Text Classification with Keras

By: Aditi Chaudhari

In [4]:
```
import tensorflow as tf
from tensorflow import keras
from keras.preprocessing.text import Tokenizer
from keras import layers, models
from keras.utils import pad_sequences
from keras.layers import Input, Dense, LSTM, Embedding
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.layers.experimental.preprocessing import TextVectorizatic
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_s
import numpy as np
from sklearn.model_selection import train_test_split
```

Reading in the Data

First, let's read in the data into a data frame. The data can be found at https://www.kaggle.com/datasets/infamouscoder/depression-reddit-cleaned. The first column of the data frame contains text taken from reddit and the second column of the data frame contains a number classifying the text as indicative of depression or not. The data set will be used for mental health classification. More specifically, it will be used to build a model that will predict whether someone is depressed or not based on what text they type.

In [1]:
```
import pandas as pd

df = pd.read_csv("depression_dataset_reddit_cleaned.csv")

df = df.sample(frac=1, random_state=1234)

df.head()
```

Out[1]:

	clean_text	is_depression
5520	i never thought that i could hate sombody but ...	0
6674	jennnnie yes and the next project s wool is hi...	0
38	i m starting to lose hope i feel like i m on a...	1
5765	watchin i m not there and missing heath ledger	0
4334	shandasaurus i see	0

Splitting the Data into Training and Testing Sets

Next, let's put 80% of the data into a training data set and the other 20% into a testing data set.

In [5]:
```
X = df.clean_text
y = df.is_depression

X_train, X_test, y_train, y_test = train_test_split(X,y,train_size=0.8,test_siz
```

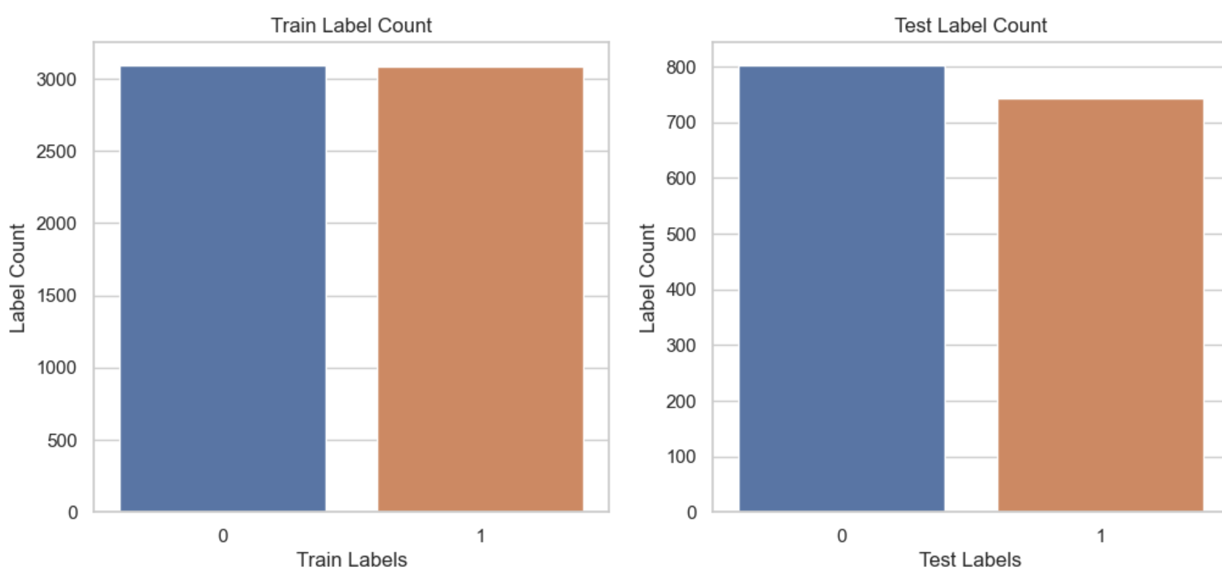
After that, let's visualize the distribution of the train/test split. As we can see, the data is mostly balanced.

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="whitegrid")

fig,ax = plt.subplots(1,2, figsize=(12,5))
for index, group in enumerate([('Train', y_train), ('Test', y_test)]):
    data = group[1].value_counts()
    sns.barplot(ax=ax[index],x=data.index, y=data.values)
    ax[index].set_title(f'{group[0]} Label Count')
    ax[index].set_xlabel(f'{group[0]} Labels')
    ax[index].set_ylabel('Label Count')

plt.show()
```



Building a Sequential Model

Firstly, we need to vectorize all the text in the data frame. We will do this with the Tokenizer() from Keras while using the mode tfidf. Next, we will use the LabelEncoder() from sklearn to encode the labels in the data.

In [21]:
```
num_labels = 2
vocab_size = 20000
batch_size = 100

tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(X_train)

X_train = tokenizer.texts_to_matrix(X_train, mode='tfidf')
```

```
X_test = tokenizer.texts_to_matrix(X_test, mode='tfidf')

encoder = LabelEncoder()
encoder.fit(y_train)

y_train = encoder.transform(y_train)
y_test = encoder.transform(y_test)

print("train shapes:", X_train.shape, y_train.shape)
print("test shapes:", X_test.shape, y_test.shape)
print("test first five labels:", y_test[:5])
```

```
train shapes: (6184, 20000) (6184,)
test shapes: (1547, 20000) (1547,)
test first five labels: [1 1 0 1 1]
```

Let's now build the Sequential model and then evaluate it on the data. After compiling the model, fitting the model on the training data, and then evaluating the model on the test data, we get an accuracy of 93%, a precision of 92%, a recall of 94%, and an F1 score of 93%.

In [ ]:

```
# building the model
seq_model = models.Sequential()
seq_model.add(layers.Dense(64, input_dim=vocab_size, kernel_initializer='normal
seq_model.add(layers.Dense(1,kernel_initializer='normal', activation='sigmoid')

# compiling the model
seq_model.compile(loss='binary_crossentropy', optimizer='adam',metrics=['accura

# fitting the model
seq_history = seq_model.fit(X_train, y_train, batch_size=batch_size,epochs=30,v
```

```
Epoch 1/30
56/56 [==============================] - 1s 8ms/step - loss: 0.4688 - accurac
y: 0.7481 - val_loss: 0.3453 - val_accuracy: 0.9192
Epoch 2/30
56/56 [==============================] - 0s 6ms/step - loss: 0.1946 - accurac
y: 0.9686 - val_loss: 0.1982 - val_accuracy: 0.9370
Epoch 3/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0761 - accurac
y: 0.9880 - val_loss: 0.1627 - val_accuracy: 0.9370
Epoch 4/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0392 - accurac
y: 0.9944 - val_loss: 0.1466 - val_accuracy: 0.9386
Epoch 5/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0240 - accurac
y: 0.9964 - val_loss: 0.1417 - val_accuracy: 0.9418
Epoch 6/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0166 - accurac
y: 0.9973 - val_loss: 0.1427 - val_accuracy: 0.9402
Epoch 7/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0123 - accurac
y: 0.9975 - val_loss: 0.1419 - val_accuracy: 0.9435
Epoch 8/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0096 - accurac
y: 0.9986 - val_loss: 0.1453 - val_accuracy: 0.9435
Epoch 9/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0077 - accurac
y: 0.9987 - val_loss: 0.1486 - val_accuracy: 0.9435
Epoch 10/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0063 - accurac
y: 0.9991 - val_loss: 0.1526 - val_accuracy: 0.9451
Epoch 11/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0053 - accurac
y: 0.9993 - val_loss: 0.1547 - val_accuracy: 0.9451
Epoch 12/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0045 - accurac
y: 0.9995 - val_loss: 0.1593 - val_accuracy: 0.9451
Epoch 13/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0038 - accurac
y: 0.9995 - val_loss: 0.1640 - val_accuracy: 0.9451
Epoch 14/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0033 - accurac
y: 0.9995 - val_loss: 0.1683 - val_accuracy: 0.9418
Epoch 15/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0029 - accurac
y: 0.9996 - val_loss: 0.1723 - val_accuracy: 0.9370
Epoch 16/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0025 - accurac
y: 0.9998 - val_loss: 0.1769 - val_accuracy: 0.9386
Epoch 17/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0022 - accurac
y: 0.9998 - val_loss: 0.1815 - val_accuracy: 0.9370
Epoch 18/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0020 - accurac
y: 1.0000 - val_loss: 0.1849 - val_accuracy: 0.9354
Epoch 19/30
56/56 [==============================] - 0s 7ms/step - loss: 0.0018 - accurac
y: 1.0000 - val_loss: 0.1902 - val_accuracy: 0.9338
Epoch 20/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0016 - accurac
y: 1.0000 - val_loss: 0.1940 - val_accuracy: 0.9321
```

```
Epoch 21/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0014 - accurac
y: 1.0000 - val_loss: 0.1983 - val_accuracy: 0.9305
Epoch 22/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0013 - accurac
y: 1.0000 - val_loss: 0.2021 - val_accuracy: 0.9289
Epoch 23/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0012 - accurac
y: 1.0000 - val_loss: 0.2056 - val_accuracy: 0.9305
Epoch 24/30
56/56 [==============================] - 0s 6ms/step - loss: 0.0011 - accurac
y: 1.0000 - val_loss: 0.2105 - val_accuracy: 0.9289
Epoch 25/30
56/56 [==============================] - 0s 6ms/step - loss: 9.8273e-04 - accu
racy: 1.0000 - val_loss: 0.2137 - val_accuracy: 0.9289
Epoch 26/30
56/56 [==============================] - 0s 6ms/step - loss: 8.9911e-04 - accu
racy: 1.0000 - val_loss: 0.2183 - val_accuracy: 0.9289
Epoch 27/30
56/56 [==============================] - 0s 6ms/step - loss: 8.2723e-04 - accu
racy: 1.0000 - val_loss: 0.2219 - val_accuracy: 0.9321
Epoch 28/30
56/56 [==============================] - 0s 6ms/step - loss: 7.6621e-04 - accu
racy: 1.0000 - val_loss: 0.2259 - val_accuracy: 0.9321
Epoch 29/30
56/56 [==============================] - 0s 6ms/step - loss: 7.0674e-04 - accu
racy: 1.0000 - val_loss: 0.2291 - val_accuracy: 0.9305
Epoch 30/30
56/56 [==============================] - 0s 6ms/step - loss: 6.5892e-04 - accu
racy: 1.0000 - val_loss: 0.2328 - val_accuracy: 0.9305
```

In [ ]:

```
# evaluating the model on the test data
seq_score = seq_model.evaluate(X_test, y_test, batch_size=batch_size, verbose=1
print("accuracy of sequential model: ", seq_score[1])
```

```
16/16 [==============================] - 0s 3ms/step - loss: 0.2652 - accurac
y: 0.9347
accuracy of sequential model:  0.9347123503684998
```

In [ ]:

```
# getting more metrics
seq_pred = seq_model.predict(X_test)
seq_pred_labels = [1 if p > 0.5 else 0 for p in seq_pred]

print('accuracy score: ', accuracy_score(y_test, seq_pred_labels))
print('precision score: ', precision_score(y_test, seq_pred_labels))
print('recall score: ', recall_score(y_test, seq_pred_labels))
print('f1 score: ', f1_score(y_test, seq_pred_labels))
```

```
49/49 [==============================] - 0s 1ms/step
accuracy score:  0.9347123464770524
precision score:  0.9224704336399474
recall score:  0.9435483870967742
f1 score:  0.9328903654485049
```

Building a RNN

Next, let's build a RNN and then evaluate it on the data. After compiling the model, fitting the model on the training data, and then evaluating the model on the test data, we get an accuracy of 96%, a precision of 95%, a recall of 97%, and an F1 score of 96%.

In [6]:
```
# setup input pipeline
BUFFER_SIZE = 10000
BATCH_SIZE = 64

train, test = train_test_split(df, train_size=0.8, random_state=1234)


train = tf.data.Dataset.from_tensor_slices((train['clean_text'].values, train['
test = tf.data.Dataset.from_tensor_slices((test['clean_text'].values, test['is_

train_dataset = train.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.A
test_dataset = test.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

# set up vectorizer
VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))

vocab = np.array(encoder.get_vocabulary())
```

Next, we can build the model.

In [7]:
```
rnn_model = models.Sequential([
    encoder,
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# compiling the model
rnn_model.compile(loss='binary_crossentropy', optimizer='adam',metrics=['accura

# fitting the model
rnn_history = rnn_model.fit(X_train, y_train, batch_size=BATCH_SIZE,epochs=10,v
```

```
Epoch 1/10
87/87 [==============================] - 37s 257ms/step - loss: 0.3381 - accur
acy: 0.8571 - val_loss: 0.1507 - val_accuracy: 0.9483
Epoch 2/10
87/87 [==============================] - 13s 153ms/step - loss: 0.0896 - accur
acy: 0.9700 - val_loss: 0.0907 - val_accuracy: 0.9693
Epoch 3/10
87/87 [==============================] - 10s 119ms/step - loss: 0.0466 - accur
acy: 0.9862 - val_loss: 0.0913 - val_accuracy: 0.9725
Epoch 4/10
87/87 [==============================] - 10s 108ms/step - loss: 0.0427 - accur
acy: 0.9851 - val_loss: 0.1355 - val_accuracy: 0.9645
Epoch 5/10
87/87 [==============================] - 8s 94ms/step - loss: 0.0292 - accurac
y: 0.9916 - val_loss: 0.0957 - val_accuracy: 0.9742
Epoch 6/10
87/87 [==============================] - 7s 80ms/step - loss: 0.0193 - accurac
y: 0.9944 - val_loss: 0.1589 - val_accuracy: 0.9645
Epoch 7/10
87/87 [==============================] - 6s 69ms/step - loss: 0.0391 - accurac
y: 0.9872 - val_loss: 0.1085 - val_accuracy: 0.9742
Epoch 8/10
87/87 [==============================] - 6s 75ms/step - loss: 0.0205 - accurac
y: 0.9941 - val_loss: 0.1337 - val_accuracy: 0.9580
Epoch 9/10
87/87 [==============================] - 5s 54ms/step - loss: 0.0146 - accurac
y: 0.9959 - val_loss: 0.1271 - val_accuracy: 0.9677
Epoch 10/10
87/87 [==============================] - 5s 57ms/step - loss: 0.0106 - accurac
y: 0.9971 - val_loss: 0.1609 - val_accuracy: 0.9628
```

We then can use the model to evaluate it on the test data and get metrics about the model.

In [9]:
```
# evaluating the model on the test data
rnn_score = rnn_model.evaluate(X_test, y_test, batch_size=BATCH_SIZE, verbose=1
print("accuracy of rnn model: ", rnn_score[1])
```

```
25/25 [==============================] - 1s 45ms/step - loss: 0.1882 - accurac
y: 0.9612
accuracy of rnn model:  0.9612152576446533
```

In [11]:
```
# getting more metrics
rnn_pred = rnn_model.predict(X_test)
rnn_pred_labels = [1 if p > 0.5 else 0 for p in rnn_pred]

print('accuracy score: ', accuracy_score(y_test, rnn_pred_labels))
print('precision score: ', precision_score(y_test, rnn_pred_labels))
print('recall score: ', recall_score(y_test, rnn_pred_labels))
print('f1 score: ', f1_score(y_test, rnn_pred_labels))
```

```
49/49 [==============================] - 2s 34ms/step
accuracy score:  0.9612152553329024
precision score:  0.9535809018567639
recall score:  0.9663978494623656
f1 score:  0.9599465954606141
```

Trying Different Embeddings

Next, let's try different embedding approaches. First, let's add a simple embedding layer.

In [7]:
```
# splits the data into an 70/20/10 train/test/validation set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, randon

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=(
```

First, we need to set up the vectorizer.

In [ ]:
```
# sets up the vectorizer
vectorizer = TextVectorization(max_tokens=20000,output_sequence_length=200)
text_ds = tf.data.Dataset.from_tensor_slices(X_train).batch(128)
vectorizer.adapt(text_ds)

# creates a dictionary in which words map to indices
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))
```

```
2023-04-22 13:25:44.461146: I tensorflow/core/platform/cpu_feature_guard.cc:19
3] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical operati
ons:  SSE4.1 SSE4.2
To enable them in other operations, rebuild TensorFlow with the appropriate co
mpiler flags.
```

Next, we need to vectorize the training data and validation data.

In [ ]:
```
# vectorizes the text
x_train = vectorizer(np.array([[s] for s in X_train])).numpy()
x_val = vectorizer(np.array([[s] for s in X_val])).numpy()

# makes the labels into np arrays
y_train = np.array(y_train)
y_val = np.array(y_val)
```

Next, we can set up the embedding layer and use it to build the model.

In [ ]:
```
from keras import Model

# sets up the embedding layer
EMBEDDING_DIM = 128
MAX_SEQUENCE_LENGTH = 200

embedding_layer = layers.Embedding(len(word_index) + 1, EMBEDDING_DIM,input_ler

# builds the model with the embedding layer
int_sequences_input = Input(shape=(None,), dtype="int64")
embedded_sequences = embedding_layer(int_sequences_input)
x = layers.Conv1D(128, 5, activation="relu")(embedded_sequences)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
preds = layers.Dense(1, activation="sigmoid")(x)
embedding_model = Model(int_sequences_input, preds)
```

```
# compiles the model
embedding_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=[

# fits the model on the training data
embedding_model.fit(x_train, y_train, batch_size=128, epochs=20, validation_dat
```

```
Epoch 1/20
43/43 [==============================] - 5s 111ms/step - loss: 0.3862 - accura
cy: 0.8305 - val_loss: 0.2940 - val_accuracy: 0.9004
Epoch 2/20
43/43 [==============================] - 5s 117ms/step - loss: 0.1582 - accura
cy: 0.9481 - val_loss: 0.1087 - val_accuracy: 0.9702
Epoch 3/20
43/43 [==============================] - 5s 117ms/step - loss: 0.0704 - accura
cy: 0.9804 - val_loss: 0.1086 - val_accuracy: 0.9677
Epoch 4/20
43/43 [==============================] - 5s 119ms/step - loss: 0.0450 - accura
cy: 0.9904 - val_loss: 0.1031 - val_accuracy: 0.9677
Epoch 5/20
43/43 [==============================] - 5s 123ms/step - loss: 0.0354 - accura
cy: 0.9932 - val_loss: 0.1120 - val_accuracy: 0.9664
Epoch 6/20
43/43 [==============================] - 5s 124ms/step - loss: 0.0281 - accura
cy: 0.9952 - val_loss: 0.1205 - val_accuracy: 0.9677
Epoch 7/20
43/43 [==============================] - 5s 125ms/step - loss: 0.0254 - accura
cy: 0.9957 - val_loss: 0.1130 - val_accuracy: 0.9690
Epoch 8/20
43/43 [==============================] - 5s 127ms/step - loss: 0.0206 - accura
cy: 0.9961 - val_loss: 0.1194 - val_accuracy: 0.9690
Epoch 9/20
43/43 [==============================] - 5s 123ms/step - loss: 0.0191 - accura
cy: 0.9961 - val_loss: 0.1317 - val_accuracy: 0.9728
Epoch 10/20
43/43 [==============================] - 5s 126ms/step - loss: 0.0150 - accura
cy: 0.9961 - val_loss: 0.1338 - val_accuracy: 0.9715
Epoch 11/20
43/43 [==============================] - 5s 120ms/step - loss: 0.0101 - accura
cy: 0.9961 - val_loss: 0.1423 - val_accuracy: 0.9728
Epoch 12/20
43/43 [==============================] - 5s 124ms/step - loss: 0.0061 - accura
cy: 0.9961 - val_loss: 0.1607 - val_accuracy: 0.9741
Epoch 13/20
43/43 [==============================] - 5s 125ms/step - loss: 0.0094 - accura
cy: 0.9943 - val_loss: 0.2021 - val_accuracy: 0.9728
Epoch 14/20
43/43 [==============================] - 6s 130ms/step - loss: 0.0153 - accura
cy: 0.9922 - val_loss: 0.2113 - val_accuracy: 0.9638
Epoch 15/20
43/43 [==============================] - 6s 132ms/step - loss: 0.0126 - accura
cy: 0.9937 - val_loss: 0.2047 - val_accuracy: 0.9677
Epoch 16/20
43/43 [==============================] - 6s 136ms/step - loss: 0.0109 - accura
cy: 0.9954 - val_loss: 0.1846 - val_accuracy: 0.9664
Epoch 17/20
43/43 [==============================] - 6s 140ms/step - loss: 0.0059 - accura
cy: 0.9980 - val_loss: 0.2336 - val_accuracy: 0.9573
Epoch 18/20
43/43 [==============================] - 6s 147ms/step - loss: 0.0058 - accura
cy: 0.9983 - val_loss: 0.1794 - val_accuracy: 0.9690
Epoch 19/20
43/43 [==============================] - 7s 159ms/step - loss: 0.0024 - accura
cy: 1.0000 - val_loss: 0.2057 - val_accuracy: 0.9677
Epoch 20/20
43/43 [==============================] - 7s 167ms/step - loss: 0.0011 - accura
cy: 1.0000 - val_loss: 0.2228 - val_accuracy: 0.9651
```

Out[ ]: <keras.callbacks.History at 0x7fb773195dc0>

In [ ]:
```
# vectorizes the test data
test_x = vectorizer(np.array([[s] for s in X_test])).numpy()

# uses the model to predict on the test data
em_pred = embedding_model.predict(test_x)
em_pred_labels = [1 if p > 0.5 else 0 for p in em_pred]

# gives metrics on the model
print('accuracy score: ', accuracy_score(y_test, em_pred_labels))
print('precision score: ', precision_score(y_test, em_pred_labels))
print('recall score: ', recall_score(y_test, em_pred_labels))
print('f1 score: ', f1_score(y_test, em_pred_labels))
```

```
49/49 [==============================] - 0s 9ms/step
accuracy score:  0.9521654815772462
precision score:  0.9608585858585859
recall score:  0.9465174129353234
f1 score:  0.9536340852130326
```

Next, let's use GloVe pretrained embedding. The GloVe pretrained embeddings were downloaded from http://nlp.stanford.edu/data/glove.6B.zip. Then, these embeddings are used to create an embeddings indexed dictionary that maps words to the GloVe embeddings.

In [ ]:
```
import os

path_to_glove_file = os.path.join(
    os.path.expanduser("~"), ".keras/datasets/glove.6B/glove.6B.100d.txt"
)

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))
```

```
Found 400000 word vectors.
```

Next, we will create an embedding matrix, which replaces the original token with a GloVe embedding.

In [ ]:
```
num_tokens = len(voc) + 2
embedding_dim = 100
hits = 0
misses = 0

# prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words rhat are not found in the embedding index will be 0
```

```
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 12103 words (3241 misses)
```

The embedding layer needs to be set to false so that the embeddings are not changed while the model is being trained.

In [ ]:
```
embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
)
```

Next, we build the model.

In [ ]:
```
# builds model with embedding layer
int_sequences_input = keras.Input(shape=(None,), dtype="int64")
embedded_sequences = embedding_layer(int_sequences_input)
x = layers.Conv1D(128, 5, activation="relu")(embedded_sequences)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
preds = layers.Dense(1, activation="sigmoid")(x)
glove_model = keras.Model(int_sequences_input, preds)

# compiles model
glove_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["acc

# fits model with training data
glove_model.fit(x_train, y_train, batch_size=128, epochs=20, validation_data=(x
```

```
Epoch 1/20
43/43 [==============================] - 5s 102ms/step - loss: 0.3236 - accura
cy: 0.8675 - val_loss: 0.1815 - val_accuracy: 0.9495
Epoch 2/20
43/43 [==============================] - 3s 68ms/step - loss: 0.1337 - accurac
y: 0.9562 - val_loss: 0.1041 - val_accuracy: 0.9690
Epoch 3/20
43/43 [==============================] - 3s 68ms/step - loss: 0.0852 - accurac
y: 0.9734 - val_loss: 0.1153 - val_accuracy: 0.9612
Epoch 4/20
43/43 [==============================] - 3s 69ms/step - loss: 0.0627 - accurac
y: 0.9782 - val_loss: 0.1062 - val_accuracy: 0.9638
Epoch 5/20
43/43 [==============================] - 3s 69ms/step - loss: 0.0379 - accurac
y: 0.9863 - val_loss: 0.0967 - val_accuracy: 0.9754
Epoch 6/20
43/43 [==============================] - 3s 69ms/step - loss: 0.0204 - accurac
y: 0.9933 - val_loss: 0.1165 - val_accuracy: 0.9625
Epoch 7/20
43/43 [==============================] - 3s 71ms/step - loss: 0.0149 - accurac
y: 0.9946 - val_loss: 0.1132 - val_accuracy: 0.9754
Epoch 8/20
43/43 [==============================] - 3s 70ms/step - loss: 0.0085 - accurac
y: 0.9969 - val_loss: 0.1353 - val_accuracy: 0.9767
Epoch 9/20
43/43 [==============================] - 3s 71ms/step - loss: 0.0039 - accurac
y: 0.9993 - val_loss: 0.1384 - val_accuracy: 0.9728
Epoch 10/20
43/43 [==============================] - 3s 71ms/step - loss: 0.0019 - accurac
y: 0.9996 - val_loss: 0.1498 - val_accuracy: 0.9728
Epoch 11/20
43/43 [==============================] - 3s 71ms/step - loss: 0.0012 - accurac
y: 0.9998 - val_loss: 0.1652 - val_accuracy: 0.9741
Epoch 12/20
43/43 [==============================] - 3s 72ms/step - loss: 9.6580e-04 - acc
uracy: 0.9998 - val_loss: 0.1833 - val_accuracy: 0.9767
Epoch 13/20
43/43 [==============================] - 3s 71ms/step - loss: 9.1450e-04 - acc
uracy: 0.9998 - val_loss: 0.1867 - val_accuracy: 0.9741
Epoch 14/20
43/43 [==============================] - 3s 70ms/step - loss: 9.3832e-04 - acc
uracy: 0.9998 - val_loss: 0.1953 - val_accuracy: 0.9741
Epoch 15/20
43/43 [==============================] - 3s 71ms/step - loss: 7.4905e-04 - acc
uracy: 0.9998 - val_loss: 0.1917 - val_accuracy: 0.9754
Epoch 16/20
43/43 [==============================] - 3s 72ms/step - loss: 7.8369e-04 - acc
uracy: 0.9998 - val_loss: 0.1899 - val_accuracy: 0.9728
Epoch 17/20
43/43 [==============================] - 3s 74ms/step - loss: 7.7855e-04 - acc
uracy: 0.9998 - val_loss: 0.1945 - val_accuracy: 0.9741
Epoch 18/20
43/43 [==============================] - 3s 72ms/step - loss: 8.5738e-04 - acc
uracy: 0.9998 - val_loss: 0.2046 - val_accuracy: 0.9741
Epoch 19/20
43/43 [==============================] - 3s 73ms/step - loss: 8.1978e-04 - acc
uracy: 0.9998 - val_loss: 0.1868 - val_accuracy: 0.9754
Epoch 20/20
43/43 [==============================] - 3s 73ms/step - loss: 6.8993e-04 - acc
uracy: 0.9998 - val_loss: 0.1905 - val_accuracy: 0.9741
```

Out[ ]: <keras.callbacks.History at 0x7f7aec1c88e0>

In [ ]:
```
# vectorizes test data
test_x = vectorizer(np.array([[s] for s in X_test])).numpy()

# uses the model to make predictions
gl_pred = glove_model.predict(test_x)
gl_pred_labels = [1 if p > 0.5 else 0 for p in gl_pred]

# prints model's metrics
print('accuracy score: ', accuracy_score(y_test, gl_pred_labels))
print('precision score: ', precision_score(y_test, gl_pred_labels))
print('recall score: ', recall_score(y_test, gl_pred_labels))
print('f1 score: ', f1_score(y_test, gl_pred_labels))
```

```
49/49 [==============================] - 0s 8ms/step
accuracy score:  0.9618616677440207
precision score:  0.981888745148771
recall score:  0.9440298507462687
f1 score:  0.962587190868738
```

Analysis

In this Python notebook, I built different deep learning models to classify whether someone is depressed or not based off of a reddit post that they made. The deep learning models that I explored were a basic Sequential Model, a Recurrent Neural Network (RNN), and models with different embedding layers.

After building the Sequential model and the RNN, I found that the RNN performed better. Prior to building the RNN, I did some research and discovered that RNNs are more commonly used for text classification. My RNN had an accuracy of 96%, a precision of 95%, a recall of 97%, and an F1 score of 96%. My Sequential model had an accuracy of 93%, a precision of 92%, a recall of 94%, and an F1 score of 93%. In all metrics, the RNN performed better. The one downside that I found to using a RNN, though, is the amount of time it took to train the model. Originally, I was working on training the model on my local machine, but it was taking nearly two hours to train. I moved over to Google Colab and it trained a lot quicker using GPU acceleration. However, I did not have to use Google Colab for my Sequential model or any of the models with different embedding layers. So, training a RNN is a lot more time consuming than just training a Sequential model.

In exploring various embedding approaches, I learned that a GloVe pretrained embedding performed a little better than a simple embedding layer. The model that used a simple embedding layer had an accuracy score of 95%, a precision score of 96%, a recall score of 95%, and a F1 score of 95%. The model that used a GloVe pretrained embedding had an accuracy score of 96%, a precision score of 98%, a recall score of 94%, and a F1 score of 96%. I am a little surprised that both models were so close, since I expected the model with a GloVe pretrained embedding to perform much better. The model that used a GloVe pretrained embedding perfomed similarly to the RNN, which I found interesting. If I had to choose just 1 model to use for my data, I would use the model with the GloVe pretrained embedding since it performed as well as the RNN but did not take nearly as much time to train.