Code, Software Used And Execution Instructions

Code:

```
import warnings
warnings.filterwarnings("ignore")

import ftfy
import matplotlib.pyplot as plt
import nltk
from nltk.tokenize import word_tokenize
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np
import pandas as pd
import re
import matplotlib.pyplot as plt
from wordcloud import WordCloud

from math import exp
from numpy import sign
from sklearn.metrics import  classification_report, confusion_matrix, accuracy_score
from gensim.models import KeyedVectors
from nltk.corpus import stopwords
from nltk import PorterStemmer


from keras.models import Model, Sequential
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.layers import Conv1D, Dense, Input, LSTM, Embedding, Dropout, Activation,
MaxPooling1D
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import plot_model

np.random.seed(1234)

DEPRES_NROWS = 3200  # number of rows to read from DEPRESSIVE_TWEETS_CSV
RANDOM_NROWS = 12000  # number of rows to read from RANDOM_TWEETS_CSV
MAX_SEQUENCE_LENGTH = 140  # Max tweet size
MAX_NB_WORDS = 20000
EMBEDDING_DIM = 300
TRAIN_SPLIT = 0.6
TEST_SPLIT = 0.2
LEARNING_RATE = 0.1
EPOCHS = 10
```

Section 1 : Load Data

Loading depressive tweets scraped from twitter using TWINT and random tweets from Kaggle dataset twitter_sentiment.

In [3]:

```
DEPRESSIVE_TWEETS_CSV =(r'C:\Users\91817\Major Project\depressive_tweets_processed.csv')
RANDOM_TWEETS_CSV = (r'C:\Users\91817\Major Project\Sentiment Analysis Dataset 2.csv')

depressive_tweets_df = pd.read_csv(DEPRESSIVE_TWEETS_CSV, sep='|', header=None, usecols=range(0, 9)
, nrows=DEPRES_NROWS)
random_tweets_df = pd.read_csv(RANDOM_TWEETS_CSV, encoding="ISO-8859-1", usecols=range(0, 4), nr
ows=RANDOM_NROWS)

EMBEDDING_FILE = (r'C:\Users\91817\Downloads\GoogleNews-vectors-negative300.bin.gz')
```

In [4]:

```
depressive_tweets_df.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8
0	989292962323615744	2018-04-25	23:59:57	Eastern Standard Time	whosalli	The lack of this understanding is a small but ...	1	0	3
1	989292959844663296	2018-04-25	23:59:56	Eastern Standard Time	estermnunes	i just told my parents about my depression and...	1	0	2
2	989292951716155392	2018-04-25	23:59:54	Eastern Standard Time	TheAlphaAries	depression is something i don't speak about ev...	0	0	0
3	989292873664393218	2018-04-25	23:59:35	Eastern Standard Time	_ojhodgson	Made myself a tortilla filled with pb&j. My de...	1	0	0
4	989292856119472128	2018-04-25	23:59:31	Eastern Standard Time	DMiller96371630	@WorldofOutlaws I am gonna need depression med...	0	0	0

In [5]:

```
random_tweets_df.head()
```

Out[5]:

	ï»¿ItemID	Sentiment	SentimentSource	SentimentText
0	1	0	Sentiment140	is so sad for my APL frie...
1	2	0	Sentiment140	I missed the New Moon trail...
2	3	1	Sentiment140	omg its already 7:30 :O
3	4	0	Sentiment140	.. Omgaga. Im sooo im gunna CRy. I'...
4	5	0	Sentiment140	i think mi bf is cheating on me!!! ...

In [6]:

```
word2vec = KeyedVectors.load_word2vec_format(EMBEDDING_FILE, binary=True)
```

Data Processing

Load Pretrained Word2Vec Model

The pretrained vectors for the Word2Vec model is from here. Using a Keyed Vectors file, we can get the embedding of any word by calling .word_vec(word) and we can get all the words in the model's vocabulary through .vocab.

Preprocessing

Preprocessing the tweets in order to:

- Remove links and images
- Remove hashtags
- Remove @ mentions
- Remove emojis
- Remove stop words
- Remove punctuation
- Get rid of stuff like "what's" and making it "what is'
- Stem words so they are all the same tense (e.g. ran -> run)

In [7]:

```
# Expand Contraction

cList = {
  "ain't": "am not",
  "aren't": "are not",
  "can't": "cannot",
  "can't've": "cannot have",
  "'cause": "because",
  "could've": "could have",
  "couldn't": "could not",
```

```
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it had",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
```

```
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there had",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we had",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
```

```
  "won't": "will not",
  "won't've": "will not have",
  "would've": "would have",
  "wouldn't": "would not",
  "wouldn't've": "would not have",
  "y'all": "you all",
  "y'alls": "you alls",
  "y'all'd": "you all would",
  "y'all'd've": "you all would have",
  "y'all're": "you all are",
  "y'all've": "you all have",
  "you'd": "you had",
  "you'd've": "you would have",
  "you'll": "you you will",
  "you'll've": "you you will have",
  "you're": "you are",
  "you've": "you have"
}

c_re = re.compile('(%s)' % '|'.join(cList.keys()))

def expandContractions(text, c_re=c_re):
    def replace(match):
        return cList[match.group(0)]
    return c_re.sub(replace, text)
```

In [8]:

```
def clean_tweets(tweets):
    cleaned_tweets = []
    for tweet in tweets:
        tweet = str(tweet)
        # if url links then dont append to avoid news articles
        # also check tweet length, save those > 10 (length of word "depression")
        if re.match("(\w+:\/\/\S+)", tweet) == None and len(tweet) > 10:
            # remove hashtag, @mention, emoji and image URLs
            tweet = ' '.join(
                re.sub("(@[A-Za-z0-9]+)|(\#[A-Za-z0-9]+)|(<Emoji:.*>)|(pic\.twitter\.com\/.*)", " ", tweet).split())

            # fix weirdly encoded texts
            tweet = ftfy.fix_text(tweet)

            # expand contraction
            tweet = expandContractions(tweet)

            # remove punctuation
```

```
        tweet = ' '.join(re.sub("([^0-9A-Za-z \t])", " ", tweet).split())

        # stop words
        stop_words = set(stopwords.words('english'))
        word_tokens = nltk.word_tokenize(tweet)
        filtered_sentence = [w for w in word_tokens if not w in stop_words]
        tweet = ' '.join(filtered_sentence)

        # stemming words
        tweet = PorterStemmer().stem(tweet)

        cleaned_tweets.append(tweet)

    return cleaned_tweets
```

In [9]:

```
def batch_clean_tweets(tweets):
    cleaned_tweets = []
    for tweet in tweets:
        tweet = str(tweet)
        if re.match("(\w+:\/\/\S+)", tweet) == None and len(tweet) > 10:
            tweet = ' '.join(
                re.sub("(@[A-Za-z0-9]+)|(\#[A-Za-z0-9]+)|(<Emoji:.*>)|(pic\.twitter\.com\/.*)", " ", tweet).split())
            tweet = ftfy.fix_text(tweet)
            tweet = expandContractions(tweet)
            tweet = ' '.join(re.sub("([^0-9A-Za-z \t])", " ", tweet).split())
            stop_words = set(stopwords.words('english'))
            word_tokens = nltk.word_tokenize(tweet)
            filtered_sentence = [w for w in word_tokens if not w in stop_words]
            tweet = ' '.join(filtered_sentence)
            tweet = PorterStemmer().stem(tweet)
            cleaned_tweets.append(tweet)
    return cleaned_tweets

depressive_tweets_arr = [x for x in depressive_tweets_df[5]]
random_tweets_arr = [x for x in random_tweets_df['SentimentText']]
X_d = clean_tweets(depressive_tweets_arr)
X_r = clean_tweets(random_tweets_arr)
```

In [10]:

```
depressive_tweets_arr = [x for x in depressive_tweets_df[5]]
random_tweets_arr = [x for x in random_tweets_df['SentimentText']]
X_d = clean_tweets(depressive_tweets_arr)
X_r = clean_tweets(random_tweets_arr)
```

Tokenizer

Using a Tokenizer to assign indices and filtering out unfrequent words. Tokenizer creates a map of every unique word and an assigned index to it. The parameter called num_words indicates that we only care about the top 20000 most frequent words.

In [11]:

```
# Tokenization
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(X_d + X_r)

sequences_d = tokenizer.texts_to_sequences(X_d)
sequences_r = tokenizer.texts_to_sequences(X_r)

word_index = tokenizer.word_index
print('Found %s unique tokens' % len(word_index))
```

```
Found 21548 unique tokens
```

Pad sequences all to the same length of 140 words.

In [12]:

```
# Padding sequences
data_d = pad_sequences(sequences_d, maxlen=MAX_SEQUENCE_LENGTH)
data_r = pad_sequences(sequences_r, maxlen=MAX_SEQUENCE_LENGTH)
print('Shape of data_d tensor:', data_d.shape)
print('Shape of data_r tensor:', data_r.shape)
```

```
Shape of data_d tensor: (2308, 140)
Shape of data_r tensor: (11911, 140)
```

In [13]:

```
def batch_tokenize_and_pad(X, tokenizer, max_sequence_length):
    sequences = tokenizer.texts_to_sequences(X)
    data = pad_sequences(sequences, maxlen=max_sequence_length)
    return data

# Batch Processing for Depressive Tweets
batch_size = 1000
num_batches = len(depressive_tweets_arr) // batch_size
if len(depressive_tweets_arr) % batch_size != 0:
    num_batches += 1

cleaned_depressive_tweets = []
for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = min((i + 1) * batch_size, len(depressive_tweets_arr))
    batch_tweets = depressive_tweets_arr[start_idx:end_idx]
    cleaned_batch_tweets = batch_clean_tweets(batch_tweets)
    cleaned_depressive_tweets.extend(cleaned_batch_tweets)
```

```
data_d = batch_tokenize_and_pad(cleaned_depressive_tweets, tokenizer, MAX_SEQUENCE_LENGTH)
```

Embedding Matrix

The embedding matrix is a n x m matrix where n is the number of words and m is the dimension of the embedding. In this case, m=300 and n=20000. We take the min between the number of unique words in our tokenizer and max words in case there are less unique words than the max we specified.

In [14]:

```
# Determine the number of words to consider
nb_words = min(MAX_NB_WORDS, len(word_index))

# Creating an embedding matrix
embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))

# Populate the embedding matrix with word vectors
for word, idx in word_index.items():
    if word in word2vec.key_to_index and idx < MAX_NB_WORDS:
        embedding_matrix[idx] = word2vec.get_vector(word)
```

In [15]:

```
#random tweets word cloud

# Join all cleaned random tweets into a single string
all_random_words = ' '.join(X_r)

# Generate the word cloud
wordcloud = WordCloud(background_color='white', colormap='jet', width=800, height=500, random_state=21,
max_font_size=110).generate(all_random_words)

# Plot the word cloud
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

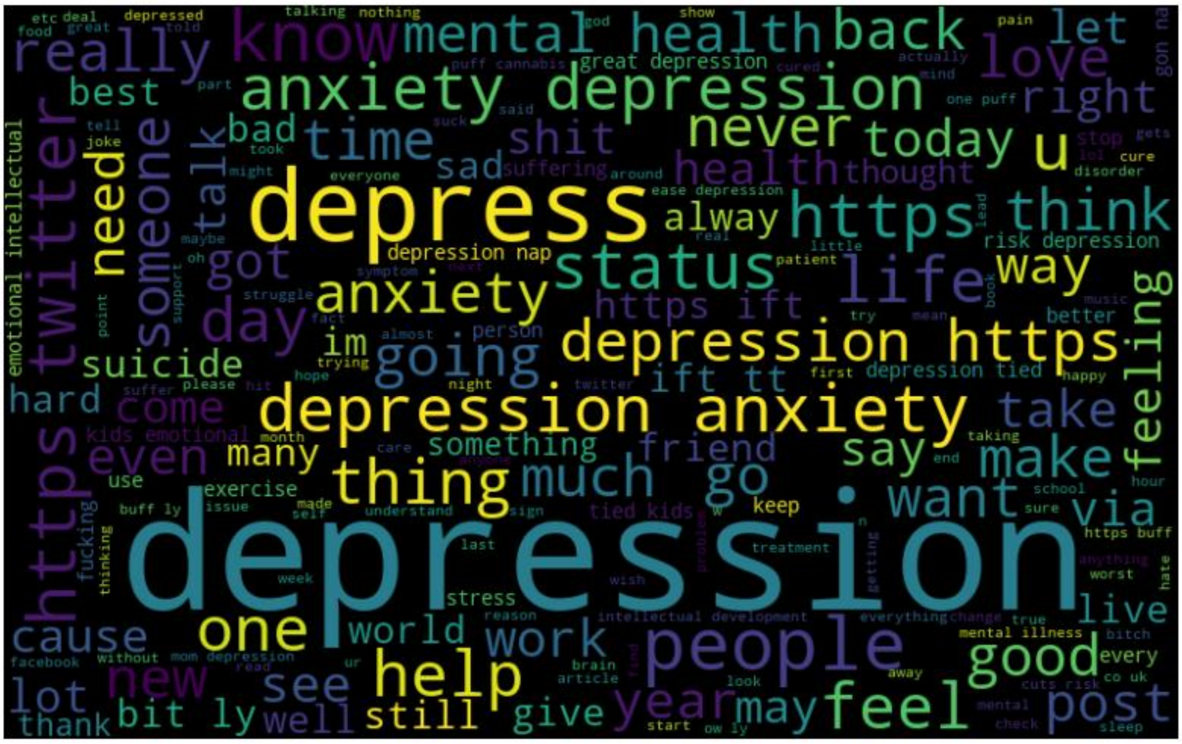
In [16]:

#depressed tweets word cloud

 # Join all cleaned depressive tweets into a single string
all_depressive_words = ' '.join(X_d)

 # Generate the word cloud
wordcloud = WordCloud(width=800, height=500, random_state=21, max_font_size=110).generate(all_depressi
ve_words)

 # Plot the word cloud
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
plt.show()



Splitting and Formatting Data

Assigning labels to the depressive tweets and random tweets data, and splitting the arrays into test (60%), validation (20%), and train data (20%). Combine depressive tweets and random tweets arrays and shuffle.

In [17]:

```
# Assigning labels to the depressive tweets and random tweets data
labels_d = np.array([1] * DEPRES_NROWS)
labels_r = np.array([0] * RANDOM_NROWS)
```

In [18]:

```
# Assigning labels to the depressive tweets and random tweets data

perm_d = np.random.permutation(len(data_d))
idx_train_d = perm_d[:int(len(data_d)*(TRAIN_SPLIT))]
idx_test_d = perm_d[int(len(data_d)*(TRAIN_SPLIT)):int(len(data_d)*(TRAIN_SPLIT+TEST_SPLIT))]
idx_val_d = perm_d[int(len(data_d)*(TRAIN_SPLIT+TEST_SPLIT)):]

perm_r = np.random.permutation(len(data_r))
idx_train_r = perm_r[:int(len(data_r)*(TRAIN_SPLIT))]
idx_test_r = perm_r[int(len(data_r)*(TRAIN_SPLIT)):int(len(data_r)*(TRAIN_SPLIT+TEST_SPLIT))]
idx_val_r = perm_r[int(len(data_r)*(TRAIN_SPLIT+TEST_SPLIT)):]

# Combine depressive tweets and random tweets arrays
data_train = np.concatenate((data_d[idx_train_d], data_r[idx_train_r]))
labels_train = np.concatenate((labels_d[idx_train_d], labels_r[idx_train_r]))
data_test = np.concatenate((data_d[idx_test_d], data_r[idx_test_r]))
```

```
labels_test = np.concatenate((labels_d[idx_test_d], labels_r[idx_test_r]))
data_val = np.concatenate((data_d[idx_val_d], data_r[idx_val_r]))
labels_val = np.concatenate((labels_d[idx_val_d], labels_r[idx_val_r]))

#Shuffling
perm_train = np.random.permutation(len(data_train))
data_train = data_train[perm_train]
labels_train = labels_train[perm_train]
perm_test = np.random.permutation(len(data_test))
data_test = data_test[perm_test]
labels_test = labels_test[perm_test]
perm_val = np.random.permutation(len(data_val))
data_val = data_val[perm_val]
labels_val = labels_val[perm_val]
```

Section 3 : Building the Model

Building Model (LSTM + CNN)

The model takes in an input and then outputs a single number representing the probability that the tweet indicates depression. The model takes in each input sentence, replace it with it's embeddings, then run the new embedding vector through a convolutional layer. CNNs are excellent at learning spatial structure from data, the convolutional layer takes advantage of that and learn some structure from the sequential data then pass into a standard LSTM layer. Last but not least, the output of the LSTM layer is fed into a standard Dense model for prediction.

In [19]:

```
model = Sequential()
# Embedded layer
model.add(Embedding(len(embedding_matrix), EMBEDDING_DIM, weights=[embedding_matrix],
                    input_length=MAX_SEQUENCE_LENGTH, trainable=False))
# Convolutional Layer
model.add(Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.2))
# LSTM Layer
model.add(LSTM(300))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Compiling Model

In [20]:

```
model.compile(loss='binary_crossentropy', optimizer='nadam', metrics=['acc'])
print(model.summary())
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 140, 300)          6000000

 conv1d (Conv1D)             (None, 140, 32)           28832
```

```
 max_pooling1d (MaxPooling1  (None, 70, 32)           0
 D)

 dropout (Dropout)          (None, 70, 32)           0

 lstm (LSTM)                (None, 300)              399600

 dropout_1 (Dropout)        (None, 300)              0

 dense (Dense)              (None, 1)                301

=================================================================
Total params: 6428733 (24.52 MB)
Trainable params: 428733 (1.64 MB)
Non-trainable params: 6000000 (22.89 MB)
_________________________________________________________________
None
```

Section 4 : Training the Model

The model is trained EPOCHS time, and Early Stopping argument is used to end training if the loss or accuracy don't improve within 3 epochs.

In [21]:

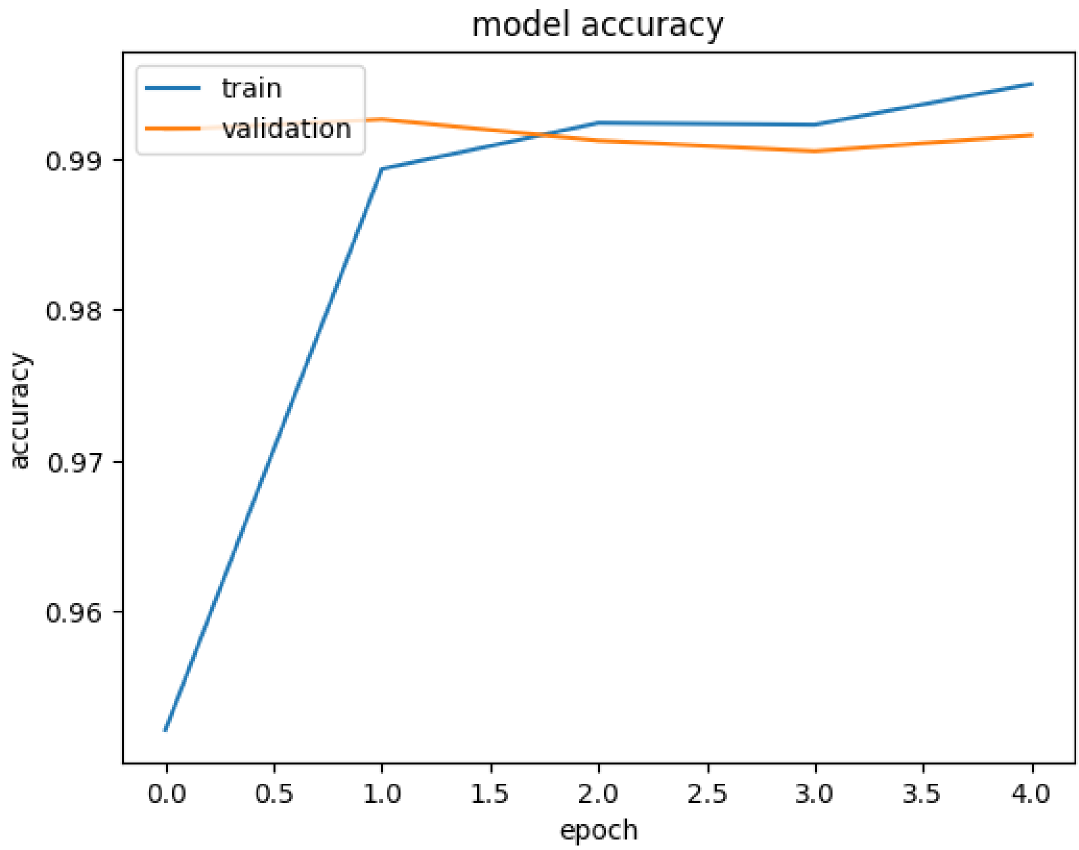
```
early_stop = EarlyStopping(monitor='val_loss', patience=3)

hist = model.fit(data_train, labels_train, \
                 validation_data=(data_val, labels_val), \
                 epochs=EPOCHS, batch_size=40, shuffle=True, \
                 callbacks=[early_stop])
```

```
Epoch 1/10
214/214 [==============================] - 101s 432ms/step - loss: 0.1420 - acc: 0.9522 - val_loss: 0
.0363 - val_acc: 0.9919
Epoch 2/10
214/214 [==============================] - 88s 410ms/step - loss: 0.0422 - acc: 0.9893 - val_loss: 0.
0302 - val_acc: 0.9926
Epoch 3/10
214/214 [==============================] - 82s 381ms/step - loss: 0.0305 - acc: 0.9924 - val_loss: 0.
0354 - val_acc: 0.9912
Epoch 4/10
214/214 [==============================] - 88s 411ms/step - loss: 0.0280 - acc: 0.9923 - val_loss: 0.
0357 - val_acc: 0.9905
Epoch 5/10
214/214 [==============================] - 88s 412ms/step - loss: 0.0207 - acc: 0.9950 - val_loss: 0.
0350 - val_acc: 0.9916
```

Section 5 : Results

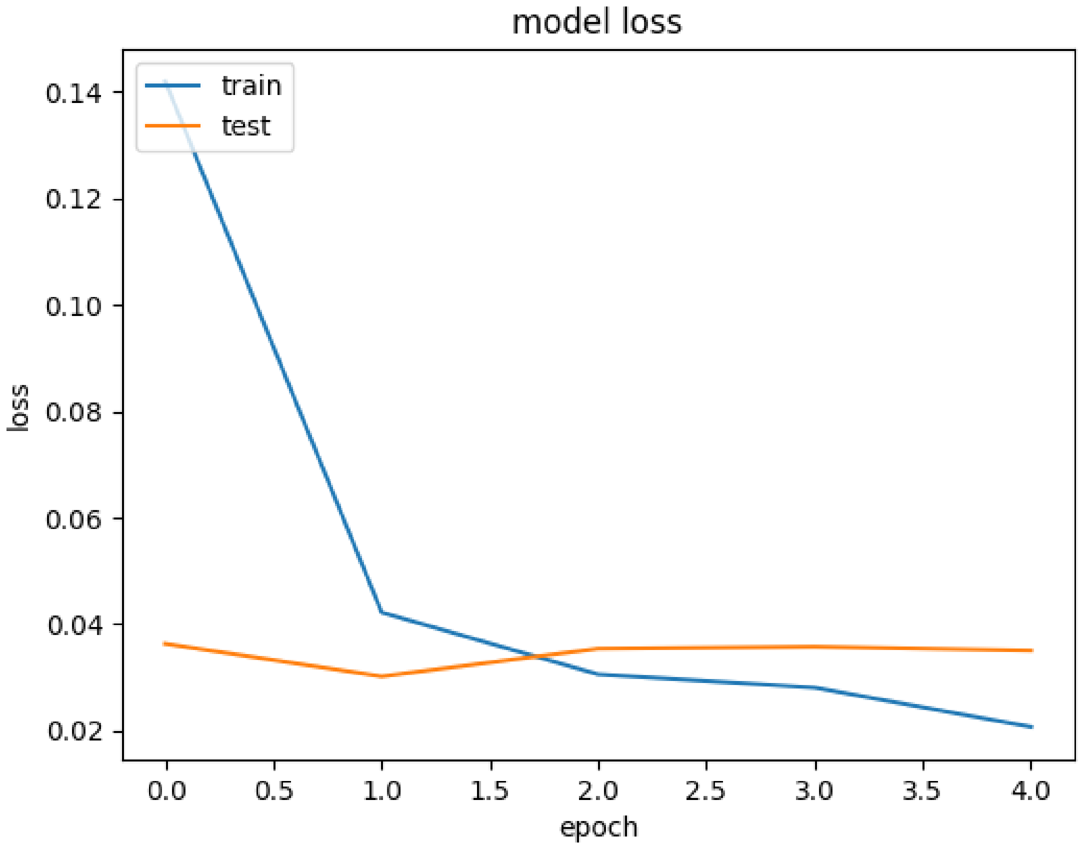
In [22]:

```
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



In [23]:

```
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Percentage accuracy of model

In [24]:

```
labels_pred = model.predict(data_test)
labels_pred = np.round(labels_pred.flatten())
accuracy = accuracy_score(labels_test, labels_pred)
print("Accuracy: %.2f%%" % (accuracy*100))
```

89/89 [==============================] - 10s 101ms/step
Accuracy: 99.02%

In [25]:

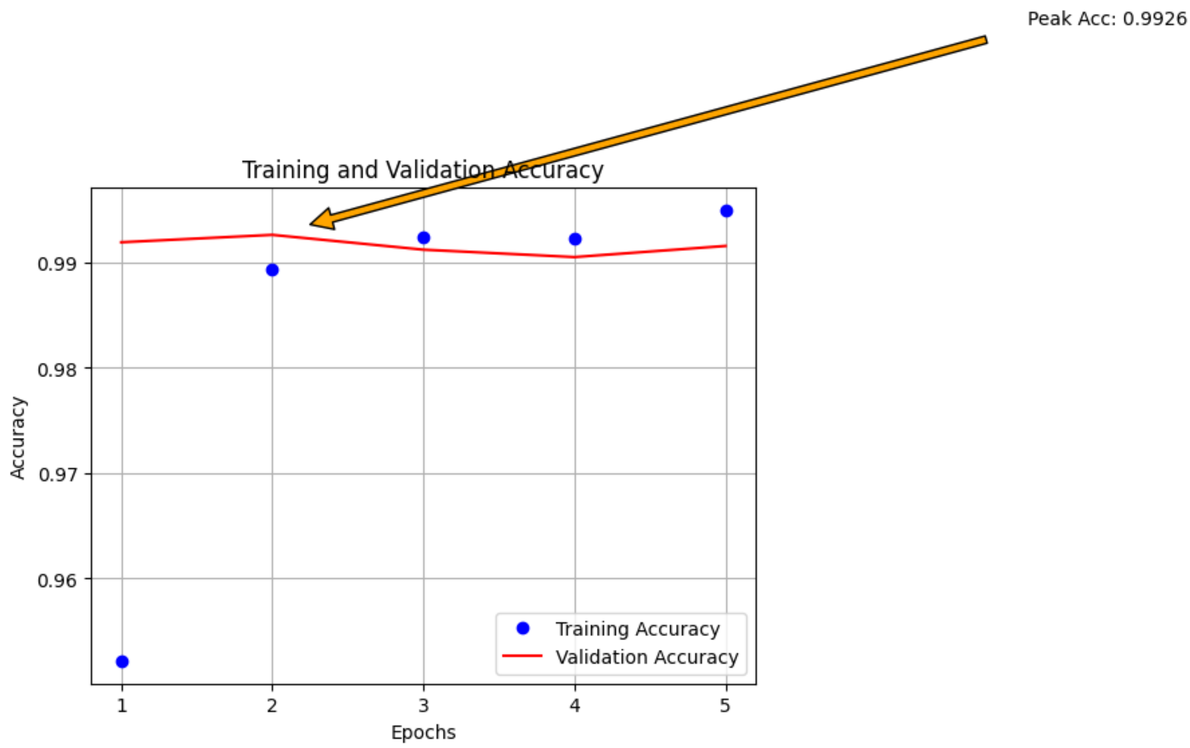
```
print(classification_report(labels_test, labels_pred))
```

	precision	recall	f1-score	support
0	0.99	1.00	0.99	2382
1	0.98	0.95	0.97	462
accuracy			0.99	2844
macro avg	0.99	0.98	0.98	2844
weighted avg	0.99	0.99	0.99	2844

In [26]:

```
history = hist

acc = history.history['acc']
val_acc = history.history['val_acc']
```

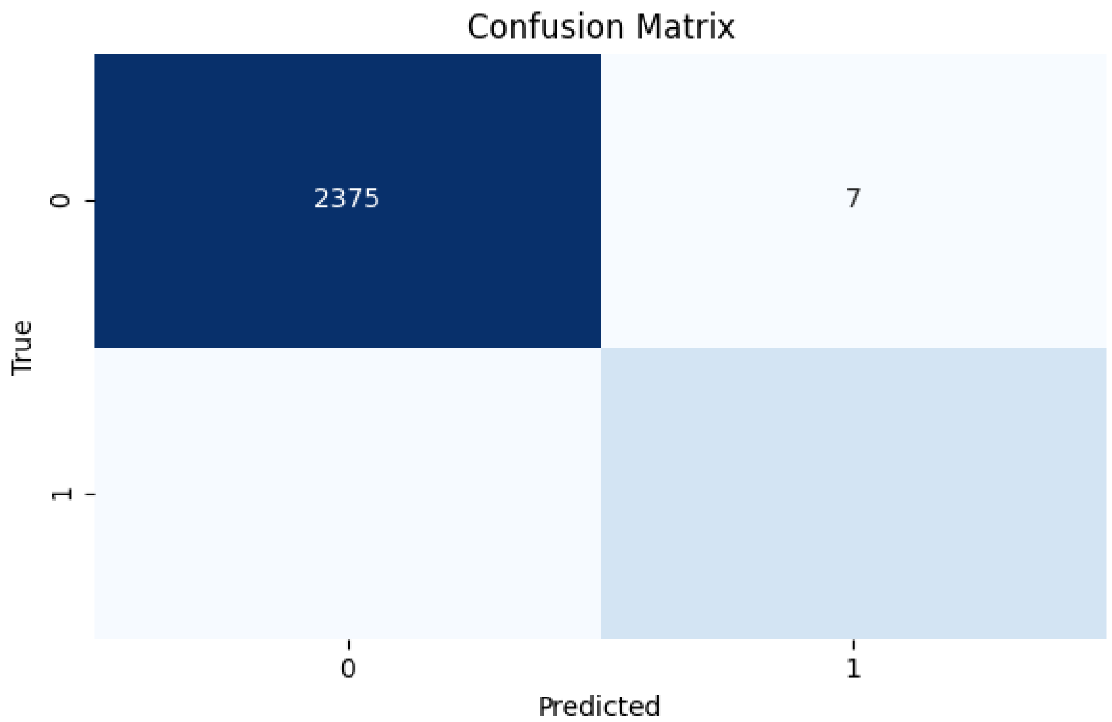
```
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training Accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)  # Adding grid

# Annotating the maximum validation accuracy point
y_arrow_acc = max(val_acc)
x_arrow_acc = val_acc.index(y_arrow_acc) + 1
plt.annotate(f'Peak Acc: {str(round(y_arrow_acc, 4))}',
             (x_arrow_acc, y_arrow_acc),
             xytext=(x_arrow_acc + 5, y_arrow_acc + .02),
             arrowprops=dict(facecolor='orange', shrink=0.05))

plt.xticks(epochs)

plt.figure()
plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)  # Adding grid

# Annotating the maximum validation loss point
y_arrow_loss = max(val_loss)
x_arrow_loss = val_loss.index(y_arrow_loss) + 1
plt.annotate(f'Peak Loss: {str(round(y_arrow_loss, 4))}',
             (x_arrow_loss, y_arrow_loss),
             xytext=(x_arrow_loss + 5, y_arrow_loss + .02),
             arrowprops=dict(facecolor='orange', shrink=0.05))

plt.xticks(epochs)
plt.show()
```

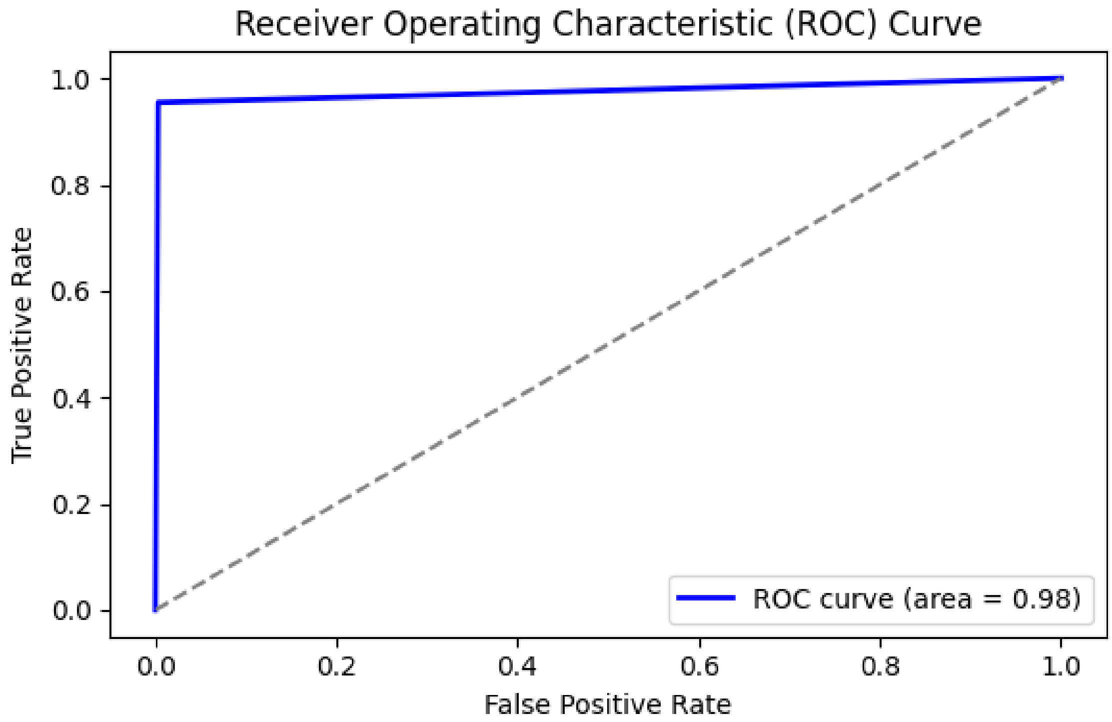


In [28]:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix, roc_auc_score, roc_cur
ve

# Calculate precision, recall, F1-score, confusion matrix, ROC-AUC score
precision = precision_score(labels_test, labels_pred)
recall = recall_score(labels_test, labels_pred)
```

```python
f1 = f1_score(labels_test, labels_pred)
conf_matrix = confusion_matrix(labels_test, labels_pred)
roc_auc = roc_auc_score(labels_test, labels_pred)

# Plot confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()

# Plot ROC curve
fpr, tpr, _ = roc_curve(labels_test, labels_pred)
plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (area = {:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

# Display metrics in tabular form
metrics_table = pd.DataFrame({
    'Metric': ['Precision', 'Recall', 'F1-Score', 'ROC-AUC Score'],
    'Value': [precision, recall, f1, roc_auc]
})
print(metrics_table)
```





	Metric	Value
0	Precision	0.984375
1	Recall	0.954545
2	F1-Score	0.969231
3	ROC-AUC Score	0.975803

Software Used:

numpy==1.23.0

pandas==1.5.0 scikit-

learn==1.1.2

streamlit==1.13

watchdog==2.1.9

xgboost==1.6.2

matplotlib==3.6.0

Streamlit