DEEP LEARNING – SE4050

ASSIGNMENT 02

B.Sc. (Hons) Degree in Information Technology

Department of Computer Science and Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

Group Members

Student ID	Name	Email
IT18149654	Rajapaksha T.N.	it18149654@my.sliit.lk
IT18148350	Naidabadu N.I.	it18148350@my.sliit.lk
IT18149272	Perera M.J.F.R.	it18149272@my.sliit.lk

Problem

Short Message Service (SMS) has become one of the most common and heavily used ways of communication in society today. According to the research conducted by Ghourabi, Mahmood, and Alzubi, mobile phone users around the world have sent more than 8.3 trillion SMS messages in the year 2017 [1]. Furthermore, the number of SMS messages sent monthly is more than 690 billion [1].

However, SMS spam has been widely spreading among most mobile phone users recently. Any undesired or unsolicited text message sent indiscriminately to your mobile phone, usually for commercial motives, is referred to as SMS spam [1, 2, 3]. According to a survey conducted in the research paper [1], more than 68% of mobile phone users around the globe are affected by SMS spam messages.

Most people face many difficulties in their day-to-day lives due to SMS spam messages. Spams annoy most people as their valuable time is wasted and their workflow is interrupted and disturbed when they have to go through unwanted messages [3]. Additionally, the important text messages can be missed due to the inbox being filled with unwanted spam SMS messages. Furthermore, these SMS spams waste network resources of the device [3]. Therefore, SMS spam can cause significant negative sociological and economical effects.

Furthermore, in many cases, SMS spams consist of malicious activities such as smishing (SMS + phishing) which is a cyber security threat for mobile users aimed at deceiving them via SMS spam messages that may include a link or malicious software [1]. Cyber-attackers are trying to steal users' secret and sensitive information such as credit card numbers, passwords, and bank account details [1] using these malicious SMS spam messages. Many individuals and even major organizations suffer huge financial losses due to the cyber security threats caused by SMS spam messages [1].

The filtration of SMS spam in smartphones is still not very robust compared to the filtration of email spam detection [1]. Identification of text spam messages is also proven to be a very hard and time-consuming task according to most research [1, 2, 3]. Most existing lexicon-based methodologies as well as traditional NLP (Natural Language Processing) and Machine Learning algorithms are not accurate and efficient enough. Therefore, addressing this real-world problem and coming up with a reliable technique to classify spam SMS messages from ham SMS messages can be very useful.

Dataset

Background

Link: http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection

The dataset which we are going to use is the **SMS Spam Collection Dataset** which is hosted on UCI Machine Learning Repository [4, 5, 6]. The dataset has been published on the 22nd of June 2012 [4]. It contains 5,574 English SMS phone messages and each of these messages have already been labeled as either ham (legitimate) or spam [4, 6] as shown in Figure 1, Figure 2, and Figure 3. The main purpose of the SMS Spam Collection Dataset is to assist NLP and computational linguistics research focused on detecting mobile phone SMS text message spams [4].

	class	sms_message
0	ham	Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 0845281007...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives around here though
...	...	...
5567	spam	This is the 2nd time we have tried 2 contact u. U have won the £750 Pound prize. 2 claim is easy, call 087187272008 NOW1! Only 10p per minute. BT-...
5568	ham	Will ü b going to esplanade fr home?
5569	ham	Pity, * was in mood for that. So...any other suggestions?
5570	ham	The guy did some bitching but I acted like i'd be interested in buying something else next week and he gave it to us for free
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Figure 1: Sample data instances of SMS Spam Collection Dataset

The dataset has been constructed using multiple free resources from the internet. Out of the total of 5,574 messages, 425 SMS messages were extracted manually from the Grumbletext website [7] which is a forum for mobile phone users to discuss SMS spam messages. Further, 3,375 SMS ham messages have been chosen randomly from the NUS SMS Corpus (NSC) [8] which consists of about 10,000 ham SMS messages collected mostly from Singaporean students studying in the Department of Computer Science at the National University of Singapore [4]. Additionally, 450 SMS ham messages have been gathered from the Ph.D. thesis of Dr. Caroline Tagg [4]. Furthermore, 1,002 SMS ham messages and 322 spam messages have been accumulated from SMS Spam Corpus v.0.1 Big [4].

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   sms_message  5572 non-null   object
 1   class        5572 non-null   object
dtypes: object(2)
memory usage: 87.2+ KB
```

Figure 2: Concise summary of the dataframe

	sms_message	class
count	5572	5572
unique	5169	2
top	Sorry, I'll call later	ham
freq	30	4825

Figure 3: Descriptive statistics of the dataframe

Attributes

The **SMS Spam Collection Dataset** is composed of one text file consisting of only two columns.

1. SMS message text

This is the only input feature of the dataset. It contains the raw text of the mobile phone SMS message. The data type of this attribute is a string.

We have analyzed the text length distribution of each of these SMS messages in terms of the character count and plotted a graph as shown in Figure 4.

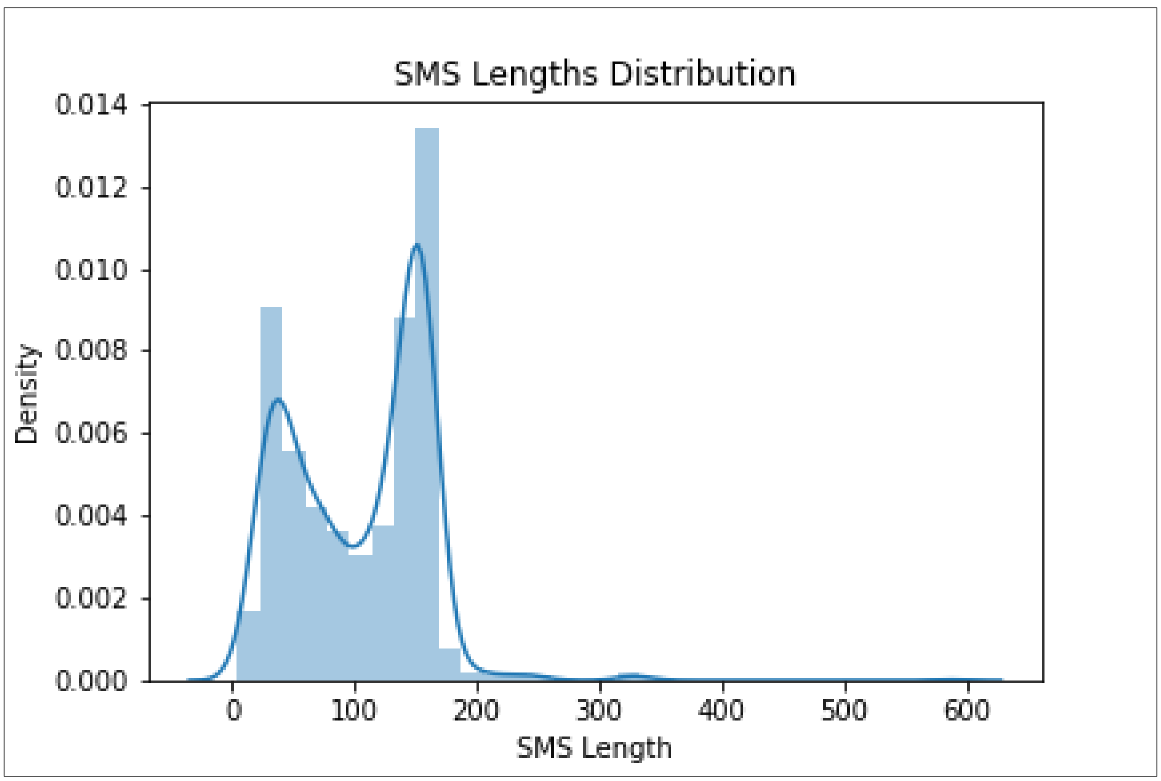

Figure 4: SMS lengths distribution plot

2. Label

This is the target attribute (class label) of the dataset. It has two distinct values: "ham" and "spam". The data type of this attribute is a string.

We have analyzed the data distribution of the target attribute of the dataset and plotted the distributions in a pie chart.

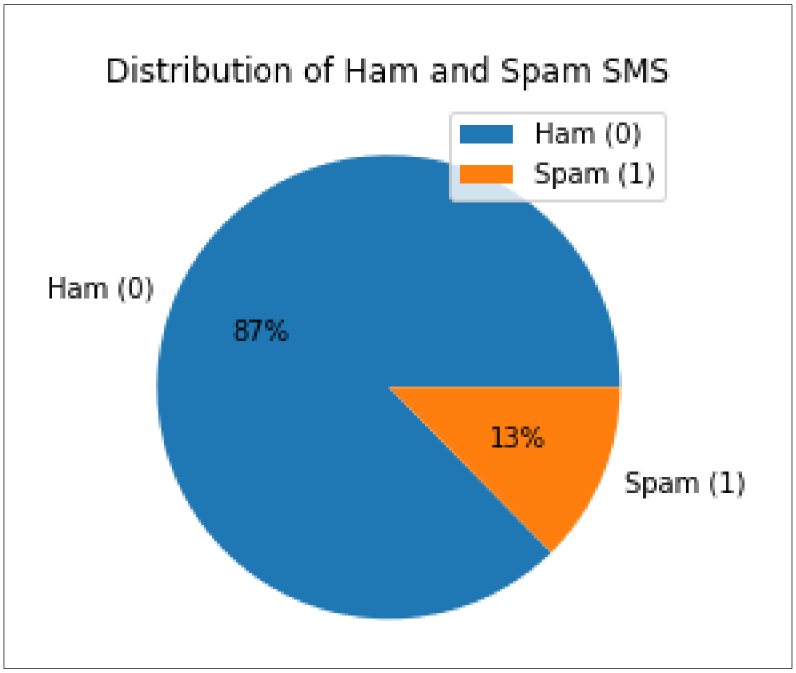

Figure 5: Distribution of ham and spam SMS plot

As shown in Figure 5, about 87% of SMS messages in the dataset are hams and the remaining 13% of SMS messages are spam. Therefore, the majority of the SMS messages in this dataset are labeled as ham as shown in Figure 6.

	class	ham	spam
sms_message	count	4825	747
	unique	4516	653
	top	Sorry, I'll call later	Please call our customer service representative on FREEPHONE 0808 145 4742 between 9am-11pm as you have WON a guaranteed £1000 cash or £5000 prize!
	freq	30	4

Figure 6: Descriptive statistics for each class of the dataframe

We further generated the Word Cloud plots for ham and spam SMS messages separately to visualize the most used words in ham and spam SMS messages as shown in Figure 3 and Figure 4 respectively.

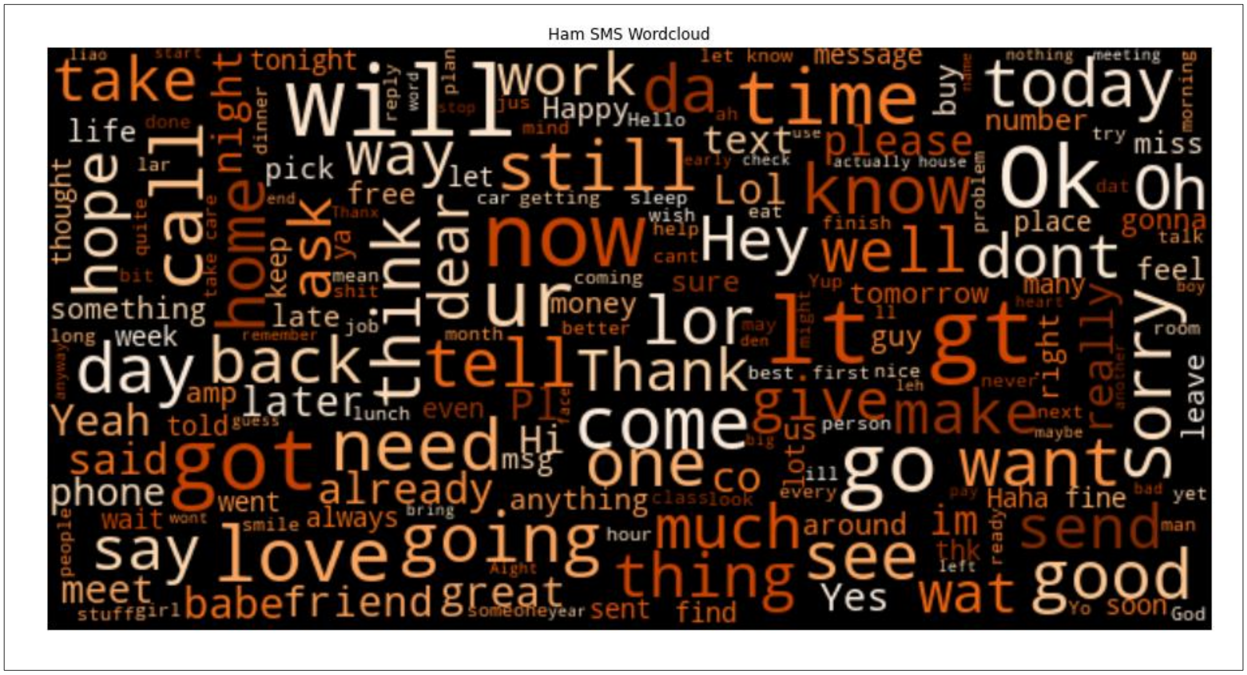

Figure 7: Ham SMS Word Cloud plot

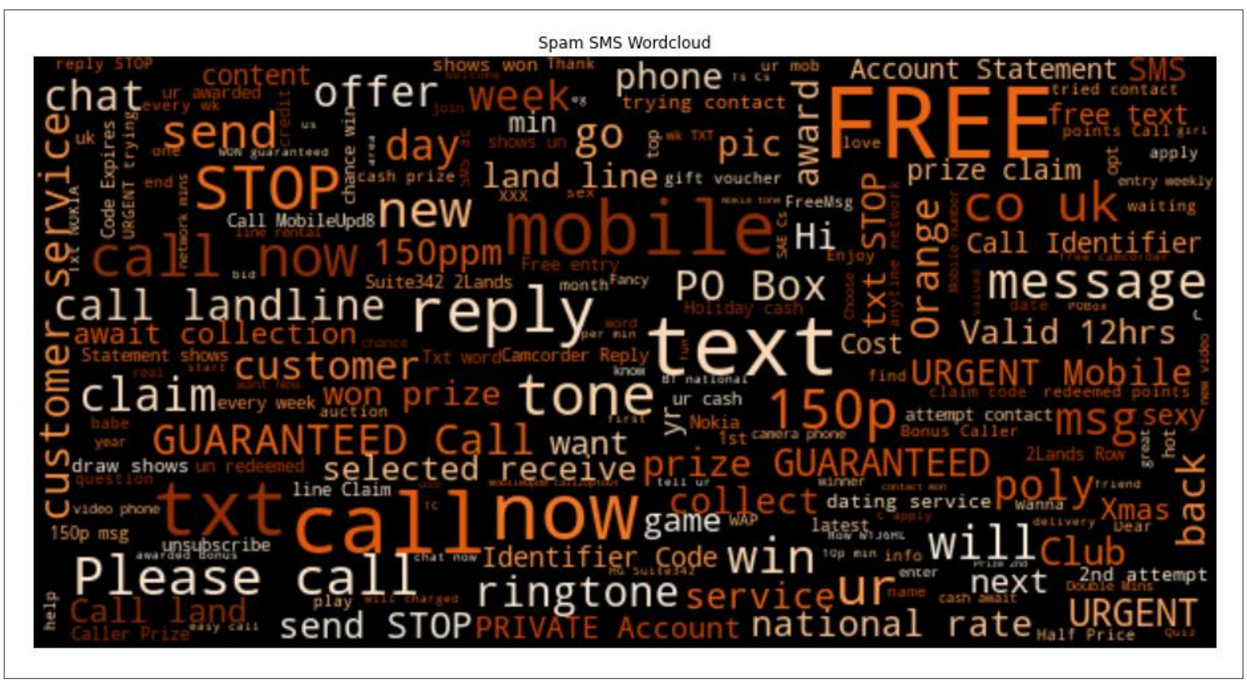

Figure 8: Spam SMS Word Cloud plot

Methodology

Data Cleaning and Preprocessing

Real-world data sets contain several problems such as missing or null values, data inconsistency, incompleteness, and outliers of the dataset. Therefore, data preprocessing is a mandatory step before feeding the dataset into a deep learning model. Data preprocessing includes transforming row data into the machine-understandable and efficient format. Since the SMS Spam Collection dataset may also contain some of these issues, we had to do preprocessing tasks before using the dataset for model training purposes.

Initially, the dataset was downloaded and extracted.

```
# downloading UCI SMS Spam Collection dataset
!wget --no-check-certificate https://archive.ics.uci.edu/ml/machine-
learning-databases/00228/smsspamcollection.zip

# extracting the downloaded dataset
!unzip /content/smsspamcollection.zip
```

Then the dataset was imported to a Pandas dataframe.

```
# importing SMSSpamCollection dataset to a pandas dataframe
sms_spam_dataframe = pd.read_csv('/content/SMSSpamCollection',
                                 sep='\t',
                                 header=None,
                                 names=['class', 'sms_message'])
```

Then the data was analyzed to check whether there are any missing or null values existed within the dataset. If any null values existed within the dataset, there are usually two ways to deal with them. The first method is to delete the particular rows which contain multiple null values. The second method is to replace the missing values with mean, median, or most frequent values. Although our analysis proved that the SMS Spam Collection dataset did not contain any missing or null values as shown in Figure 9, we implemented the null or missing values removal step programmatically.

```
sms_spam_dataframe =
sms_spam_dataframe[sms_spam_dataframe.notna().all(axis=1)]
```

```
# plotting the heatmap for missing or null values in the dataframe
sns.heatmap(sms_spam_dataframe.isnull(),
            yticklabels=False,
            cbar=False,
            cmap='viridis')
```

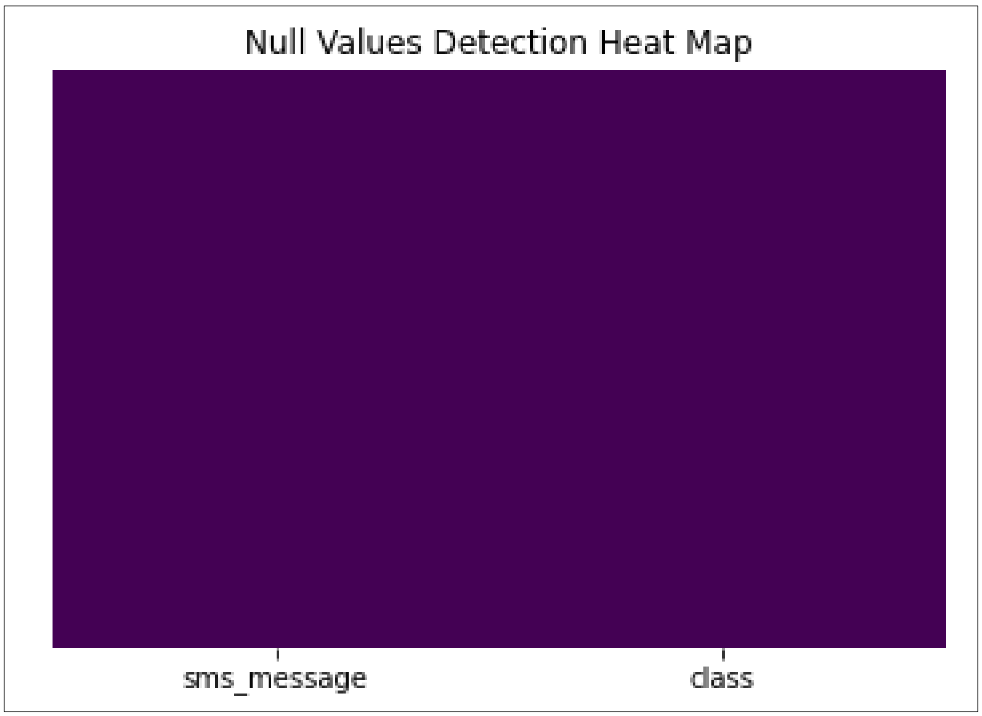

Figure 9: Null values detection heat map

Furthermore, the dataset was checked to identify duplicate rows.

```
# detecting duplicate rows exist in the dataframe before cleaning
duplicated_records = sms_spam_dataframe[sms_spam_dataframe.duplicated()]

# checking the number of duplicate rows exist in the dataframe
# before cleaning
sms_spam_dataframe.duplicated().sum()
```

Since there were 403 duplicates in the data frame, we implemented the duplicate row removal step.

```
# removing the duplicate rows from the dataframe if exist
sms_spam_dataframe = sms_spam_dataframe.drop_duplicates()
```

When we analyzed the data distribution of ham and spam SMS messages, it clearly shows that the data is imbalanced between the two classes as shown in Figure 10.

```
# printing count of values in each class of the dataframe
cleaned_sms_spam_dataframe['class'].value_counts()

ham      4516
spam      653
```

```
# plotting the distribution of target values
ax = cleaned_sms_spam_dataframe['class'].value_counts().plot(kind='pie',
                                                             labels=lbl,
                                                             autopct=pct)
```

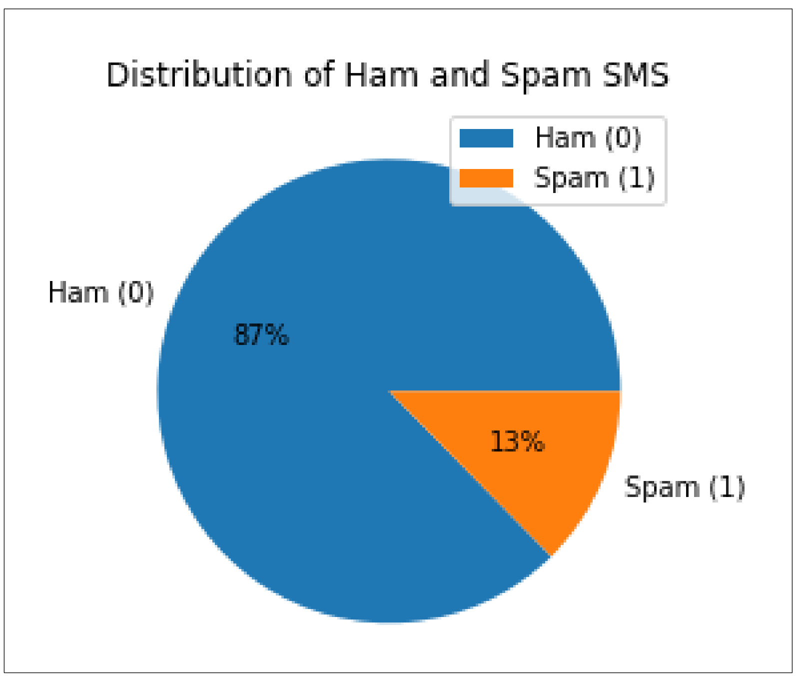

Figure 10: Distribution of ham and spam SMS plot before downsampling

To address this problem of imbalanced data, we decided to apply the downsampling technique which is a process where you randomly delete some of the observations from the majority class (ham) so that the numbers in majority and minority (spam) classes are matched. The dataset is separated into two dataframes based on the class label, and the majority class is downsampled.

```
# extracting the data instances with class label 'spam'
spam_dataframe =
cleaned_sms_spam_dataframe[cleaned_sms_spam_dataframe['class'] == 'spam']

# extracting the data instances with class label 'ham'
ham_dataframe =
cleaned_sms_spam_dataframe[cleaned_sms_spam_dataframe['class'] == 'ham']
```

```
# downsampling is a process where you randomly delete some of the
# observations from the majority class so that the numbers in majority
# and minority classes are matched
# after downsampling the ham messages (majority class), there are now
# 653 messages in each class
downsampled_ham_dataframe = ham_dataframe.sample(n=len(spam_dataframe),
                                                 random_state=44)
```

After applying downsampling and merging the two dataframes, there were the same amount of 653 messages for each class.

```
# merging the two dataframes (spam + downsampled ham dataframes)
merged_dataframe = pd.concat([downsampled_ham_dataframe, spam_dataframe])

# printing count of values in each class of the merged dataframe
merged_dataframe['class'].value_counts()
```

```
spam    653
ham     653
```

As shown in Figure 11, now the dataframe is balanced with 50% of data instances for each class.

```
# plotting the distribution of target values
ax = cleaned_sms_spam_dataframe['class'].value_counts().plot(kind='pie',
                                                             labels=lbl,
                                                             autopct=pct)
```

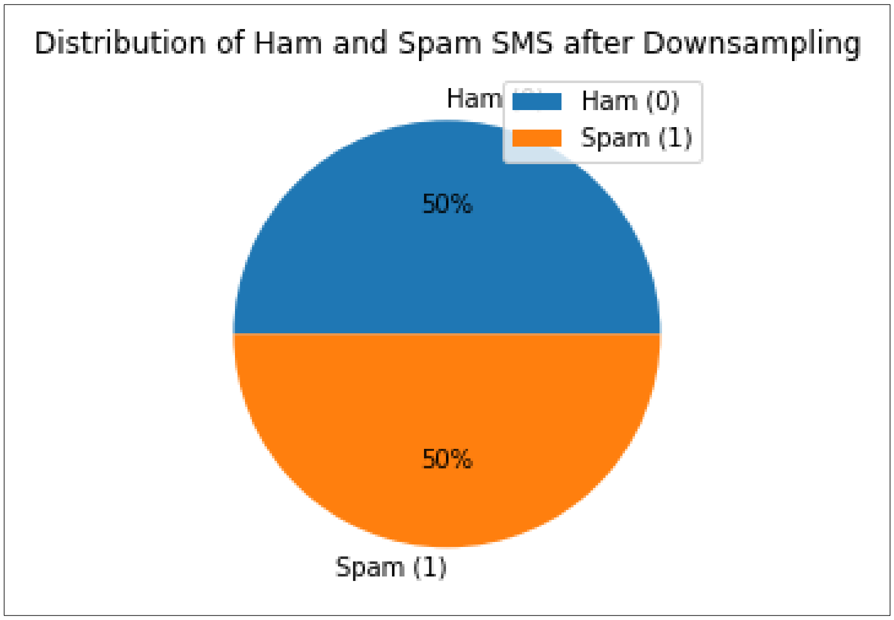

Figure 11: Distribution of ham and spam SMS plot after downsampling

Then we inserted a new column to the dataframe containing the length of the SMS message texts in terms of the number of characters.

```
# inserting a new column called 'length' to the merged dataframe
# the column contains the number of characters of the sms_message text
merged_dataframe['length'] = merged_dataframe['sms_message'].apply(len)
```

Then the text labels “ham” and “spam” were converted to numeric values 0 and 1 respectively as shown in Table 1.

```
# inserting a new column called 'label' to the merged dataframe
# if class is 'ham' label = 0
# if class is 'spam' label = 1
merged_dataframe['label'] = merged_dataframe['class'].map({'ham': 0,
'spam': 1})
```

Table 1: Numeric value for textual class labels

Class	Label
Ham	0
Spam	1

Then the dataset was split into train and test sets using the scikit-learn library [9]. The training set was 80% of the entire dataset while the test set was 20% of the entire dataset. Therefore, out of 1306 total records, 1044 records were selected as training set and 262 records were selected as testing set.

```
# splitting data into random train and test subsets
# train set - 80%, test set - 20%
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=443)
```

Table 2: Row count after train test split

Total dataset rows	**1306**
Train dataset rows	1044
Test dataset rows	262

Before feeding the data to deep learning models, we converted textual data into numerical form. Initially, we used the Tokenizer of the TensorFlow library high-level API named Keras [10] to vectorize the text corpus, by turning each text into a sequence of integers. It also performs all the preprocessing tasks including word tokenization, punctuation removal, and conversion to lower case.

The hyperparameters used for Tokenizer:

1. **oov_token = '<OOV>'**
   - oov_token defines the out of vocabulary token. It replaces the words that are not in the corpus during text_to_sequence calls.

2. **vocabulary_size = 500**
   - vocabulary_size indicates the maximum number of unique words to tokenize and load in training and testing data.

3. **char_level = False**
   - char_level indicates whether every character should be treated as a token or not. If the value of this hyperparameter is “False”, every word will be treated as a token.

```
# Tokenizer allows to vectorize a text corpus, by turning each text into
# either a sequence of integers (each integer being the index of a token
# in a dictionary) or into a vector where the coefficient for each token
# could be binary, based on word count, based on tf-idf
tokenizer = Tokenizer(num_words=vocabulary_size,
                      char_level=False,
                      oov_token=oov_token)

# updating internal vocabulary based on a list of text required before
# using texts_to_sequences
tokenizer.fit_on_texts(X_train)
```

Then, we used the texts_to_sequences() function of the Tokenizer object to represent each text of train and test sets by a sequence of numbers.

```
# transforming each text in train data to a sequence of integers
X_train_sequences = tokenizer.texts_to_sequences(X_train)


# transforming each text in test data to a sequence of integers
X_test_sequences = tokenizer.texts_to_sequences(X_test)
```

To visualize the length distribution of each sequence, we plotted two graphs for train and test sets as shown in Figure 12 and Figure 13.

```
# getting lengths of each generated sequences of integers
# in train data
x_train_length_of_sequence = [len(sequence) for sequence in
X_train_sequences]

# plotting a univariate distribution of observations for
# sequence lengths of train data
sns.distplot(x_train_length_of_sequence)


# getting lengths of each generated sequences of integers
# in test data
x_test_length_of_sequence = [len(sequence) for sequence in
X_test_sequences]

# plotting a univariate distribution of observations for sequence
# lengths of test data
sns.distplot(x_test_length_of_sequence)
```

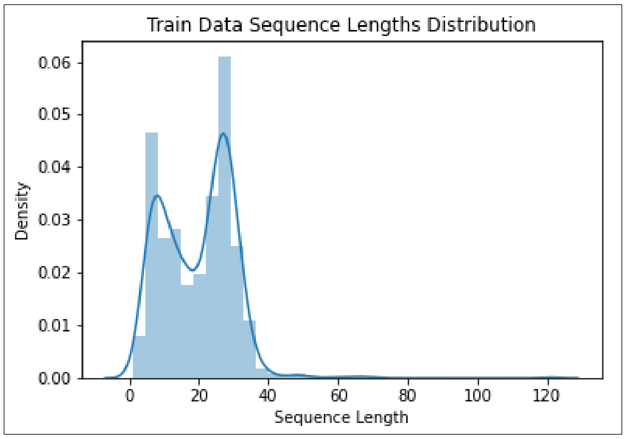

Figure 12: Train data sequence lengths distribution plot

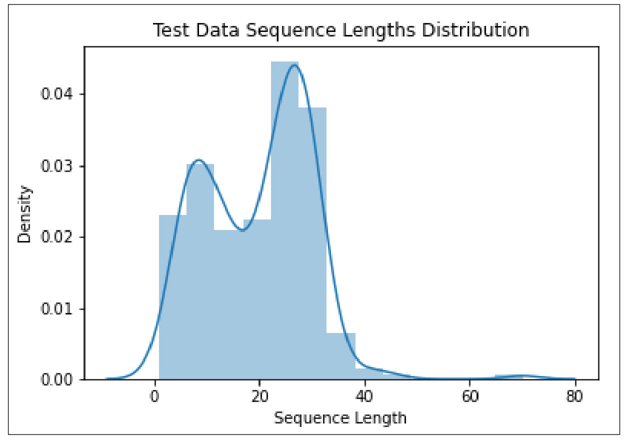

Figure 13: Train data sequence lengths distribution plot

However, after applying sequencing to the data, sequence lengths for both training and testing data were varied as shown in Figure 12 and Figure 13. Therefore, we used the pad_sequences() function to create padded sequences with the same length.

```
# padding on train data
X_train_padded = pad_sequences(X_train_sequences,
                               maxlen=maximum_length,
                               padding=padding_type,
                               truncating=truncating_type)

# padding on test data
X_test_padded = pad_sequences(X_test_sequences,
                              maxlen=maximum_length,
                              padding=padding_type,
                              truncating=truncating_type)
```

The hyperparameters used for padding:

1. **maximum_length = 50**
   - maximum_length indicates the maximum number of words considered in a text. As shown in Figure 12 and Figure 13 almost all sequence lengths were between 0-50, we chose the maximum_length as 50.

2. **truncating_type = "post"**
   - truncating_type indicates removal of values from sequences larger than maximum_length, either at the beginning ('pre') or at the end ('post') of the sequences.

3. padding_type = "post"

- padding_type indicates pad either before ('pre') or after ('post') each sequence.

To visualize the length distribution of each padded sequence, we plotted two graphs for train and test sets as shown in Figure 14 and Figure 15.

```
# getting lengths of each padded sequences of integers in train
# data
x_train_length_of_padded_sequence = [len(sequence) for sequence in
X_train_padded]

# plotting a univariate distribution of observations for sequence
# lengths of train data after padding
sns.distplot(x_train_length_of_padded_sequence)

# getting lengths of each padded sequences of integers in test
# data
x_test_length_of_padded_sequence = [len(sequence) for sequence in
X_test_padded]

# plotting a univariate distribution of observations for
# sequence lengths of test data after padding
sns.distplot(x_test_length_of_padded_sequence)
```

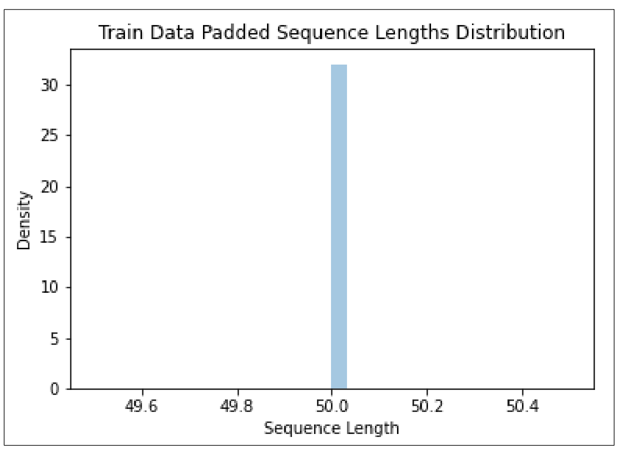

Figure 14: Train data padded sequence lengths distribution plot

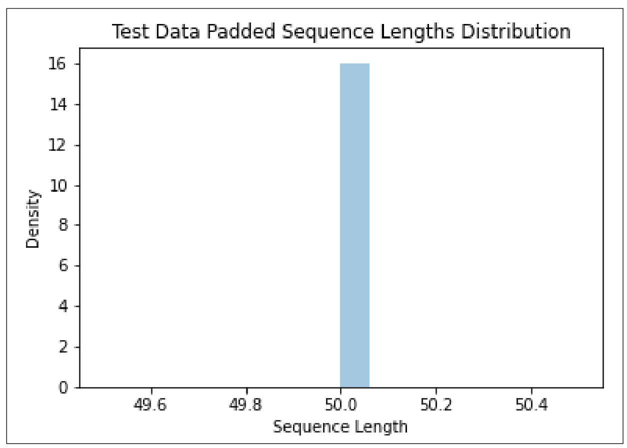

Figure 15: Test data padded sequence lengths distribution plot

Therefore, after applying padding to the data, sequence lengths for both training and testing data were the same as shown in Figure 14 and Figure 15.

Deep Learning Models

We trained two different Deep Learning models to classify ham and spam SMS messages. They are the Long Short-Term Memory (LSTM) and Densely Connected Convolutional Neural Networks (DenseNet CNN). After evaluating each trained model, the results show that the DenseNet CNN model provides the highest accuracy in predicting the correct label.

Justification for Choosing the LSTM and DenseNet CNN Models

The student performance prediction problem can be addressed through supervised learning techniques.

Introduction and Background of LSTM Model

KNN is a simple machine learning algorithm based on supervised learning, which can be used for both regression and classification problems.

Implementation of LSTM Model

We defined the LSTM sequential model architecture with four layers: an Embedding layer, two LSTM layers, and a Dense layer.

```
# LSTM model architecture

lstm_model = Sequential()

lstm_model.add(Embedding(vocabulary_size,
                         embedding_dimension,
                         input_length=maximum_length))

# return_sequences=True ensures that the LSTM cell returns all of the
# outputs from the unrolled LSTM cell through time
# if this argument is not used, the LSTM cell will simply provide the
# output of the LSTM cell from the previous step
lstm_model.add(LSTM(no_of_nodes,
                    dropout=dropout_rate,
                    return_sequences=True))

lstm_model.add(LSTM(no_of_nodes,
                    dropout=dropout_rate,
                    return_sequences=True))

# sigmoid is a non-linear and easy to work with activation function
# that takes a value as input and outputs another value between 0 and 1
lstm_model.add(Dense(1,
                     activation='sigmoid'))
```

Sigmoid is used as the activation function in the Dense layer. It is non-linear and easy to work with an activation function that takes a value as input and outputs another value between 0 and 1. Then we compiled the LSTM model with “Binary Cross-entropy” as the loss and “Adam” as the optimizer.

```
# compiling the model
# configuring the model for training
lstm_model.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

The hyperparameters used for the LSTM model:

1. **dropout_rate = 0.2**
    - dropout_rate indicates the fraction of the units to drop for the linear transformation of the inputs

2. **vocabulary_size = 500**
    - vocabulary_size indicates the maximum number of unique words to tokenize and load in training and testing data.

3. **no_of_epochs = 30**
    - no_of_epochs indicate the number of complete passes through the training dataset

4. **no_of_nodes = 20**
    - no_of_nodes indicate the number of nodes in the hidden layers within the LSTM cell

5. **embedding_dimension = 16**
    - embedding dimension indicates the dimension of the state space used for reconstruction

```
# printing a string summary of the network
lstm_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 50, 16)            8000      
_________________________________________________________________
lstm (LSTM)                  (None, 50, 20)            2960      
_________________________________________________________________
lstm_1 (LSTM)                (None, 50, 20)            3280      
_________________________________________________________________
dense (Dense)                (None, 50, 1)             21        
=================================================================
Total params: 14,261
Trainable params: 14,261
Non-trainable params: 0
_________________________________________________________________
```

We used the early stopping technique to avoid overfitting of the model. Here we configured the model to monitor the validation loss while training and if the validation loss is not improved after three epochs, then the model training is stopped.

```
# monitoring the validation loss and if the validation loss is not
# improved after three epochs, then the model training is stopped
# it helps to avoid overfitting problem and indicates when to stop
# training before the deep learning model begins overfitting
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=3)
```

Then the LSTM model was trained.

```
# training the LSTM model
history = lstm_model.fit(X_train_padded,
                         y_train,
                         epochs=no_of_epochs,
                         validation_data=(X_test_padded, y_test),
                         callbacks=[early_stopping],
                         verbose=2)
```

Then we plotted the training and validation loss against the number of epochs for the LSTM model as shown in Figure 17 and the training and validation accuracy against the number of epochs for the LSTM model as shown in Figure 18.

```
# visualizing the history results by reading as a dataframe
metrics_lstm = pd.DataFrame(history.history)

# plotting the training and validation loss by number of epochs for
# the LSTM model
metrics_lstm[['Training_Loss', 'Validation_Loss']].plot()

# plotting the training and validation accuracy by number of epochs for
# the LSTM model
metrics_lstm[['Training_Accuracy', 'Validation_Accuracy']].plot()
```

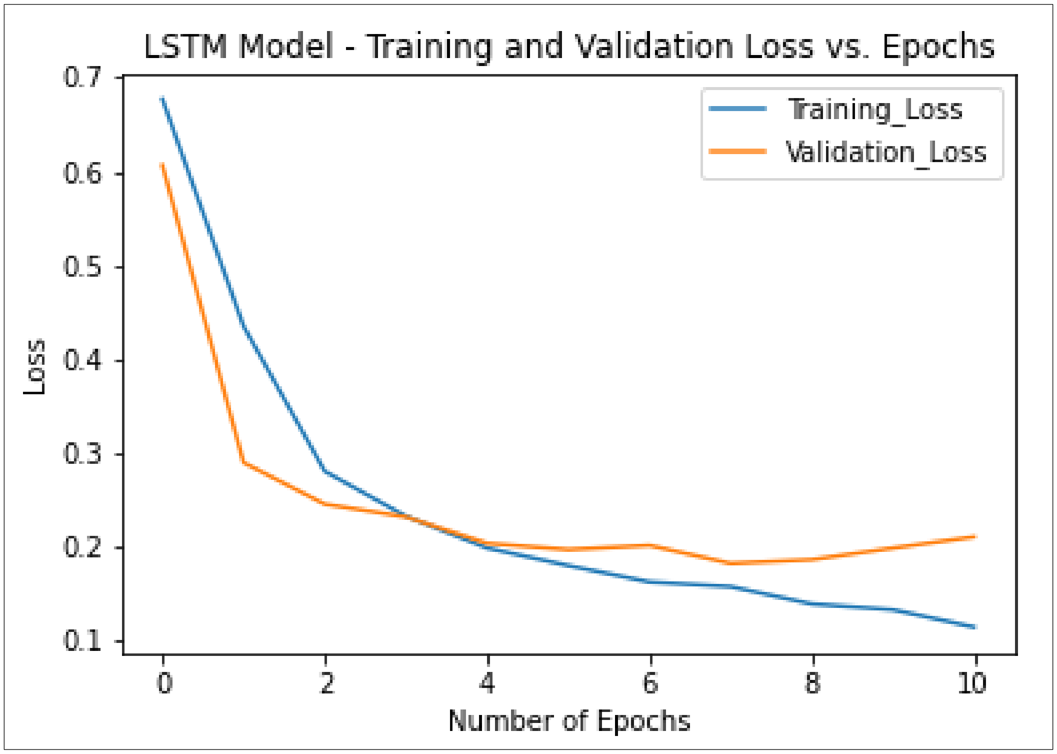

Figure 17: LSTM model - training and validation loss vs. epochs

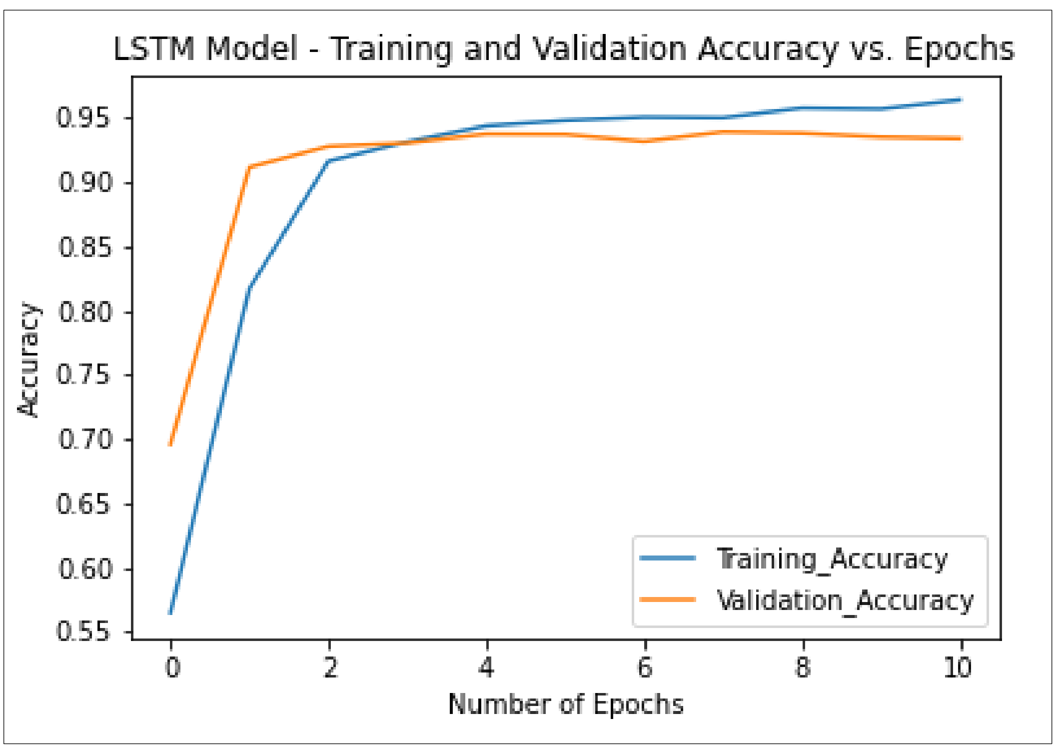

Figure 18: LSTM model - training and validation accuracy vs. epochs

The trained LSTM was saved as an h5 file which is a file format to store structured data. Then the saved model can be loaded to do evaluations and predictions.

```
# saving the trained LSTM model as an h5 file
lstm_path = 'models/lstm_model.h5'
lstm_model.save(lstm_path)

# loading the saved LSTM model
loaded_lstm_model = load_model(lstm_path)
```

KNN is a simple machine learning algorithm based on supervised learning, which can be used for both regression and classification problems.

Implementation of **DenseNet CNN** Model

We defined the DenseNet CNN sequential model architecture with five layers: an Embedding layer, a pooling layer (GlobalAveragePooling1D layer), two Dense layers, and a Dropout layer.

```
# Densely Connected CNN (DenseNet) model architecture

densenet_cnn_model = Sequential()

densenet_cnn_model.add(Embedding(vocabulary_size,
                                 embedding_dimension,
                                 input_length=maximum_length))

densenet_cnn_model.add(GlobalAveragePooling1D())

# Rectified Linear Activation Function (ReLU) is a piecewise linear
# function that will output the input directly if it is positive,
# otherwise, it will output zero
densenet_cnn_model.add(Dense(24,
                             activation='relu'))

densenet_cnn_model.add(Dropout(dropout_rate))

# sigmoid is a non-linear and easy to work with activation function
# that takes a value as input and outputs another value between 0 and 1
densenet_cnn_model.add(Dense(1,
                             activation='sigmoid'))
```

Rectified Linear Activation Function (ReLU) is used as the activation function in the first Dense layer. It is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. Sigmoid is used as the activation function in the second Dense layer. It is non-linear and easy to work with an activation function that takes a value as input and outputs another value between 0 and 1. Then we compiled the DenseNet CNN model with "Binary Cross-entropy" as the loss and "Adam" as the optimizer.

```
# compiling the model
# configuring the model for training
densenet_cnn_model.compile(loss='binary_crossentropy',
                           optimizer='adam',
                           metrics=['accuracy'])
```

The hyperparameters used for the DenseNet CNN model:

1. **dropout_rate = 0.2**
    - dropout_rate indicates the fraction of the units to drop for the linear transformation of the inputs

2. **vocabulary_size = 500**
    - vocabulary_size indicates the maximum number of unique words to tokenize and load in training and testing data.

3. **no_of_epochs = 30**
    - no_of_epochs indicate the number of complete passes through the training dataset

4. **embedding_dimension = 16**
    - embedding dimension indicates the dimension of the state space used for reconstruction

```
# printing a string summary of the network
densenet_cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 50, 16)            8000
_________________________________________________________________
global_average_pooling1d (Gl (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 24)                408
_________________________________________________________________
dropout (Dropout)            (None, 24)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 25
=================================================================
Total params: 8,433
Trainable params: 8,433
Non-trainable params: 0
_________________________________________________________________
```

We used the early stopping technique to avoid overfitting of the model. Here we configured the model to monitor the validation loss while training and if the validation loss is not improved after three epochs, then the model training is stopped.

```
# monitoring the validation loss and if the validation loss is not
# improved after three epochs, then the model training is stopped
# it helps to avoid overfitting problem and indicates when to stop
# training before the deep learning model begins overfitting
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=3)
```

Then the DenseNet CNN model was trained.

```
# training the DenseNet CNN model
history = densenet_cnn_model.fit(X_train_padded,
                                 y_train,
                                 epochs=no_of_epochs,
                                 validation_data=(X_test_padded, y_test),
                                 callbacks=[early_stopping],
                                 verbose=2)
```

Then we plotted the training and validation loss against the number of epochs for the DenseNet CNN model as shown in Figure 20 and the training and validation accuracy against the number of epochs for the DenseNet CNN model as shown in Figure 21.

```
# visualizing the history results by reading as a dataframe
densenet_cnn_metrics = pd.DataFrame(history.history)

# plotting the training and validation loss by number of epochs for
# the DenseNet CNN model
densenet_cnn_metrics[['Training_Loss', 'Validation_Loss']].plot()

# plotting the training and validation accuracy by number of epochs
# for the DenseNet CNN model
densenet_cnn_metrics[['Training_Accuracy', 'Validation_Accuracy']].plot()
```

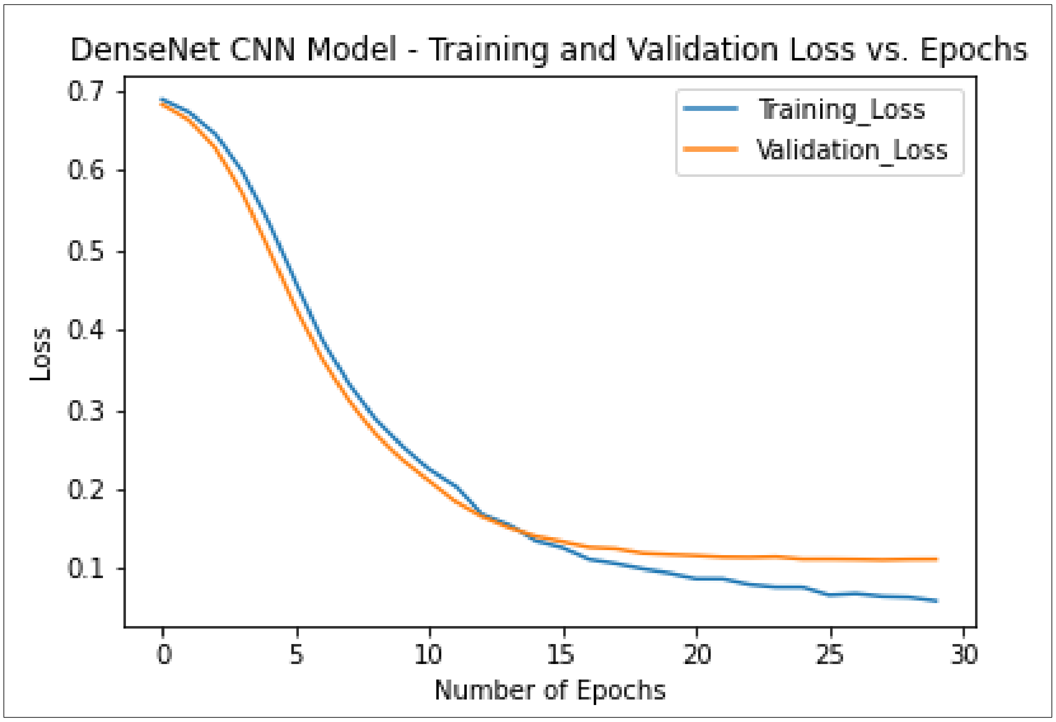

Figure 20: DenseNet CNN model - training and validation loss vs. epochs

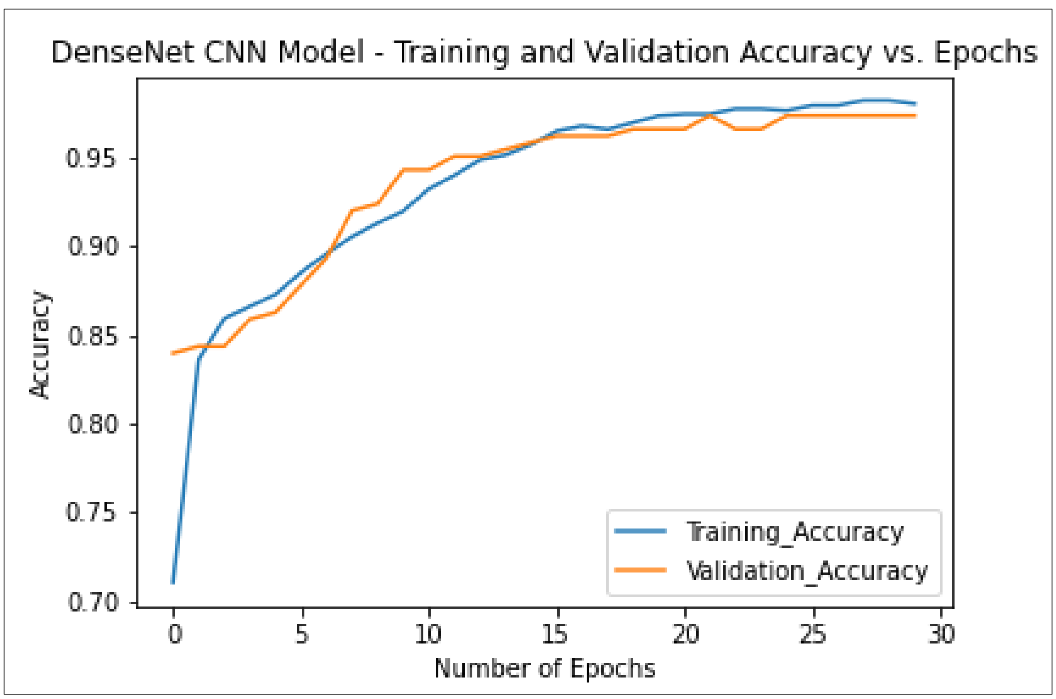

Figure 21: DenseNet CNN model - training and validation accuracy vs. epochs

The trained DenseNet CNN was saved as an h5 file which is a file format to store structured data. Then the saved model can be loaded to do evaluations and predictions.

```
# saving the trained DenseNet CNN model as an h5 file
densenet_cnn_path = 'models/densenet_cnn_model.h5'
densenet_cnn_model.save(densenet_cnn_path)

# loading the saved DenseNet CNN model
loaded_densenet_cnn_model = load_model(densenet_cnn_path)
```

References

[1] A. Ghourabi, M. A. Mahmood, and Q. M. Alzubi, “A Hybrid CNN-LSTM Model for SMS Spam Detection in Arabic and English Messages,” Future Internet, vol. 12, no. 9, p. 156, Sep. 2020, doi: 10.3390/fi12090156.

[2] J. Ma, Y. Zhang, J. Liu, K. Yu, and X. Wang, "Intelligent SMS Spam Filtering Using Topic Model," 2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS), 2016, pp. 380-383, doi: 10.1109/INCoS.2016.47.

[3] M. Popovac, M. Karanovic, S. Sladojevic, M. Arsenovic, and A. Anderla, "Convolutional Neural Network Based SMS Spam Detection," 2018 26th Telecommunications Forum (TELFOR), 2018, pp. 1-4, doi: 10.1109/TELFOR.2018.8611916.

[4] Archive.ics.uci.edu. 2012. UCI Machine Learning Repository: SMS Spam Collection Data Set. [online] Available at: http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection.

[5] "SMS Spam Collection Dataset", Kaggle.com, 2016. [online]. Available: https://www.kaggle.com/uciml/sms-spam-collection-dataset.

[6] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami, “Contributions to the study of SMS spam filtering,” presented at the 11th ACM symposium, 2011. doi: 10.1145/2034691.2034742.

[7] Grumbletext.co.uk, 2021. [Online]. Available: http://www.grumbletext.co.uk/.

[8] Comp.nus.edu.sg, 2021. [Online]. Available: https://www.comp.nus.edu.sg/.

[9] "API Reference — scikit-learn 0.24.2 documentation", *Scikit-learn.org*, 2021. [Online]. Available: https://scikit-learn.org/stable/modules/classes.html.

[10] "Module: tf.keras | TensorFlow Core v2.6.0", *TensorFlow*, 2021. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras.

[11]

[12]

[13]

[14]

[15]

[16]

[17]

[18]

Appendixes

```
#%% md

# SMS Spam Detection with Deep Learning

Short Message Service (SMS) is being heavily used as a way of
communication. However, SMS spams have targeted most mobile phone users
recently. In some cases, SMS spams contain malicious activities such as
smishing. Smishing (SMS + Phishing) is a cyber-security threat for mobile
users aimed at deceiving them via SMS spam messages that may include a
link or malicious software or both. The attackers attempt to steal users'
secret and sensitive information, like credit card numbers, bank account
details, and passwords.

The filtration of SMS spams in smartphones is still not very robust
compared to the filtration of email spams. The state-of-the-art
methodologies based on Deep Learning can be utilized for solving this
binary classification problem of SMS spam detection. We have used the two
deep neural network architectures namely Long Short-Term Memory (LSTM) and
DenseNet (Densely Connected Convolutional Neural Network (CNN)) for this
purpose.

#%%

# magic function that renders the figure in a notebook instead of
# displaying a dump of the figure object
# sets the backend of matplotlib to the 'inline' backend
# with this backend, the output of plotting commands is displayed
# inline within frontends like the Jupyter notebook, directly below
# the code cell that produced it
# the resulting plots will then also be stored in the notebook document
%matplotlib inline

#%%

# creating a new directory named plots
!mkdir plots

# creating a new directory named models
!mkdir models

# creating a new directory named processed_datasets
!mkdir processed_datasets

#%%

# importing warnings library to handle exceptions, errors, and warning
# of the program
import warnings
```

```
# ignoring potential warnings of the program
warnings.filterwarnings('ignore')

#%%

# importing pandas library to perform data manipulation and analysis
import pandas as pd

# configuring the pandas dataframes to show all columns
pd.options.display.max_columns = None

# configuring the pandas dataframes to increase the maximum column width
pd.options.display.max_colwidth = 150

#%%

# downloading UCI SMS Spam Collection dataset
!wget --no-check-certificate https://archive.ics.uci.edu/ml/machine-learning-databases/00228/smsspamcollection.zip

#%%

# extracting the downloaded dataset
!unzip /content/smsspamcollection.zip

#%%

# listing files and directories
!ls

#%% md

# SMS Spam Collection Dataset

The SMS Spam Collection Dataset was downloaded from UCI datasets. It
contains 5,574 SMS phone messages. The data were collected for the purpose
of mobile phone SMS text message spam research and have already been
labeled as either spam or ham.

Link to the dataset -
http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection

#%%

# importing SMSSpamCollection dataset to a pandas dataframe
sms_spam_dataframe = pd.read_csv('/content/SMSSpamCollection',
                                 sep='\t',
                                 header=None,
                                 names=['class', 'sms_message'])
sms_spam_dataframe

#%% md

# Dataset Analysis and Data Preprocessing
```

```
#%%

# printing the columns of the dataframe
sms_spam_dataframe.columns

#%%

# changing the order of the dataframe columns for better visualization
sms_spam_dataframe = sms_spam_dataframe[['sms_message', 'class']]
sms_spam_dataframe

#%%

# displaying the dimensionality of the dataframe
sms_spam_dataframe.shape

#%%

# printing a concise summary of the dataframe
# information such as index, data type, columns, non-null values,
# and memory usage
sms_spam_dataframe.info()

#%%

# generating descriptive statistics of the dataframe
sms_spam_dataframe.describe()

#%%

# generating descriptive statistics for each class of the dataframe
# T property is used to transpose index and columns of the dataframe
sms_spam_dataframe.groupby('class').describe().T

#%%

# checking for missing or null values in the dataframe
dataframe_null =
sms_spam_dataframe[sms_spam_dataframe.isnull().any(axis=1)]
dataframe_null

#%%

# printing the number of rows with any missing or null values
# in the dataframe
dataframe_null.shape[0]

#%%

# removing the missing or null values from the dataframe if exist
sms_spam_dataframe =
sms_spam_dataframe[sms_spam_dataframe.notna().all(axis=1)]
```

```python
# printing the count of null values in class and sms_message
# columns of the dataframe
sms_spam_dataframe[['class', 'sms_message']].isnull().sum()

#%%

# importing pyplot from matplotlib library to create interactive
# visualizations
import matplotlib.pyplot as plt

# importing seaborn library which is built on top of matplotlib to
# create statistical graphics
import seaborn as sns

# plotting the heatmap for missing or null values in the dataframe
sns.heatmap(sms_spam_dataframe.isnull(),
            yticklabels=False,
            cbar=False,
            cmap='viridis')
plt.title('Null Values Detection Heat Map')
plt.savefig('plots/null_detection_heat_map.png',
            facecolor='white')
plt.show()

#%%

# importing missingno library
# used to understand the distribution of missing values through
# informative visualizations
# visualizations can be in the form of heat maps or bar charts
# used to observe where the missing values have occurred
# used to check the correlation of the columns containing the missing
# with the target column
import missingno as msno

# plotting a matrix visualization of the nullity of the dataframe
fig = msno.matrix(sms_spam_dataframe)
fig_copy = fig.get_figure()
fig_copy.savefig('plots/msno_matrix.png',
                 bbox_inches='tight')
fig

#%%

# plotting a seaborn heatmap visualization of nullity correlation
# in the dataframe
fig = msno.heatmap(sms_spam_dataframe)
fig_copy = fig.get_figure()
fig_copy.savefig('plots/msno_heatmap.png',
                 bbox_inches='tight')
fig

#%%
```

```
# detecting duplicate rows exist in the dataframe before cleaning
duplicated_records = sms_spam_dataframe[sms_spam_dataframe.duplicated()]
duplicated_records

#%%

# checking the number of duplicate rows exist in the dataframe
# before cleaning
sms_spam_dataframe.duplicated().sum()

#%%

# removing the duplicate rows from the dataframe if exist
sms_spam_dataframe = sms_spam_dataframe.drop_duplicates()
sms_spam_dataframe

#%%

# checking the number of duplicate rows exist in the dataframe
# after cleaning
sms_spam_dataframe.duplicated().sum()

#%%

# displaying the dimensionality of the dataframe
sms_spam_dataframe.shape

#%%

# printing a concise summary of the dataframe
# information such as index, data type, columns, non-null values,
# and memory usage
sms_spam_dataframe.info()

#%%

# generating descriptive statistics of the dataframe
sms_spam_dataframe.describe()

#%%

# generating descriptive statistics for each class of the dataframe
# T property is used to transpose index and columns of the dataframe
sms_spam_dataframe.groupby('class').describe().T

#%%

# saving cleaned dataset to a csv file
file_name = 'processed_datasets/cleaned_dataset.csv'
sms_spam_dataframe.to_csv(file_name,
                          encoding='utf-8',
                          index=False)

# loading dataset from the saved csv file to a pandas dataframe
```

```
cleaned_sms_spam_dataframe = pd.read_csv(file_name)
cleaned_sms_spam_dataframe

#%%

# importing set of stopwords from wordcloud library
from wordcloud import STOPWORDS

stopwords = set(STOPWORDS)

# printing number of stopwords defined in wordcloud library
len(stopwords)

#%%

# importing random library
# used for generating random numbers
import random

# printing 10 random values of stopwords set
for i, val in enumerate(random.sample(stopwords, 10)):
    print(val)

#%%

# importing WordCloud object for generating and drawing
# wordclouds from wordcloud library
from wordcloud import WordCloud


# defining a function to return the wordcloud for a given text
def plot_wordcloud(text):
    wordcloud = WordCloud(width=600,
                          height=300,
                          background_color='black',
                          stopwords=stopwords,
                          max_font_size=50,
                          colormap='Oranges').generate(text)
    return wordcloud

#%%

# extracting the data instances with class label 'ham'
ham_dataframe =
cleaned_sms_spam_dataframe[cleaned_sms_spam_dataframe['class'] == 'ham']
ham_dataframe

#%%

# creating numpy list to visualize using wordcloud
ham_sms_message_text = '
'.join(ham_dataframe['sms_message'].to_numpy().tolist())

# generating wordcloud for ham sms messages
```

```
ham_sms_wordcloud = plot_wordcloud(ham_sms_message_text)
plt.figure(figsize=(16, 10))
plt.imshow(ham_sms_wordcloud,
           interpolation='bilinear')
plt.axis('off')
plt.title('Ham SMS Wordcloud')
plt.savefig('plots/ham_wordcloud.png',
            facecolor='white')
plt.show()

#%%

# extracting the data instances with class label 'spam'
spam_dataframe =
cleaned_sms_spam_dataframe[cleaned_sms_spam_dataframe['class'] == 'spam']
spam_dataframe

#%%

# creating numpy list to visualize using wordcloud
spam_sms_message_text = '
'.join(spam_dataframe['sms_message'].to_numpy().tolist())

# generating wordcloud for spam sms messages
spam_sms_wordcloud = plot_wordcloud(spam_sms_message_text)
plt.figure(figsize=(16, 10))
plt.imshow(spam_sms_wordcloud,
           interpolation='bilinear')
plt.axis('off')
plt.title('Spam SMS Wordcloud')
plt.savefig('plots/spam_wordcloud.png',
            facecolor='white')
plt.show()

#%%

# printing count of values in each class of the dataframe
cleaned_sms_spam_dataframe['class'].value_counts()

#%%

# plotting the distribution of target values
fig = plt.figure()
lbl = ['Ham (0)', 'Spam (1)']
pct = '%1.0f%%'
ax = cleaned_sms_spam_dataframe['class'].value_counts().plot(kind='pie',
                                                             labels=lbl,
                                                             autopct=pct)
ax.yaxis.set_visible(False)
plt.title('Distribution of Ham and Spam SMS')
plt.legend()
fig.savefig('plots/ham_spam_pie_chart.png',
            facecolor='white')
plt.show()
```

```
#%%

# downsampling is a process where you randomly delete some of the
# observations from the majority class so that the numbers in majority
# and minority classes are matched
# after downsampling the ham messages (majority class), there are now
# 653 messages in each class
downsampled_ham_dataframe = ham_dataframe.sample(n=len(spam_dataframe),
                                                 random_state=44)
downsampled_ham_dataframe

#%%

# printing the dimensions of spam and downsampled ham dataframes
print('Spam dataframe shape:', spam_dataframe.shape)
print('Ham dataframe shape:', downsampled_ham_dataframe.shape)

#%%

# merging the two dataframes (spam + downsampled ham dataframes)
merged_dataframe = pd.concat([downsampled_ham_dataframe, spam_dataframe])
merged_dataframe = merged_dataframe.reset_index(drop=True)
merged_dataframe

#%%

# printing count of values in each class of the merged dataframe
merged_dataframe['class'].value_counts()

#%%

# plotting the distribution of target values after downsampling
fig = plt.figure()
lbl = ['Ham (0)', 'Spam (1)']
pct = '%1.0f%%'
ax = merged_dataframe['class'].value_counts().plot(kind='pie',
                                                   labels=lbl,
                                                   autopct=pct)
ax.yaxis.set_visible(False)
plt.title('Distribution of Ham and Spam SMS after Downsampling')
plt.legend()
fig.savefig('plots/ham_spam_pie_chart_after_downsampling.png',
            facecolor='white')
plt.show()

#%%

# inserting a new column called 'label' to the merged dataframe
# if class is 'ham' label = 0
# if class is 'spam' label = 1
merged_dataframe['label'] = merged_dataframe['class'].map({'ham': 0,
'spam': 1})
merged_dataframe
```

```
#%%

# inserting a new column called 'length' to the merged dataframe
# the column contains the number of characters of the sms_message text
merged_dataframe['length'] = merged_dataframe['sms_message'].apply(len)
merged_dataframe

#%%

# displaying the first 5 rows of the dataframe
merged_dataframe.head()

#%%

# displaying the last 5 rows of the dataframe
merged_dataframe.tail()

#%%

# displaying the dimensionality of the dataframe
merged_dataframe.shape

#%%

# printing a concise summary of the dataframe
# information such as index, data type, columns, non-null values,
# and memory usage
merged_dataframe.info()

#%%

# generating descriptive statistics of the dataframe
merged_dataframe.describe().round(2)

#%%

# generating descriptive statistics for each class of the dataframe
# T property is used to transpose index and columns of the dataframe
merged_dataframe.groupby('label').describe().T

#%%

# generating descriptive sms text length statistics by label types
merged_dataframe.groupby('label')['length'].describe().round(2)

#%%

# plotting a univariate distribution of observations for sms lengths
sns.distplot(merged_dataframe['length'].values)
plt.title('SMS Lengths Distribution')
plt.xlabel('SMS Length')
plt.savefig('plots/sms_length.png',
            facecolor='white')
```

```python
plt.show()

#%%

# saving merged dataset to a csv file
file_name = 'processed_datasets/merged_dataset.csv'
merged_dataframe.to_csv(file_name,
                        encoding='utf-8',
                        index=False)

# loading dataset from the saved csv file to a pandas dataframe
merged_dataframe = pd.read_csv(file_name)
merged_dataframe

#%%

# assigning attributes (features) to X
X = merged_dataframe['sms_message']
X

#%%

# assigning label (target) to y
y = merged_dataframe['label'].values
y

#%%

# importing train_test_split from scikit-learn library
from sklearn.model_selection import train_test_split

# splitting data into random train and test subsets
# train set - 80%, test set - 20%
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=443)

# printing the dimension of train features dataframe
print('Shape of train features dataframe:', X_train.shape)

# printing the dimension of train target dataframe
print('Shape of train target dataframe:', y_train.shape)

# printing the dimension of test features dataframe
print('Shape of test features dataframe:', X_test.shape)

# printing the dimension of test target dataframe
print('Shape of test target dataframe:', y_test.shape)

#%%

# displaying train features dataframe
X_train
```

```
#%%

# displaying train target dataframe
y_train

#%%

# displaying test features dataframe
X_test

#%%

# displaying test target dataframe
y_test

#%%

# defining pre-processing hyperparameters

# oov_token defines the out of vocabulary token
# oov_token will be added to word index in the corpus which is used to
# build the model
# this is used to replace out of vocabulary words (words that are not
# in our corpus) during text_to_sequence calls.
oov_token = '<OOV>'

# vocabulary_size indicates the maximum number of unique words to
# tokenize and load in training and testing data
vocabulary_size = 500

#%%

# importing Tokenizer from keras library
# tensorflow is a free and open-source software library for machine
# learning used across a range of machine learning related tasks
# focus on training and inference of deep neural networks
# keras is a high-level api of tensorflow
# keras.preprocessing.text provides keras data preprocessing utils
# to pre-process datasets with textual data before they are fed to the
# machine learning model
from tensorflow.keras.preprocessing.text import Tokenizer

# Tokenizer allows to vectorize a text corpus, by turning each text into
# either a sequence of integers (each integer being the index of a token
# in a dictionary) or into a vector where the coefficient for each token
# could be binary, based on word count, based on tf-idf
tokenizer = Tokenizer(num_words=vocabulary_size,
                      char_level=False,
                      oov_token=oov_token)

# updating internal vocabulary based on a list of text required before
# using texts_to_sequences
tokenizer.fit_on_texts(X_train)
```

```
#%%

# getting the word_index
word_index = tokenizer.word_index
word_index

#%%

# printing length of the word index
len(word_index)

#%%

# transforming each text in train data to a sequence of integers
X_train_sequences = tokenizer.texts_to_sequences(X_train)

# printing the first sequence
X_train_sequences[0]

#%%

# getting lengths of each generated sequences of integers
# in train data
x_train_length_of_sequence = [len(sequence) for sequence in
X_train_sequences]

# printing the length of the first sequence
x_train_length_of_sequence[0]

#%%

# importing numpy library
# used to perform fast mathematical operations over python arrays
# and lists
import numpy as np

# printing maximum length of a sequence in the train data
np.max(x_train_length_of_sequence)

#%%

# plotting a univariate distribution of observations for
# sequence lengths of train data
sns.distplot(x_train_length_of_sequence)
plt.title('Train Data Sequence Lengths Distribution')
plt.xlabel('Sequence Length')
plt.savefig('plots/train_sequence_length.png',
            facecolor='white')
plt.show()

#%%

# transforming each text in test data to a sequence of integers
```

```python
X_test_sequences = tokenizer.texts_to_sequences(X_test)

# printing the first sequence
X_test_sequences[0]

#%%

# getting lengths of each generated sequences of integers
# in test data
x_test_length_of_sequence = [len(sequence) for sequence in
X_test_sequences]

# printing the length of the first sequence
x_test_length_of_sequence[0]

#%%

# printing maximum length of a sequence in the test data
np.max(x_test_length_of_sequence)

#%%

# plotting a univariate distribution of observations for sequence
# lengths of test data
sns.distplot(x_test_length_of_sequence)
plt.title('Test Data Sequence Lengths Distribution')
plt.xlabel('Sequence Length')
plt.savefig('plots/test_sequence_length.png',
            facecolor='white')
plt.show()

#%%

# defining pre-processing hyperparameters

# maximum_length indicates the maximum number of words considered
# in a text
maximum_length = 50

# truncating_type indicates removal of values from sequences larger
# than maxlen, either at the beginning ('pre') or at the end ('post')
# of the sequences
truncating_type = 'post'

# padding_type indicates pad either before ('pre') or after ('post')
# each sequence
padding_type = 'post'

#%%

# importing utilities for preprocessing sequence data from
# keras library
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
# padding on train data
# pad_sequences pads sequences to the same length
# padding='post' to pad after each sequence
X_train_padded = pad_sequences(X_train_sequences,
                               maxlen=maximum_length,
                               padding=padding_type,
                               truncating=truncating_type)

# printing the first padded sequence
X_train_padded[0]

#%%

# getting lengths of each padded sequences of integers in train
# data
x_train_length_of_padded_sequence = [len(sequence) for sequence in
X_train_padded]

# printing the length of the first padded sequence
x_train_length_of_padded_sequence[0]

#%%

# printing maximum length of a padded sequence in the train data
np.max(x_train_length_of_padded_sequence)

#%%

# printing the dimension of padded training dataframe
X_train_padded.shape

#%%

# plotting a univariate distribution of observations for sequence
# lengths of train data after padding
sns.distplot(x_train_length_of_padded_sequence)
plt.title('Train Data Padded Sequence Lengths Distribution')
plt.xlabel('Sequence Length')
plt.savefig('plots/train_padded_sequence_length.png',
            facecolor='white')
plt.show()

#%%

# padding on test data
# pad_sequences pads sequences to the same length
# padding='post' to pad after each sequence
X_test_padded = pad_sequences(X_test_sequences,
                              maxlen=maximum_length,
                              padding=padding_type,
                              truncating=truncating_type)

# printing the first padded sequence
X_test_padded[0]
```

```
#%%

# getting lengths of each padded sequences of integers in test
# data
x_test_length_of_padded_sequence = [len(sequence) for sequence in
X_test_padded]

# printing the length of the first padded sequence
x_test_length_of_padded_sequence[0]

#%%

# printing maximum length of a padded sequence in the test data
np.max(x_test_length_of_padded_sequence)

#%%

# printing the dimension of padded test dataframe
X_test_padded.shape

#%%

# plotting a univariate distribution of observations for
# sequence lengths of test data after padding
sns.distplot(x_test_length_of_padded_sequence)
plt.title('Test Data Padded Sequence Lengths Distribution')
plt.xlabel('Sequence Length')
plt.savefig('plots/test_padded_sequence_length.png',
            facecolor='white')
plt.show()

#%% md

# LSTM Model

Long Short Term Memory (LSTM) is a special kind of Recurrent Neural
Network (RNN). LSTM models are explicitly designed to avoid the long-term
dependency problem by remembering information for long periods of time.

#%%

# LSTM network architecture hyperparameters

# SpatialDropout1D is used to dropout the embedding layer which helps
# to drop entire 1D feature maps instead of individual elements
# dropout_rate indicates the fraction of the units to drop for the
# linear transformation of the inputs
dropout_rate = 0.2

# no_of_nodes indicates the number of nodes in the hidden layers within
# the LSTM cell
no_of_nodes = 20
```

```
# embedding_dimension indicates the dimension of the state space used for
# reconstruction
embedding_dimension = 16

# no_of_epochs indicates the number of complete passes through the
# training dataset
no_of_epochs = 30

# vocabulary_size indicates the maximum number of unique words to
# tokenize and load in training and testing data
vocabulary_size = 500

#%%

# importing Sequential class from keras
# Sequential groups a linear stack of layers into a keras Model
from tensorflow.keras.models import Sequential

# importing Embedding class from keras layers api package
# turning positive integers (indexes) into dense vectors of fixed size
# this layer can only be used as the first layer in a model
from tensorflow.keras.layers import Embedding

# importing LSTM class from keras layers api package
# LSTM - Long Short-Term Memory layer
from tensorflow.keras.layers import LSTM

# importing Dense class from keras layers api package
# Dense class is a regular densely-connected neural network layer
# Dense implements the operation:
# output = activation(dot(input, kernel) + bias)
# activation is the element-wise activation function passed as
# the activation argument
# kernel is a weights matrix created by the layer
# bias is a bias vector created by the layer
# (only applicable if use_bias is True)
from tensorflow.keras.layers import Dense

#%%

# LSTM model architecture

lstm_model = Sequential()

lstm_model.add(Embedding(vocabulary_size,
                         embedding_dimension,
                         input_length=maximum_length))

# return_sequences=True ensures that the LSTM cell returns all of the
# outputs from the unrolled LSTM cell through time
# if this argument is not used, the LSTM cell will simply provide the
# output of the LSTM cell from the previous step
lstm_model.add(LSTM(no_of_nodes,
                    dropout=dropout_rate,
```

```
                    return_sequences=True))

lstm_model.add(LSTM(no_of_nodes,
                    dropout=dropout_rate,
                    return_sequences=True))

# sigmoid is a non-linear and easy to work with activation function
# that takes a value as input and outputs another value between 0 and 1
lstm_model.add(Dense(1,
                     activation='sigmoid'))

#%%

# compiling the model
# configuring the model for training
lstm_model.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])

#%%

# printing a string summary of the network
lstm_model.summary()

#%%

# importing EarlyStopping class from callbacks module
# in keras library
# callbacks module includes utilities called at certain
# points during model training
# used to stop training when a monitored metric has
# stopped improving
from tensorflow.keras.callbacks import EarlyStopping

# monitoring the validation loss and if the validation loss is not
# improved after three epochs, then the model training is stopped
# it helps to avoid overfitting problem and indicates when to stop
# training before the deep learning model begins overfitting
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=3)

#%%

# training the LSTM model
history = lstm_model.fit(X_train_padded,
                         y_train,
                         epochs=no_of_epochs,
                         validation_data=(X_test_padded, y_test),
                         callbacks=[early_stopping],
                         verbose=2)

#%%

# visualizing the history results by reading as a dataframe
```

```
metrics_lstm = pd.DataFrame(history.history)
metrics_lstm

#%%

# renaming the column names of the dataframe
metrics_lstm.rename(columns={'loss': 'Training_Loss',
                             'accuracy': 'Training_Accuracy',
                             'val_loss': 'Validation_Loss',
                             'val_accuracy': 'Validation_Accuracy'},
                    inplace=True)
metrics_lstm

#%%

# plotting the training and validation loss by number of epochs for
# the LSTM model
metrics_lstm[['Training_Loss', 'Validation_Loss']].plot()
plt.title('LSTM Model - Training and Validation Loss vs. Epochs')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss')
plt.legend(['Training_Loss', 'Validation_Loss'])
plt.savefig('plots/lstm_loss_vs_epochs.png',
            facecolor='white')
plt.show()

#%%

# plotting the training and validation accuracy by number of epochs for
# the LSTM model
metrics_lstm[['Training_Accuracy', 'Validation_Accuracy']].plot()
plt.title('LSTM Model - Training and Validation Accuracy vs. Epochs')
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy')
plt.legend(['Training_Accuracy', 'Validation_Accuracy'])
plt.savefig('plots/lstm_accuracy_vs_epochs.png',
            facecolor='white')
plt.show()

#%%

# saving the trained LSTM model as an h5 file
# h5 is a file format to store structured data
# keras saves deep learning models in this format as it can easily store
# the weights and model configuration in a single file
lstm_path = 'models/lstm_model.h5'
lstm_model.save(lstm_path)
lstm_model

#%%

# importing load_model function from keras to load a saved keras
# deep learning model
from tensorflow.keras.models import load_model
```

```
# loading the saved LSTM model
loaded_lstm_model = load_model(lstm_path)
loaded_lstm_model

#%% md

# LSTM Model Evaluation

#%%

# evaluating the LSTM model performance on test data
# validation loss = 0.21674150228500366
# validation accuracy = 0.930610716342926
loaded_lstm_model.evaluate(X_test_padded,
                           y_test)

#%%

# predicting labels of X_test data values on the basis of the
# trained model
y_pred_lstm = [1 if x[0][0] > 0.5 else 0 for x in
loaded_lstm_model.predict(X_test_padded)]

# printing the length of the predictions list
len(y_pred_lstm)

#%%

# printing the first 25 elements of the predictions list
y_pred_lstm[:25]

#%%

# importing mean_squared_error from scikit-learn library
from sklearn.metrics import mean_squared_error

# mean squared error (MSE)
print('MSE :', mean_squared_error(y_test,
                                  y_pred_lstm))

# root mean squared error (RMSE)
# square root of the average of squared differences between predicted
# and actual value of variable
print('RMSE:', mean_squared_error(y_test,
                                  y_pred_lstm,
                                  squared=False))

#%%

# importing mean_absolute_error from scikit-learn library
from sklearn.metrics import mean_absolute_error

# mean absolute error (MAE)
```

```
print('MAE:', mean_absolute_error(y_test,
                                  y_pred_lstm))

#%%

# importing accuracy_score from scikit-learn library
from sklearn.metrics import accuracy_score

# accuracy
# ratio of the number of correct predictions to the total number of
# input samples
print('Accuracy:', accuracy_score(y_test,
                                  y_pred_lstm))

#%%

# importing precision_recall_fscore_support from scikit-learn library
from sklearn.metrics import precision_recall_fscore_support

print('\t\t\tPrecision \t\tRecall \t\tF-Measure \tSupport')

# computing precision, recall, f-measure and support for each class
# with average='micro'
print('average=micro    -', precision_recall_fscore_support(y_test,
                                                             y_pred_lstm,

average='micro'))

# computing precision, recall, f-measure and support for each class
# with average='macro'
print('average=macro    -', precision_recall_fscore_support(y_test,
                                                             y_pred_lstm,

average='macro'))

# computing precision, recall, f-measure and support for each class
# with average='weighted'
print('average=weighted -', precision_recall_fscore_support(y_test,
                                                             y_pred_lstm,

average='weighted'))

#%%

# importing classification_report from scikit-learn library
# used to measure the quality of predictions from a
# classification algorithm
from sklearn.metrics import classification_report

# report shows the main classification metrics precision, recall and
# f1-score on a per-class basis
print(classification_report(y_test,
                            y_pred_lstm))
```

```
#%%

# importing confusion_matrix from scikit-learn library
from sklearn.metrics import confusion_matrix

# confusion matrix is a summarized table used to assess the performance
# of a classification model
# number of correct and incorrect predictions are summarized with their
# count according to each class
print(confusion_matrix(y_test,
                       y_pred_lstm))

#%%

# importing plot_confusion_matrix from scikit-learn library

# plotting the confusion matrix
cm = confusion_matrix(y_test,
                      y_pred_lstm)
sns.heatmap(cm,
            annot=True,
            cbar=False,
            fmt='g')
plt.title('LSTM Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('plots/lstm_confusion_matrix.png',
            facecolor='white')
plt.show()

#%%

# plotting scatter plot to visualize overlapping of predicted and
# test target data points
plt.scatter(range(len(y_pred_lstm)),
            y_pred_lstm,
            color='red')
plt.scatter(range(len(y_test)),
            y_test,
            color='green')
plt.title('LSTM - Predicted label vs. Actual label')
plt.xlabel('SMS Messages')
plt.ylabel('Label')
plt.savefig('plots/lstm_predicted_vs_real.png',
            facecolor='white')
plt.show()

#%% md
```

Densely Connected CNN (DenseNet) Model

A DenseNet is a type of Convolutional Neural Network (CNN) that utilises dense connections between layers, where all layers with matching feature-map sizes are connected directly with each other. With the dense connections, higher accuracy is achieved with fewer parameters compared to a traditional CNN.

```
#%%

# Densely Connected CNN (DenseNet) architecture hyperparameters

# SpatialDropout1D is used to dropout the embedding layer which helps
# to drop entire 1D feature maps instead of individual elements
# dropout_rate indicates the fraction of the units to drop for the
# linear transformation of the inputs
dropout_rate = 0.2

# embedding_dimension indicates the dimension of the state space used for
# reconstruction
embedding_dimension = 16

# no_of_epochs indicates the number of complete passes through the
# training dataset
no_of_epochs = 30

# vocabulary_size indicates the maximum number of unique words to
# tokenize and load in training and testing data
vocabulary_size = 500

#%%

# importing Sequential class from keras
# Sequential groups a linear stack of layers into a keras Model
from tensorflow.keras.models import Sequential

# importing Embedding class from keras layers api package
# turning positive integers (indexes) into dense vectors of fixed size
# this layer can only be used as the first layer in a model
from tensorflow.keras.layers import Embedding

# importing GlobalAveragePooling1D class from keras layers api package
# used to perform global average pooling operation for temporal data
from tensorflow.keras.layers import GlobalAveragePooling1D

# importing Dropout class from keras layers api package
# used to apply Dropout to the input
# Dropout is one of the most effective and most commonly used
# regularization techniques for neural networks
# Dropout, applied to a layer, consists of randomly 'dropping out'
# (set to zero) a number of output features of the layer during training
from tensorflow.keras.layers import Dropout

# importing Dense class from keras layers api package
# Dense class is a regular densely-connected neural network layer
# Dense implements the operation:
# output = activation(dot(input, kernel) + bias)
# activation is the element-wise activation function passed as
# the activation argument
```

```
# kernel is a weights matrix created by the layer
# bias is a bias vector created by the layer
# (only applicable if use_bias is True)
from tensorflow.keras.layers import Dense

#%%

# Densely Connected CNN (DenseNet) model architecture

densenet_cnn_model = Sequential()

densenet_cnn_model.add(Embedding(vocabulary_size,
                                 embedding_dimension,
                                 input_length=maximum_length))

densenet_cnn_model.add(GlobalAveragePooling1D())

# Rectified Linear Activation Function (ReLU) is a piecewise linear
# function that will output the input directly if it is positive,
# otherwise, it will output zero
densenet_cnn_model.add(Dense(24,
                             activation='relu'))

densenet_cnn_model.add(Dropout(dropout_rate))

# sigmoid is a non-linear and easy to work with activation function
# that takes a value as input and outputs another value between 0 and 1
densenet_cnn_model.add(Dense(1,
                             activation='sigmoid'))

#%%

# compiling the model
# configuring the model for training
densenet_cnn_model.compile(loss='binary_crossentropy',
                           optimizer='adam',
                           metrics=['accuracy'])

#%%

# printing a string summary of the network
densenet_cnn_model.summary()

#%%

# monitoring the validation loss and if the validation loss is not
# improved after three epochs, then the model training is stopped
# it helps to avoid overfitting problem and indicates when to stop
# training before the deep learning model begins overfitting
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=3)

#%%
```

```
# training the DenseNet CNN model
history = densenet_cnn_model.fit(X_train_padded,
                                 y_train,
                                 epochs=no_of_epochs,
                                 validation_data=(X_test_padded, y_test),
                                 callbacks=[early_stopping],
                                 verbose=2)

#%%

# visualizing the history results by reading as a dataframe
densenet_cnn_metrics = pd.DataFrame(history.history)
densenet_cnn_metrics

#%%

# renaming the column names of the dataframe
densenet_cnn_metrics.rename(columns={'loss': 'Training_Loss',
                                     'accuracy': 'Training_Accuracy',
                                     'val_loss': 'Validation_Loss',
                                     'val_accuracy':
'Validation_Accuracy'},
                            inplace=True)
densenet_cnn_metrics

#%%

# plotting the training and validation loss by number of epochs for
# the DenseNet CNN model
densenet_cnn_metrics[['Training_Loss', 'Validation_Loss']].plot()
plt.title('DenseNet CNN Model - Training and Validation Loss vs. Epochs')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss')
plt.legend(['Training_Loss', 'Validation_Loss'])
plt.savefig('plots/densenet_cnn_loss_vs_epochs.png',
            facecolor='white')
plt.show()

#%%

# plotting the training and validation accuracy by number of epochs
# for the DenseNet CNN model
densenet_cnn_metrics[['Training_Accuracy', 'Validation_Accuracy']].plot()
plt.title('DenseNet CNN Model - Training and Validation Accuracy vs.
Epochs')
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy')
plt.legend(['Training_Accuracy', 'Validation_Accuracy'])
plt.savefig('plots/densenet_cnn_accuracy_vs_epochs.png',
            facecolor='white')
plt.show()

#%%
```

```
# saving the trained DenseNet CNN model as an h5 file
# h5 is a file format to store structured data
# keras saves deep learning models in this format as it can easily store
# the weights and model configuration in a single file
densenet_cnn_path = 'models/densenet_cnn_model.h5'
densenet_cnn_model.save(densenet_cnn_path)
densenet_cnn_model

#%%

# loading the saved DenseNet CNN model
loaded_densenet_cnn_model = load_model(densenet_cnn_path)
loaded_densenet_cnn_model

#%% md

# Densely Connected CNN (DenseNet) Model Evaluation

#%%

# evaluating the DenseNet CNN model performance on test data
# validation loss = 0.11119994521141052
# validation accuracy = 0.9732824563980103
loaded_densenet_cnn_model.evaluate(X_test_padded,
                                   y_test)

#%%

# predicting labels of X_test data values on the basis of the
# trained model
y_pred_densenet_cnn = [1 if x[0] > 0.5 else 0 for x in
loaded_densenet_cnn_model.predict(X_test_padded)]

# printing the length of the predictions list
len(y_pred_densenet_cnn)

#%%

# printing the first 25 elements of the predictions list
y_pred_densenet_cnn[:25]

#%%

# mean squared error (MSE)
print('MSE :', mean_squared_error(y_test,
                                  y_pred_densenet_cnn))

# root mean squared error (RMSE)
# square root of the average of squared differences between predicted
# and actual value of variable
print('RMSE:', mean_squared_error(y_test,
                                  y_pred_densenet_cnn,
                                  squared=False))
```

```python
#%%

# mean absolute error (MAE)
print('MAE:', mean_absolute_error(y_test,
                                  y_pred_densenet_cnn))

#%%

# accuracy
# ratio of the number of correct predictions to the total number of
# input samples
print('Accuracy:', accuracy_score(y_test,
                                  y_pred_densenet_cnn))

#%%

print('\t\t\tPrecision \t\tRecall \t\tF-Measure \tSupport')

# computing precision, recall, f-measure and support for each class
# with average='micro'
print('average=micro    -', precision_recall_fscore_support(y_test,

y_pred_densenet_cnn,

average='micro'))

# computing precision, recall, f-measure and support for each class
# with average='macro'
print('average=macro    -', precision_recall_fscore_support(y_test,

y_pred_densenet_cnn,

average='macro'))

# computing precision, recall, f-measure and support for each class
# with average='weighted'
print('average=weighted -', precision_recall_fscore_support(y_test,

y_pred_densenet_cnn,

average='weighted'))

#%%

# report shows the main classification metrics precision, recall and
# f1-score on a per-class basis
print(classification_report(y_test,
                            y_pred_densenet_cnn))

#%%

# confusion matrix is a summarized table used to assess the performance
# of a classification model
# number of correct and incorrect predictions are summarized with their
```

```python
# count according to each class
print(confusion_matrix(y_test,
                       y_pred_densenet_cnn))

#%%

# plotting the confusion matrix
cm = confusion_matrix(y_test,
                      y_pred_densenet_cnn)
sns.heatmap(cm,
            annot=True,
            cbar=False,
            fmt='g')
plt.title('DenseNet CNN Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.savefig('plots/densenet_cnn_confusion_matrix.png',
            facecolor='white')
plt.show()

#%%

# plotting scatter plot to visualize overlapping of predicted and
# test target data points
plt.scatter(range(len(y_pred_densenet_cnn)),
            y_pred_densenet_cnn,
            color='red')
plt.scatter(range(len(y_test)),
            y_test,
            color='green')
plt.title('DenseNet CNN - Predicted label vs. Actual label')
plt.xlabel('SMS Messages')
plt.ylabel('Label')
plt.savefig('plots/densenet_cnn_predicted_vs_real.png',
            facecolor='white')
plt.show()

#%% md

# Making Predictions with Real-World Examples



#%%

# defining pre-processing hyperparameters

# maximum_length indicates the maximum number of words considered
# in a text
maximum_length = 50

# truncating_type indicates removal of values from sequences larger
# than maxlen, either at the beginning ('pre') or at the end ('post')
# of the sequences
```

```python
truncating_type = 'post'

# padding_type indicates pad either before ('pre') or after ('post')
# each sequence
padding_type = 'post'

#%%

# defining a function to preprocess the sms message text to feed to the
# trained deep learning models
# the input text is transformed to a sequence of integers
# then the sequence is padded to the same length
# padding='post' to pad after each sequence
# the function returns the padded sequence
def preprocess_text(sms_messages):
    sequence = tokenizer.texts_to_sequences(sms_messages)
    padded_sequence = pad_sequences(sequence,
                                    maxlen=maximum_length,
                                    padding=padding_type,
                                    truncating=truncating_type)
    return padded_sequence

#%%

# defining a set of real-world samples for ham and spam sms messages
sms_messages = [
    'IMPORTANT - You could be entitled up to £3,160 in compensation from
mis-sold PPI on a credit card or loan. Please reply PPI for info or STOP
to opt out.',
    'Hello, Janith! Did you go to the school yesterday? If you did, can
you please send me the notes of all the subjects?',
    'Congratulations ur awarded 500 of CD vouchers or 125 gift guaranteed
& Free entry 2 100 wkly draw txt MUSIC to 87066.',
    'A loan for £950 is approved for you if you receive this SMS. 1 min
verification & cash in 1 hr at www.abc.co.uk to opt out reply stop',
    'If he started searching, he will get job in few days. He has great
potential and talent.',
    'One chance ONLY! Had your mobile 11mths+? You are entitled to update
to the latest colour camera mobile for FREE! Call The Mobile Update Co
FREE on 08002986906.',
    'Valentines Day Special! Win over 1000 USD in cash in our quiz and
take your partner on the trip of a lifetime! Send GO to 83600 now. 150
p/msg rcvd.',
    'Now I am better. Made up for Friday and stuffed myself like a pig
yesterday. Now I feel bad.',
    'I got another job! The one at the hospital, doing data analysis or
something, starts on Monday! Not sure when my thesis will finish.'
]

#%%

# invoking the preprocess_text function to preprocess and get the
# padded sequences of the set of real-world samples for ham and
# spam sms messages
```

```
padded_sequences = preprocess_text(sms_messages)
padded_sequences

#%%

# making prediction for the given set of real-world sms messages
# using the trained LSTM model

print('LSTM Model Predictions',
      end='\n\n')

lstm_prediction = []

for index, sms_message in enumerate(sms_messages):
    prediction = loaded_lstm_model.predict(padded_sequences)[index][0][0]
    lstm_prediction.append(prediction)
    if prediction > 0.5:
        print('SPAM -', prediction, '-', sms_message)
    else:
        print('HAM  -', prediction, '-', sms_message)

#%%

# making prediction for the given set of real-world sms messages
# using the trained DenseNet CNN model

print('DenseNet CNN Model Predictions',
      end='\n\n')

densenet_cnn_prediction = []

for index, sms_message in enumerate(sms_messages):
    prediction =
loaded_densenet_cnn_model.predict(padded_sequences)[index][0]
    densenet_cnn_prediction.append(prediction)
    if prediction > 0.5:
        print('SPAM -', prediction, '-', sms_message)
    else:
        print('HAM  -', prediction, '-', sms_message)

#%% md

# Comparison of Deep Learning Models

#%%

# plotting line graphs of the predicted values for LSTM and
# DenseNet CNN deep learning models to compare the separation
# of classes by each model
plt.plot(list(range(len(sms_messages))),
         lstm_prediction,
         label='LSTM',
         color='blue',
         marker='o')
```

```python
plt.plot(list(range(len(sms_messages))),
         densenet_cnn_prediction,
         label='CNN',
         color='red',
         marker='o')
plt.plot(list(range(len(sms_messages))),
         [0.5 for x in range(len(sms_messages))],
         color='black',
         linestyle='dashed')
plt.title('Comparison of Predicted Values by LSTM and DenseNet Models')
plt.xlabel('SMS Message ID')
plt.ylabel('Predicted Value')
plt.legend(loc='upper right',
           bbox_to_anchor=(1, 1))
plt.savefig('plots/prediction_values_comparison.png',
            facecolor='white')
plt.show()

#%%

# importing recall_score from scikit-learn library
from sklearn.metrics import recall_score

# importing precision_score from scikit-learn library
from sklearn.metrics import precision_score

# importing f1_score from scikit-learn library
from sklearn.metrics import f1_score

accuracy = {}
recall = {}
precision = {}
f1 = {}
rmse = {}
mae = {}

y_pred_dict = {
    'LSTM': y_pred_lstm,
    'CNN': y_pred_densenet_cnn
}

for y_pred in y_pred_dict:
    accuracy[y_pred] = accuracy_score(y_test,
                                      y_pred_dict[y_pred])
    recall[y_pred] = recall_score(y_test,
                                  y_pred_dict[y_pred],
                                  average='weighted')
    precision[y_pred] = precision_score(y_test,
                                        y_pred_dict[y_pred],
                                        average='weighted')
    f1[y_pred] = f1_score(y_test,
                          y_pred_dict[y_pred],
                          average='weighted')
    rmse[y_pred] = mean_squared_error(y_test,
```

```
                                          y_pred_dict[y_pred],
                                          squared=False)
    mae[y_pred] = mean_absolute_error(y_test,
                                      y_pred_dict[y_pred])

# ratio of the number of correct predictions to the total number
# of input samples
print('Accuracy:', accuracy)

# recall is the ratio tp / (tp + fn)
# tp is the number of true positives
# fn the number of false negatives
print('Recall:', recall)

# precision is the ratio tp / (tp + fp)
# tp is the number of true positives
# fp the number of false positives
print('Precision:', precision)

# F1 score is also known as balanced F-score or F-measure
# F1 score can be interpreted as a weighted average of the
# precision and recall
# F1 = 2 * (precision * recall) / (precision + recall)
print('F1 Score:', f1)

# root mean squared error (RMSE)
# square root of the average of squared differences between predicted
# and actual value of variable
print('RMSE:', rmse)

# mean absolute error (MAE)
print('MAE:', mae)

#%%

# sorting the accuracy scores of two deep learning models in
# descending order
sorted(accuracy.items(),
       key=lambda kv: kv[1],
       reverse=True)

#%%

# plotting the accuracy comparison bar chart for the two
# deep learning models
fig, ax = plt.subplots(figsize=(5, 5))
plt.bar(y_pred_dict.keys(),
        accuracy.values(),
        color='rg')
plt.title('Accuracy Comparison')
plt.xlabel('Algorithm')
plt.ylabel('Accuracy')
plt.savefig('plots/accuracy_comparison.png',
            facecolor='white')
```

```python
plt.show()

#%%

# defining a function to plot a bar chart with multiple bars
def bar_plot(ax,
             data,
             colors=None,
             total_width=0.8,
             single_width=1,
             legend=True):
    if colors is None:
        colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
    n_bars = len(data)
    bar_width = total_width / n_bars
    bars = []
    for i, (name, values) in enumerate(data.items()):
        x_offset = (i - n_bars / 2) * bar_width + bar_width / 2
        for x, y in enumerate(values):
            bar = ax.bar(x + x_offset,
                         y,
                         width=bar_width * single_width,
                         color=colors[i % len(colors)])
        bars.append(bar[0])
    if legend:
        ax.legend(bars,
                  data.keys())

#%%

# plotting the algorithm comparison chart for all evaluation metrics

data = {}

for key in y_pred_dict.keys():
    data[key] = [accuracy[key],
                 recall[key],
                 precision[key],
                 f1[key],
                 rmse[key],
                 mae[key]]

fig, ax = plt.subplots(figsize=(7, 5))

bar_plot(ax,
         data,
         total_width=0.9,
         single_width=0.9)

plt.title('Algorithm Comparison')
plt.xlabel('Metrics')
plt.ylabel('Scores')
plt.xticks(range(len(data[key])),
           ['Accuracy', 'Recall', 'Precision', 'F1-Score', 'RMSE', 'MAE'])
```

```
plt.savefig('plots/algorithm_comparison.png',
            facecolor='white')
plt.show()

#%%
```