```
from google.colab import files
uploaded = files.upload()
```

Choose Files data_rt.csv

- **data_rt.csv**(text/csv) - 1284122 bytes, last modified: 4/22/2023 - 100% done

```
Saving data_rt.csv to data_rt.csv
```

```
import io
import pandas as pd

df = pd.read_csv(io.BytesIO(uploaded['data_rt.csv']))
```

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models, preprocessing
```

```
(train_data, train_labels), (test_data, test_labels) = datasets.imdb.load_data(num_words=10000)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 [==============================] - 0s 0us/step
```

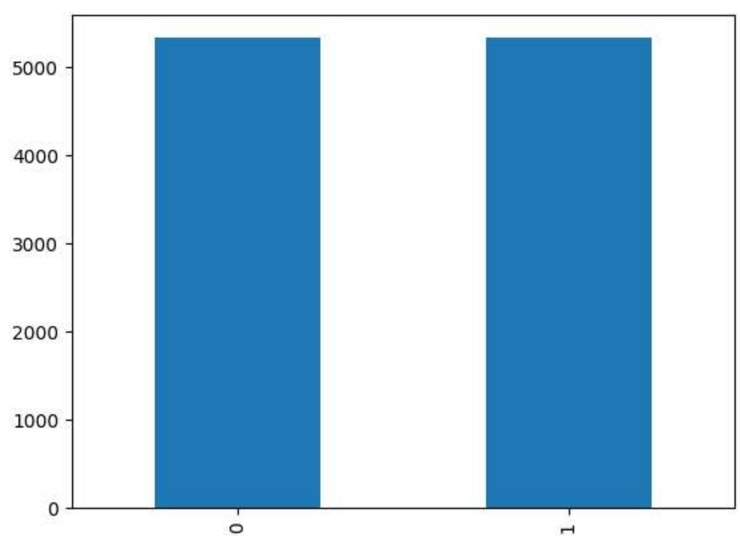
```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

```
import matplotlib.pyplot as plt
# Use the line below to show inline in a notebook
%matplotlib inline
```

```
fig, ax = plt.subplots()
df['labels'].value_counts().plot(ax=ax, kind='bar')
```

```
<Axes: >
```



This data set consists of movie reviews from Rotten Tomatoes and whether they are positive or negative. There is an equal number of positive and negative reviews and they are organized by this attribute. This is why we shuffled the data set before using it. This model will be able to predict whether a movie review is positive or negative.

Sequential Model

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

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

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    min_delta=0.001, # min change in metrics
    patience=4,     # min epochs
    restore_best_weights=True,
)
```

```
model.fit(x_train, y_train, callbacks=[early_stopping], batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
49/49 [==============================] - ETA: 0s - loss: 0.2057 - accuracy: 0.9454WARNING:tensorflow:Early stopping conditioned on metri
49/49 [==============================] - 1s 30ms/step - loss: 0.2057 - accuracy: 0.9454
782/782 [==============================] - 2s 2ms/step - loss: 0.4065 - accuracy: 0.8602
```

```
#Loss metric and accuracy
results
```

```
[0.4064764082431793, 0.8602399826049805]
```

```
from sklearn.metrics import classification_report
```

```
pred = model.predict(x_test)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
print(classification_report(y_test, pred))
```

```
782/782 [==============================] - 2s 3ms/step
              precision    recall  f1-score   support

         0.0       0.84      0.88      0.86     12500
         1.0       0.88      0.84      0.86     12500

    accuracy                           0.86     25000
   macro avg       0.86      0.86      0.86     25000
weighted avg       0.86      0.86      0.86     25000
```

```
#pos/neg distribution (set is 50/50)
pos = sum(y_test[y_test>0]) / len(y_test)
pos
```

```
0.5
```

The sequential model did better with this dataset than Naive Bayes, logistic regression, and neural networks, which we used in our last text classification exercise. The sequential model had an accuracy of 0.86, compared to Naive Baye's 0.76, logistic regression's 0.756, and neural network's 0.73.

RNN

```
maxlen = 500
batch_size = 32

# load the data
train_data = preprocessing.sequence.pad_sequences(train_data, maxlen=maxlen)
test_data = preprocessing.sequence.pad_sequences(test_data, maxlen=maxlen)
```

```
rnn_model = models.Sequential()
rnn_model.add(layers.Embedding(max_features, 32))
rnn_model.add(layers.SimpleRNN(32))
rnn_model.add(layers.Dense(1, activation='sigmoid'))
```

```
rnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 32)          320000

 simple_rnn_1 (SimpleRNN)    (None, 32)                2080

 dense_4 (Dense)             (None, 1)                 33

=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
_________________________________________________________________
```

```
rnn_model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
history = rnn_model.fit(train_data,
                    train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
Epoch 1/10
157/157 [==============================] - 32s 197ms/step - loss: 0.6844 - accuracy: 0.5505 - val_loss: 0.6601 - val_accuracy: 0.6088
Epoch 2/10
157/157 [==============================] - 32s 204ms/step - loss: 0.5073 - accuracy: 0.7621 - val_loss: 0.4873 - val_accuracy: 0.7656
Epoch 3/10
157/157 [==============================] - 30s 194ms/step - loss: 0.3567 - accuracy: 0.8473 - val_loss: 0.5419 - val_accuracy: 0.7242
Epoch 4/10
157/157 [==============================] - 30s 192ms/step - loss: 0.2687 - accuracy: 0.8903 - val_loss: 0.3801 - val_accuracy: 0.8394
Epoch 5/10
157/157 [==============================] - 30s 194ms/step - loss: 0.1967 - accuracy: 0.9259 - val_loss: 0.3807 - val_accuracy: 0.8566
Epoch 6/10
157/157 [==============================] - 35s 221ms/step - loss: 0.1322 - accuracy: 0.9531 - val_loss: 0.4428 - val_accuracy: 0.8444
Epoch 7/10
157/157 [==============================] - 43s 270ms/step - loss: 0.0945 - accuracy: 0.9692 - val_loss: 0.5862 - val_accuracy: 0.7662
Epoch 8/10
157/157 [==============================] - 29s 184ms/step - loss: 0.0686 - accuracy: 0.9784 - val_loss: 0.5401 - val_accuracy: 0.8292
Epoch 9/10
157/157 [==============================] - 29s 186ms/step - loss: 0.0415 - accuracy: 0.9871 - val_loss: 0.7299 - val_accuracy: 0.7606
Epoch 10/10
157/157 [==============================] - 30s 193ms/step - loss: 0.0244 - accuracy: 0.9938 - val_loss: 0.8209 - val_accuracy: 0.7402
```

```
pred = rnn_model.predict(test_data)
pred = [1.0 if p>= 0.5 else 0.0 for p in pred]
print(classification_report(test_labels, pred))
```

```
782/782 [==============================] - 24s 31ms/step
              precision    recall  f1-score   support

           0       0.74      0.76      0.75     12500
           1       0.75      0.74      0.74     12500
```

```
    accuracy                           0.75     25000
   macro avg       0.75      0.75      0.75     25000
weighted avg       0.75      0.75      0.75     25000
```

The RNN model performed worse than the previous sequential model by more than .10. RNN does not do well with long sequences like the dataset we are using which again consists of movie reviews.

Embeddings

```
emb_model = models.Sequential()
emb_model.add(layers.Embedding(max_features, 8, input_length=maxlen))
emb_model.add(layers.Flatten())
emb_model.add(layers.Dense(16, activation='relu'))
emb_model.add(layers.Dense(1, activation='sigmoid'))

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

history = emb_model.fit(train_data, train_labels, epochs=10, batch_size=32, validation_split=0.2)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 500, 8)            80000

 flatten (Flatten)           (None, 4000)              0

 dense_5 (Dense)             (None, 16)                64016

 dense_6 (Dense)             (None, 1)                 17

=================================================================
Total params: 144,033
Trainable params: 144,033
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
625/625 [==============================] - 5s 7ms/step - loss: 0.5054 - acc: 0.7286 - val_loss: 0.3119 - val_acc: 0.8696
Epoch 2/10
625/625 [==============================] - 4s 6ms/step - loss: 0.2515 - acc: 0.8998 - val_loss: 0.2964 - val_acc: 0.8756
Epoch 3/10
625/625 [==============================] - 5s 9ms/step - loss: 0.1859 - acc: 0.9296 - val_loss: 0.2921 - val_acc: 0.8836
Epoch 4/10
625/625 [==============================] - 7s 11ms/step - loss: 0.1397 - acc: 0.9481 - val_loss: 0.3161 - val_acc: 0.8846
Epoch 5/10
625/625 [==============================] - 6s 9ms/step - loss: 0.0974 - acc: 0.9668 - val_loss: 0.3663 - val_acc: 0.8794
Epoch 6/10
625/625 [==============================] - 4s 6ms/step - loss: 0.0628 - acc: 0.9809 - val_loss: 0.4250 - val_acc: 0.8716
Epoch 7/10
625/625 [==============================] - 4s 6ms/step - loss: 0.0375 - acc: 0.9900 - val_loss: 0.5062 - val_acc: 0.8632
Epoch 8/10
625/625 [==============================] - 5s 9ms/step - loss: 0.0211 - acc: 0.9939 - val_loss: 0.6123 - val_acc: 0.8570
Epoch 9/10
625/625 [==============================] - 4s 7ms/step - loss: 0.0110 - acc: 0.9972 - val_loss: 0.7327 - val_acc: 0.8532
Epoch 10/10
625/625 [==============================] - 4s 7ms/step - loss: 0.0053 - acc: 0.9983 - val_loss: 0.8766 - val_acc: 0.8502
```

The validation accuracy with this approach ended up quite high at 0.998. This is more than any of the previous approaches we tried on this dataset.

Final Thoughts

On our smaller dataset populated with text strings, our approach with embeddings performed best. The RNN approach did worst, on par with the approaches in our previous text classification exercise, and took the longest to train. Our first approach with the sequential model also had better precision and recall than the RNN approach.