$\textbf{from} \ \texttt{sklearn.preprocessing} \ \textbf{import} \ \texttt{StandardScaler}$ from tensorflow.keras.preprocessing.text import Tokenizer from sklearn.model selection import train test split from sklearn.linear\_model import SGDClassifier from sklearn.metrics import roc\_auc\_score, accuracy\_score,plot\_roc\_curve, confusion\_me from sklearn.metrics import mean\_squared\_error, precision\_recall\_curve, auc, accuracy from tqdm import tqdm warnings.filterwarnings('ignore') Data and Pre-Processing The dataset consists of a .csv that contains both the text from each post, and the classification of that post as either suicidal or non-suicidal. To convert the data into a format that can be taken in by the model, I convert each text post into what I've dubbed a "frequency vector". This is a vector where each index represents a different word, and where the value of the i-th element is the number of times that the i-th word has been detected in the text. This is a simple method that encodes the frequency of each word. In order to avoid low-frequency words (as these may be words that are typos, and/or only pop up in a handful of texts and therefore may be situation-specific) I filter for the 10,000 words with the highest frequency in the training set. My final input for each sample is thus a 10,000 element vector, containing the frequency of each of the 10,000 words. After sampling 30% of the full data (due to memory limitations), I use a 60/40 split to allocate training and testing data, respectively. I then encode the label of each sample as 1 (or True) to indicate a post showing suicidal ideation, and 0 (or False) to indicate a post not showing suicidal ideation. Reading data #Load in data data = pd.read csv('Suicide Detection.csv',index col=0) #Due to memory limitations, I can only sample for 30% of the full data. data = data.sample(frac=0.3, random state=10) train\_data,test\_data=train\_test\_split(data,test\_size=0.4,random\_state=10) data.loc[data['class'] == "non-suicide",].head(3) text class 37204 ANYBODY WHO CAN PHOTO SHOP(simply cropping out... non-suicide 30955 question to my fellow asian-americans & amp; fi... non-suicide 231290 Merry Christmas <3 non-suicide Above is an example of how the dataset is stored in the dataframe. The class column contains either "suicide" or "non-suicide" label, and the text column contains a string of text that has been scraped. In order to increase consistency between different pieces of texts, I convert the text to all lower case, and remove any stop words (common words like: as, for, with). This is shown in the clean\_text() function below. def clean\_text(text): #Function for cleaning multiple pieces of text text\_length=[] cleaned\_text=[] for sent in tqdm(text): sent=sent.lower() sent=nfx.remove\_special\_characters(sent) #This function removes special characters sent=nfx.remove\_stopwords(sent) #This function removes stop words from the telephone cleaned\_text.append(sent) return cleaned\_text #Return cleaned word in the text train data['class'].value counts() Out[12]: non-suicide 21011 suicide 20762 Name: class, dtype: int64 As you can see, there is a good balance in our training data between positive and negative cases. #Cleaning training and testing data cleaned\_train\_text=clean\_text(train\_data.text) cleaned\_test\_text=clean\_text(test\_data.text) #Converting data into useable format t = Tokenizer(num\_words=10000) t.fit\_on\_texts(cleaned\_train\_text) X\_train = t.texts\_to\_matrix(cleaned\_train\_text, mode='count') #Converts multiple text X\_test = t.texts\_to\_matrix(cleaned\_test\_text, mode='count') #Normalize and scale data scaler = StandardScaler() scaler.fit(X train) X train = scaler.transform(X train) X\_test = scaler.transform(X\_test) #Binarizing text labels into number format for classification y train=train data['class']=='suicide' y\_train=y\_train.to\_numpy() y\_test=test\_data['class']=='suicide' y\_test=y\_test.to\_numpy() 100%| | 41773/ 41773 [00:02<00:00, 19996.67it/s] 27849 [00:01<00:00, 20477.23it/s] The Tokenizer class is a keras class that can convert texts into formats for NLP usage. In this case, the Tokenizer class determines the frequency of each word in each sample, returning a NxP matrix where N is the number of samples and P is the number of words. The Nth,Pth element of the matrix is thus the frequency that the Pth word occurs in sample N. The num\_words argument is used so that the Tokenizer will only take the most 10,000 frequently occuring words. The feature data thus consists of an NxP frequency matrix, that has been Z-score normalized, and the label data consists of a 0 or 1, where 1 denotes a case of possible suicidal ideation, and 0 denotes a non-suicidal text. **Model Training** Now that we have our formatted data, we can begin to test different models. For this assignment, I wanted to test Random Forests, Logistic Regression, and a neural network model and compare their performance. For the Logistic Regression and Random Forest models, I use the built-in sklearn classes, while for the neural network I use a dense neural network with 4 layers and one final sigmoid activation layer. In [14]: from sklearn.ensemble import RandomForestClassifier rf = RandomForestClassifier(max\_depth=2, random\_state=0) rf.fit(X\_train, y\_train) Out[14]: RandomForestClassifier(max depth=2, random state=0) I train my Random Forests model with a max depth of 2, to limit the degree of overfitting. from sklearn.linear model import SGDClassifier LR = SGDClassifier(max iter=1000, tol=1e-3,loss='log',penalty='elasticnet') LR.fit(X\_train,y\_train) Out[15]: SGDClassifier(loss='log', penalty='elasticnet') For the logistic regression model, I use the SGDClassifier class with the 'log' loss, which gives a logistic regression model that can predict probabilities. I also use elastic-net regularization for my model, which is a type of regularization that combines both L1 and L2 regularization in regression models. #Build deep neural network using stochastic gradient decscent and 4 dense, fully-conne opt = tf.keras.optimizers.SGD(learning rate=0.005) DNN = tf.keras.Sequential() DNN.add(tf.keras.layers.Dense(2000, input shape=(10000,))) DNN.add(tf.keras.layers.Dense(500)) DNN.add(tf.keras.layers.Dense(125)) DNN.add(tf.keras.layers.Dense(1,activation='sigmoid')) #Final layer uses sigmoid activation='sigmoid') DNN.compile(optimizer=opt, loss='binary\_crossentropy', metrics=['accuracy']) #Model training history=DNN.fit(X\_train, y\_train, epochs=10, batch size=128, validation split=0.2) Epoch 1/10 8021 - val loss: 0.3465 - val accuracy: 0.8612 Epoch 2/10 9054 - val loss: 0.3088 - val accuracy: 0.8840 Epoch 3/10 9309 - val\_loss: 0.3050 - val\_accuracy: 0.8930 Epoch 4/10 9435 - val\_loss: 0.3145 - val\_accuracy: 0.8940 Epoch 5/10 9510 - val loss: 0.3323 - val accuracy: 0.8973 Epoch 6/10 9573 - val\_loss: 0.3475 - val\_accuracy: 0.8975 Epoch 7/10 9621 - val\_loss: 0.3703 - val\_accuracy: 0.8953 Epoch 8/10 9645 - val loss: 0.3916 - val accuracy: 0.8949 Epoch 9/10 262/262 [========================= ] - 23s 87ms/step - loss: 0.0899 - accuracy: 0. 9689 - val loss: 0.4121 - val accuracy: 0.8946 Epoch 10/109704 - val loss: 0.4373 - val accuracy: 0.8937 For the deep neural network model, I built my network using 4 fully connected layers and a final sigmoid activation layer. I also used a binary cross entropy loss, and tracked both the validation and training loss and accuracy using a validation split of 20%. The model was trained over 10 epochs and a batch size of 256. #Reformatting data for plotting purposes using seaborn history df = pd.DataFrame(history.history) history df['Epoch']=history df.index+1 loss\_df=pd.melt(history\_df[['loss','Epoch','val\_loss']],id\_vars='Epoch',value\_name='Loss'] accuracy\_df=pd.melt(history\_df[['accuracy','Epoch','val\_accuracy']],id\_vars='Epoch',val\_accuracy'] In [19]: p=sns.lineplot(data=accuracy\_df,x='Epoch',y='Accuracy',hue='variable') =p.set(title="Model training and validation accuracy over 10 epochs") Model training and validation accuracy over 10 epochs 0.975 0.950 0.925 0.900 0.875 0.850 variable 0.825 accuracy val accuracy 0.800 10 Epoch The above plot shows the training and validation accuracy for the model over the course of 10 epochs. The validation accuracy and training accuracy remain about the same until after the 2nd epoch, after which the model's training accuracy overtakes the validation accuracy, and the model begins to overfit to the training set. sns.lineplot(data=loss\_df,x='Epoch',y='Loss',hue='variable') Out[20]: <AxesSubplot:xlabel='Epoch', ylabel='Loss'> 0.45 0.40 0.35 0.30 0.25 0.20 variable 0.15 loss 0.10 val loss 10 Epoch The above plot shows the training and validation binary cross-entropy loss for the model over the course of 10 epochs. The validation accuracy and training accuracy remain about the same until after the 2th epoch, after which the model's training loss overtakes the validation loss. The above 2 plots are good indications that the model begins to overfit after the 5th epoch, and continuing to fit the data is redundant

Final Assignment: Suicide Text Classification

on both the individual, and those around them.

those indicative of generally poor mental health/state.

import matplotlib.pyplot as plt

import neattext.functions as nfx import matplotlib.pyplot as plt

import numpy as np import pandas as pd

import warnings

import seaborn as sns

import tensorflow as tf

According to the World Health Organization, over 700,000 people die by suicide annually worldwide, and for each suicide death, more than 20 suicide attempts are made. Suicides and suicide attempts create ripple affects to close families, friends and communities, making the prevention of attempts vital. Detection of attempts before they occur could save numerous lives, and help prevent the large tolls that attempts take

For this project, my aim was to leverage machine learning to detect suicidal thought, and assess multiple models and their ability to detect thought. The dataset I used was taken from this Kaggle dataset, which is a collection of text posts scraped from 2 different subreddits called "SuicideWatch" and "depression". All posts that were made to "SuicideWatch" from December 16, 2008 (creation of the subreddit) until January 2, 2021, were collected while "depression" posts were collected from January 1, 2009, to January 2, 2021.

In this dataset, posts that are collected from the SuicideWatch subreddit were categorized as suicidal, while those from the depression subreddit were categorized as non-suicidal. The intuition behind this is that texts from either subreddit would be closer in general tone and vocabulary than a random text scraped from another subreddit. This makes the task more difficult, but also gives credit to the classification ability of our machine learning methods, as they are able to distinguish texts indicating suicidal ideation from

## at that point. **Model Evaluation** y\_pred=DNN.predict(X\_test) auc = roc\_auc\_score(y\_test,y\_pred) tn, fp, fn, tp = confusion\_matrix(y\_test, y\_pred>0.5).ravel() sensitivity = tp / (tp+fn)NPV = tn/(tn+fn)accuracy = accuracy\_score(y\_test,y\_pred>0.5) DNN\_metrics=['DNN', accuracy, auc, sensitivity, NPV] y pred=LR.predict(X test) auc = roc auc score(y test, y pred) tn, fp, fn, tp = confusion\_matrix(y\_test, y\_pred>0.5).ravel() sensitivity = tn / (tn+fp)NPV = tn/(tn+fn)accuracy = accuracy\_score(y\_test,y\_pred>0.5) LR metrics=['LR', accuracy, auc, sensitivity, NPV] y\_pred=rf.predict(X\_test) auc = roc\_auc\_score(y\_test,y\_pred) tn, fp, fn, tp = confusion\_matrix(y\_test, y\_pred>0.5).ravel() sensitivity = tn / (tn+fp)NPV = tn/(tn+fn)accuracy = accuracy\_score(y\_test,y\_pred>0.5) RF metrics=['RF', accuracy, auc, sensitivity, NPV] results\_df=pd.DataFrame([DNN\_metrics,LR\_metrics,RF\_metrics],columns=['Model','Accuracy The metrics I used to assess the performance of my models include accuracy, specificity, negative predictive value, and area under the receiver-operator curve (AUROC). Accuracy and AUROC are both standard metrics used to assess model performance in a binary classification setting. However, I also decided to assess sensitivity and negative predictive value, since having false negative in this case could mean a lost suicide attempt, and is therefore a priority to avoid. melted\_df=pd.melt(results\_df,id\_vars="Model") sns.set(style="darkgrid") plt.figure(figsize=(8, 16)) p=sns.catplot(x="variable", y="value", hue="Model", data=melted df, kind="bar", height=4, aspect=3.0); \_=p.set(title="Model performance of Random Forests, Elastic Net and Deep Neural Network xlabel="Metric", ylabel="Metric value") <Figure size 576x1152 with 0 Axes> Model performance of Random Forests, Elastic Net and Deep Neural Network for predicting suicidality in text samples 0.8 o.6 Metric 0.4 0.2 0.0 Accuracy Based on the above results, it seems like the best method differs depending on the metric used. While the deep neural network performed best in terms of accuracy and AUROC, when taking into account the

Model DNN

Negative Predictive Value

AUROC

performs the best for this task.

Sensitivity

Metric

importance of sensitivity and negative predictive value, the DNN falls behind other methods. For example,

preventing false negatives, I would argue that maximizing negative predictive value should be the most import indicator of the model's performance. with respect to this metric, the logistic regression model

improvements to this project however. Firstly, the data scraped was simply taken from both subreddits, and not screened nor classified with any systematic rigor. As a result, there may have been some cases where an individual in the depression subreddit expressed suicidal ideation and that text was misclassified, or vice versa. Additionally, the feature engineering was relatively simple (just counting the frequency of words), and there many other ways of encoding text that could be explored. Finally, the models used here are

logistic regression has the highest negative predictive value, while random forests has the highest sensitivity. Thus, it is difficult to say which model is necessarily the best as no model seems to totally outclass the others, and each model has metrics where it performs well. Given the importance of

Overall, all models perform well in their ability to classify the texts. There could be a number of

relatively simple, and there are other architectures that could have been used such as RNNs, or

autoencoders, that are more complex and may perform better.

LR