Estimating Patient Satisfaction Using Drug Reviews

IBM Deep Learning And Reinforcement Learning

Course Assignment

Vladas Alesius

2021-08-02

Main Objective

The purpose of this analysis is to use Recurrent Neural Network (RNN) models for a Natural Language Processing task. The task is to predict patient ratings of drugs given their reviews (in text). Three types of RNNs will be tried:

1. Simple RNN
2. Gated Recurrent Unit (GRU)
3. Long Short-Term Memory (LSTM)

In all three cases, the same training dataset and model parameters will be used.

After fitting the models, their efficiency will be compared with two main metrics - Mean Squared Error and Coefficient of Determination (R^2 score). These metrics are commonly used for regression tasks, and will be used in our models as well. However, our ratings are discrete, and regression analysis is typically applied to continuous dependent variables. This might impact the model quality.

As per the description in Machine Learning Repository webpage, "The dataset provides patient reviews on specific drugs along with related conditions and a 10 star patient rating reflecting overall patient satisfaction. The data was obtained by crawling online pharmaceutical review sites. The intention was to study

(1) sentiment analysis of drug experience over multiple facets, i.e. sentiments learned on specific aspects such as effectiveness and side effects,

(2) the transferability of models among domains, i.e. conditions, and

(3) the transferability of models among different data sources (see 'Drug Review Dataset (Druglib.com)')."

This dataset was used in the following publication:

Felix Gräßer, Surya Kallumadi, Hagen Malberg, and Sebastian Zaunseder. 2018. Aspect-Based Sentiment Analysis of Drug Reviews Applying Cross-Domain and Cross-Data Learning. In Proceedings of the 2018 International Conference on Digital Health (DH '18). ACM, New York, NY, USA, 121-125. DOI: Web Link

Data Description

Before starting our work, all necessary modules and functions will be imported, and random seeds will be set to ensure reproducibility:

```
from numpy.random import seed
seed(33)
import tensorflow as tf
tf.random.set_seed(33)

import pandas as pd
import numpy as np
import seaborn as sns

import keras
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding, Activation, ReLU
from keras.layers import SimpleRNN, GRU, LSTM
from keras import initializers

from keras import backend as K
from keras.utils.generic_utils import get_custom_objects
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.models import load_model

from sklearn.metrics import mean_squared_error, r2_score

from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
import warnings
warnings.filterwarnings('ignore')
```

The data is split into a train (75%) and a test (25%) partition (see the publication) and stored in two .tsv (tab-separated-values) files, respectively.

```
train_data = pd.read_table('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_lea
test_data = pd.read_table('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_lea
```

Attribute Information:

1. drugName (categorical): name of drug
2. condition (categorical): name of condition
3. review (text): patient review
4. rating (numerical): 10 star patient rating
5. date (date): date of review entry
6. usefulCount (numerical): number of users who found review useful

The first five entries of the training dataset are presented below:

```
train_data.head()
```

	Unnamed: 0	drugName	condition	review	rating	date	usefulCount
0	206461	Valsartan	Left Ventricular Dysfunction	"It has no side effect, I take it in combinati...	9.0	May 20, 2012	27
1	95260	Guanfacine	ADHD	"My son is halfway through his fourth week of ...	8.0	April 27, 2010	192
2	92703	Lybrel	Birth Control	"I used to take another oral contraceptive, wh...	5.0	December 14, 2009	17
3	138000	Ortho Evra	Birth Control	"This is my first time using any form of birth...	8.0	November 3, 2015	10
4	35696	Buprenorphine / naloxone	Opiate Dependence	"Suboxone has completely turned my life around...	9.0	November 27, 2016	37

Data Exploration

Reviews from both training and testing datasets will be converted to lists in order to enable further preprocessing:

```
training_data = train_data["review"].values.tolist()
testing_data = test_data["review"].values.tolist()
```

Training dataset has over 161 000 reviews:

```
train_data.shape
```

```
(161297, 7)
```

It contains more than 1.3 million total and almost 200 000 unique words:

```
len([i for item in training_data for i in item.split()])
```

```
13461824
```

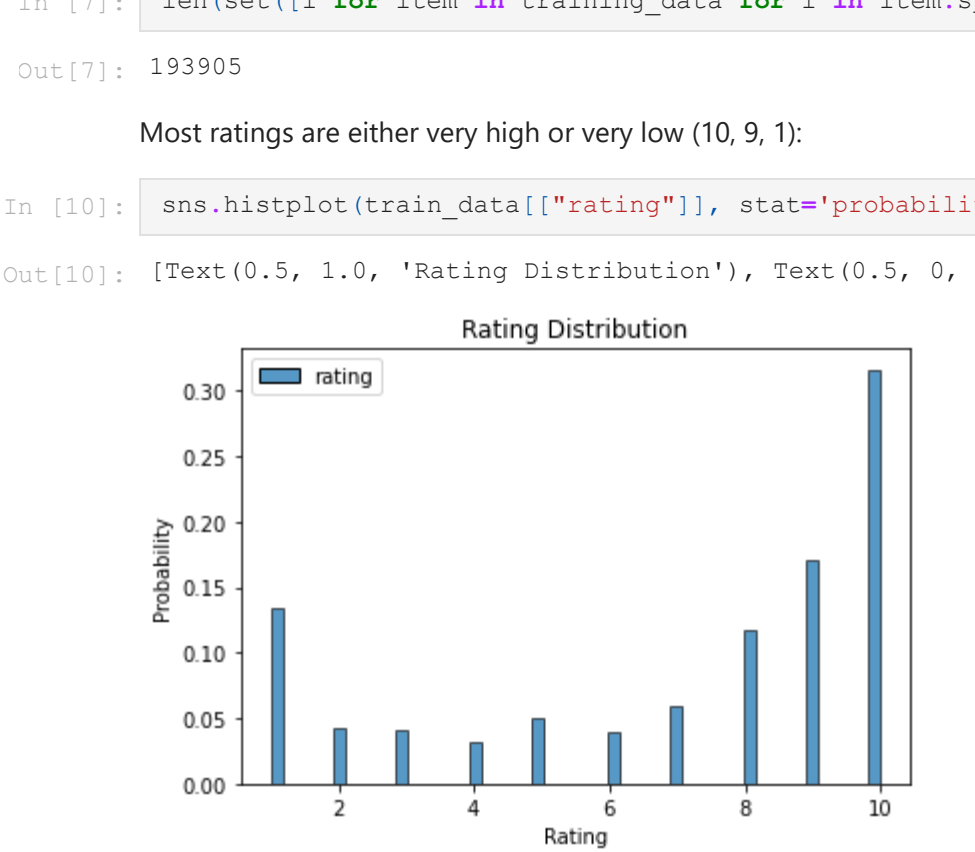
```
len(set([i for item in training_data for i in item.split()]))
```

```
193905
```

Most ratings are either very high or very low (10, 9, 1):

```
sns.histplot(train_data["rating"], stat='probability').set(title='Rating Distribution', xlabel='Rating')
```

```
[Text(0.5, 1.0, 'Rating Distribution'), Text(0.5, 0, 'Rating')]
```



The average rating is about 7, but the median one is 8:

```
train_data["rating"].describe()
```

```
count    161297.000000
mean          6.994377
std           3.272329
min           1.000000
25%           5.000000
50%           8.000000
75%          10.000000
max          10.000000
Name: rating, dtype: float64
```

A number of words for each review is calculated and added to the training dataset below:

```
train_data["word_counts"] = train_data.review.apply(lambda x: len(str(x).split(" ")))
```

```
train_data.head()
```

	Unnamed: 0	drugName	condition	review	rating	date	usefulCount	word_counts
0	206461	Valsartan	Left Ventricular Dysfunction	"It has no side effect, I take it in combinati...	9.0	May 20, 2012	27	17
1	95260	Guanfacine	ADHD	"My son is halfway through his fourth week of ...	8.0	April 27, 2010	192	141
2	92703	Lybrel	Birth Control	"I used to take another oral contraceptive, wh...	5.0	December 14, 2009	17	133
3	138000	Ortho Evra	Birth Control	"This is my first time using any form of birth...	8.0	November 3, 2015	10	89
4	35696	Buprenorphine / naloxone	Opiate Dependence	"Suboxone has completely turned my life around...	9.0	November 27, 2016	37	134

Reviews have between 1 and 1857 words, and about 86 words on average:

```
train_data["word_counts"].describe()
```

```
count    161297.000000
mean         85.646788
std          45.507389
min           1.000000
25%          49.000000
50%          84.000000
75%         127.000000
max        1857.000000
Name: word_counts, dtype: float64
```

Average word counts do not differ significantly among reviews with different ratings:

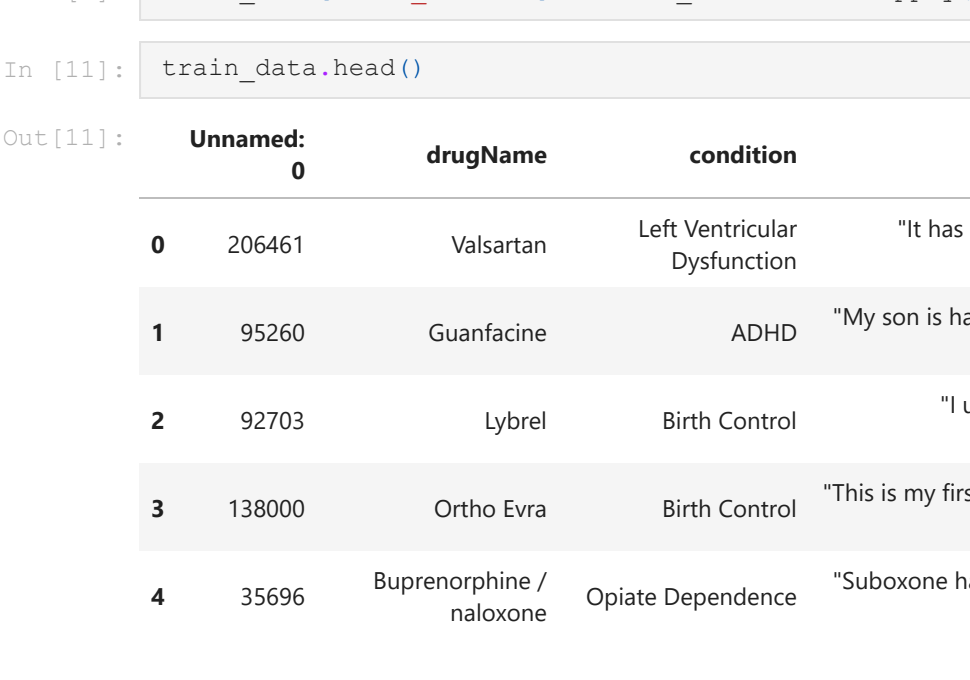
```
pd.pivot_table(train_data, values = 'word_counts', index = 'rating', aggfunc = [np.min, np.max, np.mean])
```

	amin	amax	mean
	word_counts	word_counts	word_counts
rating			
1.0	1	706	79.696471
2.0	2	1857	84.116722
3.0	1	881	85.681713
4.0	1	555	86.353751
5.0	1	373	89.092600
6.0	1	368	87.334384
7.0	1	587	90.594014
8.0	1	770	90.212017
9.0	1	1099	89.141441
10.0	1	1222	83.056600

The vast majority of word counts are below 200, only a tiny number of reviews have more words:

```
sns.displot(train_data["word_counts"]).set(title='Numbers Of Words In Reviews', xlabel='Word Counts')
```

```
<seaborn.axisgrid.FacetGrid at 0x1d889928670>
```



Only about 3% of all reviews have more than 150 words:

```
np.sum(train_data["word_counts"] > 150)
```

```
4827
```

After exploring the training dataset, we will vectorize the review texts to process them later. For this purpose, a Text vectorization layer will be used.

According to Tensorflow tutorial, "It transforms a batch of strings (one sample = one string) into either a list of token indices (one sample = 1D tensor of integer token indices).

If desired, the user can call this layer's adapt() method on a dataset. When this layer is adapted, it will analyze the dataset, determine the frequency of individual string values, and create a 'vocabulary' from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary."

The maximum size of vocabulary will be 50000 (about a quarter of all words), and each review will be padded or truncated to 150 elements:

```
vectorizer = TextVectorization(max_tokens=50000, output_mode="int", output_sequence_length=150)
vectorizer.adapt(training_data)
x_train = vectorizer(training_data)
x_test = vectorizer(testing_data)
print(x_train)
```

```
tf.Tensor(
[[   8   37   27 ...    0    0    0]
 [   6  639   19 ...    0    0    0]
 [   2  144    5 ...    0    0    0]
 ...
 [2198  569   26 ...    0    0    0]
 [  39   24   13 ...    0    0    0]
 [  39   18  443 ...    0    0    0]], shape=(161297, 150), dtype=int64)
```

These tensors will be additionally transformed into NumPy arrays using pad_sequences method:

```
x_train = sequence.pad_sequences(x_train)
x_test = sequence.pad_sequences(x_test)
```

An example of these arrays is displayed below:

```
x_train[158]
```

```
array([  24,   35,   14,   77,    9,  156,   38,    2,   11,    7,  115,
       4207,    3,    8,   10,  142,  127,    6,   78,   64,    2,  299,
       5549,    6,  955, 1323,  224,    5,    7,  955,   21,   84,   22,
         11,    4,  491, 3943,   21,    8,  217,   18,    8,  143,  119,
          9,  209,   12,  222,   38,   16,  224,    5,    6,  594, 2984,
          2,  201,  214,  219,    8,    5,    4,  650,  566,  353,    5,
       2137,   33,  139,    5,  206,   35,    8,    3,  210,   61,  157,
         19,    7,  383,   15, 6025,    4,  650,   80,    2,   11,    5,
        522,  123,   12,    7,  347,  955,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0])
```

Model Training

Recurrent neural networks (RNN) are a class of neural networks that is powerful for modeling sequence data such as time series or natural language. For example, RNNs can be used for sentiment classification with binary outcome. We will try them to predict numeric ratings (integer values from 1 to 10) based on text data.

There are three built-in RNN layers in Keras:

1. keras.layers.SimpleRNN, a fully-connected RNN where the output from the previous timestep is to be fed to the next timestep.
2. keras.layers.GRU (Gated Recurrent Unit)
3. keras.layers.LSTM (Long Short-Term Memory)

In the analysis below, these three model types will be examined on how well they predict ratings from the drugs dataset. In order to compare model efficiency, their structure and parameters will be the same.

Each network will have 3 layers, they will be of the same shape for each model:

1. An embedding layer.
2. A hidden layer for the respective model.
3. A dense layer for the final output.

Each word in the vocabulary (50000 in total) will be represented as a 50-dimensional numeric vector with the Embedding layer. This output will serve as an input to the hidden layer for the respective model, which will be 5-dimensional. Finally, the Dense layer will output the number (which will be the predicted rating).

For the middle layer, no activation function will be used ('linear'), while a modified version of Rectified Linear Unit (ReLU) function will be used for the final output. According to this source, the He initializer works better with ReLU activation, so it will be used to initialize weights in our models.

All models will be trained in batches of 32 samples. Adam optimizer with a learning rate of 0.0001 will be used (the default rate is 0.001). According to this source, "Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process." So we deliberately choose a lower value, to increase chances of finding an optimal set of weights.

Since ReLU activation function will be used for the output node, we have to modify it to suit our needs. The standard ReLU function has non-negative values, but we need rating values between 1 and 10, so we define a custom ReLU function below.

After each iteration, the mean squared error will be checked on both datasets - used for training and validation, the latter making 20% of all initial training dataset. For each model, a different number of epochs might be necessary to train the model, so the training will be terminated (using EarlyStopping) when mean squared error does not decrease on validation data subset for 3 epochs in a row. On the other hand, the number of epochs should not exceed 30.

Also, for each type of layer, the best model will be saved with ModelCheckpoint.

```
def relu_advanced(x):
    return K.relu(x, max_value=10)

get_custom_objects().update({'relu_advanced': Activation(relu_advanced)})

es = EarlyStopping(monitor='val_mean_squared_error', patience=3)
mc_rnn = ModelCheckpoint('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_lear
mc_gru = ModelCheckpoint('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_lear
mc_lstm = ModelCheckpoint('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_lear
```

Simple RNN

We will try a Simple RNN layer first.

```
rnn_hidden_dim = 5
word_embedding_dim = 50
batch_size = 32
```

```
model_rnn = Sequential()
model_rnn.add(Embedding(vectorizer.vocabulary_size(), word_embedding_dim))  #This layer takes each integer in
model_rnn.add(SimpleRNN(rnn_hidden_dim,
                        activation='linear',
                        kernel_initializer='he_uniform',
                        input_shape=x_train.shape[1:]))

model_rnn.add(Dense(1, kernel_initializer='he_uniform', activation=relu_advanced))
```

As for other models, the network will consist of three layers - embedding (with output dimension of 50), Simple RNN (with 5 output nodes) and Dense layer (1 final output node). Since our vocabulary consists of 50000 words, we will have about 2.5 million parameters to estimate.

```
model_rnn.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 50)          2500000
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 5)                 280
_________________________________________________________________
dense (Dense)                (None, 1)                 6
=================================================================
Total params: 2,500,286
Trainable params: 2,500,286
Non-trainable params: 0
_________________________________________________________________
```

```
model_rnn.compile(loss='mean_squared_error',
                  optimizer=keras.optimizers.Adam(0.0001),
                  metrics=['mean_squared_error'])
```

The lowest mean squared error on validation subset was reached on epoch 3, so the training stopped after 6 epochs:

```
hist_rnn = model_rnn.fit(x_train, train_data[["rating"]],
          batch_size=batch_size,
          epochs=30,
          validation_split=0.2,
          callbacks=[es, mc_rnn],
          verbose=1)
```

```
Epoch 1/30
4033/4033 [==============================] - 167s 38ms/step - loss: 8.7428 - mean_squared_error: 8.7428 - val_l
oss: 5.4998 - val_mean_squared_error: 5.4998

Epoch 00001: val_mean_squared_error improved from inf to 5.49979, saving model to C:\Users\Lenovo\Desktop\ibm_m
achine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_simple_rnn_model.h5
Epoch 2/30
4033/4033 [==============================] - 152s 38ms/step - loss: 5.0638 - mean_squared_error: 5.0638 - val_l
oss: 5.3877 - val_mean_squared_error: 5.3877

Epoch 00002: val_mean_squared_error improved from 5.49979 to 5.38773, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_simple_rnn_model.h5
Epoch 3/30
4033/4033 [==============================] - 151s 37ms/step - loss: 4.6207 - mean_squared_error: 4.6207 - val_l
oss: 5.2324 - val_mean_squared_error: 5.2324

Epoch 00003: val_mean_squared_error improved from 5.38773 to 5.23243, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_simple_rnn_model.h5
Epoch 4/30
4033/4033 [==============================] - 145s 36ms/step - loss: 4.1856 - mean_squared_error: 4.1856 - val_l
oss: 5.4084 - val_mean_squared_error: 5.4084

Epoch 00004: val_mean_squared_error did not improve from 5.23243
Epoch 5/30
4033/4033 [==============================] - 144s 36ms/step - loss: 3.8842 - mean_squared_error: 3.8842 - val_l
oss: 5.3484 - val_mean_squared_error: 5.3484

Epoch 00005: val_mean_squared_error did not improve from 5.23243
Epoch 6/30
4033/4033 [==============================] - 144s 36ms/step - loss: 3.5398 - mean_squared_error: 3.5398 - val_l
oss: 5.3248 - val_mean_squared_error: 5.3248

Epoch 00006: val_mean_squared_error did not improve from 5.23243
```

Rating predictions are made for both training and testing datasets. Also, they are rounded to have integer values, just like the true ratings:

```
rnn_saved_model = load_model('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_
```

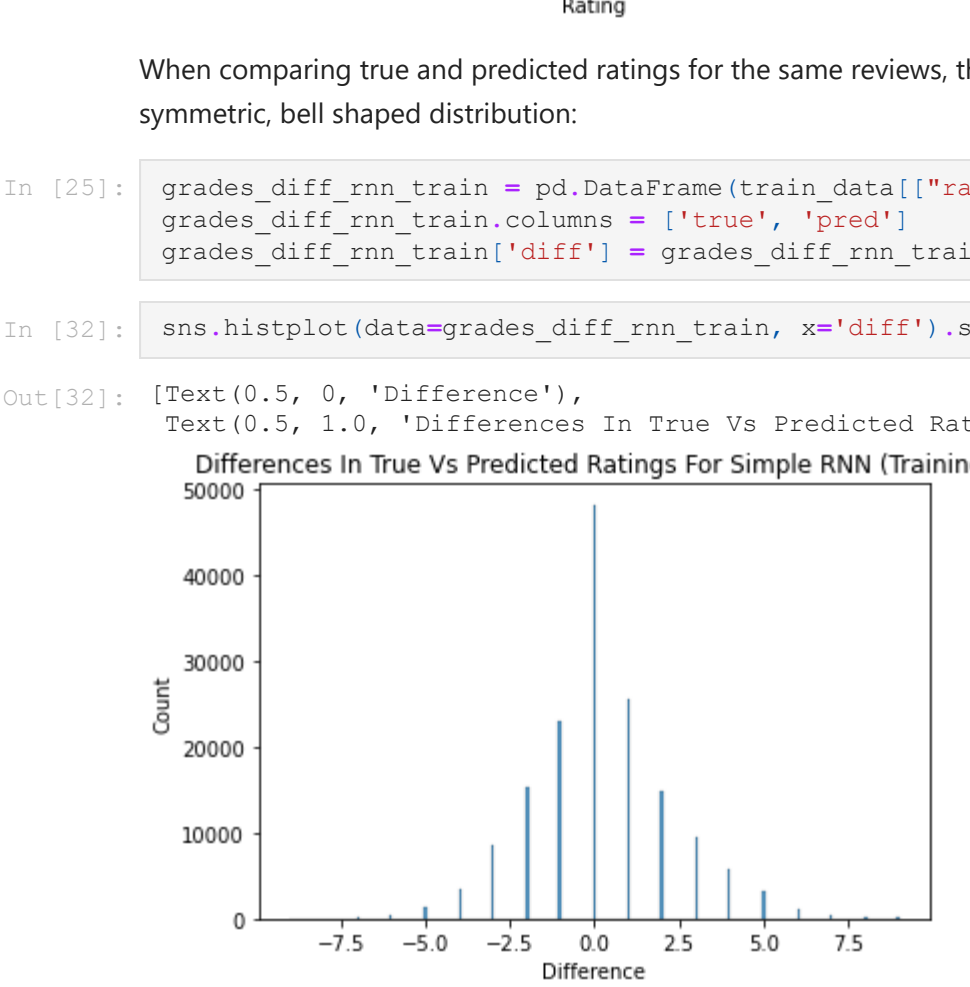
```
rnn_predict_train = rnn_saved_model.predict(x_train)
rnn_predict_train = np.round(rnn_predict_train)
```

```
rnn_predict_test = rnn_saved_model.predict(x_test)
rnn_predict_test = np.round(rnn_predict_test)
```

The histograms below show that the Simple RNN model tends to underestimate the number of high and low ratings (1, 2, 9 and 10), and overestimates the number of middle ratings (4-7):

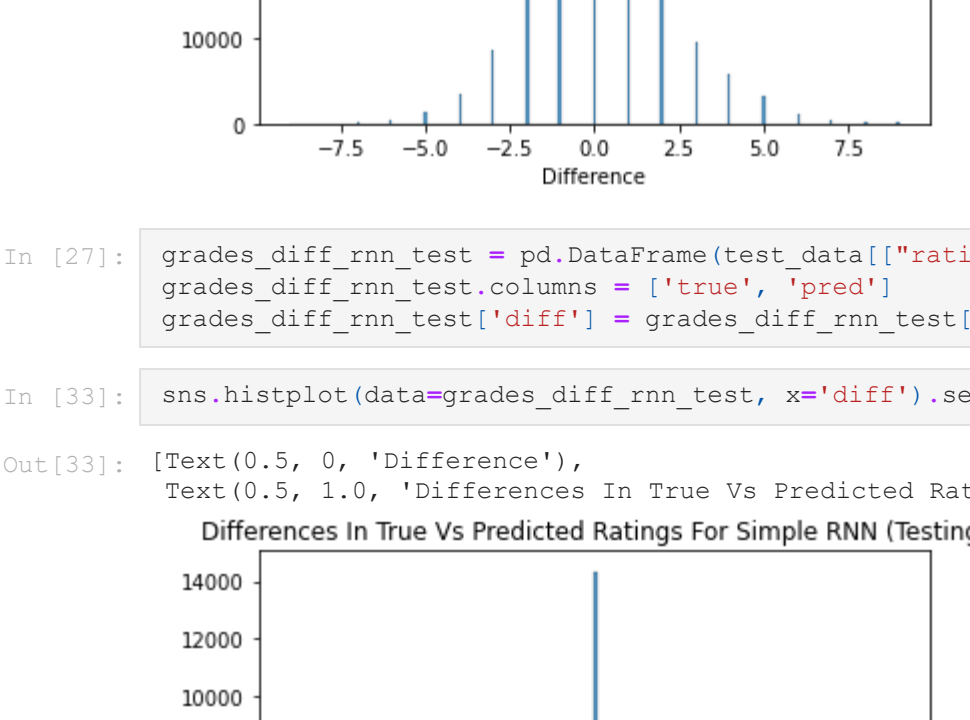
```
grades_true = pd.DataFrame(train_data[["rating"]])
grades_true["type"]="true"
grades_pred = pd.DataFrame(rnn_predict_train)
grades_pred.columns = ['rating']
grades_pred["type"]="pred"
grades_rnn_train = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_rnn_train, x='rating', hue='type', multiple="dodge").set(xlabel='Rating', title='True
```

```
[Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For Simple RNN (Training)')]
```




```
grades_true = pd.DataFrame(test_data[["rating"]])
grades_true["type"]="true"
grades_pred = pd.DataFrame(rnn_predict_test)
grades_pred.columns = ['rating']
grades_pred["type"]="pred"
grades_rnn_test = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_rnn_test, x='rating', hue='type', multiple="dodge").set(xlabel='Rating', title='True V
```

```
[Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For Simple RNN (Testing)')]
```

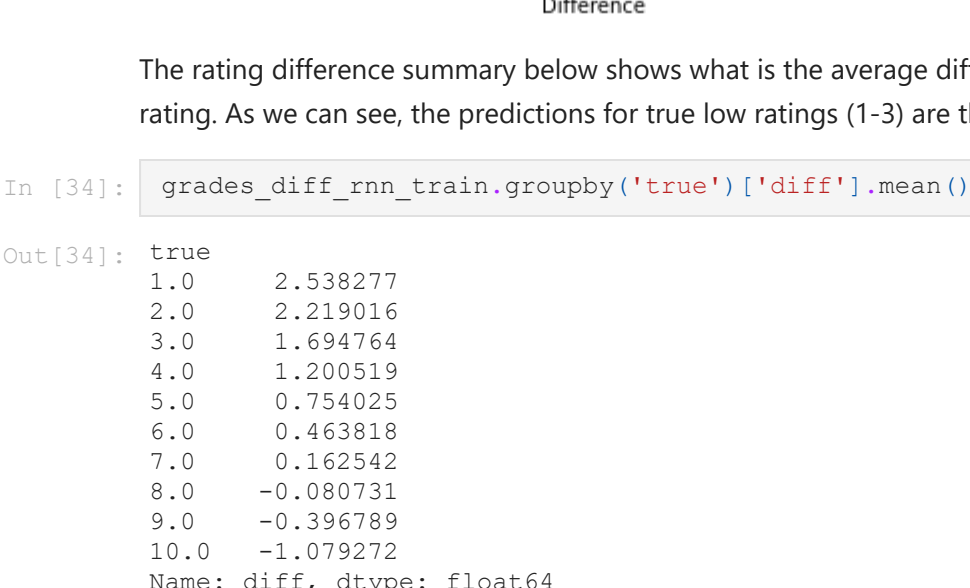


When comparing true and predicted ratings for the same reviews, the most common difference is zero, and the differences have a symmetric, bell shaped distribution:

```
grades_diff_rnn_train = pd.DataFrame(train_data[["rating"]]).join(pd.DataFrame(rnn_predict_train))
grades_diff_rnn_train.columns = ['true', 'pred']
grades_diff_rnn_train['diff'] = grades_diff_rnn_train['pred'] - grades_diff_rnn_train['true']
```

```
sns.histplot(data=grades_diff_rnn_train, x='diff').set(xlabel='Difference', title='Differences In True Vs Predi
```

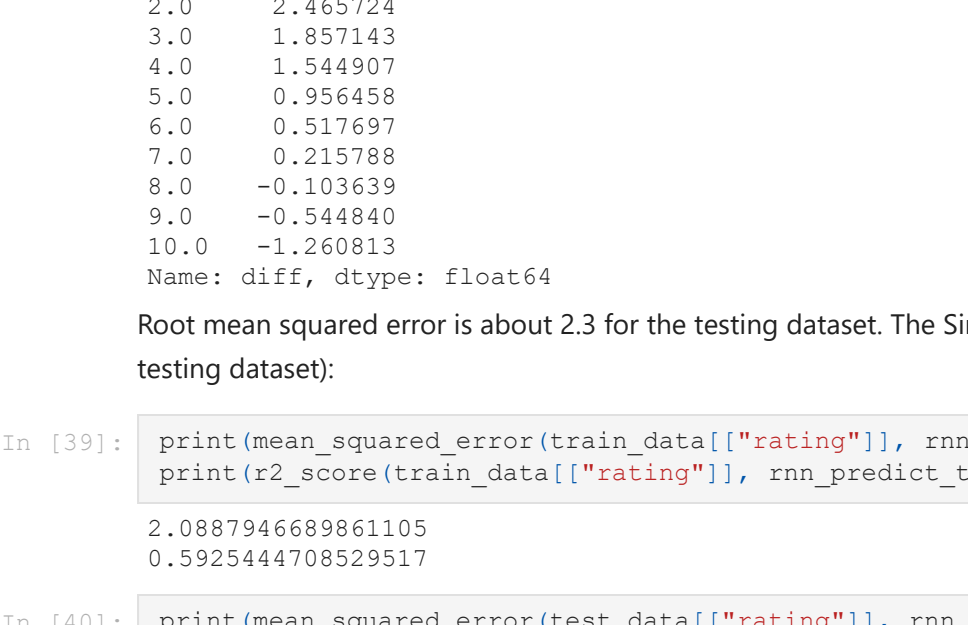
```
[Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For Simple RNN (Training)')]
```




```
grades_diff_rnn_test = pd.DataFrame(test_data[["rating"]]).join(pd.DataFrame(rnn_predict_test))
grades_diff_rnn_test.columns = ['true', 'pred']
grades_diff_rnn_test['diff'] = grades_diff_rnn_test['pred'] - grades_diff_rnn_test['true']
```

```
sns.histplot(data=grades_diff_rnn_test, x='diff').set(xlabel='Difference', title='Differences In True Vs Predic
```

```
[Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For Simple RNN (Testing)')]
```



The rating difference summary below shows what is the average difference between true and predicted ratings, for each value of the true rating. As we can see, the predictions for true low ratings (1-3) are the least accurate.

```
grades_diff_rnn_train.groupby('true')['diff'].mean()
```

```
true
1.0     2.538277
2.0     2.219016
3.0     1.694764
4.0     1.200519
5.0     0.754025
6.0     0.463818
7.0     0.162542
8.0    -0.080731
9.0    -0.396789
10.0   -1.079272
Name: diff, dtype: float64
```

```
grades_diff_rnn_test.groupby('true')['diff'].mean()
```

```
true
1.0     2.888889
2.0     2.465724
3.0     1.857143
4.0     1.554907
5.0     0.956458
6.0     0.517697
7.0     0.215788
8.0    -0.103639
9.0    -0.544840
10.0   -1.260813
Name: diff, dtype: float64
```

Root mean squared error is about 2.3 for the testing dataset. The Simple RNN model explains about half of the rating variance (for the testing dataset):

```
print(mean_squared_error(train_data[["rating"]], rnn_predict_train, squared=False))
print(r2_score(train_data[["rating"]], rnn_predict_train))
```

```
2.0877946689861105
0.5925447085259517
```

```
print(mean_squared_error(test_data[["rating"]], rnn_predict_test, squared=False))
print(r2_score(test_data[["rating"]], rnn_predict_test))
```

```
2.331271906525229
0.4964199956679445
```

GRU Model

The next model we will try is Gated Recurrent Unit (GRU). The workflow is the same as for the Simple RNN model. A three layer neural network is created, and the model is trained for 30 epochs using Adam as the optimizer.

```
rnn_hidden_dim = 5
word_embedding_dim = 50
model_gru = Sequential()
model_gru.add(Embedding(vectorizer.vocabulary_size(), word_embedding_dim))  #This layer takes each integer in
model_gru.add(GRU(rnn_hidden_dim,
                  activation='linear',
                  kernel_initializer='he_uniform',
                  input_shape=x_train.shape[1:]))

model_gru.add(Dense(1, kernel_initializer='he_uniform', activation=relu_advanced))
```

```
model_gru.compile(loss='mean_squared_error',
                  optimizer=keras.optimizers.Adam(0.0001),
                  metrics=['mean_squared_error'])
```

This time, model training took 28 epochs in total, achieving the lowest mean squared error for validation subset on epoch 25. However, it took few epochs to achieve a great improvement - 3 epochs were enough to reach a value lower than 6. The lowest reached value is about 4.5:

```
model_gru.fit(x_train, train_data[["rating"]],
          batch_size=batch_size,
          epochs=30,
          validation_split=0.2,
          callbacks=[es, mc_gru],
          verbose=1)
```

```
Epoch 1/30
4033/4033 [==============================] - 195s 48ms/step - loss: 22.6208 - mean_squared_error: 22.6208 - val
_loss: 10.2997 - val_mean_squared_error: 10.2997

Epoch 00001: val_mean_squared_error improved from inf to 10.29967, saving model to C:\Users\Lenovo\Desktop\ibm_
machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 2/30
4033/4033 [==============================] - 193s 48ms/step - loss: 8.7835 - mean_squared_error: 8.7835 - val_l
oss: 6.4588 - val_mean_squared_error: 6.4588

Epoch 00002: val_mean_squared_error improved from 10.29967 to 6.45884, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 3/30
4033/4033 [==============================] - 193s 48ms/step - loss: 6.1702 - mean_squared_error: 6.1702 - val_l
oss: 5.9537 - val_mean_squared_error: 5.9537

Epoch 00003: val_mean_squared_error improved from 6.45884 to 5.95375, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 4/30
4033/4033 [==============================] - 194s 48ms/step - loss: 5.5883 - mean_squared_error: 5.5883 - val_l
oss: 5.7707 - val_mean_squared_error: 5.7707

Epoch 00004: val_mean_squared_error improved from 5.95375 to 5.77068, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 5/30
4033/4033 [==============================] - 208s 52ms/step - loss: 5.2290 - mean_squared_error: 5.2290 - val_l
oss: 5.6523 - val_mean_squared_error: 5.6523

Epoch 00005: val_mean_squared_error improved from 5.77068 to 5.65230, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 6/30
4033/4033 [==============================] - 206s 51ms/step - loss: 4.9707 - mean_squared_error: 4.9707 - val_l
oss: 5.6463 - val_mean_squared_error: 5.6463

Epoch 00006: val_mean_squared_error improved from 5.65230 to 5.64632, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 7/30
4033/4033 [==============================] - 207s 51ms/step - loss: 4.6791 - mean_squared_error: 4.6791 - val_l
oss: 5.5597 - val_mean_squared_error: 5.5597

Epoch 00007: val_mean_squared_error improved from 5.64632 to 5.55968, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 8/30
4033/4033 [==============================] - 195s 48ms/step - loss: 4.5271 - mean_squared_error: 4.5271 - val_l
oss: 5.2827 - val_mean_squared_error: 5.2827

Epoch 00008: val_mean_squared_error improved from 5.55968 to 5.28266, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 9/30
4033/4033 [==============================] - 196s 49ms/step - loss: 4.3265 - mean_squared_error: 4.3265 - val_l
oss: 5.2739 - val_mean_squared_error: 5.2739

Epoch 00009: val_mean_squared_error improved from 5.28266 to 5.27393, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 10/30
4033/4033 [==============================] - 196s 48ms/step - loss: 4.1021 - mean_squared_error: 4.1021 - val_l
oss: 5.1444 - val_mean_squared_error: 5.1444

Epoch 00010: val_mean_squared_error improved from 5.27393 to 5.14440, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 11/30
4033/4033 [==============================] - 194s 48ms/step - loss: 3.9567 - mean_squared_error: 3.9567 - val_l
oss: 5.0817 - val_mean_squared_error: 5.0817

Epoch 00011: val_mean_squared_error improved from 5.14440 to 5.08170, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 12/30
4033/4033 [==============================] - 196s 49ms/step - loss: 3.7982 - mean_squared_error: 3.7982 - val_l
oss: 5.0281 - val_mean_squared_error: 5.0281

Epoch 00012: val_mean_squared_error improved from 5.08170 to 5.02813, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 13/30
4033/4033 [==============================] - 196s 48ms/step - loss: 3.7052 - mean_squared_error: 3.7052 - val_l
oss: 5.1443 - val_mean_squared_error: 5.1443

Epoch 00013: val_mean_squared_error did not improve from 5.02813
Epoch 14/30
4033/4033 [==============================] - 196s 48ms/step - loss: 3.5013 - mean_squared_error: 3.5013 - val_l
oss: 4.9434 - val_mean_squared_error: 4.9434

Epoch 00014: val_mean_squared_error improved from 5.02813 to 4.94342, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 15/30
4033/4033 [==============================] - 207s 51ms/step - loss: 3.3994 - mean_squared_error: 3.3994 - val_l
oss: 4.8779 - val_mean_squared_error: 4.8779

Epoch 00015: val_mean_squared_error improved from 4.94342 to 4.87787, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 16/30
4033/4033 [==============================] - 208s 52ms/step - loss: 3.2513 - mean_squared_error: 3.2513 - val_l
oss: 4.8801 - val_mean_squared_error: 4.8801

Epoch 00016: val_mean_squared_error did not improve from 4.87787
Epoch 17/30
4033/4033 [==============================] - 209s 52ms/step - loss: 3.1418 - mean_squared_error: 3.1418 - val_l
oss: 4.8645 - val_mean_squared_error: 4.8645

Epoch 00017: val_mean_squared_error improved from 4.87787 to 4.86446, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 18/30
4033/4033 [==============================] - 203s 50ms/step - loss: 3.0178 - mean_squared_error: 3.0178 - val_l
oss: 4.7282 - val_mean_squared_error: 4.7282

Epoch 00018: val_mean_squared_error improved from 4.86446 to 4.72820, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 19/30
4033/4033 [==============================] - 203s 50ms/step - loss: 2.8993 - mean_squared_error: 2.8993 - val_l
oss: 4.6558 - val_mean_squared_error: 4.6558

Epoch 00019: val_mean_squared_error improved from 4.72820 to 4.65584, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 20/30
4033/4033 [==============================] - 202s 50ms/step - loss: 2.7531 - mean_squared_error: 2.7531 - val_l
oss: 4.6682 - val_mean_squared_error: 4.6682

Epoch 00020: val_mean_squared_error did not improve from 4.65584
Epoch 21/30
4033/4033 [==============================] - 204s 51ms/step - loss: 2.6658 - mean_squared_error: 2.6658 - val_l
oss: 4.6474 - val_mean_squared_error: 4.6474

Epoch 00021: val_mean_squared_error improved from 4.65584 to 4.64736, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 22/30
4033/4033 [==============================] - 207s 51ms/step - loss: 2.5682 - mean_squared_error: 2.5682 - val_l
oss: 4.6193 - val_mean_squared_error: 4.6193

Epoch 00022: val_mean_squared_error improved from 4.64736 to 4.61931, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 23/30
4033/4033 [==============================] - 202s 50ms/step - loss: 2.4553 - mean_squared_error: 2.4553 - val_l
oss: 4.6066 - val_mean_squared_error: 4.6066

Epoch 00023: val_mean_squared_error improved from 4.61931 to 4.60659, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 24/30
4033/4033 [==============================] - 203s 50ms/step - loss: 2.4157 - mean_squared_error: 2.4157 - val_l
oss: 4.5705 - val_mean_squared_error: 4.5705

Epoch 00024: val_mean_squared_error improved from 4.60659 to 4.57049, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 25/30
4033/4033 [==============================] - 196s 49ms/step - loss: 2.3243 - mean_squared_error: 2.3243 - val_l
oss: 4.5127 - val_mean_squared_error: 4.5127

Epoch 00025: val_mean_squared_error improved from 4.57049 to 4.51273, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_gru_model.h5
Epoch 26/30
4033/4033 [==============================] - 194s 48ms/step - loss: 2.2156 - mean_squared_error: 2.2156 - val_l
oss: 4.6129 - val_mean_squared_error: 4.6129

Epoch 00026: val_mean_squared_error did not improve from 4.51273
Epoch 27/30
4033/4033 [==============================] - 191s 47ms/step - loss: 2.1572 - mean_squared_error: 2.1572 - val_l
oss: 4.5169 - val_mean_squared_error: 4.5169

Epoch 00027: val_mean_squared_error did not improve from 4.51273
Epoch 28/30
4033/4033 [==============================] - 192s 48ms/step - loss: 2.0764 - mean_squared_error: 2.0764 - val_l
oss: 4.5558 - val_mean_squared_error: 4.5558

Epoch 00028: val_mean_squared_error did not improve from 4.51273
```

```
<keras.callbacks.History at 0x1e0e5bce2e0>
```

```
gru_saved_model = load_model('C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_
```

Rating predictions are made for both train and testing datasets. Also, they are rounded to have integer values, just like the true ratings:

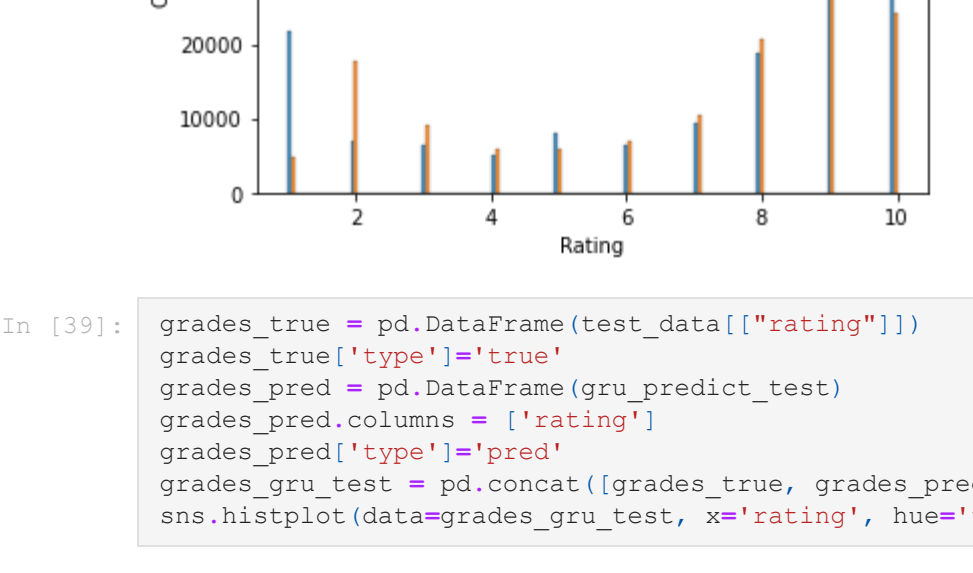
```
gru_predict_train = gru_saved_model.predict(x_train)
gru_predict_train = np.round(gru_predict_train)
```

```
gru_predict_test = gru_saved_model.predict(x_test)
gru_predict_test = np.round(gru_predict_test)
```

The histograms below show that true and predicted rating counts for middle values (3 to 8) are similar. Ratings 2 and 9 are overpredicted, while 1 and 10 - underpredicted:

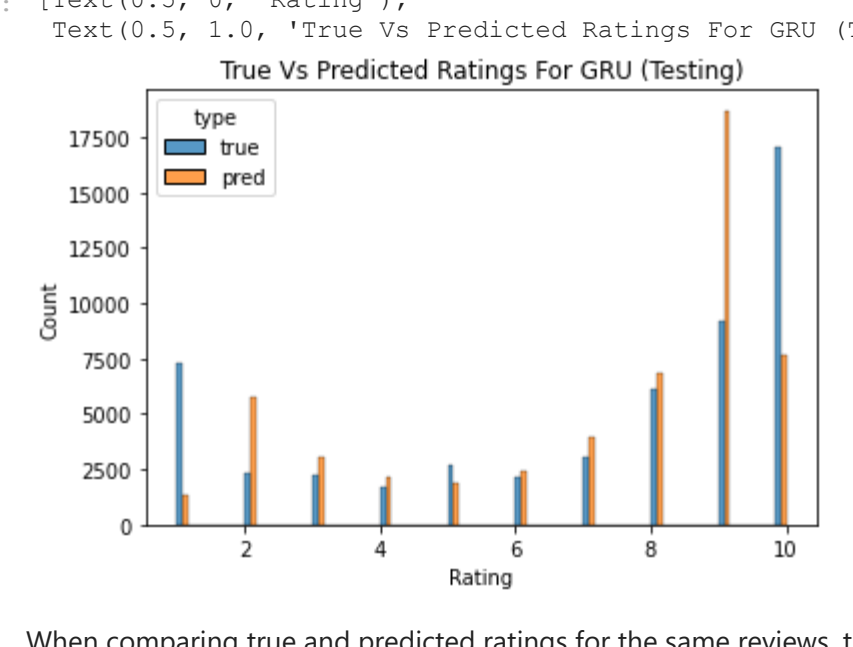
```
grades_true = pd.DataFrame(train_data[["rating"]])
grades_true["type"]="true"
grades_pred = pd.DataFrame(gru_predict_train)
grades_pred.columns = ['rating']
grades_pred["type"]="pred"
grades_gru_train = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_gru_train, x='rating', hue='type', multiple='dodge').set(xlabel='Rating', title='True
```

```
[Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For GRU (Training)')]
```




```
grades_true = pd.DataFrame(test_data[["rating"]])
grades_true["type"]="true"
grades_pred = pd.DataFrame(gru_predict_test)
grades_pred.columns = ['rating']
grades_pred["type"]="pred"
grades_gru_test = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_gru_test, x='rating', hue='type', multiple='dodge').set(xlabel='Rating', title='True V
```

Out[38]: [Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For GRU (Testing)')]

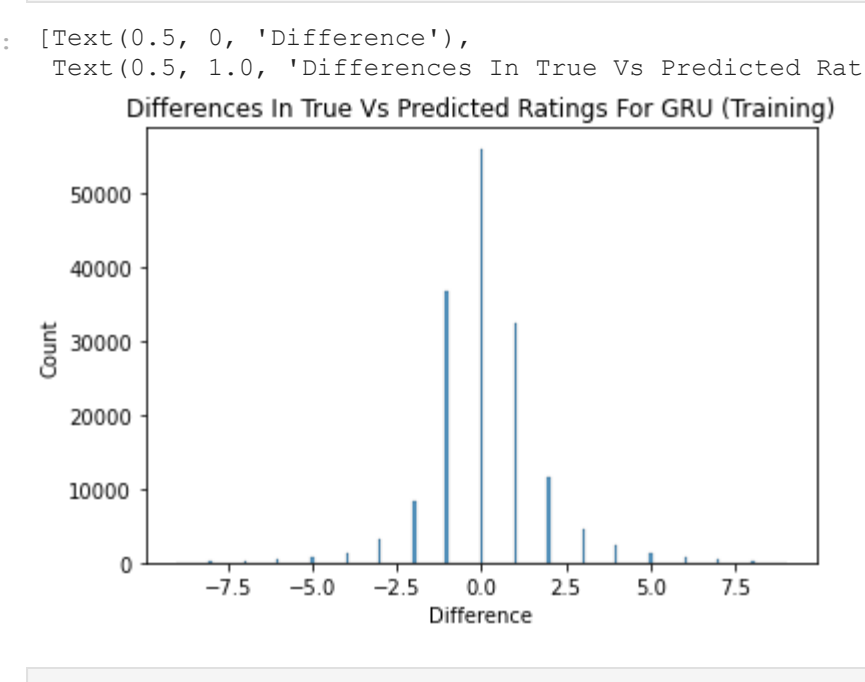


When comparing true and predicted ratings for the same reviews, the most common difference is zero, and the differences have a symmetric, bell shaped distribution (similar to Simple RNN). On the other hand, there are many differences equal to 1 or -1 (which probably reflects the misprediction for 1/2 and 9/10 grades demonstrated above):

In [41]:
```
grades_diff_gru_train = pd.DataFrame(train_data[["rating"]]).join(pd.DataFrame(gru_predict_train))
grades_diff_gru_train.columns = ['true', 'pred']
grades_diff_gru_train['diff'] = grades_diff_gru_train['pred'] - grades_diff_gru_train['true']
```

In [43]:
```
sns.histplot(data=grades_diff_gru_train, x='diff').set(xlabel='Difference', title='Differences In True Vs Predic
```

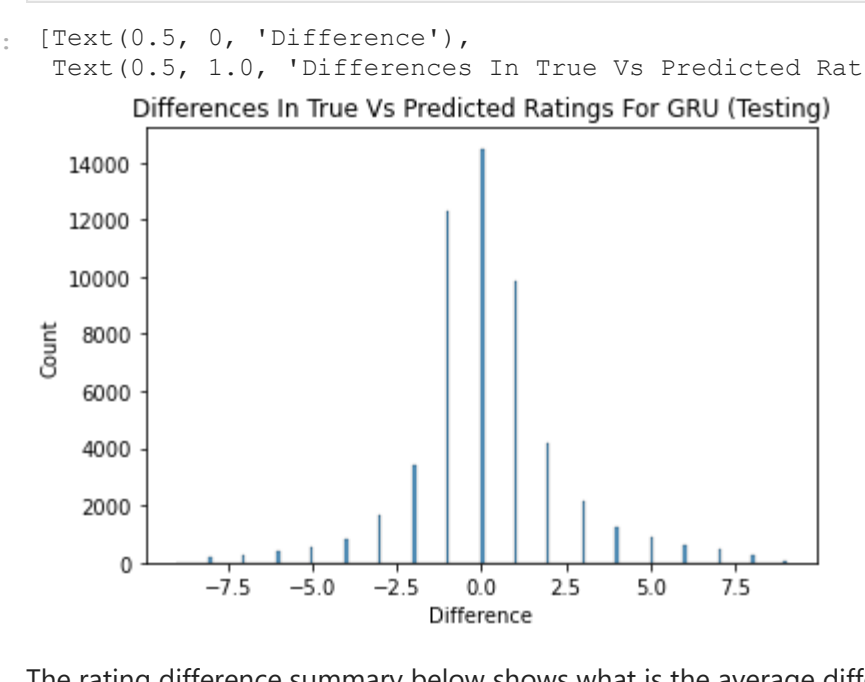
Out[43]: [Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For GRU (Training)')]



In [45]:
```
grades_diff_gru_test = pd.DataFrame(test_data[["rating"]]).join(pd.DataFrame(gru_predict_test))
grades_diff_gru_test.columns = ['true', 'pred']
grades_diff_gru_test['diff'] = grades_diff_gru_test['pred'] - grades_diff_gru_test['true']
```

In [46]:
```
sns.histplot(data=grades_diff_gru_test, x='diff').set(xlabel='Difference', title='Differences In True Vs Predic
```

Out[46]: [Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For GRU (Testing)')]



The rating difference summary below shows what is the average difference between true and predicted ratings, for each value of the true rating. As we can see, the predictions for true low ratings (1-3) and rating 10 are the least accurate (similar to the Simple RNN model).

In [50]:
```
grades_diff_gru_train.groupby('true')['diff'].mean()
```

Out[50]:
```
true
1.0     1.571349
2.0     1.170105
3.0     0.824044
4.0     0.641860
5.0     0.592038
6.0     0.571023
7.0     0.479061
8.0     0.229751
9.0    -0.194217
10.0   -0.909118
Name: diff, dtype: float64
```

In [54]:
```
grades_diff_gru_test.groupby('true')['diff'].mean()
```

Out[54]:
```
true
1.0     2.244554
2.0     1.797344
3.0     1.236735
4.0     1.124171
5.0     0.848339
6.0     0.559698
7.0     0.428017
8.0     0.108512
9.0    -0.396644
10.0   -1.177598
Name: diff, dtype: float64
```

Root mean squared error for testing dataset is about 2.18, and GRU model explains about 56% of rating variance. In other words, the results are a bit better than for the Simple RNN model (2.33 and 50% respectively).

In [55]:
```
print(mean_squared_error(train_data[["rating"]], gru_predict_train, squared=False))
print(r2_score(train_data[["rating"]], gru_predict_train))
```

```
1.6605694324174551
0.7424848447722434
```

In [56]:
```
print(mean_squared_error(test_data[["rating"]], gru_predict_test, squared=False))
print(r2_score(test_data[["rating"]], gru_predict_test))
```

```
2.184468003322141
0.5578455578320375
```

LSTM

The last model to experiment with is the Long Short Term Memory (LSTM) model. The workflow is the same as for Simple RNN and GRU models - a three layer neural network is created, and the model is trained for 30 epochs using Adam as the optimizer.

In [57]:
```
rnn_hidden_dim = 5
word_embedding_dim = 50
model_lstm = Sequential()
model_lstm.add(Embedding(vectorizer.vocabulary_size(), word_embedding_dim))  #This layer takes each integer in
model_lstm.add(LSTM(rnn_hidden_dim,
                activation='linear',
                kernel_initializer='he_uniform',
                input_shape=x_train.shape[1:]))

model_lstm.add(Dense(1, kernel_initializer='he_uniform', activation=relu_advanced))
```

In [58]:
```
model_lstm.compile(loss='mean_squared_error',
              optimizer=keras.optimizers.Adam(0.0001),
              metrics=['mean_squared_error'])
```

LSTM model was trained slower than the previous ones - it took 12 epochs to reach a mean squared error on validation subset lower than 6. But the total number of epochs is 28, the lowest value was reached on epoch 25 (4.43):

In [59]:
```
model_lstm.fit(x_train, train_data[["rating"]],
          batch_size=batch_size,
          epochs=30,
          validation_split=0.2,
          callbacks=[es, mc_lstm],
          verbose=1)
```

```
Epoch 1/30
4033/4033 [==============================] - 181s 44ms/step - loss: 28.7854 - mean_squared_error: 28.7854 - val
_loss: 19.5838 - val_mean_squared_error: 19.5838

Epoch 00001: val_mean_squared_error improved from inf to 19.58378, saving model to C:\Users\Lenovo\Desktop\ibm_
machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 2/30
4033/4033 [==============================] - 178s 44ms/step - loss: 25.5417 - mean_squared_error: 25.5417 - val
_loss: 18.9546 - val_mean_squared_error: 18.9546

Epoch 00002: val_mean_squared_error improved from 19.58378 to 18.95464, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 3/30
4033/4033 [==============================] - 178s 44ms/step - loss: 18.6916 - mean_squared_error: 18.6916 - val
_loss: 17.7724 - val_mean_squared_error: 17.7724

Epoch 00003: val_mean_squared_error improved from 18.95464 to 17.77243, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 4/30
4033/4033 [==============================] - 178s 44ms/step - loss: 16.1179 - mean_squared_error: 16.1179 - val
_loss: 10.7450 - val_mean_squared_error: 10.7450

Epoch 00004: val_mean_squared_error improved from 17.77243 to 10.74496, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 5/30
4033/4033 [==============================] - 178s 44ms/step - loss: 10.7484 - mean_squared_error: 10.7484 - val
_loss: 10.5083 - val_mean_squared_error: 10.5083

Epoch 00005: val_mean_squared_error improved from 10.74496 to 10.50829, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 6/30
4033/4033 [==============================] - 178s 44ms/step - loss: 10.4891 - mean_squared_error: 10.4891 - val
_loss: 10.1590 - val_mean_squared_error: 10.1590

Epoch 00006: val_mean_squared_error improved from 10.50829 to 10.15898, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 7/30
4033/4033 [==============================] - 182s 45ms/step - loss: 9.6658 - mean_squared_error: 9.6658 - val_l
oss: 8.4039 - val_mean_squared_error: 8.4039

Epoch 00007: val_mean_squared_error improved from 10.15898 to 8.40393, saving model to C:\Users\Lenovo\Desktop
\ibm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 8/30
4033/4033 [==============================] - 183s 45ms/step - loss: 8.6142 - mean_squared_error: 8.6142 - val_l
oss: 8.4499 - val_mean_squared_error: 8.4499

Epoch 00008: val_mean_squared_error did not improve from 8.40393
Epoch 9/30
4033/4033 [==============================] - 209s 52ms/step - loss: 7.9338 - mean_squared_error: 7.9338 - val_l
oss: 7.3564 - val_mean_squared_error: 7.3564

Epoch 00009: val_mean_squared_error improved from 8.40393 to 7.35642, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 10/30
4033/4033 [==============================] - 199s 49ms/step - loss: 7.2320 - mean_squared_error: 7.2320 - val_l
oss: 6.9430 - val_mean_squared_error: 6.9430

Epoch 00010: val_mean_squared_error improved from 7.35642 to 6.94305, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 11/30
4033/4033 [==============================] - 181s 45ms/step - loss: 6.5223 - mean_squared_error: 6.5223 - val_l
oss: 6.4742 - val_mean_squared_error: 6.4742

Epoch 00011: val_mean_squared_error improved from 6.94305 to 6.47417, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 12/30
4033/4033 [==============================] - 182s 45ms/step - loss: 5.7859 - mean_squared_error: 5.7859 - val_l
oss: 5.6192 - val_mean_squared_error: 5.6192

Epoch 00012: val_mean_squared_error improved from 6.47417 to 5.61922, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 13/30
4033/4033 [==============================] - 180s 45ms/step - loss: 4.8196 - mean_squared_error: 4.8196 - val_l
oss: 5.1371 - val_mean_squared_error: 5.1371

Epoch 00013: val_mean_squared_error improved from 5.61922 to 5.13709, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 14/30
4033/4033 [==============================] - 180s 45ms/step - loss: 4.1779 - mean_squared_error: 4.1779 - val_l
oss: 4.7804 - val_mean_squared_error: 4.7804

Epoch 00014: val_mean_squared_error improved from 5.13709 to 4.78039, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 15/30
4033/4033 [==============================] - 180s 45ms/step - loss: 3.8637 - mean_squared_error: 3.8637 - val_l
oss: 4.8310 - val_mean_squared_error: 4.8310

Epoch 00015: val_mean_squared_error did not improve from 4.78039
Epoch 16/30
4033/4033 [==============================] - 180s 45ms/step - loss: 3.5918 - mean_squared_error: 3.5918 - val_l
oss: 4.6991 - val_mean_squared_error: 4.6991

Epoch 00016: val_mean_squared_error improved from 4.78039 to 4.69905, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 17/30
4033/4033 [==============================] - 179s 44ms/step - loss: 3.3730 - mean_squared_error: 3.3730 - val_l
oss: 4.7866 - val_mean_squared_error: 4.7866

Epoch 00017: val_mean_squared_error did not improve from 4.69905
Epoch 18/30
4033/4033 [==============================] - 181s 45ms/step - loss: 3.1807 - mean_squared_error: 3.1807 - val_l
oss: 4.5333 - val_mean_squared_error: 4.5333

Epoch 00018: val_mean_squared_error improved from 4.69905 to 4.53334, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 19/30
4033/4033 [==============================] - 183s 45ms/step - loss: 2.9968 - mean_squared_error: 2.9968 - val_l
oss: 4.4682 - val_mean_squared_error: 4.4682

Epoch 00019: val_mean_squared_error improved from 4.53334 to 4.46824, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 20/30
4033/4033 [==============================] - 184s 46ms/step - loss: 2.8104 - mean_squared_error: 2.8104 - val_l
oss: 4.4930 - val_mean_squared_error: 4.4930

Epoch 00020: val_mean_squared_error did not improve from 4.46824
Epoch 21/30
4033/4033 [==============================] - 185s 46ms/step - loss: 2.6787 - mean_squared_error: 2.6787 - val_l
oss: 4.4854 - val_mean_squared_error: 4.4854

Epoch 00021: val_mean_squared_error did not improve from 4.46824
Epoch 22/30
4033/4033 [==============================] - 190s 47ms/step - loss: 2.5963 - mean_squared_error: 2.5963 - val_l
oss: 4.3931 - val_mean_squared_error: 4.3931

Epoch 00022: val_mean_squared_error improved from 4.46824 to 4.39312, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 23/30
4033/4033 [==============================] - 186s 46ms/step - loss: 2.4585 - mean_squared_error: 2.4585 - val_l
oss: 4.5230 - val_mean_squared_error: 4.5230

Epoch 00023: val_mean_squared_error did not improve from 4.39312
Epoch 24/30
4033/4033 [==============================] - 184s 46ms/step - loss: 2.4029 - mean_squared_error: 2.4029 - val_l
oss: 4.4392 - val_mean_squared_error: 4.4392

Epoch 00024: val_mean_squared_error did not improve from 4.39312
Epoch 25/30
4033/4033 [==============================] - 183s 45ms/step - loss: 2.2901 - mean_squared_error: 2.2901 - val_l
oss: 4.3785 - val_mean_squared_error: 4.3785

Epoch 00025: val_mean_squared_error improved from 4.39312 to 4.37845, saving model to C:\Users\Lenovo\Desktop\i
bm_machine_learning\5_deep_learning_and_reinforcement_learning\week6\assignment\best_lstm_model.h5
Epoch 26/30
4033/4033 [==============================] - 188s 47ms/step - loss: 2.1740 - mean_squared_error: 2.1740 - val_l
oss: 4.4302 - val_mean_squared_error: 4.4302

Epoch 00026: val_mean_squared_error did not improve from 4.37845
Epoch 27/30
4033/4033 [==============================] - 180s 45ms/step - loss: 2.0926 - mean_squared_error: 2.0926 - val_l
oss: 4.4027 - val_mean_squared_error: 4.4027

Epoch 00027: val_mean_squared_error did not improve from 4.37845
Epoch 28/30
4033/4033 [==============================] - 183s 45ms/step - loss: 2.0208 - mean_squared_error: 2.0208 - val_l
oss: 4.3847 - val_mean_squared_error: 4.3847

Epoch 00028: val_mean_squared_error did not improve from 4.37845
```

Out[59]: <keras.callbacks.History at 0x1e5886404c0>

In [47]:
```
lstm_saved_model = load_model(r'C:\Users\Lenovo\Desktop\ibm_machine_learning\5_deep_learning_and_reinforcement_
```

Rating predictions are made for both training and testing datasets. Also, they are rounded to have integer values, just like the true ratings:

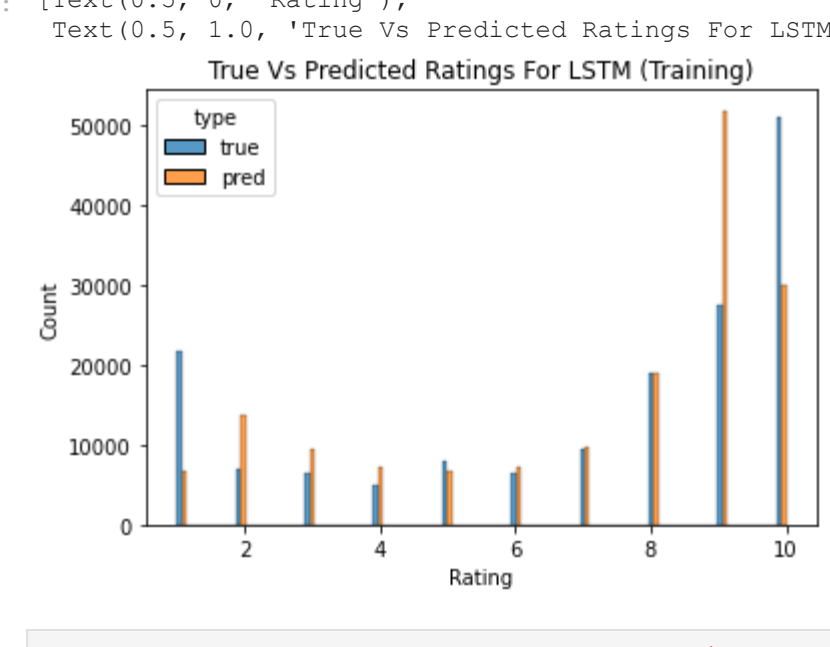
In [48]:
```
lstm_predict_train = lstm_saved_model.predict(x_train)
lstm_predict_train = np.round(lstm_predict_train)
```

In [49]:
```
lstm_predict_test = lstm_saved_model.predict(x_test)
lstm_predict_test = np.round(lstm_predict_test)
```

The histograms below show that true and predicted rating counts for middle values (3 to 8) are similar. Ratings 2 and 9 are overpredicted, while 1 and 10 - underpredicted (similar to GRU model):

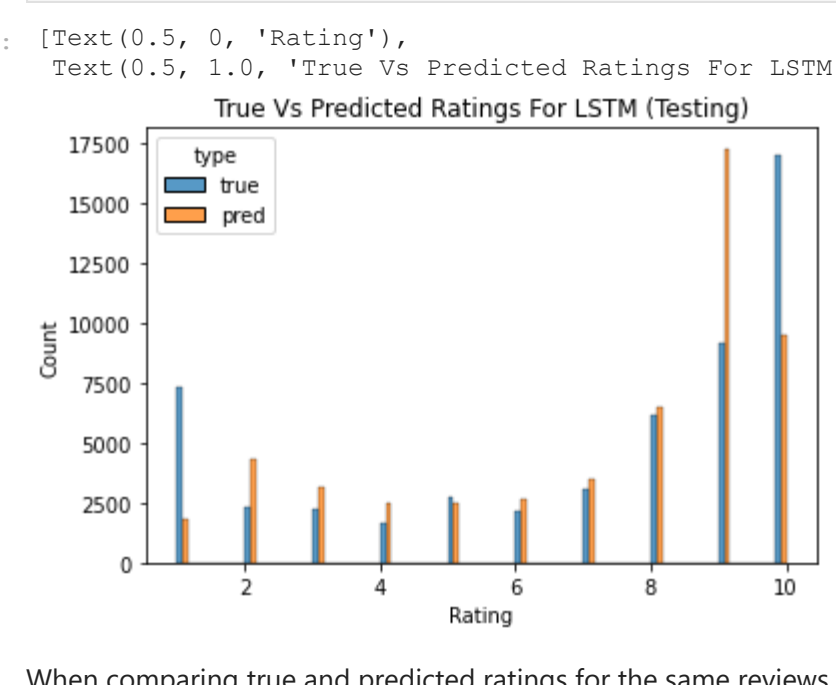
In [50]:
```
grades_true = pd.DataFrame(train_data[["rating"]])
grades_true['type']='true'
grades_pred = pd.DataFrame(lstm_predict_train)
grades_pred.columns = ['rating']
grades_pred['type']='pred'
grades_lstm_train = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_lstm_train, x='rating', hue='type', multiple='dodge').set(xlabel='Rating', title='True
```

Out[50]: [Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For LSTM (Training)')]



In [52]:
```
grades_true = pd.DataFrame(test_data[["rating"]])
grades_true['type']='true'
grades_pred = pd.DataFrame(lstm_predict_test)
grades_pred.columns = ['rating']
grades_pred['type']='pred'
grades_lstm_test = pd.concat([grades_true, grades_pred])
sns.histplot(data=grades_lstm_test, x='rating', hue='type', multiple='dodge').set(xlabel='Rating', title='True
```

Out[52]: [Text(0.5, 0, 'Rating'),
 Text(0.5, 1.0, 'True Vs Predicted Ratings For LSTM (Testing)')]

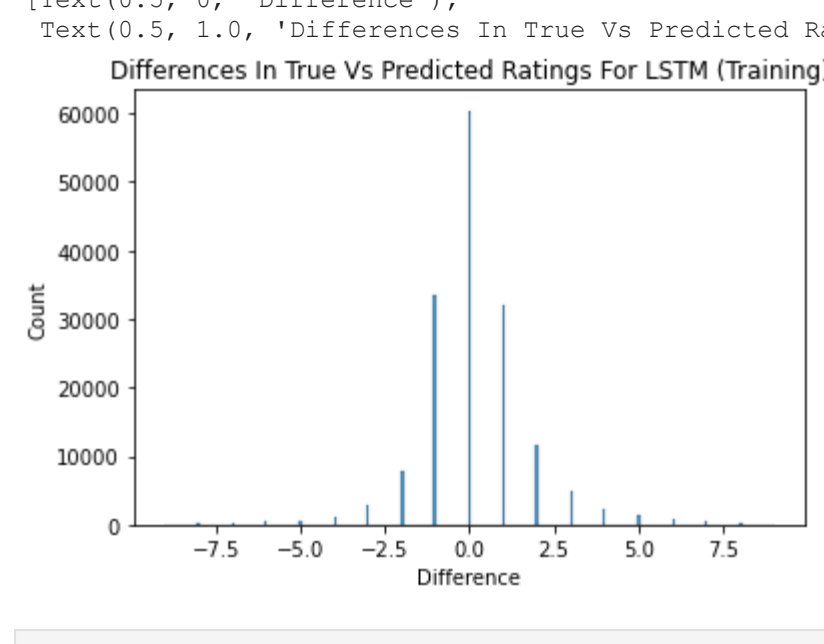


When comparing true and predicted ratings for the same reviews, the most common difference is zero, and the differences have a symmetric, bell shaped distribution (similar to previous models):

In [53]:
```
grades_diff_lstm_train = pd.DataFrame(train_data[["rating"]]).join(pd.DataFrame(lstm_predict_train))
grades_diff_lstm_train.columns = ['true', 'pred']
grades_diff_lstm_train['diff'] = grades_diff_lstm_train['pred'] - grades_diff_lstm_train['true']
```

In [54]:
```
sns.histplot(data=grades_diff_lstm_train, x='diff').set(xlabel='Difference', title='Differences In True Vs Pred
```

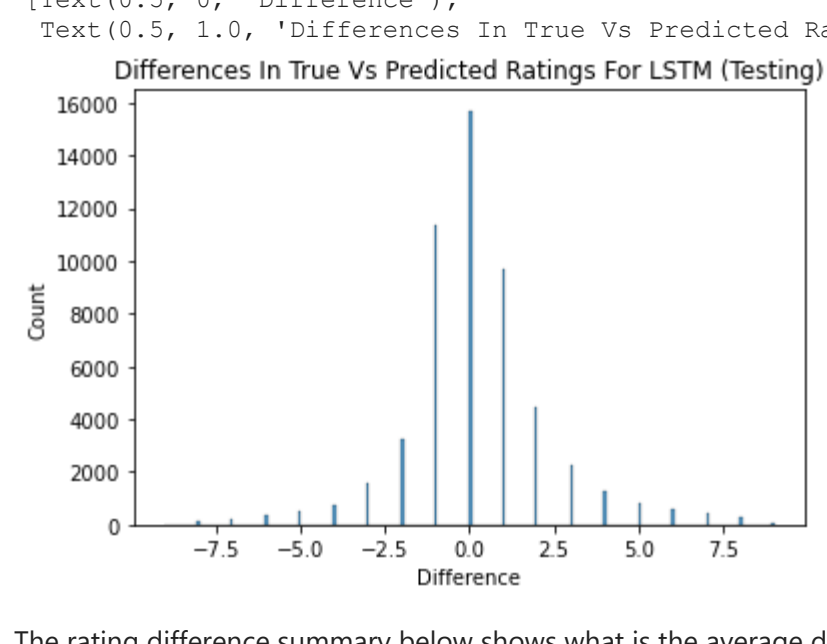
Out[54]: [Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For LSTM (Training)')]



In [56]:
```
grades_diff_lstm_test = pd.DataFrame(test_data[["rating"]]).join(pd.DataFrame(lstm_predict_test))
grades_diff_lstm_test.columns = ['true', 'pred']
grades_diff_lstm_test['diff'] = grades_diff_lstm_test['pred'] - grades_diff_lstm_test['true']
```

In [57]:
```
sns.histplot(data=grades_diff_lstm_test, x='diff').set(xlabel='Difference', title='Differences In True Vs Predi
```

Out[57]: [Text(0.5, 0, 'Difference'),
 Text(0.5, 1.0, 'Differences In True Vs Predicted Ratings For LSTM (Testing)')]



The rating difference summary below shows what is the average difference between true and predicted ratings, for each value of the true rating. As we can see, the predictions for true low ratings (1-3) are the least accurate (similar to the previous models).

In [66]:
```
grades_diff_lstm_train.groupby('true')['diff'].mean()
```

Out[66]:
```
true
1.0     1.525047
2.0     1.286972
3.0     0.988638
4.0     0.725259
5.0     0.597404
6.0     0.574492
7.0     0.473985
8.0     0.278560
9.0    -0.117976
10.0   -0.798133
Name: diff, dtype: float64
```

In [70]:
```
grades_diff_lstm_test.groupby('true')['diff'].mean()
```

Out[70]:
```
true
1.0     2.228524
2.0     1.898029
3.0     1.435828
4.0     1.268234
5.0     0.906273
6.0     0.586126
7.0     0.449693
8.0     0.125406
9.0    -0.323853
10.0   -1.073578
Name: diff, dtype: float64
```

Root mean squared error for testing dataset is about 2.16, and GRU model explains about 57% of rating variance. In other words, the results are a bit better than for the GRU model (2.18 and 56% respectively). The GRU model was better than the Simple RNN model, so LSTM model gives the best results:

In [71]:
```
print(mean_squared_error(train_data[["rating"]], lstm_predict_train, squared=False))
print(r2_score(train_data[["rating"]], lstm_predict_train))
```

```
1.6290849971065933
0.7521572593261975
```

In [72]:
```
print(mean_squared_error(test_data[["rating"]], lstm_predict_test, squared=False))
print(r2_score(test_data[["rating"]], lstm_predict_test))
```

```
2.15596362368956
0.5693093190057756
```

Model Recommendation

In the analysis above, three types of Recurrent Neural Networks (Simple RNN, GRU, LSTM) were applied to predict the patient satisfaction rating given their feedback on drugs. Mean Squared Error and R^2 score were used to assess model quality. Among those three models, LSTM results in the lowest Mean Squared Error and the highest R^2 score, so is the best one.

Summary And Key Findings

In this analysis, Recurrent Neural Network model was used to predict drug ratings with patient reviews (integer values and text data, respectively). Three types of RNNs were tried - Simple RNN, Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM) model.

The best model among the three is LSTM, whose Root Mean Squared Error equals 2.16, and R^2 score is 0.57. The second best model is GRU (2.18 and 0.56), followed by Simple RNN (2.33 and 0.50).

The results for GRU and LSTM are very similar not only in the metrics mentioned above, but in the distribution of predicted ratings as well. For both models, the distribution of predicted ratings is similar to true ratings for middle values (3-8). However, there are much fewer predicted 1 and 10 rating values and much more predicted 2 and 9 rating values, compared to the true ones.

The predicted values for Simple RNN are distributed differently - rating values on both ends (1, 2, 9 and 10) are predicted less frequently, and middle values (4-8) are predicted more frequently, compared to the true values.

However, for all three models, reviews have significantly higher predicted values on the lower end (about 2 points on average for grades 1 and 2), and lower predicted values on the higher end (more than 1 point for grade 10). Rating values between 6 and 9 are predicted the most accurately. This is not suprising, given the mean of grades around 7, and the median equal to 8.

There are some differences in how these models were trained. The Simple RNN model took only 3 epochs to reach the lowest mean squared error (on validation subset). However, both GRU and LSTM models reached the minimum value on epoch 25. The GRU model reached MSE value lower than 6 on epoch 3, while it took 12 epochs for LSTM to reach the same value. So in this case, the LSTM model was harder to train, despite slightly better final results.

One more observation - differences between training duration. For Simple RNN, each epoch takes about 2.5 minutes, which is the shortest time. On the other hand, GRU is the most time demanding, with about 3.5 minutes for each epoch. LSTM is in between, with about 3 minutes on average.

Suggestions

For the best (LSTM) model, root mean squared error for rating prediction is about 2.16, and R^2 score equals about 0.57. The model does a relatively poor job in predicting either high or low ratings (1, 2, 9 and 10), while does a better job for average grades. So there is room for improvement. Possible ways to achieve it are discussed below.

Since model weights are initialized randomly, it depends on the selected random seed in our case. Trying a different random seed might yield different results, and probably a more effective model. Apart from that, there are multiple ways to initialize weights (we chose He uniform), trying others might be useful as well.

All our models contained three layers - the embedding layer, the model layer and the output. Adding more layers (or maybe experimenting with combinations of model layers) might be another option to try.

We selected Adam as an optimizer, with a learning rate of 0.0001 (which is lower than the default rate of 0.001). It might be worth trying other optimizers and/or learning rates in order to either improve model accuracy, or to save time for model training.

We selected our vocabulary to consist of 50000 words, and each sequence to be of length 150. Selecting lower numbers will probably increase the training speed, and might not even reduce the accuracy.

We relied on arbitrary selection of some other model parameters as well - such as layer output dimensions. Applying KerasTuner to do the optimal hyperparameter search for output dimensions and other parameters discussed above might be another option to improve the model.

Finally, other types of neural networks (such as Convolutional) might be used for Natural Language Processing related tasks, such as this one.