How I chose my dataset for this assignment

When looking through the datasets on Kaggle.com, I decided to pick a dataset that was different than the one I chose for Text Classification 1. For the first assignment I had picked an Imdb review dataset. However, I wanted to pick a dataset that we had not used in class, plus I wanted to work with some new data.

After looking through, I decided to pick the "Text Emotion Recognition" dataset because there was a lot of data (~283,000 entries), the dataset was relatively balanced, and the data already came preprocessed.

Dataset Description

This dataset holds conversational text from people. There are two columns:

- text - a person's dialog. stopwords are already removed and any slang is changed to what it actually means (ex. lol changed to Laughed out Loud)
- emotion - an emoji representation of the user's feelings when writing the text. 🙂 for a positive emotion, ☹ for a negative emotion

Given the data, I want to predict the user's emotional state based solely on their text.

For splitting data between test and train, I initially tried to test on all ~283,000 entries, but I kept crashing and running out of RAM. I eventually decided to only test on 110,000 entries just so I could get it to run without crashing. I also wanted to split the data with 10% being test and the rest being training, but I again struggled with my training set being too large. I eventually settled on having it be 20% test to reduce the size of my training set.

```
# set up import from local file
from google.colab import files
uploaded = files.upload()
```

```
Choose Files No file chosen Upload widget is only available when the cell has been executed in
browser session. Please rerun this cell to enable.
Saving glove.6B.100d.txt to glove.6B.100d.txt
```

```
import pandas as pd
import io
import numpy as np
import seaborn as sb

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
from tensorflow.keras import layers, models

from tensorflow import keras

from sklearn.preprocessing import LabelEncoder


# read in file
df = pd.read_csv('../content/Text_Emotion.csv', header=0)
print('rows and columns:', df.shape)
print(df.head())
print(df.dtypes)

# map the emotion emojis to numbers
df['emotion'] = df['emotion'].map({
    '☹' : 0,
    '🙂' : 1
})
print(df.head())
```

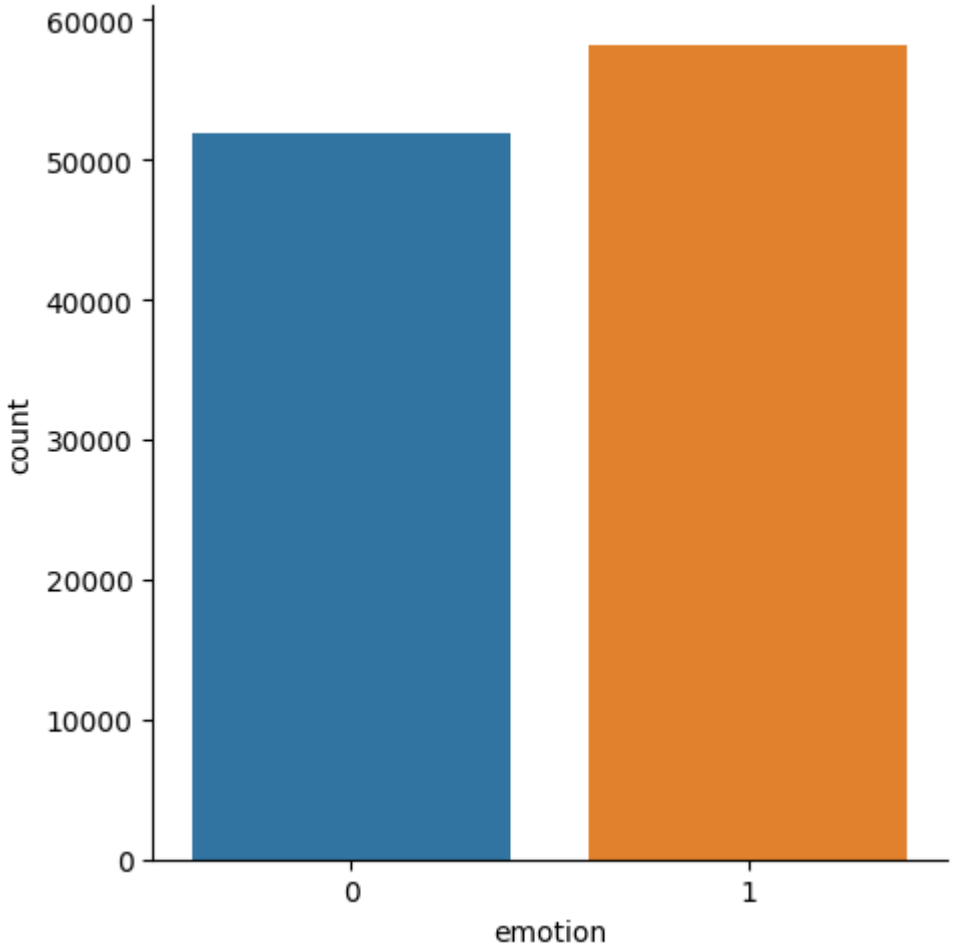
```
rows and columns: (282822, 2)
                                                text emotion
0  carefully word blog posts amount criticism hea...       ☹
1  cannot remember little mermaid feeling carefre...       🙂
2  not feeling super well turns cold knocked next...       🙂
3            feel honored part group amazing talents       🙂
4  think helping also began feel pretty lonely lo...       ☹
text       object
emotion    object
dtype: object
                                                text  emotion
0  carefully word blog posts amount criticism hea...        0
1  cannot remember little mermaid feeling carefre...        1
2  not feeling super well turns cold knocked next...        1
3            feel honored part group amazing talents        1
4  think helping also began feel pretty lonely lo...        0
```

```
# set seed for reproducibility
np.random.seed(1234)

# only perform on a portion of the original dataset
# I tried running on the entire dataset but it kept running out of RAM
df = df[:110000]

# split df into train and test
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:  (88028, 2)
test data size:  (21972, 2)
```

```
# plot the distribution of the datasets
sb.catplot(x="emotion", kind="count", data=df)

# for the size of the data, the data is relatively balanced
```

```
<seaborn.axisgrid.FacetGrid at 0x7fc87003ae20>
```



because my dataset is fairly large, I will keep the vocab size and batch size relatively small. On previous attempts I had tried to make them larger, but I struggled a lot with using too much RAM so I needed to make them smaller so that I wouldn't run out of RAM when it came to running the model.

```
# prepare data

# set up X and Y
num_labels = 2
# make vocab size and batch size small because it kept crashing on larger sizes
vocab_size = 10000
batch_size = 4
```

```
# 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.emotion)
y_train = encoder.transform(train.emotion)
y_test = encoder.transform(test.emotion)
# 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: (88028, 10000) (88028,)
test shapes: (21972, 10000) (21972,)
test first five labels: [0 1 0 1 0]
```

Sequential Model Attempt 1

I will add one internal layes to start since the data is so large. A sigmoid activation is used on the output layer since the results are binary. I will have the optimizer as adam for large datasets, and will calculate the binary_crossentropy between the true and predicted labels.

I will train over 3 epochs and see the results.

```
# fit model
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(10000,), kernel_initializer='normal', activation='rel
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))

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

# train the model
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=3,
                    verbose=1,
                    validation_split=0.1)
```

```
Epoch 1/3
19807/19807 [==============================] - 105s 5ms/step - loss: 0.1698 - accuracy:
Epoch 2/3
19807/19807 [==============================] - 114s 6ms/step - loss: 0.0978 - accuracy:
```

```
Epoch 3/3
19807/19807 [==============================] - 109s 6ms/step - loss: 0.0820 - accuracy:
```

```
# use sklearn evaluation

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))
```

```
687/687 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.95      0.95      0.95     10340
           1       0.95      0.96      0.96     11632

    accuracy                           0.95     21972
   macro avg       0.95      0.95      0.95     21972
weighted avg       0.95      0.95      0.95     21972
```

It seems like despite the fact that I could only run for 3 epochs, it did a pretty good job predicting the values in x_test.

Sequential Model Attempt 2

I want to add one more internal layer and see how the results change. I will keep the overall number of nodes to 32, but will split it between the input and internal layer.

```
# fit model
model = models.Sequential()
model.add(layers.Dense(20, input_shape=(10000,), kernel_initializer='normal', activation='rel
model.add(layers.Dense(12, kernel_initializer='normal', activation='relu'))
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))

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

# train the model
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=3,
                    verbose=1,
                    validation_split=0.1)
```

```
Epoch 1/3
19807/19807 [==============================] - 86s 4ms/step - loss: 0.1744 - accuracy:
Epoch 2/3
19807/19807 [==============================] - 83s 4ms/step - loss: 0.1003 - accuracy:
Epoch 3/3
19807/19807 [==============================] - 85s 4ms/step - loss: 0.0828 - accuracy:
```

```
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))
```

```
687/687 [==============================] - 2s 2ms/step
              precision    recall  f1-score   support

           0       0.95      0.95      0.95     10340
           1       0.96      0.96      0.96     11632

    accuracy                           0.95     21972
   macro avg       0.95      0.95      0.95     21972
weighted avg       0.95      0.95      0.95     21972
```

Interestingly, splitting the nodes between a few layers not only slightly improved the accuracy on positive texts, but it also didn't take nearly as much RAM to run.

Now I want to try to input a new value not in the dataset, and I want to see if its able to detect whether it is positive or not

```
new_test_messages = ['stressful moment test tomorrow not ready', 'won free meal restaurant']
x_test_new = tokenizer.texts_to_matrix(new_test_messages, mode='tfidf')

new_predictions = model.predict(x_test_new)
new_predictions
```

```
1/1 [==============================] - 0s 40ms/step
array([[0.3823218],
       [0.8492132]], dtype=float32)
```

On these two messages, it did a great job determining which one was negative (the first one) and which one was positive (the second one). It would need more verification, but since it did an okay job verifying those messages then the data probably did not overfit too much.

Attempt with RNN architecture

I have decided to first use the RNN architecture since this one works well with textual data. I am going to keep things mostly the same as the simple dense architecture on attempt 2, except I will change the layers a bit to have some embedding

I have noticed that training on RNNs take a very long time. For this reason, I will only train on 1 epoch. I also increased my batch size to 500 as that seemed to have sped things up a bit, at the cost of reducing the number of updates.

```
model = models.Sequential()
model.add(layers.Embedding(vocab_size, 32))
model.add(layers.SimpleRNN(32))
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))

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

# train the model
history = model.fit(x_train, y_train,
                    batch_size=500,
                    epochs=1,
                    verbose=1,
                    validation_split=0.1)
```

```
159/159 [==============================] - 3299s 21s/step - loss: 0.6920 - accuracy: 0.
```

```
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))
```

```
687/687 [==============================] - 317s 462ms/step
              precision    recall  f1-score   support

           0       0.00      0.00      0.00     10340
           1       0.53      1.00      0.69     11632

    accuracy                           0.53     21972
   macro avg       0.26      0.50      0.35     21972
weighted avg       0.28      0.53      0.37     21972

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

The results of this model did significantly worse than the results of the dense sequential model. It seems like the presence or absence of words from the Dense Sequential is more helpful when it comes to determining the positivity or negativity of a statement. In addition, while RNN uses previous words to determine its solution, its not good at remembering things long-term across many iterations.

However, I don't think it is a fair comparison since I dramatically increased the batch size and decreased the epochs, causing it to update far less, which for an iterative model like RNN would of course decrease its performance. I will give the next architecture a smaller batch size to help it perform a bit better.

Attempt with GRU architecture

My prediction is that GRU should perform a bit better than RNN not just because of its smaller batch size but also because it has the capability of determining whether or not data is irrelevant and can drop it if it is, making it a bit more efficient than RNN.

This model takes significantly more RAM to run, so I needed to drop the batch size significantly in order to get it to run.

```
model = models.Sequential()
model.add(layers.Embedding(vocab_size, 32))
model.add(layers.GRU(32))
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))
```

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

# train the model
history = model.fit(x_train, y_train,
                    batch_size=200,
                    epochs=1,
                    verbose=1,
                    validation_split=0.1)
```

```
397/397 [==============================] - 7496s 19s/step - loss: 0.6917 - accuracy: 0.
```

```
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))
```

```
687/687 [==============================] - 493s 717ms/step
              precision    recall  f1-score   support

           0       0.00      0.00      0.00     10340
           1       0.53      1.00      0.69     11632

    accuracy                           0.53     21972
   macro avg       0.26      0.50      0.35     21972
weighted avg       0.28      0.53      0.37     21972

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

GRU performed the exact same as RNN. However, based on the precision distribution I would say that my model just predicts 1s the entire time. I would assume that it did the same for RNN as well. I think having it run for more epochs and maybe randomizing the data again could help it out.

Creating my own embedding

I will create an embedding to run on this data. I will need to create a vectorizer set up with the vocab in order to vectorize the training and test sets. I will be using some Conv1D layers since

those layers tend to work well with text. I will also be using some max pooling and a dropout layer to prevent overfitting. I will end in a normal dense output layer.

```
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
# creating the vectorizer
vectorizer = TextVectorization(max_tokens=10000, output_sequence_length=200)
text_ds = tf.data.Dataset.from_tensor_slices(train.text).batch(128)
vectorizer.adapt(text_ds)
```

```
# get the vocab so that you can create a dict for determining word frequency
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))

# verifying to see if my dict was created properly
test = ["sad", "happy", "stressed", "scared", "fit"]
[word_index[w] for w in test]
```

```
[83, 40, 245, 696, 864]
```

Now my own embedding will be created

```
# next, the embedding layer will be created
from tensorflow.keras import layers

# determines the dimension of the final embedding
EMBEDDING_DIM = 128

# fixed length of each sample. If sample does not reach this length, it will be padded so it
MAX_SEQUENCE_LENGTH = 200

embedding_layer = layers.Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

```
# creating the model
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.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
# I picked 1 because that was the size of the dense output layer in previous models
preds = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(int_sequences_input, preds)
```

In order to use my data on this model, I need to vectorize the data so it can be utilized with the embedding layer.

To keep things more consistent with other models, I will not have validation data since my initial divide was only with test and train.

```
x_train = vectorizer(np.array([[s] for s in train.text])).numpy()
y_train = np.asarray(train.emotion).astype('float32').reshape((-1,1))

model.compile(
    loss='binary_crossentropy', optimizer="adam", metrics=["accuracy"]
)
model.fit(x_train, y_train, batch_size=128, epochs=3, verbose=1)
```

```
Epoch 1/3
688/688 [==============================] - 317s 458ms/step - loss: 0.6165 - accuracy: 0
Epoch 2/3
688/688 [==============================] - 304s 443ms/step - loss: 0.6007 - accuracy: 0
Epoch 3/3
688/688 [==============================] - 301s 437ms/step - loss: 0.5979 - accuracy: 0
<keras.callbacks.History at 0x7fcc8af63760>
```

```
from sklearn.metrics import classification_report

test_x = vectorizer(np.array([[s] for s in test.text])).numpy()

preds = model.predict(test_x)
emotion_labels = [np.argmax(p) for p in preds]
print(emotion_labels)
print(test.emotion)

print(classification_report(test.emotion, emotion_labels))
```

```
687/687 [==============================] - 22s 31ms/step
[1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1,
7         0
8         1
9         0
19        1
24        0
         ..
109978    0
109979    0
109983    0
109993    1
109996    0
Name: emotion, Length: 21972, dtype: int64
              precision    recall  f1-score   support

           0       0.06      0.07      0.06     10340
```

```
              1       0.02      0.02      0.02     11632
              2       0.00      0.00      0.00         0

    accuracy                           0.04     21972
   macro avg       0.03      0.03      0.03     21972
weighted avg       0.04      0.04      0.04     21972

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

It strikes me as strange that the values are as low as they are. It was also strange that there was an index 2 despite being binary data. To me this said that there could be something wrong with the outputted labels, so I double checked the predicted labels and actual labels and saw that they are comparable (meaning the predicted labels looked similar to the actual labels). So my model wasn't spitting out invalid data, my model just predicted very bad.

Its also possible that the precision is super low because the values while training got reset to 0 too often before they could be learned properly. Also, I could have let my model run for longer to see if any issues got sorted out. Perhaps the layers that I used for the model require a decent amount of epochs in order to get good results.

Initially, the values were even worse, with 0 having a 0.03 precision. By modifying some values I was able to improve it a little bit, but its still very low.

Adding a layer to the Embedding Model

I realized that I left out a GlobalMaxPooling1D layer out of my model, which takes the max value over the time dimension. That may help the values to converge and learn better than my previous model

```
# creating the model
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.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
# I picked 1 because that was the size of the dense output layer in previous models
```

```
preds = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(int_sequences_input, preds)


x_train = vectorizer(np.array([[s] for s in train.text])).numpy()
y_train = np.asarray(train.emotion).astype('float32').reshape((-1,1))

model.compile(
    loss='binary_crossentropy', optimizer="adam", metrics=["accuracy"]
)
model.fit(x_train, y_train, batch_size=128, epochs=3, verbose=1)
```

```
Epoch 1/3
688/688 [==============================] - 277s 401ms/step - loss: 0.6154 - accuracy: 0
Epoch 2/3
688/688 [==============================] - 267s 388ms/step - loss: 0.6006 - accuracy: 0
Epoch 3/3
688/688 [==============================] - 273s 396ms/step - loss: 0.5979 - accuracy: 0
<keras.callbacks.History at 0x7fc7d5bc7f40>
```

```
from sklearn.metrics import classification_report

test_x = vectorizer(np.array([[s] for s in test.text])).numpy()

preds = model.predict(test_x)
emotion_labels = [np.argmax(p) for p in preds]

print(classification_report(test.emotion, emotion_labels))
```

```
687/687 [==============================] - 18s 26ms/step
              precision    recall  f1-score   support

           0       0.04      0.05      0.04     10340
           1       0.03      0.02      0.03     11632
           2       0.00      0.00      0.00         0

    accuracy                           0.03     21972
   macro avg       0.02      0.02      0.02     21972
weighted avg       0.03      0.03      0.03     21972

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

The extra layer actually caused it to perform slightly worse, making me think that it didn't help to get it to converge at all. I think the only thing that can help it is to run it for more epochs, since it

seemed to be getting gradually better the longer it ran.

Using GloVe for a pretrained embedding

I will use the same vectorizer and word_index dict that was created when I made my own embedding. I will then import GloVe's pretrained embedding to use

```
import os

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

# getting the word vectors from the file
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.
```

```
num_tokens = len(voc) + 2
embedding_dim = 100
hits = 0
misses = 0

# Creating embedding matrix that will contain points for all the vocab + 2
# by the embedding dimension that we set
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
    # using a word we grabbed from our vocab, try to find that word in
    # the GloVe vectors
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # word from vocab was found in the GloVe vectors
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        # that entry in the vocab will remain as 0
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

```
Converted 9906 words (94 misses)
```

```
# create the embedding, keeping the GloVe vectors as they are by setting trainable
# to false
from tensorflow.keras.layers import Embedding

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

I will use the same layer structure used from my own embeddings for the GloVe embeddings

```
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)
# I picked 1 because that was the size of the dense output layer in previous models
preds = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(int_sequences_input, preds)
```

Now I will train on this model and see how performance is

```
x_train = vectorizer(np.array([[s] for s in train.text])).numpy()
y_train = np.asarray(train.emotion).astype('float32').reshape((-1,1))

model.compile(
    loss='binary_crossentropy', optimizer="adam", metrics=["accuracy"]
)
model.fit(x_train, y_train, batch_size=128, epochs=3, verbose=1)
```

```
Epoch 1/3
688/688 [==============================] - 176s 254ms/step - loss: 0.2364 - accuracy: 0
Epoch 2/3
688/688 [==============================] - 179s 261ms/step - loss: 0.1217 - accuracy: 0
Epoch 3/3
688/688 [==============================] - 178s 259ms/step - loss: 0.0916 - accuracy: 0
<keras.callbacks.History at 0x7fc5b9d7d190>
```

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

```
test_x = vectorizer(np.array([[s] for s in test.text])).numpy()

preds = model.predict(test_x)
emotion_labels = [np.argmax(p) for p in preds]
print(emotion_labels)
print(test.emotion)

print(classification_report(test.emotion, emotion_labels))
```

```
687/687 [==============================] - 20s 29ms/step
[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,
7         0
8         1
9         0
19        1
24        0
         ..
109978    0
109979    0
109983    0
109993    1
109996    0
Name: emotion, Length: 21972, dtype: int64
              precision    recall  f1-score   support

           0       0.47      1.00      0.64     10340
           1       0.00      0.00      0.00     11632

    accuracy                           0.47     21972
   macro avg       0.24      0.50      0.32     21972
weighted avg       0.22      0.47      0.30     21972

/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.9/dist-packages/sklearn/metrics/_classification.py:1344: Undefin
  _warn_prf(average, modifier, msg_start, len(result))
```

I got kind of suspicious about the precision percentage of the 0, as that was about the same percentage of negative values in the dataset overall. So I decided to print out the predicted labels and, sure enough, it guessed 0 for everything. It must have decided that was the best approach during the training phase, however it doesn't result in a very good model.

I think if I were to run it for longer this problem may potentially sort itself out, or perhaps randomize the data again and see if the distribution of 0s to 1s caused it to decide to just predict 0s

✓ 8m 25s completed at 2:15 PM