Dementia Classification based on Linguistic Analysis

Yunxi (Claire) Cheng, Tripti Sharma, Linting (Linsey) Wang, Sifan(Ellen) Zhu

Project Overview

Our goal is to classify ***if a patient has dementia /Alzheimers based on linguistic data from the DementiaBank dataset***. The label is a binary value indicating if the patient has dementia (1) or not (0). The predictor is the transcripts with audio data containing sentences from patients with or without dementia.

Motivation & Novelty

Dementia is a debilitating neurological disorder that affects millions of people worldwide. Early diagnosis and intervention can significantly improve patients' quality of life, but it can be challenging to identify dementia in its early stages. Machine learning models can help to classify individuals with dementia based on language use patterns, which can serve as an early diagnostic tool.

Healthcare professionals, researchers, and caregivers who work with individuals with dementia would find such models useful. Implementing these models could improve the quality of care and treatment for individuals with dementia while also supporting research into the underlying mechanisms of the disease.

Data Source

Dataset for this project is retrived from **DementiaBank**.

The dataset contains audio recordings and transcriptions of conversations between individuals with dementia and no dementia as the control. Transcriptions were transcribed using CLAN (Computerized Language Analysis) software tool which provides a user-friendly interface for transcribing audio data and allows researchers to create coded transcripts for detailed linguistic analysis. In this project, the audio data from the DementiaBank corpus was transcribed using CLAN and then preprocessed for use in the neural network. The `.cha` files were obtained from TalkBank and preprocessed to remove irrelevant information and convert them to a format suitable for training a machine learning model.

The basic motive behind the neural network and classification is to identify patterns in language use that are characteristic of individuals with dementia. The neural network uses the transcriptions of conversations between individuals with dementia and their caregivers to learn these patterns and classify individuals as either having dementia or not. By accurately classifying individuals with dementia, the model can serve as an early diagnostic tool, allowing for earlier intervention and improved outcomes for patients.

Data cleaning

```
# mount drive
from google.colab import drive
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from keras import layers
import io
import numpy as np


drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tru
```

```
# check the version of tensorflow and upgrade if needed
!pip -q install --upgrade tensorflow
```

Our first step of data cleaning is to import the data, and parse the `.cha` formatted data into strings by: Removing unwanted lines from the investigator, which was also recorded along

1. Removing unwanted lines from the investigator. We kept speech opens with `*PAR:` which indicates that it is from the participants only.
2. Removing the encoding that fails to retain any meaning when transformed from `.cha` file to strings using `re`.

```
import os
import re
import chardet

# parse .cha files to string
def parse_cha_file(content):
  parsed_data = []
  lines = content.split("\n")
  for line in lines:
    if line.startswith("*PAR:"): # keeps only participants' speech
      line = re.sub('\d+', '', line)
      line = re.sub('_', '', line)
      speaker = re.findall(r'\*(\w+):', line)[0]
      utterance = line.split(":", 1)[1].strip()

      """ Need to figure out a way to remove the number_number at the end of each line"""

      parsed_data.append({"speaker": speaker, "utterance": utterance})
  return parsed_data

# control folder
folder_path_cnt = '/content/drive/Shareddrives/Dementia Classification based on Linguistic Analysis/dementia/Control'
folder_path_dmt = '/content/drive/Shareddrives/Dementia Classification based on Linguistic Analysis/dementia/Dementia'

speech_data_cnt = []
speech_data_dmt = []

for file in os.listdir(folder_path_cnt):
  if file.endswith(".cha"):
    file_path = os.path.join(folder_path_cnt, file)
    speech = ""


    # Detect the file encoding using chardet
    with open(file_path, "rb") as f:
      file_content = f.read()
      detected_encoding = chardet.detect(file_content)["encoding"]

    # Read the file with the detected encoding
    with open(file_path, "r", encoding=detected_encoding) as f:
      content = f.read()

    parsed_data = parse_cha_file(content)

    for entry in parsed_data:
      speech += entry["utterance"]
      speech += " "
    speech_data_cnt.append([speech, 0])

for file in os.listdir(folder_path_dmt):
  if file.endswith(".cha"):
    file_path = os.path.join(folder_path_dmt, file)
    speech = ""

    # Detect the file encoding using chardet
    with open(file_path, "rb") as f:
      file_content = f.read()
      detected_encoding = chardet.detect(file_content)["encoding"]

    # Read the file with the detected encoding
    with open(file_path, "r", encoding=detected_encoding) as f:
      content = f.read()

    parsed_data = parse_cha_file(content)

    for entry in parsed_data:
      speech += entry["utterance"]
      speech += " "
    speech_data_dmt.append([speech, 1])

data = speech_data_dmt + speech_data_cnt

df = pd.DataFrame(data, columns=['speech', 'has_dementia'])
df.head(5)
```

	speech	has_dementia
0	animals . [+ exc] a horse . a cow . a...	1
1	sharpen up (.) the pencil „ please . the tr...	1
2	please sharpen my pencil . (.) be careful o...	1
3	I write with a pencil . the tree has some l...	1
4	you are using a pencil . (.) a tree is gree...	1

Data Description

The data consists of text samples in the `'speech'` column, and a binary label in the `'has_dementia'` column indicating whether the speaker has been diagnosed with dementia or not.

The data has 1040 records of dementia patients, and 247 records of control with class imbalance.

In total, there are 1287 observations.

```
#check the total number of demential and control
len(speech_data_dmt),len(speech_data_cnt)
```

```
(1040, 247)
```

```
# shuffle the rows
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
df.head(5)
```

	speech	has_dementia
0	okay . [+ exc] well the mother is drying th...	0
1	(..) well ‡ Melanie [//] Melanie's grandfather...	1
2	you ready ? [+ exc] the little girl's goin(...	1
3	I can't +/. [+ exc] oh ‡ to eat ? [+ exc] ...	1
4	(.) well the mother's &-uh doing the dishes an...	0

```
print('total samples:',df.shape[0])
df_text = df['speech'].to_numpy()
labels = df["has_dementia"].to_numpy()
data_train = df_text[:800]
labels_train = labels[:800]
data_test = df_text[1000:]
labels_test = labels[1000:]
data_val = df_text[800:1000]
labels_val = labels[800:1000]
```

```
total samples: 1287
```

Neural Network Models

LSTM Model without Embedding

The nature of the data suggests that our predictor consists of long strings, which makes LSTM (Long Short Term Memory) a good fit, as LSTM model is capable of dealing with long string of data.

We started by tokenizing and vectorizing the textual data.

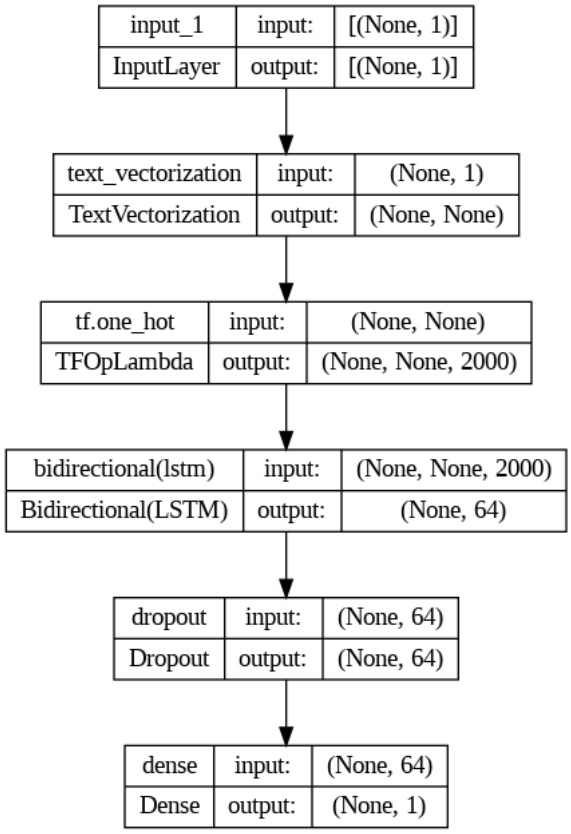
```
text_vectorization = keras.layers.TextVectorization(
    max_tokens=2000, # We will use the 2000 most frequent terms; doing this automatically pads the                    seque
    output_mode="int", # This is requesting integer encodings (which means we'll have a sequence of integers)
    #ragged=True, # This returns a ragged list (sequences of variable length); absent this, the vectorizer will pad.
)
text_vectorization.adapt(df_text)
```

```
processing = text_vectorization(data_train)
print(processing)
```

```
tf.Tensor(
[[ 57   4   2 ...   0   0   0]
 [ 32  12 165 ...   0   0   0]
 [ 19 341   4 ...   0   0   0]
 ...
 [288  23  43 ...   0   0   0]
 [ 57   4   2 ...   0   0   0]
 [144   2 135 ...   0   0   0]], shape=(800, 615), dtype=int64)
```

Then we construct the model with 32 LSTMs. We also experimented with 16 and 64 LSTMs and found that 32 LSTMs gives the most accurate results. We also added `layers.Dropout(0.5)` to decrease overfitting, and used `activation="sigmoid"` in output layer because this is a classification prediction.

```
def build_model():
    inputs = keras.layers.Input(shape=(1), dtype="string") # We take our strings as input
    processing = text_vectorization(inputs)
    one_hot = tf.one_hot(processing, depth=2000)
    x = layers.Bidirectional(layers.LSTM(32))(one_hot)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)

    model = keras.Model(inputs,outputs)
    model.compile(optimizer="rmsprop",loss="binary_crossentropy",metrics=['accuracy'])
    return model

model = build_model()

keras.utils.plot_model(model, show_shapes=True)
```



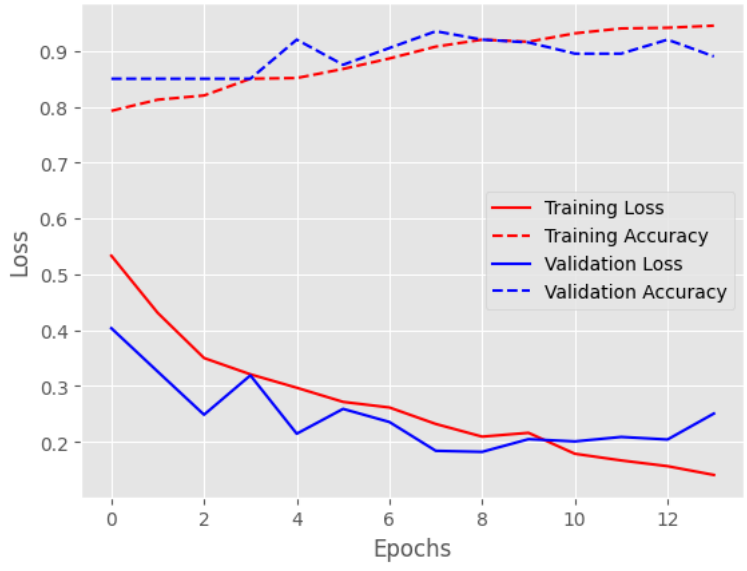
When fitting the model, we also used `EarlyStopping` to reduce overfitting. The results are accurate, with `- loss: 0.1409 - accuracy: 0.9450 - val_loss: 0.2508 - val_accuracy: 0.8900`.

```
from tensorflow.keras.callbacks import EarlyStopping

num_epochs = 25
batch_sizes = 25
```

```
5
6 model = build_model()
7 early_stopping = EarlyStopping(monitor='val_loss', patience=5)
8
9 history = model.fit(data_train, labels_train,
10                       validation_data = (data_val,labels_val),
11                       epochs=num_epochs, batch_size=batch_sizes,callbacks=[early_stopping])
12 val_loss_history = history.history['val_loss']
13 val_acc_history = history.history['val_accuracy']
14 loss_history = history.history['loss']
15 acc_history = history.history['accuracy']
```

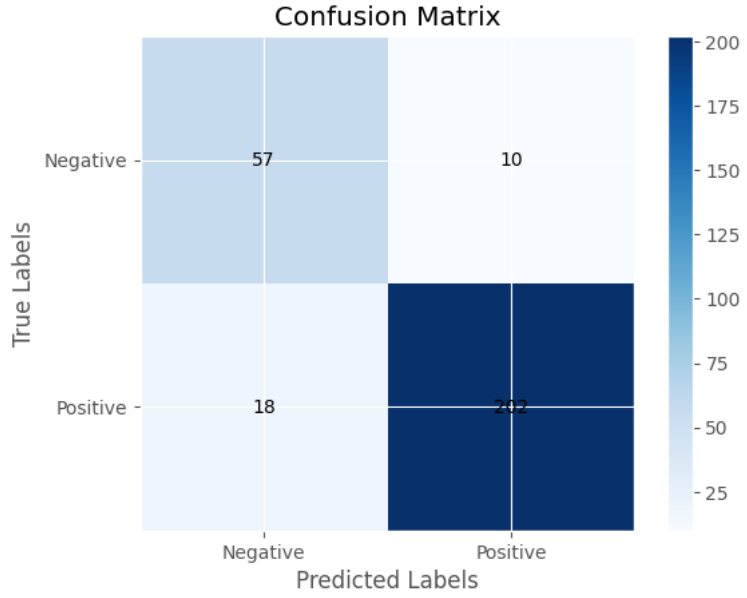
```
Epoch 1/25
32/32 [==============================] - 63s 2s/step - loss: 0.5332 - accuracy: 0.7925 - val_loss: 0.4035 - val_accuracy: 0.8
Epoch 2/25
32/32 [==============================] - 33s 1s/step - loss: 0.4310 - accuracy: 0.8125 - val_loss: 0.3259 - val_accuracy: 0.8
Epoch 3/25
32/32 [==============================] - 32s 997ms/step - loss: 0.3501 - accuracy: 0.8200 - val_loss: 0.2486 - val_accuracy:
Epoch 4/25
32/32 [==============================] - 37s 1s/step - loss: 0.3210 - accuracy: 0.8500 - val_loss: 0.3191 - val_accuracy: 0.8
Epoch 5/25
32/32 [==============================] - 33s 1s/step - loss: 0.2969 - accuracy: 0.8512 - val_loss: 0.2146 - val_accuracy: 0.9
Epoch 6/25
32/32 [==============================] - 34s 1s/step - loss: 0.2716 - accuracy: 0.8675 - val_loss: 0.2591 - val_accuracy: 0.8
Epoch 7/25
32/32 [==============================] - 35s 1s/step - loss: 0.2616 - accuracy: 0.8863 - val_loss: 0.2356 - val_accuracy: 0.9
Epoch 8/25
32/32 [==============================] - 34s 1s/step - loss: 0.2321 - accuracy: 0.9075 - val_loss: 0.1840 - val_accuracy: 0.9
Epoch 9/25
32/32 [==============================] - 33s 1s/step - loss: 0.2094 - accuracy: 0.9200 - val_loss: 0.1822 - val_accuracy: 0.9
Epoch 10/25
32/32 [==============================] - 35s 1s/step - loss: 0.2162 - accuracy: 0.9162 - val_loss: 0.2049 - val_accuracy: 0.9
Epoch 11/25
32/32 [==============================] - 38s 1s/step - loss: 0.1788 - accuracy: 0.9312 - val_loss: 0.2009 - val_accuracy: 0.8
Epoch 12/25
32/32 [==============================] - 58s 2s/step - loss: 0.1668 - accuracy: 0.9400 - val_loss: 0.2089 - val_accuracy: 0.8
Epoch 13/25
32/32 [==============================] - 40s 1s/step - loss: 0.1565 - accuracy: 0.9413 - val_loss: 0.2043 - val_accuracy: 0.9
Epoch 14/25
32/32 [==============================] - 33s 1s/step - loss: 0.1409 - accuracy: 0.9450 - val_loss: 0.2508 - val_accuracy: 0.8
```

```
1 import matplotlib.pyplot as plt
2
3 plt.style.use('ggplot')
4
5 plt.plot(loss_history,c='r')
6 plt.plot(acc_history,c="r",linestyle="dashed")
7 plt.plot(val_loss_history,c='b')
8 plt.plot(val_acc_history,c='b',linestyle="dashed")
9 plt.xlabel("Epochs")
10 plt.ylabel("Loss")
11 plt.legend(['Training Loss','Training Accuracy','Validation Loss','Validation Accuracy'])
12 plt.show()
```



```
from sklearn.metrics import confusion_matrix
predictions = model.predict(data_test)

# build confusion matrix
cm = confusion_matrix(labels_test, np.ravel(predictions)>0.5)
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0, 1], ["Negative", "Positive"])
plt.yticks([0, 1], ["Negative", "Positive"])
# plot numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

plt.show()
```

```
9/9 [==============================] - 8s 845ms/step
```

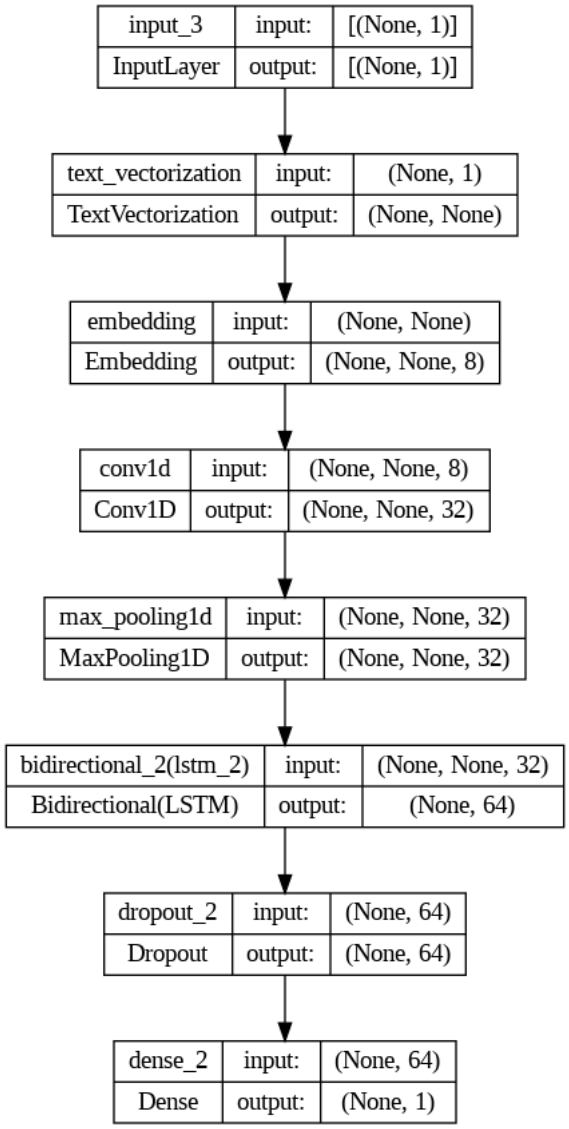


Word Embedding with 1D Convolution

Our second model of choice is LSTM with word embedding layer. We added a 1D convolution layer. We then added a pooling layer along with the convolution layer, followed by the LSTM model.

```
def build_model_embed():
    inputs = keras.layers.Input(shape=(1), dtype="string")
    processing = text_vectorization(inputs)
    one_hot = tf.one_hot(processing, depth=2000)
    embedding = keras.layers.Embedding(input_dim=2000,output_dim=8,input_length=615, mask_zero=True)(processing)


    # Add a 1D convolutional layer
    conv_layer = keras.layers.Conv1D(filters=32, kernel_size=3, padding='same', activation='relu')(embedding)
    pool_layer = keras.layers.MaxPooling1D(pool_size=2)(conv_layer)

    # Add a bidirectional LSTM layer
    lstm_layer = layers.Bidirectional(layers.LSTM(32))(pool_layer)
    dropout_layer = layers.Dropout(0.5)(lstm_layer)
    outputs = layers.Dense(1, activation="sigmoid")(dropout_layer)

    model = keras.Model(inputs,outputs)
    model.compile(optimizer="rmsprop",loss="binary_crossentropy",metrics=['accuracy'])
    return model


model = build_model_embed()
```

```
23
24 keras.utils.plot_model(model, show_shapes=True)
```



LSTM with word embedding layer results are: `- loss: 0.1313 - accuracy: 0.9650 - val_loss: 0.2402 - val_accuracy: 0.8800`. However, from the confusion matrix, we can see that there are more false positive predictions.

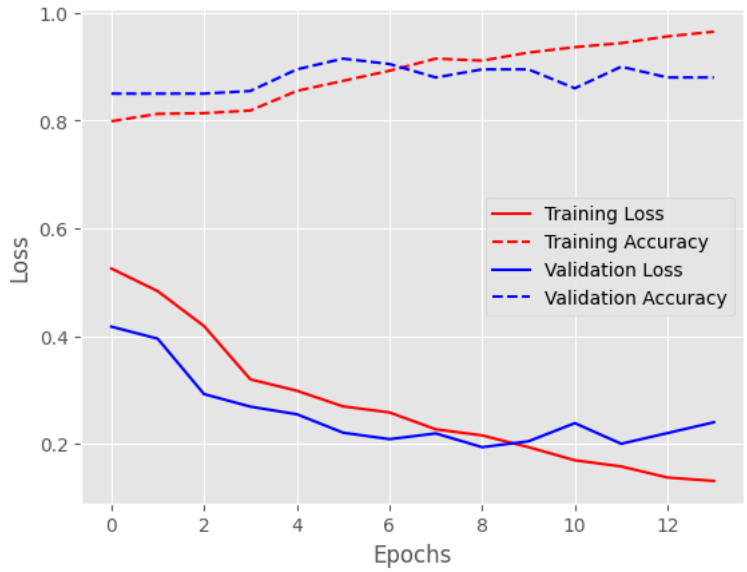
One possible explanation is that the participants are asked to read the same script, or describing the same picture; as a result, the word embedding or the context will not affect the predictions as much as the tokens themselves do.

```
 1 num_epochs = 25
 2 batch_sizes = 25
 3
 4 model_embed = build_model_embed()
 5 early_stopping = EarlyStopping(monitor='val_loss', patience=5)
 6
 7 history = model_embed.fit(data_train, labels_train,
 8                          validation_data = (data_val,labels_val),
 9                          epochs=num_epochs, batch_size=batch_sizes,callbacks=[early_stopping])
10 val_loss_history = history.history['val_loss']
11 val_acc_history = history.history['val_accuracy']
12 loss_history = history.history['loss']
13 acc_history = history.history['accuracy']
```

```
Epoch 1/25
32/32 [==============================] - 10s 181ms/step - loss: 0.5254 - accuracy: 0.7987 - val_loss: 0.4176 - val_accuracy:
Epoch 2/25
32/32 [==============================] - 8s 255ms/step - loss: 0.4835 - accuracy: 0.8125 - val_loss: 0.3953 - val_accuracy: (
Epoch 3/25
32/32 [==============================] - 4s 131ms/step - loss: 0.4187 - accuracy: 0.8138 - val_loss: 0.2925 - val_accuracy: (
Epoch 4/25
```

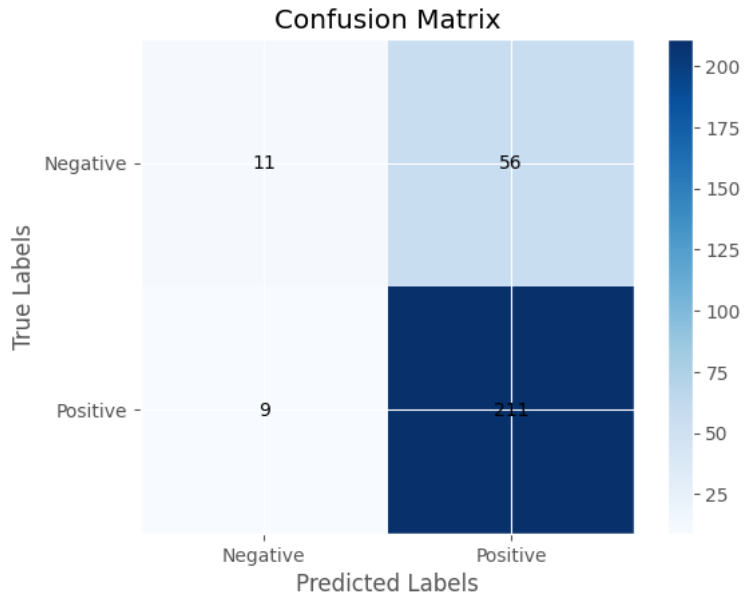
```
32/32 [==============================] - 7s 212ms/step - loss: 0.3198 - accuracy: 0.8188 - val_loss: 0.2692 - val_accuracy: (
Epoch 5/25
32/32 [==============================] - 4s 130ms/step - loss: 0.2988 - accuracy: 0.8550 - val_loss: 0.2551 - val_accuracy: (
Epoch 6/25
32/32 [==============================] - 4s 128ms/step - loss: 0.2697 - accuracy: 0.8737 - val_loss: 0.2209 - val_accuracy: (
Epoch 7/25
32/32 [==============================] - 6s 177ms/step - loss: 0.2585 - accuracy: 0.8925 - val_loss: 0.2089 - val_accuracy: (
Epoch 8/25
32/32 [==============================] - 5s 135ms/step - loss: 0.2271 - accuracy: 0.9150 - val_loss: 0.2195 - val_accuracy: (
Epoch 9/25
32/32 [==============================] - 4s 137ms/step - loss: 0.2158 - accuracy: 0.9112 - val_loss: 0.1940 - val_accuracy: (
Epoch 10/25
32/32 [==============================] - 6s 186ms/step - loss: 0.1940 - accuracy: 0.9262 - val_loss: 0.2049 - val_accuracy: (
Epoch 11/25
32/32 [==============================] - 5s 146ms/step - loss: 0.1697 - accuracy: 0.9362 - val_loss: 0.2387 - val_accuracy: (
Epoch 12/25
32/32 [==============================] - 4s 129ms/step - loss: 0.1582 - accuracy: 0.9438 - val_loss: 0.2003 - val_accuracy: (
Epoch 13/25
32/32 [==============================] - 6s 181ms/step - loss: 0.1377 - accuracy: 0.9563 - val_loss: 0.2201 - val_accuracy: (
Epoch 14/25
32/32 [==============================] - 5s 154ms/step - loss: 0.1313 - accuracy: 0.9650 - val_loss: 0.2402 - val_accuracy: (
```

```
import matplotlib.pyplot as plt

plt.style.use('ggplot')

plt.plot(loss_history,c='r')
plt.plot(acc_history,c="r",linestyle="dashed")
plt.plot(val_loss_history,c='b')
plt.plot(val_acc_history,c='b',linestyle="dashed")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(['Training Loss','Training Accuracy','Validation Loss','Validation Accuracy'])
plt.show()
```




```
from sklearn.metrics import confusion_matrix
predictions = model.predict(data_test)

# build confusion matrix
cm = confusion_matrix(labels_test, np.ravel(predictions)>0.5)
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0, 1], ["Negative", "Positive"])
plt.yticks([0, 1], ["Negative", "Positive"])
# plot numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

plt.show()
```

```
9/9 [==============================] - 1s 39ms/step
```



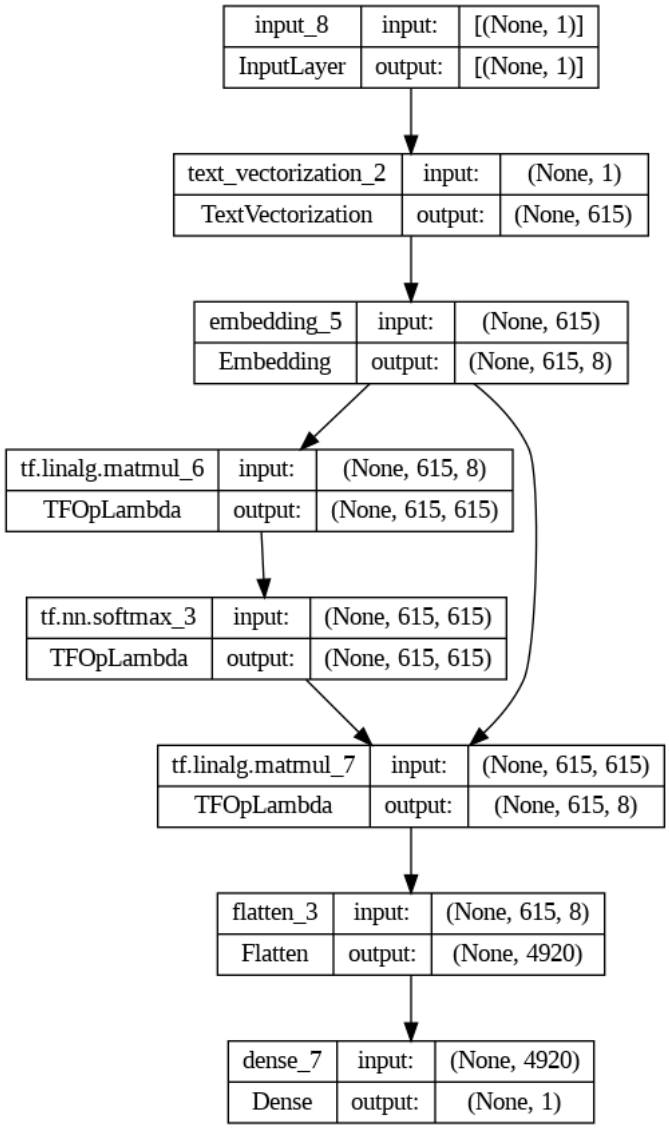
Self-Attention Model

Our last model is the Attention Model, which is also capable of capturing and analyzing long sequence of data for predictions. We used 8-dimension embedding to transform the train data and applied dot product for the weights.

```
text_vectorization = keras.layers.TextVectorization(
    max_tokens=2000, # This means we work with 2000 most common tokens.
    output_mode="int", # This is requesting integer encodings (which means we'll have a sequence of integers)
    output_sequence_length = 615,
)

text_vectorization.adapt(df_text)
```

```
def build_scaled_dotprod_model():
    inputs = keras.layers.Input(shape=(1,), dtype=tf.string) # We take a tensor of strings as input.
    integer_seqs = text_vectorization(inputs)

    # Our integer indices will be converted into 8-dimensional embeddings.
    embeddings = layers.Embedding(2000,8)(integer_seqs)

    ### NOW LET'S BUILD THE ATTENTION MECHANISM

    # In Step 1, we need to compute pairwise dot-products between each word-embedding, and each other word embedding in the tex
    # We have 666 word embeddings, so we should end up with 615 x 615 dot products.
    pairwise_dot_products = tf.matmul(embeddings, embeddings, transpose_b=True)

    # In Step 2, we softmax the resulting dot products, yielding an element-wise transformation.
    # By specifying axis = -1 it means it softmaxes over the final axis of the tensor (rather than softmaxing over the entire t
    # We'll end up with 615 * 615 scaled weights.
    attention_weights = tf.nn.softmax(pairwise_dot_products, axis=-1)

    # Finally, in Step 3, in place of each input word embedding, we obtain the weighted sum of its peer-term attention_weights
    # Here, we are summing over terms (rather than over the embedding dimensions)
    attention_output = tf.matmul(attention_weights, embeddings)

    # Flatten the attention output
    attention_output = layers.Flatten()(attention_output)

    # Output layer
    outputs = layers.Dense(1, activation='sigmoid')(attention_output)

    # Compile the model
    model = keras.Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

```
34
35 model = build_scaled_dotprod_model()
36 keras.utils.plot_model(model,show_shapes=True)
```



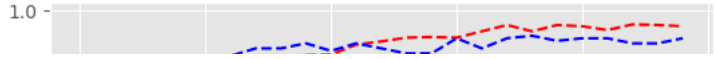
Our results indicate that our attention model has the best performance: `- loss: 0.0840 - accuracy: 0.9688 - val_loss: 0.1642 - val_accuracy: 0.9450`. And it has the least overfitting comparing to the previous 2 models. Our confusion matrics also shows that with attention model, we get the highest numbers of true negative and true positives.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="dotprod.tf",
                                    save_best_only=True)
]

model = build_scaled_dotprod_model()
history = model.fit(data_train, labels_train,
                        validation_data = (data_val,labels_val),
                        epochs=25, batch_size=25,callbacks=callbacks)
```

```
Epoch 1/25
32/32 [==============================] - ETA: 0s - loss: 0.5084 - accuracy: 0.8125WARNING:absl:Found untraced functions such
32/32 [==============================] - 11s 329ms/step - loss: 0.5084 - accuracy: 0.8125 - val_loss: 0.4104 - val_accuracy:
Epoch 2/25
32/32 [==============================] - ETA: 0s - loss: 0.4525 - accuracy: 0.8125WARNING:absl:Found untraced functions such
32/32 [==============================] - 8s 244ms/step - loss: 0.4525 - accuracy: 0.8125 - val_loss: 0.3767 - val_accuracy: (
Epoch 3/25
32/32 [==============================] - ETA: 0s - loss: 0.3971 - accuracy: 0.8163WARNING:absl:Found untraced functions such
32/32 [==============================] - 11s 357ms/step - loss: 0.3971 - accuracy: 0.8163 - val_loss: 0.3249 - val_accuracy:
Epoch 4/25
32/32 [==============================] - ETA: 0s - loss: 0.3402 - accuracy: 0.8288WARNING:absl:Found untraced functions such
32/32 [==============================] - 11s 340ms/step - loss: 0.3402 - accuracy: 0.8288 - val_loss: 0.2663 - val_accuracy:
```

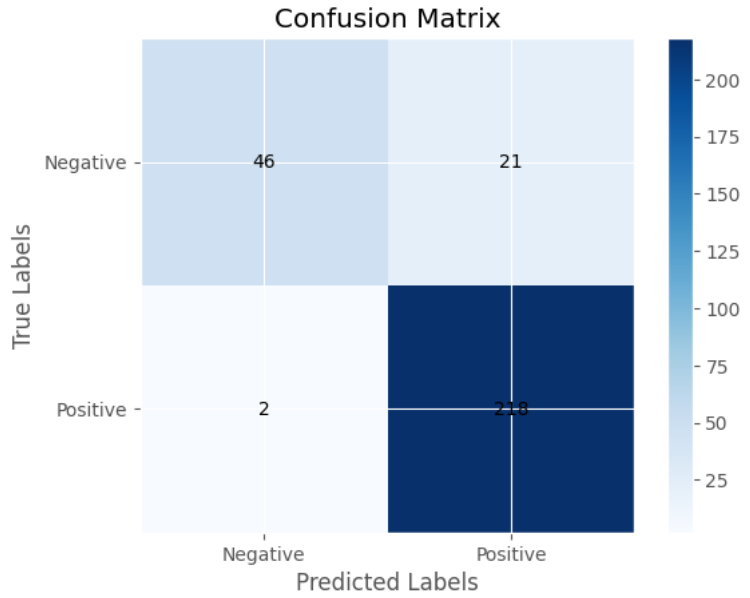
```
Epoch 5/25
32/32 [==============================] - ETA: 0s - loss: 0.2947 - accuracy: 0.8475WARNING:absl:Found untraced functions such
32/32 [==============================] - 8s 248ms/step - loss: 0.2947 - accuracy: 0.8475 - val_loss: 0.2378 - val_accuracy: (
Epoch 6/25
32/32 [==============================] - ETA: 0s - loss: 0.2734 - accuracy: 0.8687WARNING:absl:Found untraced functions such
32/32 [==============================] - 10s 304ms/step - loss: 0.2734 - accuracy: 0.8687 - val_loss: 0.2203 - val_accuracy:
Epoch 7/25
32/32 [==============================] - ETA: 0s - loss: 0.2444 - accuracy: 0.8913WARNING:absl:Found untraced functions such
32/32 [==============================] - 10s 328ms/step - loss: 0.2444 - accuracy: 0.8913 - val_loss: 0.2059 - val_accuracy:
Epoch 8/25
32/32 [==============================] - 6s 202ms/step - loss: 0.2294 - accuracy: 0.8850 - val_loss: 0.2199 - val_accuracy: (
Epoch 9/25
32/32 [==============================] - ETA: 0s - loss: 0.2161 - accuracy: 0.9075WARNING:absl:Found untraced functions such
32/32 [==============================] - 10s 305ms/step - loss: 0.2161 - accuracy: 0.9075 - val_loss: 0.1900 - val_accuracy:
Epoch 10/25
32/32 [==============================] - ETA: 0s - loss: 0.2024 - accuracy: 0.9112WARNING:absl:Found untraced functions such
32/32 [==============================] - 9s 290ms/step - loss: 0.2024 - accuracy: 0.9112 - val_loss: 0.1774 - val_accuracy: (
Epoch 11/25
32/32 [==============================] - ETA: 0s - loss: 0.1981 - accuracy: 0.9125WARNING:absl:Found untraced functions such
32/32 [==============================] - 8s 257ms/step - loss: 0.1981 - accuracy: 0.9125 - val_loss: 0.1697 - val_accuracy: (
Epoch 12/25
32/32 [==============================] - ETA: 0s - loss: 0.1741 - accuracy: 0.9325WARNING:absl:Found untraced functions such
32/32 [==============================] - 10s 332ms/step - loss: 0.1741 - accuracy: 0.9325 - val_loss: 0.1654 - val_accuracy:
Epoch 13/25
32/32 [==============================] - 7s 230ms/step - loss: 0.1649 - accuracy: 0.9388 - val_loss: 0.1904 - val_accuracy: (
Epoch 14/25
32/32 [==============================] - ETA: 0s - loss: 0.1522 - accuracy: 0.9463WARNING:absl:Found untraced functions such
32/32 [==============================] - 11s 355ms/step - loss: 0.1522 - accuracy: 0.9463 - val_loss: 0.1569 - val_accuracy:
Epoch 15/25
32/32 [==============================] - 9s 287ms/step - loss: 0.1421 - accuracy: 0.9475 - val_loss: 0.1623 - val_accuracy: (
Epoch 16/25
32/32 [==============================] - ETA: 0s - loss: 0.1416 - accuracy: 0.9463WARNING:absl:Found untraced functions such
32/32 [==============================] - 8s 262ms/step - loss: 0.1416 - accuracy: 0.9463 - val_loss: 0.1551 - val_accuracy: (
Epoch 17/25
32/32 [==============================] - ETA: 0s - loss: 0.1276 - accuracy: 0.9588WARNING:absl:Found untraced functions such
32/32 [==============================] - 11s 333ms/step - loss: 0.1276 - accuracy: 0.9588 - val_loss: 0.1505 - val_accuracy:
Epoch 18/25
32/32 [==============================] - 10s 324ms/step - loss: 0.1201 - accuracy: 0.9712 - val_loss: 0.1612 - val_accuracy:
Epoch 19/25
32/32 [==============================] - 8s 252ms/step - loss: 0.1121 - accuracy: 0.9588 - val_loss: 0.1579 - val_accuracy: (
Epoch 20/25
32/32 [==============================] - 10s 312ms/step - loss: 0.1029 - accuracy: 0.9712 - val_loss: 0.1679 - val_accuracy:
Epoch 21/25
32/32 [==============================] - 10s 309ms/step - loss: 0.1011 - accuracy: 0.9688 - val_loss: 0.1639 - val_accuracy:
Epoch 22/25
32/32 [==============================] - ETA: 0s - loss: 0.1103 - accuracy: 0.9613WARNING:absl:Found untraced functions such
```

```
import matplotlib.pyplot as plt
plt.style.use('ggplot')

plt.plot(history.history['loss'],c="r")
plt.plot(history.history['val_loss'],c="b")
plt.plot(history.history['accuracy'],c="r",linestyle="--")
plt.plot(history.history['val_accuracy'],c="b",linestyle="--")
plt.legend(['Training Loss','Validation Loss','Training Acc','Validation Acc'])
plt.show()

model = keras.models.load_model("dotprod.tf")
print(f"Val acc: {model.evaluate(data_val,labels_val)[1]:.3f}")
```




```
from sklearn.metrics import confusion_matrix
predictions = model.predict(data_test)

# build confusion matrix
cm = confusion_matrix(labels_test, np.ravel(predictions)>0.5)
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0, 1], ["Negative", "Positive"])
plt.yticks([0, 1], ["Negative", "Positive"])
# plot numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

plt.show()
```

```
9/9 [==============================] - 1s 144ms/step
```



Self-Attention Model with GloVe embeddings

We also try to use the GloVe pre-trained embeddings. however, it is overfitting very quickly. The origional attention model is better than attention mdoel with pre-trained embeddings here.

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

```
--2023-05-02 23:57:39--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2023-05-02 23:57:39--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2023-05-02 23:57:40--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.01MB/s    in 2m 39s
```

```
2023-05-03 00:00:19 (5.17 MB/s) - 'glove.6B.zip' saved [862182613/862182613]
```

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1) # Split off only the first element in the row, i.e., the word, keep the remaining
        coefs = np.fromstring(coefs, "f", sep=" ") # Convert the set of numeric values into a numpy array, splitting elements b
        embeddings_index[word] = coefs # Populate our dictionary - for this word (key), the vector representation is this vecto

glove_vec_lengths = len(coefs)

print(f"Found {len(embeddings_index)} word vectors.")
print(f"GloVe vector representations are {glove_vec_lengths} elements long.")

```

```
Found 400000 word vectors.
GloVe vector representations are 100 elements long.
```

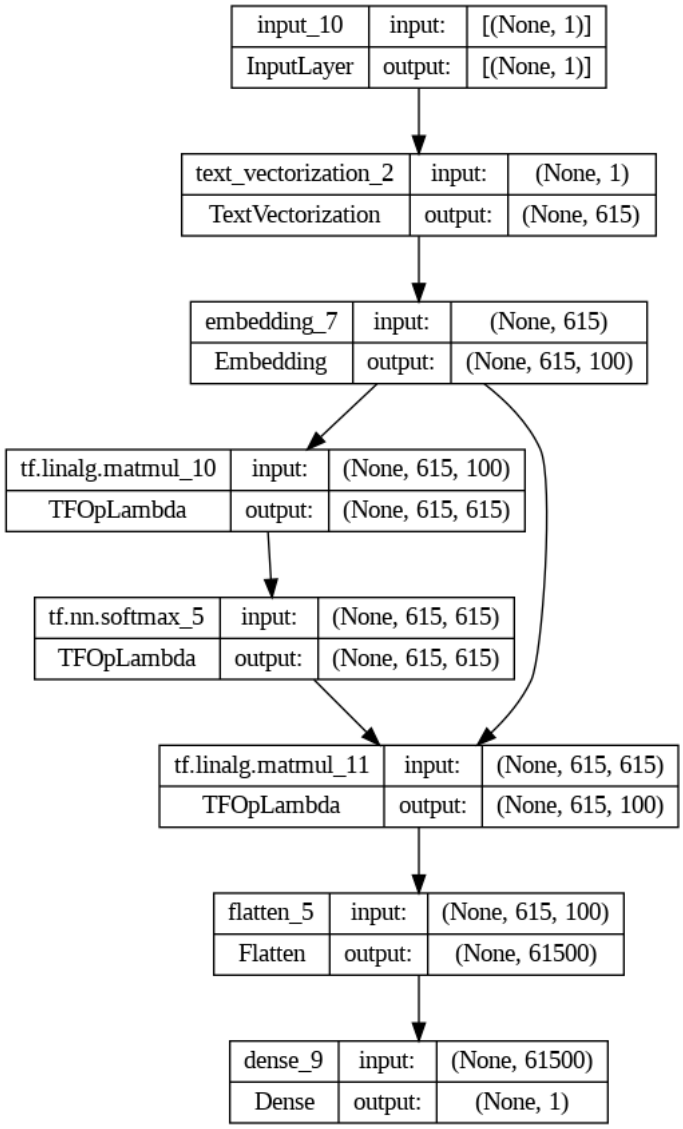
```
embedding_dim = glove_vec_lengths # The length of the vector representations; the latent embedding space will be in 100 dimensi

vocabulary = text_vectorization.get_vocabulary() # From the vocabulary our text vectorizer learned from our dataset, go over ev
word_index = dict(zip(vocabulary, range(len(vocabulary)))) # Make a dictionary, key is word to value is index.

# Instantiate a matrix of values (these will be the 'weights' in our embedding layer)
embedding_matrix = np.zeros((len(vocabulary), embedding_dim)) # It will be the 2000 tokens by 100 (length of GloVe vectors)
for word, i in word_index.items():
    if i < len(vocabulary):
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
embedding_matrix
```

```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 0.74746001, -0.36522999,  0.22048   , ..., -0.30627999,
         0.43299001,  0.015122  ],
       [ 0.040128  , -0.57393998,  0.17842001, ..., -0.0096184 ,
         0.90460998, -0.46077001],
       [ 0.87342   , -0.49823999,  0.18340001, ...,  0.089029  ,
        -0.025995  ,  0.014179  ]])
```

```
embedding_layer = layers.Embedding(
    len(vocabulary), # 2000 words.
    glove_vec_lengths, # Vectors of 100 elements per word.
    embeddings_initializer=keras.initializers.Constant(embedding_matrix), # Initialize with fixed values from our matrix.
    trainable=False, # These are weights, but they will be frozen so they don't update during training.
    mask_zero=True, # If we have a 0 token, for padding, we don't pass it through the layer.
)
```

```
def build_model_GloVe():
    inputs = keras.layers.Input(shape=(1,), dtype=tf.string) # We take a tensor of strings as input.
    integer_seqs = text_vectorization(inputs)

    # Our integer indices will be converted into 100-dimensional embeddings with GloVe.
    embeddings = embedding_layer(integer_seqs)
    pairwise_dot_products = tf.matmul(embeddings, embeddings, transpose_b=True)

    # In Step 2, we softmax the resulting dot products, yielding an element-wise transformation.
    # By specifying axis = -1 it means it softmaxes over the final axis of the tensor (rather than softmaxing over the entire t
    # We'll end up with 666 * 666 scaled weights.
    attention_weights = tf.nn.softmax(pairwise_dot_products, axis=-1)

    # Finally, in Step 3, in place of each input word embedding, we obtain the weighted sum of its peer-term attention_weights
    # Here, we are summing over terms (rather than over the embedding dimensions)
    attention_output = tf.matmul(attention_weights, embeddings)

    # Flatten the attention output
```

```
    attention_output = layers.Flatten()(attention_output)

    # Output layer
    outputs = layers.Dense(1, activation='sigmoid')(attention_output)

    # Compile the model
    model = keras.Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    return model

model = build_model_GloVe()
keras.utils.plot_model(model,show_shapes=True)
```

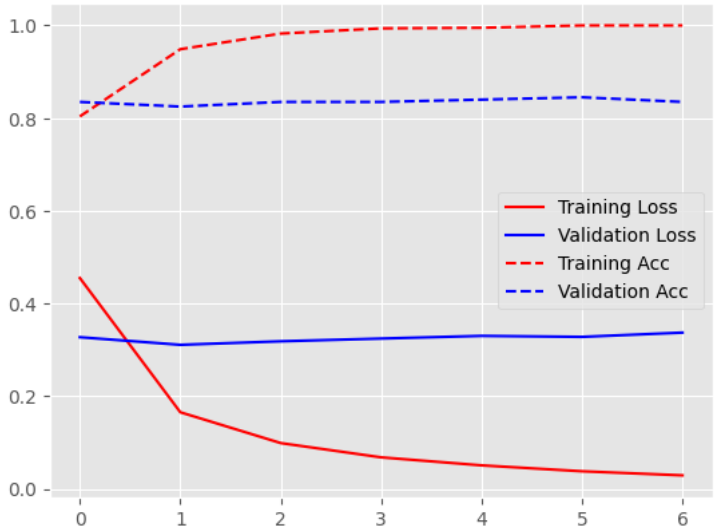



```
callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="dotprod.tf",
                                    save_best_only=True)
]

early_stopping = EarlyStopping(monitor='val_loss', patience=5)

model = build_model_GloVe()
history = model.fit(data_train, labels_train,
                        validation_data = (data_val,labels_val),
                        epochs=25, batch_size=25,callbacks=[early_stopping])
```

```
Epoch 1/25
32/32 [==============================] - 7s 185ms/step - loss: 0.4550 - accuracy: 0.8037 - val_loss: 0.3273 - val_accuracy: (
Epoch 2/25
32/32 [==============================] - 8s 238ms/step - loss: 0.1655 - accuracy: 0.9488 - val_loss: 0.3109 - val_accuracy: (
Epoch 3/25
```

```
32/32 [==============================] - 6s 179ms/step - loss: 0.0989 - accuracy: 0.9825 - val_loss: 0.3185 - val_accuracy: (
Epoch 4/25
32/32 [==============================] - 8s 240ms/step - loss: 0.0681 - accuracy: 0.9937 - val_loss: 0.3245 - val_accuracy: (
Epoch 5/25
32/32 [==============================] - 6s 176ms/step - loss: 0.0509 - accuracy: 0.9950 - val_loss: 0.3302 - val_accuracy: (
Epoch 6/25
32/32 [==============================] - 7s 218ms/step - loss: 0.0381 - accuracy: 1.0000 - val_loss: 0.3281 - val_accuracy: (
Epoch 7/25
32/32 [==============================] - 6s 183ms/step - loss: 0.0293 - accuracy: 1.0000 - val_loss: 0.3373 - val_accuracy: (
```

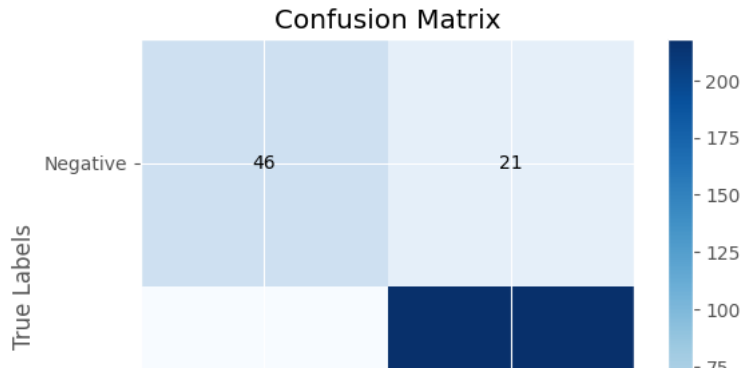
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')

plt.plot(history.history['loss'],c="r")
plt.plot(history.history['val_loss'],c="b")
plt.plot(history.history['accuracy'],c="r",linestyle="--")
plt.plot(history.history['val_accuracy'],c="b",linestyle="--")
plt.legend(['Training Loss','Validation Loss','Training Acc','Validation Acc'])
plt.show()

model = keras.models.load_model("dotprod.tf")
print(f"Val acc: {model.evaluate(data_val,labels_val)[1]:.3f}")
```




```
7/7 [==============================] - 1s 135ms/step - loss: 0.1488 - accuracy: 0.9350
Val acc: 0.935
```

```
from sklearn.metrics import confusion_matrix
predictions = model.predict(data_test)

# build confusion matrix
cm = confusion_matrix(labels_test, np.ravel(predictions)>0.5)
plt.imshow(cm, cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.xticks([0, 1], ["Negative", "Positive"])
plt.yticks([0, 1], ["Negative", "Positive"])
# plot numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

plt.show()
```

```
9/9 [==============================] - 1s 97ms/step
```



Results Comparison

The results from the three models are as follows:

1. ***LSTM without Embedding*** `- loss: 0.1751 - accuracy: 0.9450 - val_loss: 0.2275 - val_accuracy: 0.9076.`
2. ***LSTM with 1D Convolution Layer for Embedding*** `- loss: 0.1038 - accuracy: 0.9638 - val_loss: 0.2412 - val_accuracy: 0.9117` but has more false positive predictions then the other two models.
3. ***Attention model*** `- loss: 0.0879 - accuracy: 0.9725 - val_loss: 0.2149 - val_accuracy: 0.9220`, and have the highest number of true positives and true negative predicts.

As a result, we recommend using attention model for predicting dementia with the DementiaBank dataset. It is the most accurate in predicting if patients have dementia or not out of the 3 models we used. It is also the best choice for healthcare providers to better allocate resource and give patients the correct treatment they need.

Conclusion and Future Work

The dementia classification model can be employed in real-life settings to assist medical professionals in accurately identifying patients with dementia. Its potential uses may include:

- *Early detection of dementia*: The model can be used to identify patients with early stages of dementia, even before the onset of symptoms. This can enable healthcare providers to initiate early interventions and treatments, which can help slow down the progression of the disease.
- *Improving accuracy of diagnosis*: The model can serve as a complementary tool to assist medical professionals in making accurate diagnosis of dementia. This can help reduce the number of misdiagnosed cases, and provide a more reliable and accurate diagnosis.
- *Streamlining diagnostic process*: The model can help streamline the diagnostic process by automating certain aspects of the diagnosis. This can help reduce the workload of healthcare providers, and also save time and resources.
- *Remote screening*: The model can also be used for remote screening of patients, especially in rural or remote areas where access to healthcare providers may be limited. This can help increase access to diagnosis and treatment for patients with dementia

To further enhance the developed dementia classification model, future research could explore various approaches, such as

- Expand the training dataset by integrating speech data collected from everyday dialogues.
- Future research on the dementia classification model could investigate the feasibility of utilizing audio data to eliminate the necessity of CLAN software training, and supplement the existing speech training data with additional samples obtained from everyday discourse.
- One potential area for future work is to leverage the developed model to classify different stages of dementia. This could involve incorporating additional data and features related to the progression of the disease, as well as exploring different neural network architectures and training techniques to optimize the model's performance in this specific task. By accurately identifying the stage of dementia, medical professionals can better tailor treatments and care plans to the needs of individual patients, potentially improving outcomes and quality of life.