 24/50
303/303 [==============================] - 3s 9ms/step - loss: 0.8388 - root_mean_squared_error: 0.9159 - val_loss: 0.8774 - va
l_root_mean_squared_error: 0.9367
```

Our minimum RMSE for this execution is 0.**9298**. The model, as well as the .IPNYB file, are attached to the TP file.