

**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].

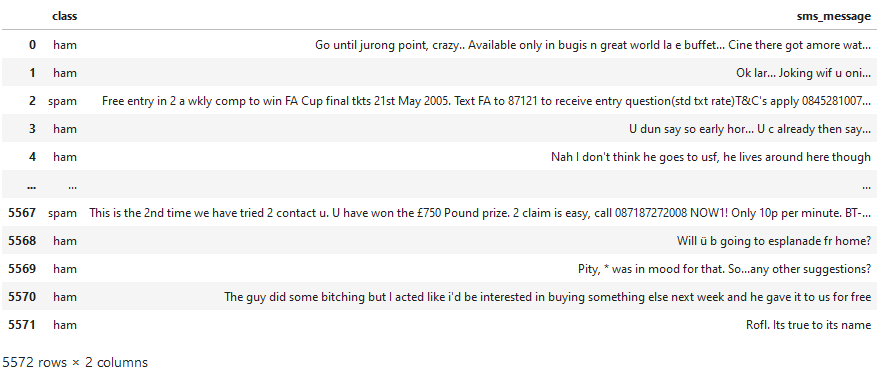


Figure 1: Sample data instances of SMS Spam Collection Dataset

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].

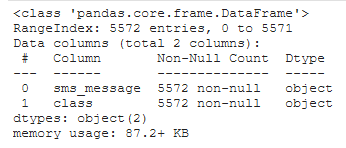


Figure 2: Concise summary of the dataframe

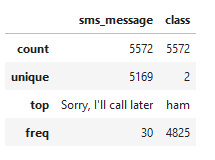


Figure 3: Descriptive statistics of the dataframe

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.

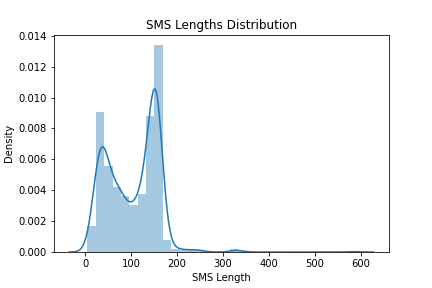


Figure 4: SMS lengths distribution plot

1. **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.

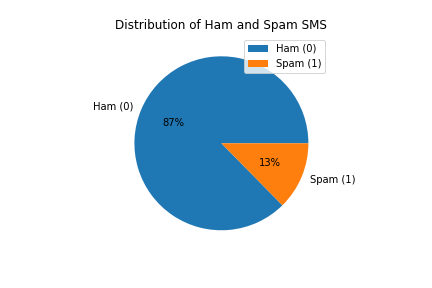


Figure 5: Distribution of ham and spam SMS plot

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.

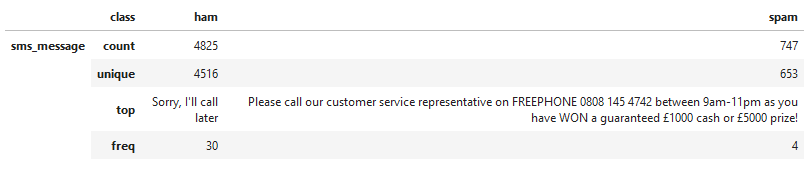


Figure 6: Descriptive statistics for each class of the dataframe

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')

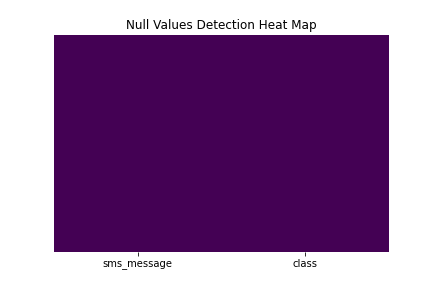


Figure 9: Null values detection heat map

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)

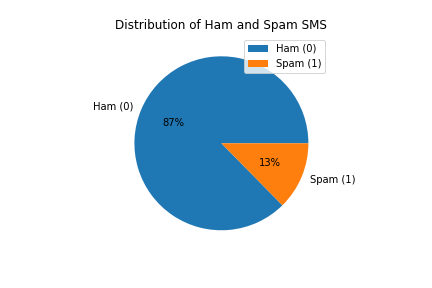


Figure 10: Distribution of ham and spam SMS plot before downsampling

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)

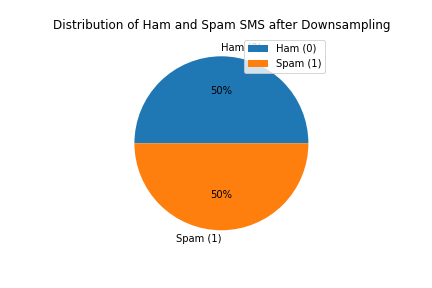


Figure 11: Distribution of ham and spam SMS plot after downsampling

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)

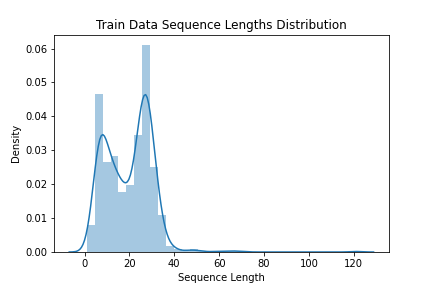
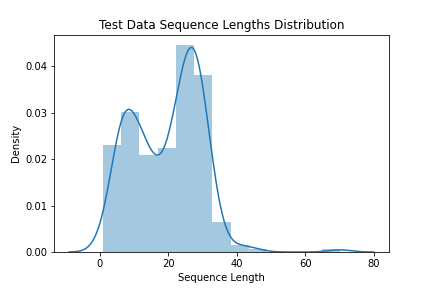


Figure 13: Train data sequence lengths distribution plot

Figure 12: 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)

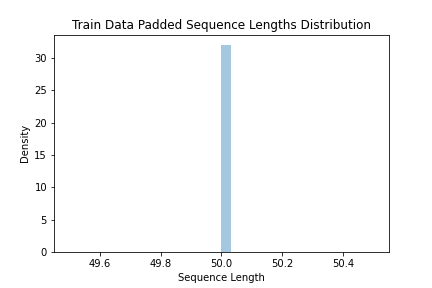
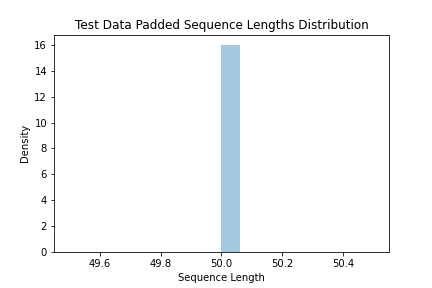


Figure 15: Test data padded sequence lengths distribution plot

Figure 14: Train 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 SMS message spam detection problem can be addressed through supervised learning techniques. In supervised learning, a data set, with a set of features and labels, is fed to the learning algorithm. When the algorithm is fully trained, it will correctly identify the relationships between features and their labels. Therefore, it will be able to accurately predict the label for a previously unseen set of features. Since the SMS Spam Collection dataset contains labeled data a supervised learning algorithm can be used for this problem. Therefore, we have decided to experiment with the two Deep Learning algorithms namely Long Short-Term Memory (LSTM) and Densely Connected Convolutional Neural Networks (DenseNet CNN) which are widely used for supervised learning problems.

There are two main types of supervised learning problems called classification problems and regression problems. Regression is used when the supervised learning problem is predicting a numerical label. Classification is used when the supervised learning problem is predicting a class label. Since the SMS Spam Collection dataset included a column called “class” with two labels called ham (0) and spam (1), this problem should be addressed through a classification algorithm. There are two types of classification algorithms called binary classification and multi-class classification. Binary classification categorizes data into one of two categories such as True (1) or False (0). Multiclass classification categorizes data into one of the multiple number of classes. Since the spam detection problem’s target variable contains only two classes (ham - 0 and spam - 1), binary classification is applicable for this scenario. Both LSTM and DenseNet neural network architectures are applicable for binary classification problems.

The SMS Spam Collection dataset contains raw text as the feature attribute. These SMS message texts consist of complex and unstructured text data with letters, numbers, punctuations, and special characters. Further, the analysis we performed on the dataset revealed that there were texts even exceeding 600 characters for a single value. The majority of text data at least had about 50 to 200 characters. Most NLP research and industry applications have proven that both LSTM and DenseNet neural network architectures are heavily used in these types of text classification problems because of their ability to process this kind of complex data structure and provide accurate results.

Further, both LSTM and DenseNet neural network architectures are very simple, easy-to-understand algorithms that are also easier to implement. Only a few parameters are there to fine-tune. They are also known to provide higher accuracy when compared to other Deep Learning models in terms of text classification problems. The evaluation of the models proved that the DenseNet outperformed the LSTM model which will be discussed deeply in the Results and Discussion section of this report.

Introduction and Background of LSTM Model

Long Short-Term Memory (LSTM) networks are a special kind of Recurrent Neural Network (RNN) which was introduced by Hochreiter and Schmidhuber in 1997 [11]. LSTM networks are capable of learning long-term dependencies [11] and can process complex sequences of unstructured data such as text. They are widely used to solve many real-world complex problems especially in NLP and computational linguistics domains such as machine translation, sentiment analysis, text classification, spam detection, speech recognition, time series prediction, and handwritten character recognition [12].

An LSTM unit consists of a cell, an input gate, an output gate and a forget gate as shown in Figure 16. The three gates control the flow of information into and out of the cell, and the cell remembers values across random time intervals [13]. The forget gate decides what relevant information from the previous steps is required. The input gate determines which relevant information could be added from the current step, and the output gate is responsible for finalizing the next hidden state [13].

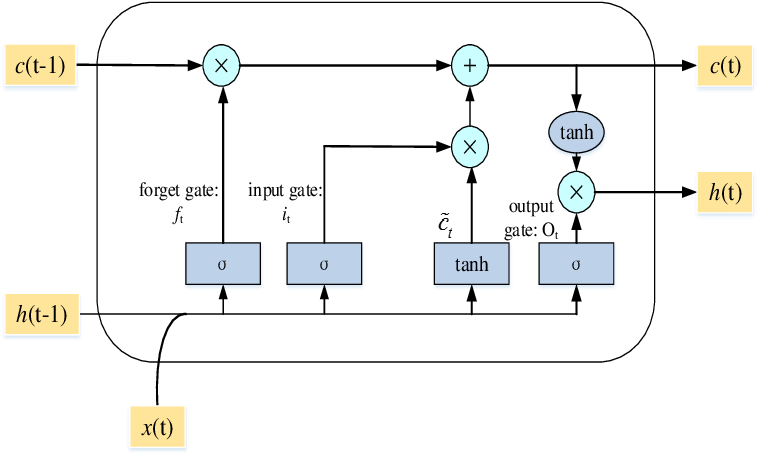


Figure 16: LSTM architecture

LSTM networks have many advantages over traditional RNNs. It is explicitly designed to avoid the long-term dependency problem of RNN [14]. LSTM networks use special units in addition to standard units within their architecture. LSTM units include a “memory cell” and they can maintain information in memory for longer periods. Therefore, the memory cell helps the LSTM model to learn longer-term dependencies [14]. Furthermore, LSTM models are capable of dealing with vanishing and exploding gradient problems successfully unlike the traditional RNN models. It introduces new gates like input and forget gates which allows for better control over the gradient flow and enables better preservation of “long-range dependencies” [14]. Another advantage of LSTM networks is that it provides a large range of parameters like learning rates and input-output biases. Therefore, fine adjustments for the model are not required [15].

Even though LSTM networks provide many advantages, there are some drawbacks as well [15]. LSTM models usually require lots of resources and time for the training process. Further, they are highly prone to overfitting, and applying dropout regularizations is difficult [15]. With the rapid advancement of data mining methods, most Deep Learning researchers are looking for models that can keep past information for a longer time than LSTMs [15].

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()

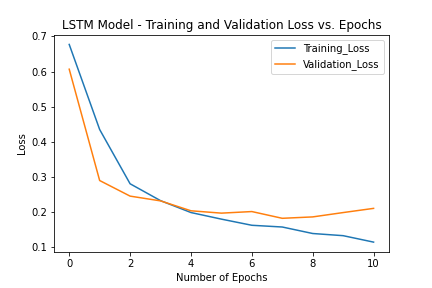


Figure 17: LSTM model - training and validation loss vs. epochs

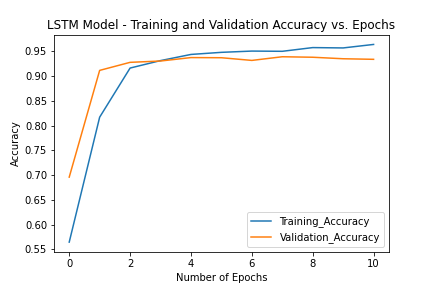


Figure 18: LSTM model - training and validation accuracy vs. epochs

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)

Introduction and Background of DenseNet CNN Model

Dense Convolutional Network (DenseNet) is a special type of Convolutional Neural Network (CNN) which was introduced very recently by Huang, Liu, Van Der Maaten, and Weinberger in 2017 [16]. DenseNets connect all of its layers with each other by using the feature-maps of all preceding layers as the inputs for each layer [16] in the network. DenseNets are used for many complex applications like image classification, object detection, and text classification.

In DenseNet architecture, each layer gets additional inputs from all preceding layers and provides its feature-maps to all subsequent layers [17]. Therefore, each layer in the network receives a “collective knowledge” from all preceding layers as shown in Figure 17. Even though traditional CNNs with n number of layers have only n number of connections (one connection between each layer and its subsequent layer), DenseNet has n (n + 1) / 2 connections [16, 17].

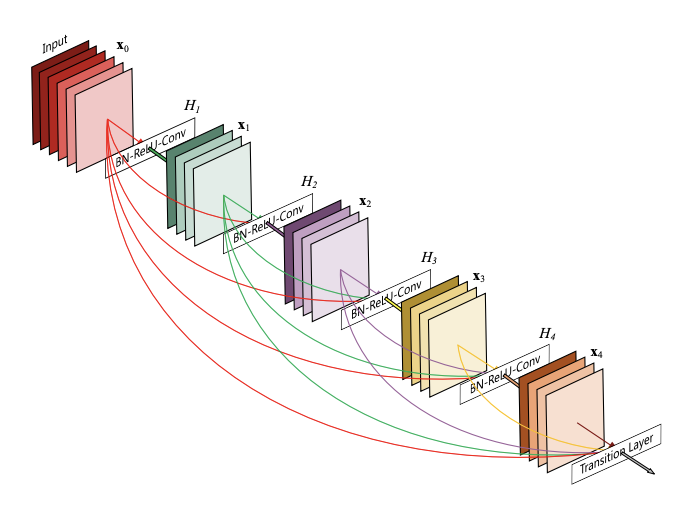


Figure 17: DenseNet architecture

As a novel Deep Learning neural network architecture, DenseNet provides many advantages for real-world problem-solving. DenseNet reduces the vanishing-gradient problem that exists in traditional deep learning approaches [16]. Furthermore, this architecture reduces the number of parameters and required computational power while providing higher performances [16]. Further, the dense connections in the DenseNet models have a regularizing effect. Therefore, overfitting is reduced in training processed with smaller training set sizes [16].

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()

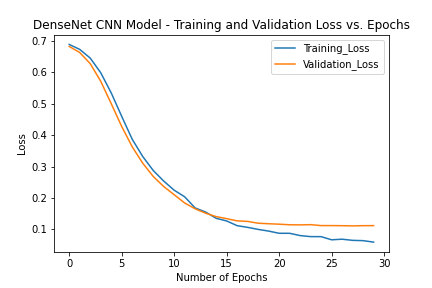


Figure 20: DenseNet CNN model - training and validation loss vs. epochs

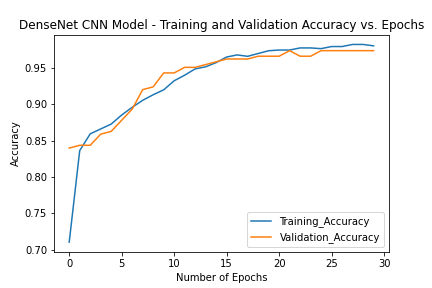


Figure 21: DenseNet CNN model - training and validation accuracy vs. epochs

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)

Results and Discussion (Deep Learning Models Evaluation and Comparison)

Short Message Service (SMS) has become one of the most common and heavily used ways of communication in society today.

Accuracy Improvement Techniques, Limitations, and Future Work

Short Message Service (SMS) has become one of the most common and heavily used ways of communication in society today.

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] C. Olah, "Understanding LSTM Networks -- colah's blog", Colah.github.io, 2021. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[12] J. Brownlee, "A Gentle Introduction to Long Short-Term Memory Networks by the Experts", Machine Learning Mastery, 2017. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/.

[13] G. Singhal, "Introduction to LSTM Units in RNN | Pluralsight", Pluralsight.com, 2020. [Online]. Available: https://www.pluralsight.com/guides/introduction-to-lstm-units-in-rnn.

[14] A. Tripathi, "What is the main difference between RNN and LSTM | NLP | RNN vs LSTM", ashutoshtripathi.com, 2021. [Online]. Available: https://ashutoshtripathi.com/2021/07/02/what-is-the-main-difference-between-rnn-and-lstm-nlp-rnn-vs-lstm/.

[15] "Understanding of LSTM Networks - GeeksforGeeks", swww.geeksforgeeks.org, 2021. [Online]. Available: https://www.geeksforgeeks.org/understanding-of-lstm-networks/.

[16] G. Huang, Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2261-2269, doi: 10.1109/CVPR.2017.243.

[17] S. Tsang, "Review: DenseNet — Dense Convolutional Network (Image Classification)", towardsdatascience.com, 2018. [Online]. Available: https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803.

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)]  
  
*# 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'**)  
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*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**))  
  
*#%%  
  
# 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  
# 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*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'**)  
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'**)  
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()  
  
*#%%*