Portfolio 10: Text Classification 2

Bushra Rahman

Text Classification on Spam Data

"NLP Email Classification" is a dataset of email bodies and labels for text classification, uploaded to Kaggle (URL https://www.kaggle.com/datasets/datatattle/email-classification-nlp). This dataset consists of a folder named **archive** which contains 2 CSV files, **SMS_train** and **SMS_test**. SMS_train contains **957** observations, and SMS_test contains **125** observations. Both CSV files have 3 columns: the row number of the observation, a string containing the **message body**, and the binary label for **spam/non-spam**. Altogether, the two CSV files contain **1,082** observations. The percentage of the train/test split between CSV_train and CSV_test is about 88/12. However, for the purposes of this program, we will ignore the preset train/test split and create our own.

Google Colab is used to classify this text using Tensorflow. In order to use a **GPU** with TensorFlow in this notebook, go to Edit >> Notebook settings >> and select *GPU* under **Hardware accelerator**.

Opening & Exploring the CSV Files

In order to upload files in Google Colab, first run this code block:

In [1]:
```
from google.colab import files
uploaded = files.upload()
# Select SMS_test.csv and SMS_train.csv, which are under 'archive'
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving SMS_test.csv to SMS_test (3).csv
Saving SMS_train.csv to SMS_train (3).csv
```

After uploading the selected CSV files, Pandas is used to read the CSV files into data frames, which are then concatenated into one data frame.

In [2]:
```
import pathlib
import pandas as pd
import io

# Convert CSV files into 2 Pandas dataframes
train_df = pd.read_csv(io.BytesIO(uploaded['SMS_train.csv']), usecols=[1,2], encoding=
test_df = pd.read_csv(io.BytesIO(uploaded['SMS_test.csv']), usecols=[1,2], encoding='

# Concatenate into one df
df = pd.concat([train_df, test_df])
print('Rows and columns of concatenated df:', df.shape)
```

```
# Replace Non-Spam/Spam with 0/1
df['Label'].replace(['Non-Spam', 'Spam'], [0, 1], inplace=True)

print('Rows and columns of concatenated df:', df.shape)
print(df.head())
```

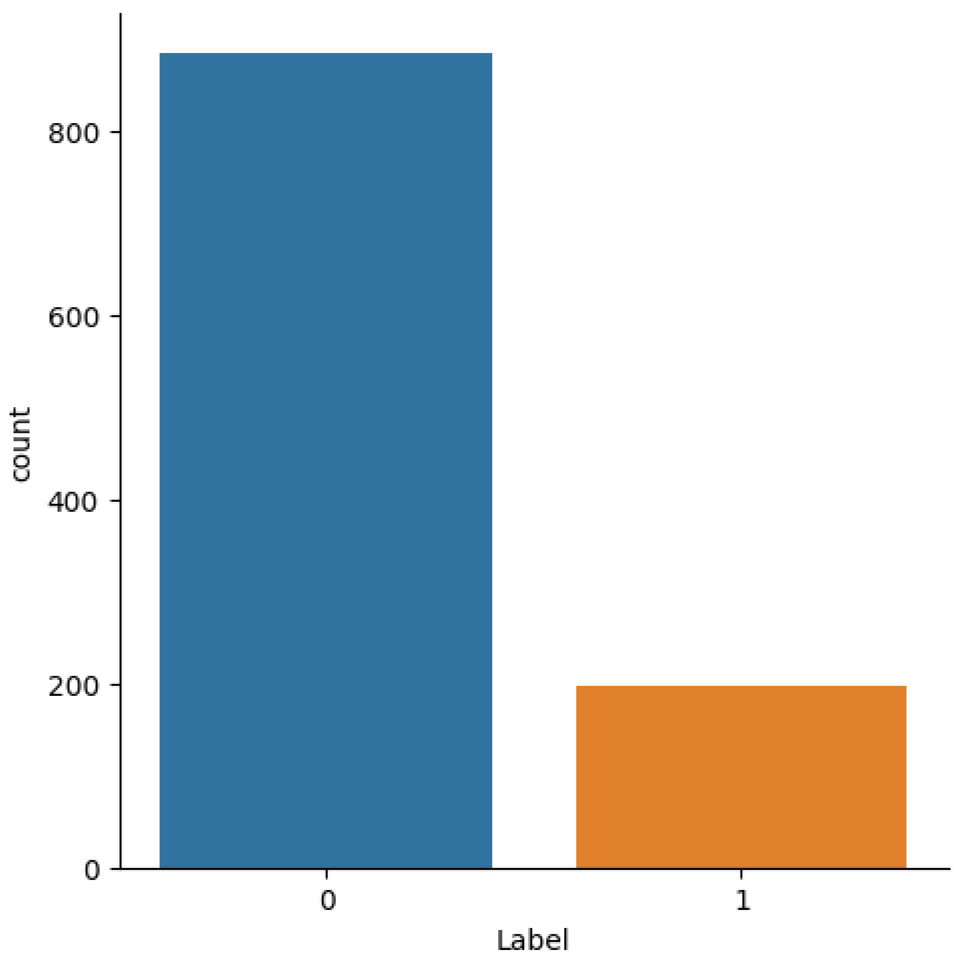
```
Rows and columns of concatenated df: (1082, 2)
Rows and columns of concatenated df: (1082, 2)
                                        Message_body  Label
0                           Rofl. Its true to its name      0
1  The guy did some bitching but I acted like i'd...      0
2  Pity, * was in mood for that. So...any other s...      0
3                Will ü b going to esplanade fr home?      0
4  This is the 2nd time we have tried 2 contact u...      1
```

Target Distribution Graph

The following code uses Seaborn to show the distribution of the target class **Label**.

In [3]:
```
import seaborn as sb
import matplotlib.pyplot as plt  # Plots created using seaborn need to be displayed li

sb.catplot(x='Label', kind='count', data=df)
plt.show()
```



Train/Test Split

The following code uses Numpy to split the data into 80/20 train/test sets.

```
In [4]: import numpy as np

        # Set seed for reproducibility
        np.random.seed(1234)

        # Divide into train/test sets on an 80/20 split
        i = np.random.rand(len(df)) < 0.8
        train = df[i]
        test = df[~i]
        print("train data size: ", train.shape)
        print("test data size: ", test.shape)
```

```
train data size:  (845, 2)
test data size:  (237, 2)
```

Data Preprocessing

The core structures in TensorFlow are layers and models. A **layer** is an object or representing a transformation step whose function is to input and output tensors. The **model** defines how the layers work together. In order to apply neural network models to the training data, first the data must be preprocessed and vectorized.

```
In [5]: import tensorflow as tf
        from tensorflow import keras
        from keras.preprocessing.text import Tokenizer
        from keras import layers, models, preprocessing

        # Check tf version for this notebook
        print('Currently using version', tf.__version__, 'of TensorFlow')
```

```
Currently using version 2.12.0 of TensorFlow
```

```
In [6]: # Set up X and Y:
        # train_data = x_train, test_data = x_test,
        # train_labels = y_train, test_labels = y_test.
        num_labels = 2
        vocab_size = 25000
        batch_size = 100

        # Fit the tokenizer on the training data
        tokenizer = Tokenizer(num_words=vocab_size)
        tokenizer.fit_on_texts(train.Message_body)

        # Vectorized training data
        x_train = tokenizer.texts_to_matrix(train.Message_body, mode='tfidf')
        x_test = tokenizer.texts_to_matrix(test.Message_body, mode='tfidf')

        from sklearn.preprocessing import LabelEncoder
        encoder = LabelEncoder()

        # Vectorized labels
        encoder.fit(train.Label)
        y_train = encoder.transform(train.Label)
        y_test = encoder.transform(test.Label)
```

```
# Check shape
print("Train shapes:", x_train.shape, y_train.shape)
print("Test shapes:", x_test.shape, y_test.shape)
print("Test - first ten labels:", y_test[:10])  # The first of the randomly selected l
```

```
Train shapes: (845, 25000) (845,)
Test shapes: (237, 25000) (237,)
Test - first ten labels: [0 0 0 0 0 1 0 0 0 0]
```

Sequential Model

Sequential models are usually the first and easiest approach to try when implementing a neural network. A **sequential model** is simply a linear stack of layers, where each layer has 1 input tensor and 1 output tensor. Sequential models are typically the first option for text classification because language is inherently sequential, making sequential models well-suited for learning the underlying patterns in text data.

Hidden layers in these models are usually Dense (densely connected layers), and each layer can have its own **activation function**. The following code builds a sequential model with only 2 Dense layers — since this is a small dataset, a large amount of hidden layers would risk overfitting the data. The first intermediate hidden layer uses the **relu** (rectified linear unit) activation function, which is usually the choice function for intermediate layers. This layer has 32 nodes, which is fine because each intermediate layer should more units (nodes) than the next layer (which in this case is the output layer with 1 node).

The activation function of the second layer is the **sigmoid** activation function, which is usually used for the output layer of binary classification tasks because it outputs a probability between 0 and 1. Since our text classification task is binary between Spam/Non-Spam, the sigmoid activation function is most suitable for the output layer. Finally, the model makes 10 forward and backward passes through the network, or **epochs**.

In [7]:

```
# Build the Sequential Model
model = models.Sequential()
model.add(layers.Dense(32, input_dim=vocab_size, kernel_initializer='normal', activati
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, batch_size=batch_size, epochs=10, verbose=1, val
```

```
Epoch 1/10
8/8 [==============================] - 6s 66ms/step - loss: 0.6390 - accuracy: 0.7776
- val_loss: 0.6859 - val_accuracy: 0.5412
Epoch 2/10
8/8 [==============================] - 0s 24ms/step - loss: 0.5445 - accuracy: 0.9039
- val_loss: 0.6827 - val_accuracy: 0.5412
Epoch 3/10
8/8 [==============================] - 0s 21ms/step - loss: 0.4546 - accuracy: 0.9237
- val_loss: 0.6865 - val_accuracy: 0.5529
Epoch 4/10
8/8 [==============================] - 0s 27ms/step - loss: 0.3676 - accuracy: 0.9395
- val_loss: 0.7000 - val_accuracy: 0.6118
Epoch 5/10
8/8 [==============================] - 0s 26ms/step - loss: 0.2913 - accuracy: 0.9605
- val_loss: 0.7200 - val_accuracy: 0.6824
Epoch 6/10
8/8 [==============================] - 0s 23ms/step - loss: 0.2275 - accuracy: 0.9789
- val_loss: 0.7355 - val_accuracy: 0.6824
Epoch 7/10
8/8 [==============================] - 0s 25ms/step - loss: 0.1756 - accuracy: 0.9882
- val_loss: 0.7511 - val_accuracy: 0.6941
Epoch 8/10
8/8 [==============================] - 0s 29ms/step - loss: 0.1356 - accuracy: 0.9934
- val_loss: 0.7633 - val_accuracy: 0.7059
Epoch 9/10
8/8 [==============================] - 0s 26ms/step - loss: 0.1045 - accuracy: 0.9974
- val_loss: 0.7814 - val_accuracy: 0.7059
Epoch 10/10
8/8 [==============================] - 0s 30ms/step - loss: 0.0814 - accuracy: 0.9987
- val_loss: 0.7994 - val_accuracy: 0.7059
```

In [8]:

```
# Evaluate sequential model
score = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
print(score)
```

```
3/3 [==============================] - 0s 11ms/step - loss: 0.2174 - accuracy: 0.9325
Accuracy:  0.9324894547462463
[0.21736228466033936, 0.9324894547462463]
```

In [9]:

```
# Get predictions and use them to calculate more metrics
pred = model.predict(x_test)
pred_labels = [1 if p>0.5 else 0 for p in pred]
```

```
8/8 [==============================] - 0s 5ms/step
```

In [10]:

```
from sklearn.metrics import classification_report, accuracy_score, precision_score, re

# Evaluation metrics for the sequential model
print(classification_report(y_test, pred_labels))
print('_____________________________________________________')
print('_____________________________________________________')
print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred_labels, pos_label=0))
print('Recall score: ', recall_score(y_test, pred_labels, pos_label=0))
print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred_labels))
print('Recall score: ', recall_score(y_test, pred_labels))
```

```
              precision    recall  f1-score   support

           0       0.92      1.00      0.96       192
           1       1.00      0.64      0.78        45

    accuracy                           0.93       237
   macro avg       0.96      0.82      0.87       237
weighted avg       0.94      0.93      0.93       237

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.9230769230769231
Recall score:  1.0

Precision and Recall for Spam (1):
Precision score:  1.0
Recall score:  0.6444444444444445
```

CNN Model

Convolutional neural networks (**CNNs** or convnets for short) learn patterns in small windows, like in 'tiles' of input as stated earlier. While sequential modeling required the input data to be preprocessed, CNNs confer the advantage of being able to automatically learn relevant features from raw input data. Although CNNs are usually used for **image processing** tasks, they can also be used to great effect on simple text classification tasks.

Keras provides 1D, 2D, and 3D Conv layers for CNNs, with the dimensionality of the layer corresponding to the dimensionality of the input data. When using CNNs for text classification, **1D layers** work best for 1D sequential data like text sequences. Likewise, 2D layers work best for 2D spatial data like images, and 3D layers work best for 3D volumetric data like videos.

The following code creates a CNN model that is a modification of the sequential model. This model is built as a sequential model 1D convnet.

In [11]:

```
# Step 1: Build CNN model
max_features = 10000
maxlen = 500  # Representing the first 500 words of each training sample
batch_size = 32

model = models.Sequential()
# The embedding layer
model.add(layers.Embedding(max_features, 128, input_length=maxlen))
# A Conv1D with MaxPooling
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
# Another Conv1D followed by GlobalMaxPooling, which returns the max value over the er
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```

In [12]:

```
# Step 2: Compile CNN model

model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),  # Set learni
```

```
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [13]:
```
# Step 3: Train CNN model
from keras import utils

# Pad the data to maxlen
train_data = utils.pad_sequences(x_train, maxlen=maxlen)
test_data = utils.pad_sequences(x_test, maxlen=maxlen)

history = model.fit(train_data, y_train, epochs=10, batch_size=128, validation_split=0
```

```
Epoch 1/10
6/6 [==============================] - 4s 122ms/step - loss: 2.0080 - accuracy: 0.869
8 - val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 2/10
6/6 [==============================] - 0s 18ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 3/10
6/6 [==============================] - 0s 21ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 4/10
6/6 [==============================] - 0s 18ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 5/10
6/6 [==============================] - 0s 18ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 6/10
6/6 [==============================] - 0s 19ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 7/10
6/6 [==============================] - 0s 15ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 8/10
6/6 [==============================] - 0s 18ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 9/10
6/6 [==============================] - 0s 18ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
Epoch 10/10
6/6 [==============================] - 0s 23ms/step - loss: 2.0080 - accuracy: 0.8698
- val_loss: 5.9327 - val_accuracy: 0.6154
```

In [14]:
```
# Step 4: Get predictions
pred = model.predict(test_data)
pred_labels = [1.0 if p>= 0.5 else 0.0 for p in pred]
```

```
8/8 [==============================] - 0s 12ms/step
```

In [15]:
```
# Step 5: Evaluation metrics for the CNN model
print(classification_report(y_test, pred_labels))
print('_____________________________________________________')
print('_____________________________________________________')
print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred_labels, pos_label=0))
print('Recall score: ', recall_score(y_test, pred_labels, pos_label=0))
print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred_labels))
print('Recall score: ', recall_score(y_test, pred_labels))
```

```
              precision    recall  f1-score   support

           0       0.81      1.00      0.90       192
           1       0.00      0.00      0.00        45

    accuracy                           0.81       237
   macro avg       0.41      0.50      0.45       237
weighted avg       0.66      0.81      0.73       237

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.810126582278481
Recall score:  1.0

Precision and Recall for Spam (1):
Precision score:  0.0
Recall score:  0.0
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Different Embedding Approaches

Different word embedding approaches can be used to vectorize the dataset, with the goal being to make the vectors similar for words that occur in similar contexts. Words that tend to occur together likely have some kind of relation to each other, so their vectors tend to be similar. Word embeddings can be learned while training happens. The following code adds an embedding layer to the sequential model.

In [16]:

```python
# Set up the embedding layer in a sequential model
model = models.Sequential()
model.add(layers.Embedding(max_features, 8, input_length=maxlen))
model.add(layers.Flatten())
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(train_data, y_train, epochs=10, batch_size=32, validation_split=0.
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding_1 (Embedding)     (None, 500, 8)            80000     
                                                                 
 flatten (Flatten)           (None, 4000)              0         
                                                                 
 dense_3 (Dense)             (None, 16)                64016     
                                                                 
 dense_4 (Dense)             (None, 1)                 17        
                                                                 
=================================================================
Total params: 144,033
Trainable params: 144,033
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
22/22 [==============================] - 1s 13ms/step - loss: 0.4330 - acc: 0.8373 - 
val_loss: 0.7728 - val_acc: 0.6154
Epoch 2/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3932 - acc: 0.8698 - v
al_loss: 0.9297 - val_acc: 0.6154
Epoch 3/10
22/22 [==============================] - 0s 7ms/step - loss: 0.3929 - acc: 0.8698 - v
al_loss: 0.8090 - val_acc: 0.6154
Epoch 4/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3913 - acc: 0.8698 - v
al_loss: 1.0129 - val_acc: 0.6154
Epoch 5/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3932 - acc: 0.8698 - v
al_loss: 0.9018 - val_acc: 0.6154
Epoch 6/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3903 - acc: 0.8698 - v
al_loss: 0.7695 - val_acc: 0.6154
Epoch 7/10
22/22 [==============================] - 0s 7ms/step - loss: 0.3900 - acc: 0.8698 - v
al_loss: 1.0270 - val_acc: 0.6154
Epoch 8/10
22/22 [==============================] - 0s 5ms/step - loss: 0.3919 - acc: 0.8698 - v
al_loss: 0.7365 - val_acc: 0.6154
Epoch 9/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3934 - acc: 0.8698 - v
al_loss: 0.9592 - val_acc: 0.6154
Epoch 10/10
22/22 [==============================] - 0s 6ms/step - loss: 0.3904 - acc: 0.8698 - v
al_loss: 0.8657 - val_acc: 0.6154
```

In [17]:

```python
# Get predictions
pred = model.predict(test_data)
pred_labels = [1.0 if p>= 0.5 else 0.0 for p in pred]
```

```
8/8 [==============================] - 0s 2ms/step
```

In [18]:

```python
# Evaluation metrics for embedding layer model
print(classification_report(y_test, pred_labels))
print('_____________________________________________________')
print('_____________________________________________________')
print('\nPrecision and Recall for Non-Spam (0):')
print('Precision score: ', precision_score(y_test, pred_labels, pos_label=0))
```

```
print('Recall score: ', recall_score(y_test, pred_labels, pos_label=0))
print('\nPrecision and Recall for Spam (1):')
print('Precision score: ', precision_score(y_test, pred_labels))
print('Recall score: ', recall_score(y_test, pred_labels))
```

```
              precision    recall  f1-score   support

           0       0.81      1.00      0.90       192
           1       0.00      0.00      0.00        45

    accuracy                           0.81       237
   macro avg       0.41      0.50      0.45       237
weighted avg       0.66      0.81      0.73       237

_____________________________________________________
_____________________________________________________

Precision and Recall for Non-Spam (0):
Precision score:  0.810126582278481
Recall score:  1.0

Precision and Recall for Spam (1):
Precision score:  0.0
Recall score:  0.0
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in labe
ls with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undef
inedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Analysis of the Various Models

Of all the models, the sequential model performed the best, which is unusual. This is due to many constraining factors: the overall simplicity of the models, the smallness of the dataset, the low number of training epochs, etc. Results could likely be improved if the CNN and embedding layer models were had more complex layer architecture and used more epochs than 10 for training. As each model trained through each epoch, the output of the training shows the accuracy (indicated by 'accuracy' or 'acc') at each epoch. This metric, when compared across each epoch for each model, shows that the sequential model reached an accuracy of 0.9987, while the two latter models reached an accuracy of 0.8698. So overall, all three models performed relatively well given their constraints, although the sequential model had a nearly perfect performance.