MXS170018

Humor Data set text classification

In [1]:
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models, preprocessing
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.corpus import stopwords
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.preprocessing import LabelEncoder
np.random.seed(1234)
```

Humor Dataset

It contains a text comlumn and a humor classifier, if the text is a joke, it is classified as true.

This program should be able to take in text and output whether it is a joke or not

200k samples, 100k are humorous and 100k are not

In [2]:
```
df = pd.read_csv('dataset.csv', header=0, usecols=[0,1], encoding='latin-1')
print('rows and columns:', df.shape)
print(df.head())
```

```
rows and columns: (200000, 2)
                                                text  humor
0  Joe biden rules out 2020 bid: 'guys, i'm not r...  False
1  Watch: darvish gave hitter whiplash with slow ...  False
2  What do you call a turtle without its shell? d...   True
3      5 reasons the 2016 election feels so personal  False
4  Pasco police shot mexican migrant from behind,...  False
```

In [3]:
```
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)
print(train.head())
print(test.head())
```

```
train data size:  (160192, 2)
test data size:  (39808, 2)
                                                text  humor
0  Joe biden rules out 2020 bid: 'guys, i'm not r...  False
1  Watch: darvish gave hitter whiplash with slow ...  False
2  What do you call a turtle without its shell? d...   True
3      5 reasons the 2016 election feels so personal  False
4  Pasco police shot mexican migrant from behind,...  False
                                                text  humor
7   Why do native americans hate it when it rains ...   True
8   Obama's climate change legacy is impressive, i...  False
9       My family tree is a cactus, we're all pricks.   True
19  I just ended a 5 year relationship i'm fine, i...   True
24  Swimming toward a brighter future: how i was i...  False
```

One hot encoding - TFIDF

Tokenizing

Encoding

Here we use a tokenizer to reduce the number of words used. Then we convert it to one-hot encoding using tfidf to turn it into a matrix so the model can work with the text. It then encodes the classification y train and test sets.

In [36]:
```
# set up X and Y
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.text)

x_train = tokenizer.texts_to_matrix(train.text, mode='tfidf')
x_test = tokenizer.texts_to_matrix(test.text, mode='tfidf')

encoder = LabelEncoder()
encoder.fit(train.humor)
y_train = encoder.transform(train.humor)
y_test = encoder.transform(test.humor)
```

In [37]:
```
# check shape
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: (160192, 25000) (160192,)
test shapes: (39808, 25000) (39808,)
test first five labels: [1 0 1 1 0]
```

Standard sequential model

here we fit a sequential model using the Adam optimizer with the accuracy metric. We do this to get a baseline to compare our other models with.

In [113…
```
#fit 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=12,
                    verbose=1,
                    validation_split=0.5)
```

```
Epoch 1/12
801/801 [==============================] - 17s 21ms/step - loss: 0.2300 - accuracy:
0.9106 - val_loss: 0.1618 - val_accuracy: 0.9356
Epoch 2/12
801/801 [==============================] - 3s 4ms/step - loss: 0.0990 - accuracy: 0.9
631 - val_loss: 0.1678 - val_accuracy: 0.9358
Epoch 3/12
801/801 [==============================] - 3s 4ms/step - loss: 0.0556 - accuracy: 0.9
804 - val_loss: 0.1964 - val_accuracy: 0.9338
Epoch 4/12
801/801 [==============================] - 3s 4ms/step - loss: 0.0300 - accuracy: 0.9
903 - val_loss: 0.2351 - val_accuracy: 0.9311
Epoch 5/12
801/801 [==============================] - 3s 4ms/step - loss: 0.0151 - accuracy: 0.9
960 - val_loss: 0.2735 - val_accuracy: 0.9307
Epoch 6/12
801/801 [==============================] - 3s 3ms/step - loss: 0.0074 - accuracy: 0.9
984 - val_loss: 0.3066 - val_accuracy: 0.9298
Epoch 7/12
801/801 [==============================] - 3s 4ms/step - loss: 0.0035 - accuracy: 0.9
995 - val_loss: 0.3415 - val_accuracy: 0.9288
Epoch 8/12
801/801 [==============================] - 3s 3ms/step - loss: 0.0018 - accuracy: 0.9
998 - val_loss: 0.3711 - val_accuracy: 0.9279
Epoch 9/12
801/801 [==============================] - 3s 4ms/step - loss: 9.9328e-04 - accuracy:
0.9999 - val_loss: 0.4014 - val_accuracy: 0.9277
Epoch 10/12
801/801 [==============================] - 3s 4ms/step - loss: 5.6634e-04 - accuracy:
1.0000 - val_loss: 0.4296 - val_accuracy: 0.9274
Epoch 11/12
801/801 [==============================] - 3s 3ms/step - loss: 3.5442e-04 - accuracy:
1.0000 - val_loss: 0.4575 - val_accuracy: 0.9272
Epoch 12/12
801/801 [==============================] - 3s 3ms/step - loss: 2.2896e-04 - accuracy:
1.0000 - val_loss: 0.4862 - val_accuracy: 0.9268
```

In [115…
```
# evaluate
score = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
399/399 [==============================] - 1s 2ms/step - loss: 0.4958 - accuracy: 0.9
247
Accuracy:  0.9246633648872375
```

One hot encoding - Keras text vectorization

Encoding

Here we do what we did in the previous section, except here we use Keras's text vectorization instead of tfidf encoding

In [116…

```
VOCAB_SIZE = 750

maxlen = 500
batch_size = 32

from tensorflow import keras


encoder = LabelEncoder()
encoder.fit(train.humor)
y_train = encoder.transform(train.humor)
y_test = encoder.transform(test.humor)

encoder = keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)
encoder.adapt(train.text)

x_train = encoder(train.text)
x_test = encoder(test.text)

x_train = tf.keras.utils.to_categorical(x_train, num_classes = VOCAB_SIZE)
x_test = tf.keras.utils.to_categorical(x_test, num_classes = VOCAB_SIZE)

from keras.utils.np_utils import to_categorical
y_train = to_categorical(y_train, num_classes=None)
y_test = to_categorical(y_test, num_classes=None)
```

Keras Gated Recurrent Unit (GRU) with dense layers

In [117…

```
gru = keras.Sequential()
gru.add(layers.GRU(48, input_dim=VOCAB_SIZE))
gru.add(layers.Dense(32, activation='sigmoid'))
gru.add(layers.Dense(16, activation='sigmoid'))
gru.add(layers.Dense(8, activation='sigmoid'))
gru.add(layers.Dense(2, activation='softmax'))

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

history = gru.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=12,
                    verbose=1,
                    validation_split=0.5)
```

```
Epoch 1/12
2503/2503 [==============================] - 26s 10ms/step - loss: 0.2966 - accuracy:
0.8922 - val_loss: 0.2047 - val_accuracy: 0.9243
Epoch 2/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1910 - accuracy:
0.9270 - val_loss: 0.1818 - val_accuracy: 0.9299
Epoch 3/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1742 - accuracy:
0.9334 - val_loss: 0.1851 - val_accuracy: 0.9284
Epoch 4/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1629 - accuracy:
0.9378 - val_loss: 0.1720 - val_accuracy: 0.9338
Epoch 5/12
2503/2503 [==============================] - 26s 10ms/step - loss: 0.1524 - accuracy:
0.9417 - val_loss: 0.1703 - val_accuracy: 0.9341
Epoch 6/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1435 - accuracy:
0.9462 - val_loss: 0.1764 - val_accuracy: 0.9334
Epoch 7/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1355 - accuracy:
0.9494 - val_loss: 0.1758 - val_accuracy: 0.9317
Epoch 8/12
2503/2503 [==============================] - 25s 10ms/step - loss: 0.1263 - accuracy:
0.9533 - val_loss: 0.1770 - val_accuracy: 0.9342
Epoch 9/12
2503/2503 [==============================] - 24s 10ms/step - loss: 0.1189 - accuracy:
0.9559 - val_loss: 0.1915 - val_accuracy: 0.9338
Epoch 10/12
2503/2503 [==============================] - 26s 11ms/step - loss: 0.1111 - accuracy:
0.9592 - val_loss: 0.1890 - val_accuracy: 0.9333
Epoch 11/12
2503/2503 [==============================] - 26s 11ms/step - loss: 0.1024 - accuracy:
0.9635 - val_loss: 0.1966 - val_accuracy: 0.9295
Epoch 12/12
2503/2503 [==============================] - 26s 10ms/step - loss: 0.0942 - accuracy:
0.9665 - val_loss: 0.2047 - val_accuracy: 0.9295
```

In [118…

```
score = gru.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
1244/1244 [==============================] - 4s 4ms/step - loss: 0.2049 - accuracy:
0.9300
Accuracy:  0.9299889206886292
```

LSTM with multiple dense layers

In [119…

```
lstm = keras.Sequential()
lstm.add(layers.LSTM(48, input_dim=VOCAB_SIZE))
lstm.add(layers.Dense(32, activation='sigmoid'))
lstm.add(layers.Dense(16, activation='sigmoid'))
lstm.add(layers.Dense(8, activation='sigmoid'))
lstm.add(layers.Dense(2, activation='softmax'))

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

history = lstm.fit(x_train, y_train,
                   batch_size=batch_size,
```

```
                    epochs=12,
                    verbose=1,
                    validation_split=0.5)
```

```
Epoch 1/12
2503/2503 [==============================] - 23s 9ms/step - loss: 0.3271 - accuracy:
0.8651 - val_loss: 0.2001 - val_accuracy: 0.9239
Epoch 2/12
2503/2503 [==============================] - 21s 9ms/step - loss: 0.1846 - accuracy:
0.9312 - val_loss: 0.1825 - val_accuracy: 0.9307
Epoch 3/12
2503/2503 [==============================] - 22s 9ms/step - loss: 0.1660 - accuracy:
0.9359 - val_loss: 0.1770 - val_accuracy: 0.9327
Epoch 4/12
2503/2503 [==============================] - 22s 9ms/step - loss: 0.1547 - accuracy:
0.9408 - val_loss: 0.1771 - val_accuracy: 0.9305
Epoch 5/12
2503/2503 [==============================] - 21s 8ms/step - loss: 0.1442 - accuracy:
0.9451 - val_loss: 0.1754 - val_accuracy: 0.9326
Epoch 6/12
2503/2503 [==============================] - 21s 8ms/step - loss: 0.1344 - accuracy:
0.9488 - val_loss: 0.1818 - val_accuracy: 0.9318
Epoch 7/12
2503/2503 [==============================] - 22s 9ms/step - loss: 0.1263 - accuracy:
0.9531 - val_loss: 0.1884 - val_accuracy: 0.9286
Epoch 8/12
2503/2503 [==============================] - 22s 9ms/step - loss: 0.1183 - accuracy:
0.9559 - val_loss: 0.1926 - val_accuracy: 0.9305
Epoch 9/12
2503/2503 [==============================] - 21s 8ms/step - loss: 0.1105 - accuracy:
0.9597 - val_loss: 0.1939 - val_accuracy: 0.9295
Epoch 10/12
2503/2503 [==============================] - 21s 8ms/step - loss: 0.1023 - accuracy:
0.9627 - val_loss: 0.2055 - val_accuracy: 0.9319
Epoch 11/12
2503/2503 [==============================] - 21s 9ms/step - loss: 0.0946 - accuracy:
0.9660 - val_loss: 0.2121 - val_accuracy: 0.9305
Epoch 12/12
2503/2503 [==============================] - 21s 9ms/step - loss: 0.0880 - accuracy:
0.9689 - val_loss: 0.2209 - val_accuracy: 0.9257
```

In [120…

```
score = lstm.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
1244/1244 [==============================] - 4s 3ms/step - loss: 0.2241 - accuracy:
0.9241
Accuracy:  0.9240856170654297
```

RNN with some dense layers

In [121…

```
rnn = keras.Sequential()
rnn.add(layers.SimpleRNN(48))
rnn.add(layers.Dense(32, activation='sigmoid'))
rnn.add(layers.Dense(16, activation='sigmoid'))
rnn.add(layers.Dense(8, activation='sigmoid'))
rnn.add(layers.Dense(2, activation='softmax'))

rnn.compile(loss='binary_crossentropy',
              optimizer='adam',
```

```
              metrics=['accuracy'])

history = rnn.fit(x_train, y_train,
                  batch_size=batch_size,
                  epochs=12,
                  verbose=1,
                  validation_split=0.5)
```

```
Epoch 1/12
2503/2503 [==============================] - 17s 6ms/step - loss: 0.6789 - accuracy:
0.5451 - val_loss: 0.6698 - val_accuracy: 0.5497
Epoch 2/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.6638 - accuracy:
0.5601 - val_loss: 0.6625 - val_accuracy: 0.5663
Epoch 3/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.6628 - accuracy:
0.5639 - val_loss: 0.6621 - val_accuracy: 0.5674
Epoch 4/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.6441 - accuracy:
0.6031 - val_loss: 0.6292 - val_accuracy: 0.6140
Epoch 5/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.6194 - accuracy:
0.6253 - val_loss: 0.6161 - val_accuracy: 0.6292
Epoch 6/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.5385 - accuracy:
0.7356 - val_loss: 0.4781 - val_accuracy: 0.8153
Epoch 7/12
2503/2503 [==============================] - 16s 6ms/step - loss: 0.3098 - accuracy:
0.8794 - val_loss: 0.2313 - val_accuracy: 0.9135
Epoch 8/12
2503/2503 [==============================] - 16s 6ms/step - loss: 0.2673 - accuracy:
0.9006 - val_loss: 0.3222 - val_accuracy: 0.8831
Epoch 9/12
2503/2503 [==============================] - 16s 6ms/step - loss: 0.2610 - accuracy:
0.8989 - val_loss: 0.2494 - val_accuracy: 0.8902
Epoch 10/12
2503/2503 [==============================] - 16s 6ms/step - loss: 0.2139 - accuracy:
0.9203 - val_loss: 0.2219 - val_accuracy: 0.9133
Epoch 11/12
2503/2503 [==============================] - 15s 6ms/step - loss: 0.1987 - accuracy:
0.9246 - val_loss: 0.2120 - val_accuracy: 0.9156
Epoch 12/12
2503/2503 [==============================] - 16s 6ms/step - loss: 0.1930 - accuracy:
0.9274 - val_loss: 0.1922 - val_accuracy: 0.9267
```

In [122…

```
score = rnn.evaluate(x_test, y_test, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
1244/1244 [==============================] - 3s 3ms/step - loss: 0.1982 - accuracy:
0.9231
Accuracy:  0.9230808019638062
```

Sequential model embedding

GRU with embedding

LSTM with embedding

RNN with embedding

I tried multiple formats for getting embedding to work. I was getting so many shape errors, and I tried so many things that I am the third day of this assignment. I didn't find anything helpful online (it felt like gibberish to me), so I am going to stop here and do the analysis.

Analysis

Baseline Accuracy: 92.46% GRU Accuracy: 92.99% LSTM Accuracy: 92.41% RNN Accuracy: 92.30%

It is interesting that all 4 have the same 92.xx% accuracy. This might be because there are 200k samples. The more samples there are the more likely to find a better model (up to a certain extent). It seems that without further input processing this seems to be the average to shoot for. Regardless, GRU seemed to marginally be the best. It is worth venturing into the embedding world to see how it affects these models.