COMSATS University Islamabad (Lahore Campus)

Assignment <2> FALL 2022

Course Title:	Computational Intelligence			Course Code:		CSC	Credit Hours: 3(2,1)
Course Instructor:	Dr. Atifa Athar			Programme Name:		BS Computer Science	
Semester:	7th	Batch:	FA19	Section:	**B**	Date:	14-1-2022
Due Date:	**04-11-2022**			**Maximum Marks:**			**10**
Student Name:	**Muhammad Awais Abdullah Muhammad Huzaifa Tariq**			**Registration No.**			**FA19-BCS-105 FA19-BCS-114 FA19-BCS-047**

Important Instructions / Guidelines:

- **No late submissions will be accepted.**
- **All assignments are required to be submitted using attached template only.**

Question No 1. **Marks: 10**

CLO: <3>; Bloom Taxonomy Level: <Applying>

Select a textual dataset of your choice (for example IMDB movies review data). Design and implement RNN and LSTM Neural Network, Bert model (along with its variants) in Python using your selected dataset.

Submit a complete report including an introduction to the dataset. implementation, Results (accuracy tables and confusion matrix), and comparisons. Along with the Report, you are required to submit your code as well.

Problem Statement:

Design, implement, evaluate and compare different NLP techniques for tweet classification such as RNN & LSTM, CNN, CNN+LSTM, and BERT in python on the COVID-19 vaccines dataset.

Introduction to Dataset:

The dataset we have selected for this assignment is “Tweets Dataset on COVID-19 Vaccines”. It was collected by social media posts on COVID-19 vaccines using Twitter’s streaming API from December 9, 2020 - Feb 24, 2021, with keywords related to the vaccines ("Vaccine", "Pfizer", "BioNTech", "Moderna", "Janssen", "AstraZeneca", "Sinopharm"). This dataset includes CSV files that contain IDs, text, labels, and sixty other columns related to the COVID-19 tweets. On social media, content propagates through the network when accounts engage with posts by re-sharing (retweeting), replying to tweets, and quoting (quote tweets are retweets without a comment). There are 376K entries in the raw dataset with most of the tweets as retweeted, so there are unique 14.6k tweets, roughly 10,377 labeled as reliable and 4,267 labeled as unreliable/conspiracy.

Data Info:

```
RangeIndex: 14644 entries, 0 to 14643
Data columns (total 69 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   cascade_sno             14644 non-null  int64
 1   cascade_tid             14644 non-null  int64
 2   tweetid                 14644 non-null  int64
 3   userid                  14644 non-null  int64
 4   screen_name             14644 non-null  object
 5   date                    14644 non-null  object
 6   lang                    14644 non-null  object
 7   location                10398 non-null  object
 8   place_id                52 non-null     object
 9   place_url               52 non-null     object
 10  place_type              52 non-null     object
 11  place_name              52 non-null     object
 12  place_full_name         52 non-null     object
 13  place_country_code      52 non-null     object
 14  place_country           52 non-null     object
 15  place_bounding_box      52 non-null     object
 16  text                    14644 non-null  object
 17  extended                14644 non-null  object
 18  coord                   0 non-null      float64
 19  reply_userid            388 non-null    float64
 20  reply_screen            388 non-null    object
 21  reply_statusid          388 non-null    float64
 22  tweet_type              14644 non-null  object
 23  friends_count           14644 non-null  int64
 24  listed_count            14644 non-null  int64
 25  followers_count         14644 non-null  int64
 26  favourites_count        14644 non-null  int64
 27  statuses_count          14644 non-null  int64
 28  verified                14644 non-null  bool
 29  hashtag                 14644 non-null  object
 30  urls_list               14644 non-null  object
 31  profile_pic_url         12200 non-null  object
 32  profile_banner_url      12200 non-null  object
 33  display_name            14643 non-null  object
 34  date_first_tweet        14644 non-null  object
 35  account_creation_date   14644 non-null  object
 36  rt_urls_list            14644 non-null  object
 37  mentionid               14644 non-null  object
 38  mentionsn               14644 non-null  object
 39  rt_screen               8821 non-null   object
 40  rt_userid               8821 non-null   float64
 41  rt_text                 8889 non-null   object
 42  rt_hashtag              14644 non-null  object
 43  rt_qtd_count            14644 non-null  int64
 44  rt_rt_count             14644 non-null  int64
 45  rt_reply_count          14644 non-null  int64
 46  rt_fav_count            14644 non-null  int64
 47  rt_tweetid              8889 non-null   float64
 48  rt_location             7181 non-null   object
 49  qtd_screen              129 non-null    object
 50  qtd_userid              129 non-null    float64
 51  qtd_text                132 non-null    object
```

```
52  qtd_hashtag            14644 non-null  object
53  qtd_qtd_count          14644 non-null  int64
54  qtd_rt_count           14644 non-null  int64
55  qtd_reply_count        14644 non-null  int64
56  qtd_fav_count          14644 non-null  int64
57  qtd_tweetid            132 non-null    float64
58  qtd_urls_list          14644 non-null  object
59  qtd_location           97 non-null     object
60  sent_vader             14644 non-null  float64
61  token                  14641 non-null  object
62  media_urls             14644 non-null  object
63  rt_media_urls          14644 non-null  object
64  q_media_urls           14644 non-null  object
65  corrected_tweet_type   14644 non-null  object
66  ns_label               14644 non-null  object
67  ns_url                 14644 non-null  object
68  timestamp              14644 non-null  float64
```

Data Preprocessing:

1. Clean all the tweets by converting them into lowercase and remove all unnecessary characters (and some unnecessarily repeated words).
2. Encode the string labels as “1” for unreliable/conspiracy and “0” for reliable.
3. Split the data into training and testing sets.
4. Generate tokens of the tweets data.
5. Generate sequences and add padding to the max length.

NLP techniques for Text Classification:

There are several natural language processing (NLP) techniques that can be used for text classification:

1. **Bag-of-Words:** This technique represents each text as a bag of its words, disregarding grammar and word order but keeping track of occurrences. This can be used as input for a classifier such as Naive Bayes.

2. **Term Frequency-Inverse Document Frequency (TF-IDF):** This technique assigns weight to each word in the text, with more weight given to words that are more informative and less common words. This can also be used as input for a classifier.

3. **Word Embeddings:** This technique represents each word in a text as a dense vector, which captures semantic and syntactic information about the word. These vectors can be used as input for a classifier such as a feed-forward neural network or a convolutional neural network.

4. **Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM):** These are types of neural networks that process sequential data, such as text. They can be trained in text classification tasks and can output a probability of the text belonging to each class.

5. **Transformers:** These are deep learning models that are particularly well suited for NLP tasks, such as text classification. They have been shown to be very effective in achieving state-of-the-art performance on a wide range of NLP tasks.

LSTM

LSTM stands for Long Short-Term Memory. It is a type of Recurrent Neural Network (RNN) that can capture long-term dependencies in sequential data. Unlike traditional RNNs, which tend to forget past information as the network unfolds over time, LSTMs have a built-in memory cell that can retain information for prolonged periods. This makes them well-suited for tasks such as language modeling, machine translation, and speech recognition.

Implementation: Implementing an LSTM network typically involves the following steps:

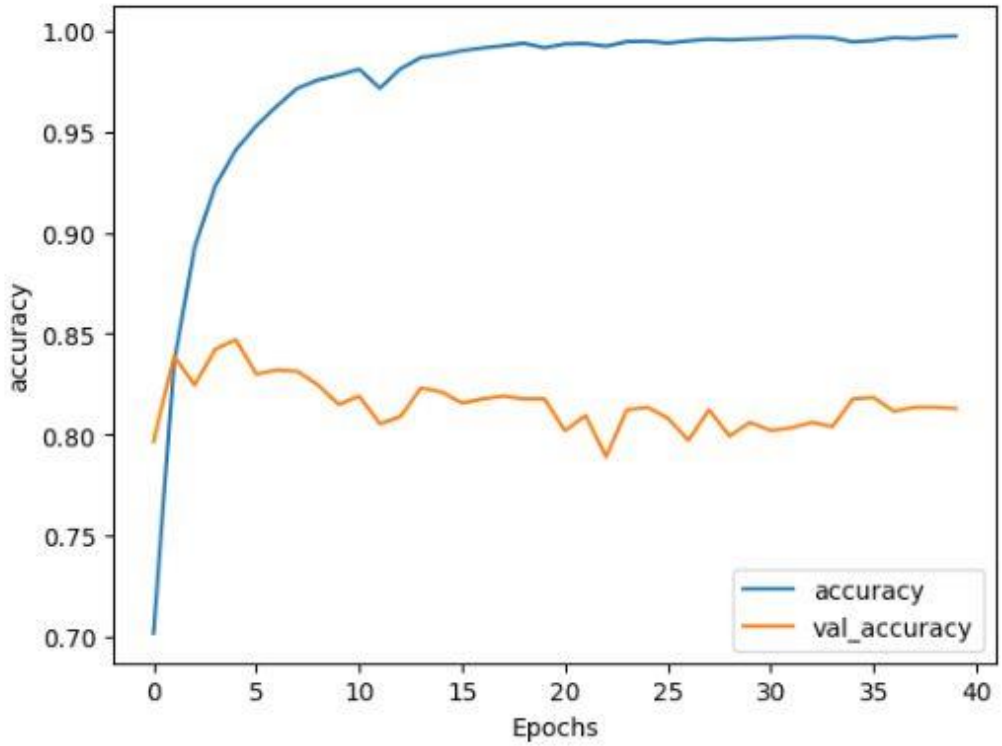
1. **Prepare the data:** The first step is to prepare the data by tokenizing and indexing the text, and then dividing it into training and testing sets.
2. **Choose the LSTM architecture:** The next step is to choose the LSTM architecture, which includes deciding on the number of layers, the number of units in each layer, and whether to use a bidirectional or unidirectional LSTM.
3. **Define the model:** The LSTM model can be defined using a deep learning library such as TensorFlow, Keras, or PyTorch. The model consists of an LSTM layer, followed by one or more fully connected layers.
4. **Compile the model:** Once the model is defined, it needs to be compiled by specifying the loss function, optimizer, and metrics.
5. **Train the model:** The model is then trained on the training data using the fit() function. The model will learn to associate the input sequences with the corresponding output labels.
6. **Evaluate the model:** After the model is trained, it can be evaluated on the test data using the evaluate() function, which will return the loss and any other metrics that were specified during compilation.

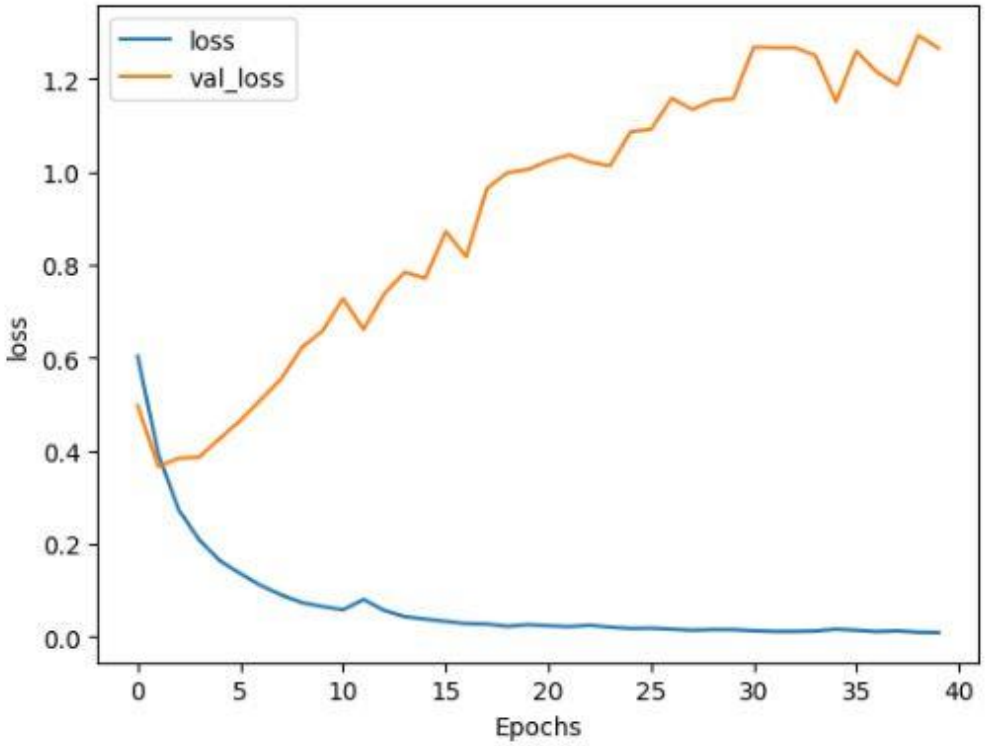
It's important to note that the specific implementation details of an LSTM will depend on the deep learning framework and the specific use case.

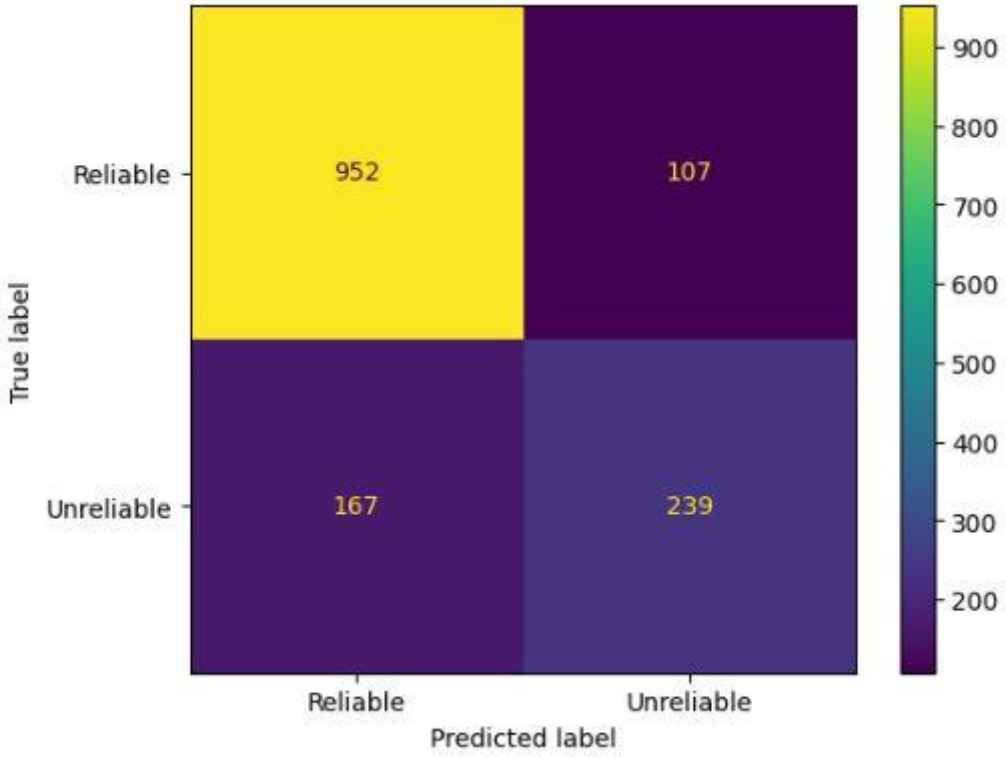
LSTM Model Summary:

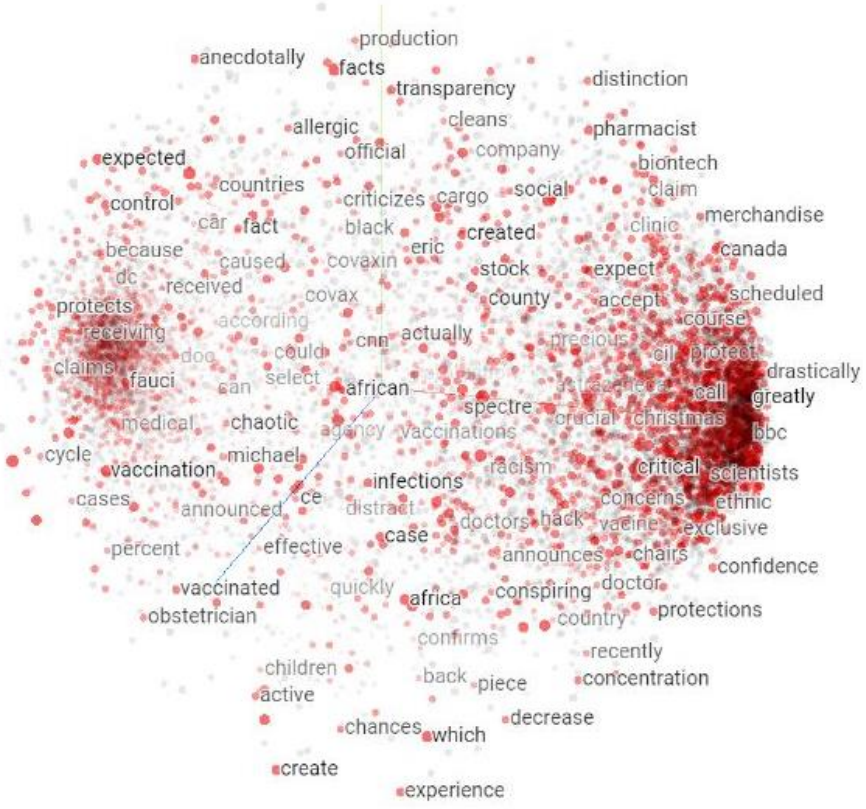
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 331, 16)           224000

 spatial_dropout1d (SpatialD  (None, 331, 16)          0
 ropout1D)

 bidirectional (Bidirectiona  (None, 64)               12544
 l)

 dense (Dense)               (None, 6)                 390

 dense_1 (Dense)             (None, 1)                 7

=================================================================
Total params: 236,941
Trainable params: 236,941
Non-trainable params: 0
_________________________________________________________________
```

	precision	recall	f1-score	support
Reliable	0.90	0.85	0.87	1119
UnReliable	0.59	0.69	0.64	346
accuracy			0.81	1465
macro avg	0.74	0.77	0.75	1465
weighted avg	0.83	0.81	0.82	1465


1.00
0.95
0.90
0.85
0.80
0.75
0.70
accuracy
0
5
10
15
20
25
30
35
40
Epochs
accuracy
val_accuracy


loss
val_loss
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
5
10
15
20
25
30
35
40
Epochs


Reliable
Unreliable
952
107
167
239
True label
Reliable
Unreliable
Predicted label
900
800
700
600
500
400
300
200


production
anecdotally
facts
transparency
distinction
allergic
cleans
pharmacist
official
company
biontech
expected
countries
social
claim
control
criticizes
cargo
merchandise
car
fact
black
created
clinic
because
eric
canada
caused
covaxin
stock
expect
dc
received
covax
county
accept
scheduled
protects
according
course
receiving
cnn
actually
precious
could
claims
doo
select
african
drastically
fauci
can
spectre
greatly
medical
chaotic
agency
vaccinations
christmas
bbc
cycle
michael
racism
critical
vaccination
infections
scientists
cases
announced
distract
concerns
ethnic
doctors
hack
vaccine
exclusive
case
percent
effective
announces
chairs
confidence
vaccinated
quickly
africa
conspiring
doctor
obstetrician
country
protections
confirms
recently
children
back
piece
concentration
active
decrease
chances
which
create
experience

GRU

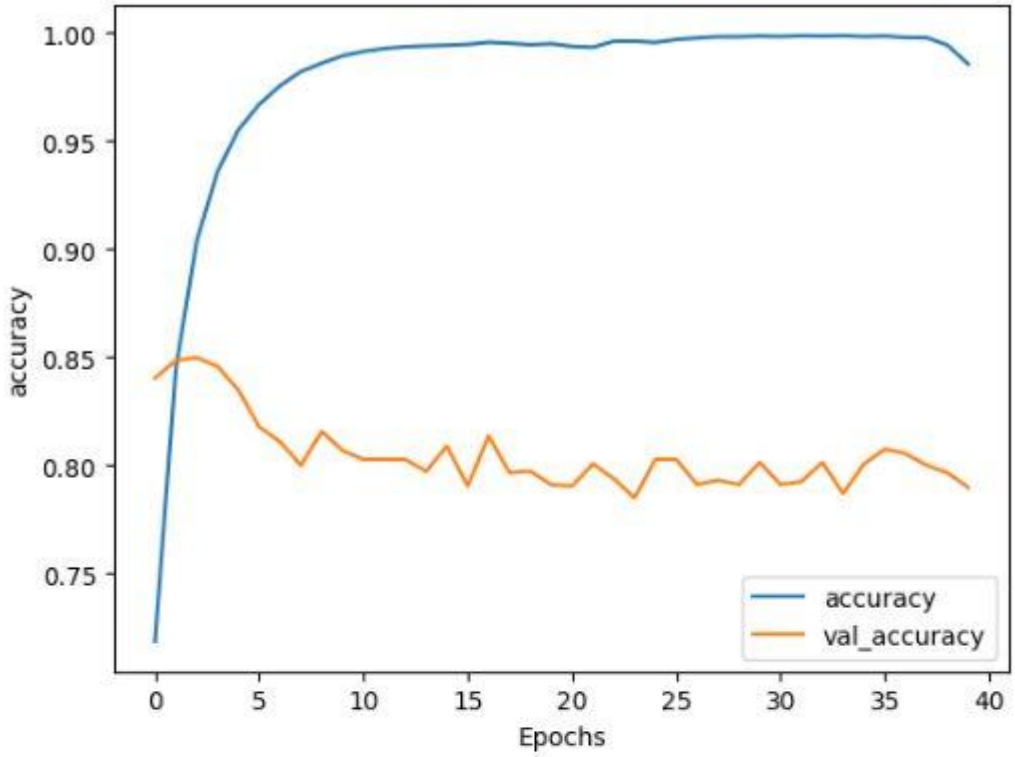
The Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture that addresses some of the limitations of traditional RNNs, such as the vanishing gradient problem. The GRU architecture utilizes gating mechanisms to control the flow of information through the network, allowing for better preservation of long-term dependencies and improved performance on tasks such as language modeling and speech recognition. The GRU was introduced by Cho et al. in 2014 and has since become a popular choice in RNN architectures.

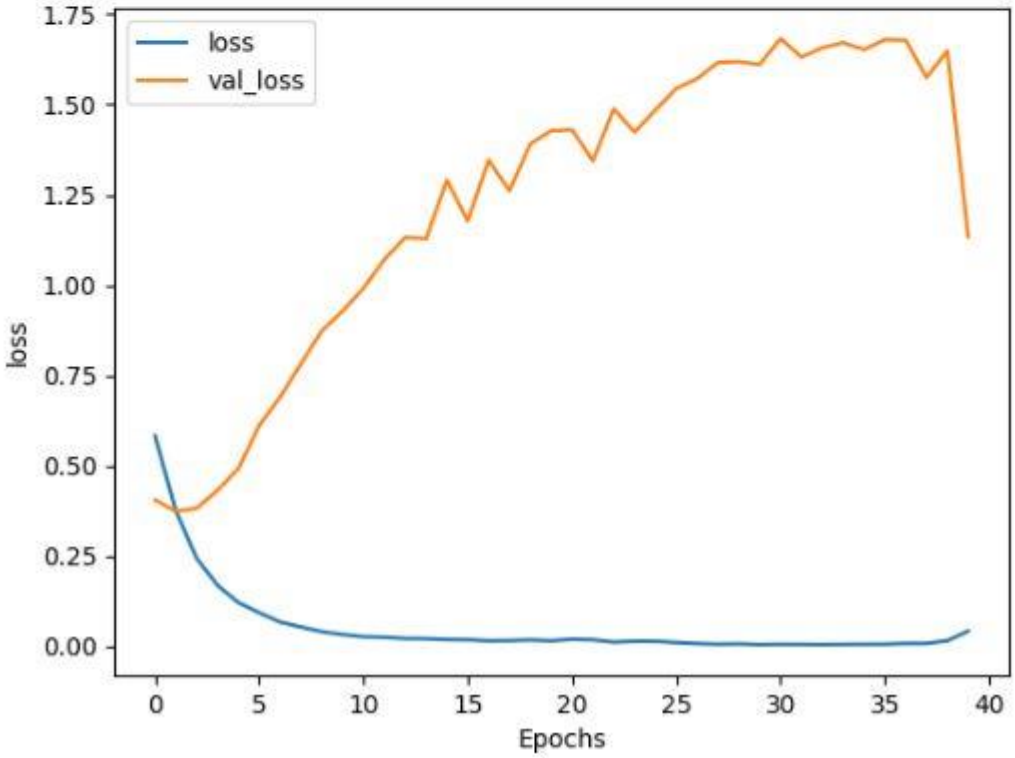
Implementation: Implementing a GRU in code involves several steps:

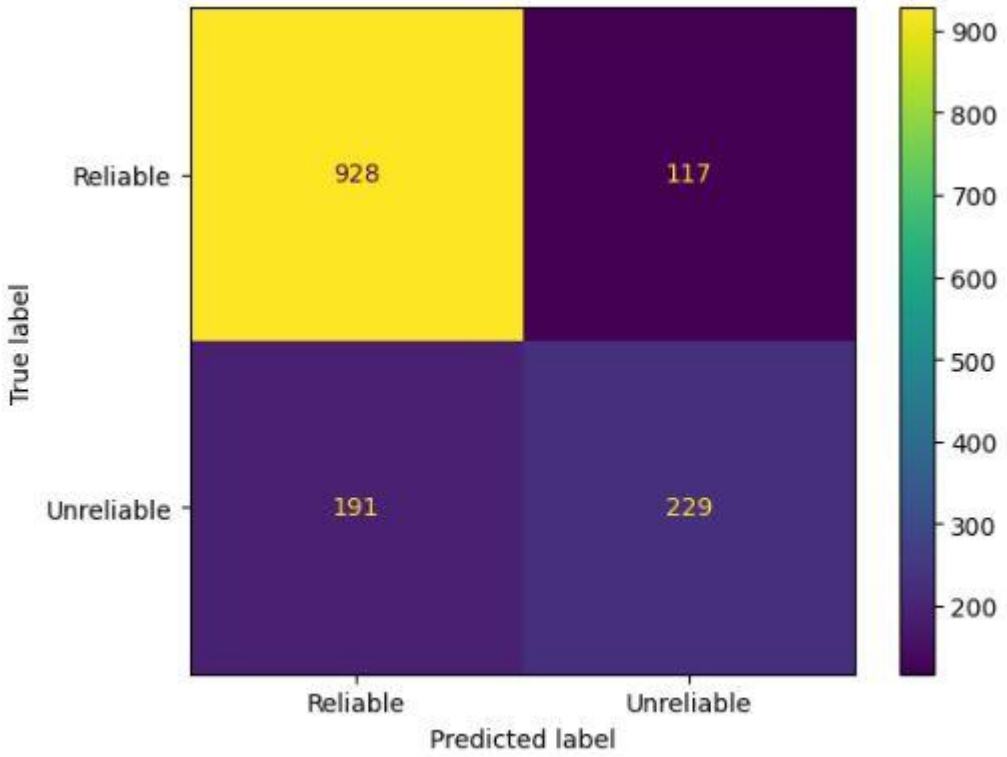
1. Initializing the GRU model. This typically includes defining the number of hidden units, the input dimension, and the number of layers in the model.
2. Defining the input and output layers. The input layer is typically an embedding layer, which converts the input words or sequences into a dense vector representation. The output layer is typically a linear layer that maps the hidden state of the GRU to the output of the model.
3. Defining the GRU layers. This typically involves creating instances of the GRU class, specifying the number of hidden units, and connecting the input and output layers to the GRU layers.
4. Training the model. This typically involves defining a loss function and an optimizer and then training the model on a dataset using a training loop.
5. Evaluating the model. This typically involves evaluating the model on a validation or test dataset and measuring its performance using metrics such as accuracy or perplexity.

GRU Model Summary:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 331, 16)           224000

 bidirectional_1 (Bidirectio  (None, 64)               9600
 nal)

 dense_2 (Dense)             (None, 6)                 390

 dense_3 (Dense)             (None, 1)                 7

=================================================================
Total params: 233,997
Trainable params: 233,997
Non-trainable params: 0
```


1.00
0.95
0.90
0.85
0.80
0.75
accuracy
0
5
10
15
20
25
30
35
40
Epochs
accuracy
val_accuracy


1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
loss
0
5
10
15
20
25
30
35
40
Epochs
loss
val_loss



	precision	recall	f1-score	support
Reliable	0.89	0.83	0.86	1119
UnReliable	0.55	0.66	0.60	346
accuracy			0.79	1465
macro avg	0.72	0.75	0.73	1465
weighted avg	0.81	0.79	0.80	1465

CNN

A Convolutional Neural Network (CNN) is a type of deep learning neural network that is designed to process and analyze data that has a grid-like structure, such as images, videos, and audio signals. CNNs are composed of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The convolutional layers are responsible for analyzing the input data and extracting features, such as edges, textures, and patterns. The pooling layers are responsible for reducing the spatial dimensions of the data while maintaining the important features. The fully connected layers are responsible for classifying the data based on the features extracted by the convolutional and pooling layers.

CNNs are particularly useful for image classification, object detection, and image segmentation tasks, as they are able to learn and extract features from images that are not easily represented by traditional image processing techniques. They have been widely adopted in many computer vision applications such as self-driving cars, image search engines, and facial recognition. CNNs can also be used for text classification tasks, in which the goal is to classify a piece of text into one or more predefined categories. Text classification is a widely used technique in natural language processing (NLP) and is used in tasks such as sentiment analysis, topic classification, and spam detection. In text classification, a CNN is typically applied to the embedding of words in a sentence or a document, rather than to the raw input text. The embedding is a dense vector representation of each word, which captures the semantic meaning of the word. The embedding is typically obtained by training a word embedding model, such as word2vec or GloVe, on a large corpus of text.

The CNN architecture for text classification is similar to the CNN architecture for image classification, but with some modifications to adapt it to text data. The convolutional layers are typically 1-dimensional (1D) convolutional layers, which are applied to the embedding of the words in a sentence or a document. The pooling layers are typically max-pooling layers, which are applied to reduce the spatial dimensions of the data and to maintain the important features. The fully connected layers are responsible for classifying the data based on the features extracted by the convolutional and pooling layers.

In summary, CNNs are a powerful tool for analyzing grid-like data and are particularly useful for image and video processing tasks but they can be used for text classification tasks by applying 1D convolutional layers to the embedding of words in a sentence or a document and using max-pooling and fully connected layers to classify the data based on the features extracted by the convolutional and pooling layers. It is important to note that this type of architecture, while it can work well, is not the most common way to perform text classification and other architectures such as LSTM, GRU, or Transformer architectures are commonly used instead.

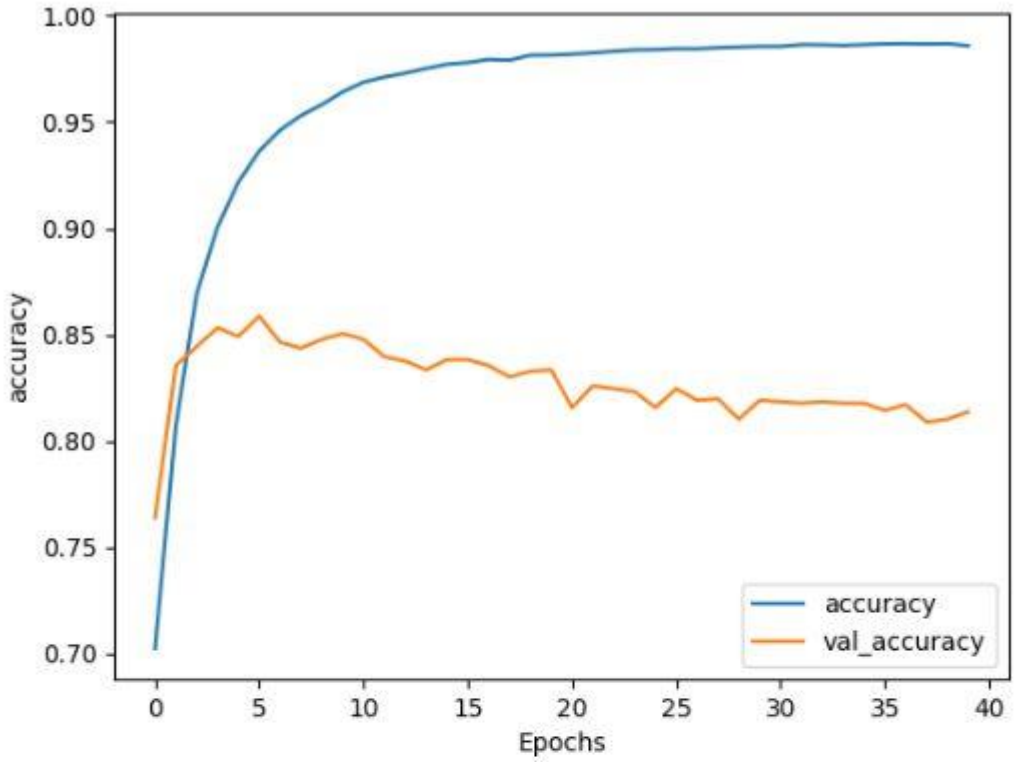
Implementation: Here are the general steps for implementing a CNN for text classification:

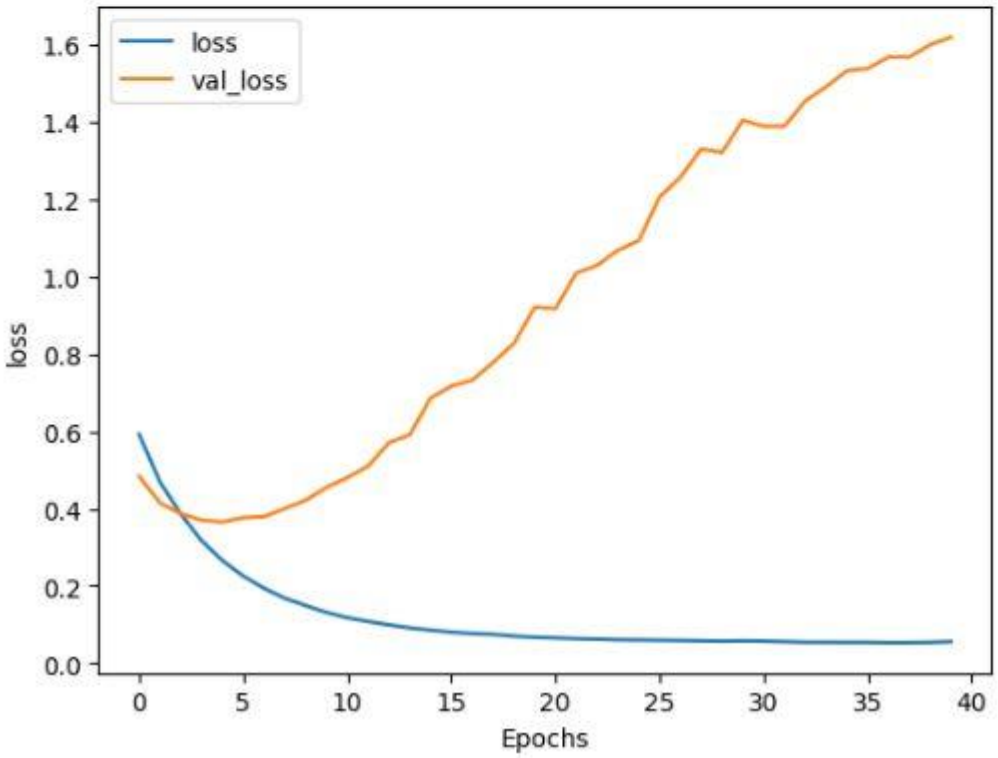
1. **Data preprocessing:** Clean and preprocess the text data, such as removing stop words, stemming, or lemmatizing the words, and tokenizing the text.
2. **Word embedding:** Train a word embedding model, such as word2vec or GloVe, on a large corpus of text, and use the trained model to obtain the embedding of the words in the text data.
3. **Define the CNN model:** Initialize the CNN model and define the architecture, including the number of convolutional layers, the number of filters, the filter size, the pooling layers, and the fully connected layers.

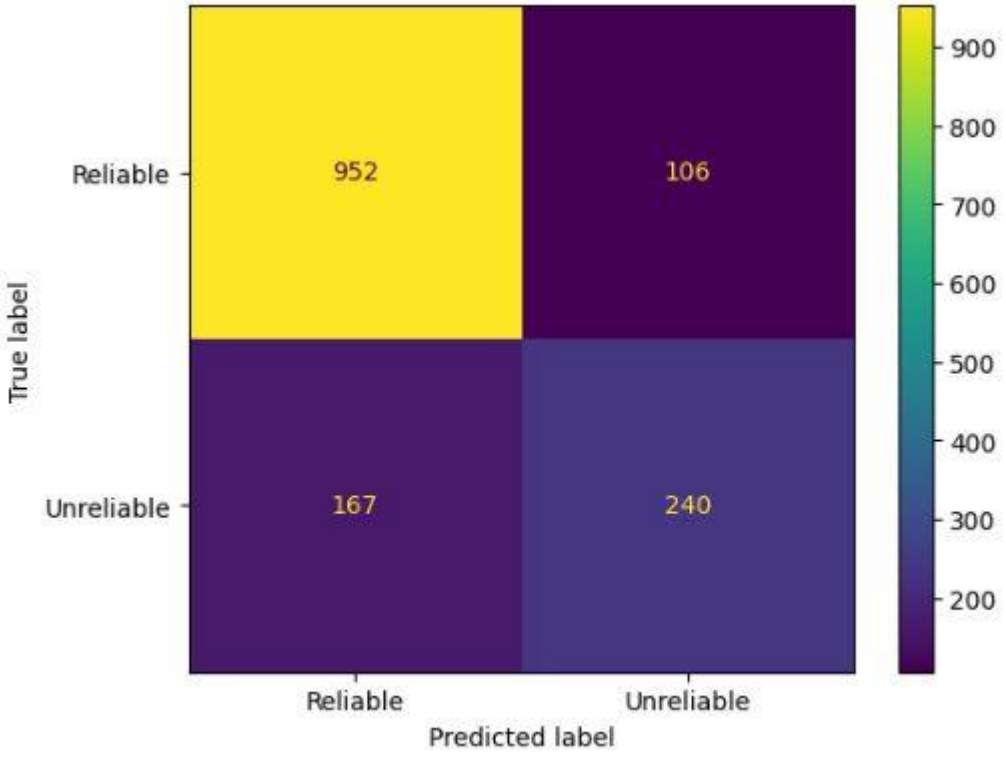
4. **Train the model:** Train the model on the text data and the corresponding labels, using a loss function such as cross-entropy loss and an optimizer such as Adam.
5. Evaluate the model: Evaluate the model on a validation or test dataset and measure its performance using metrics such as accuracy or F1-score.

CNN Model Summary:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 331, 16)           224000

 conv1d (Conv1D)             (None, 327, 128)          10368

 global_average_pooling1d (G  (None, 128)              0
 lobalAveragePooling1D)

 dense_4 (Dense)             (None, 6)                 774

 dense_5 (Dense)             (None, 1)                 7

=================================================================
Total params: 235,149
Trainable params: 235,149
Non-trainable params: 0
_________________________________________________________________
```


1.00
0.95
0.90
0.85
0.80
0.75
0.70
accuracy
0
5
10
15
20
25
30
35
40
Epochs
accuracy
val_accuracy


loss
val_loss
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
loss
0
5
10
15
20
25
30
35
40
Epochs



	precision	recall	f1-score	support
Reliable	0.90	0.85	0.87	1119
UnReliable	0.59	0.69	0.64	346
accuracy			0.81	1465
macro avg	0.74	0.77	0.76	1465
weighted avg	0.83	0.81	0.82	1465

CNN + LSTM

A CNN-LSTM (Convolutional Neural Network-Long Short-Term Memory) network is a type of deep learning neural network that combines the strengths of CNNs and LSTMs. CNNs are particularly good at analyzing grid-like data, such as images and videos, and extracting features from the data, while LSTMs are particularly good at preserving long-term dependencies in sequential data, such as time series and natural language.

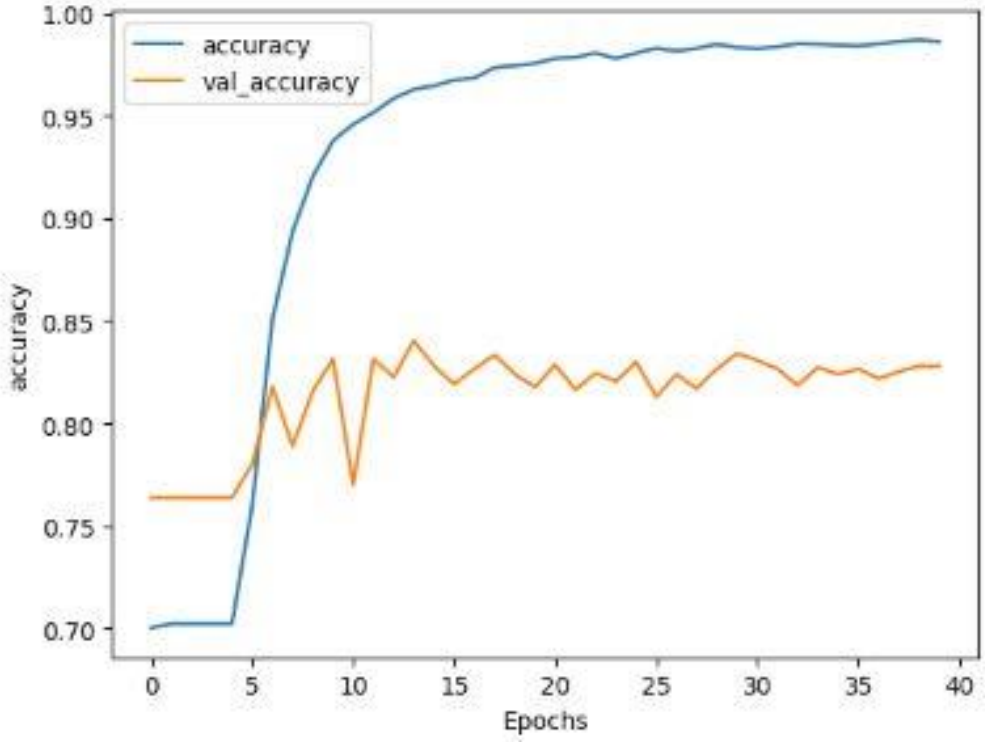
The idea behind a CNN-LSTM network is to use a CNN to extract features from the input data, and then pass the extracted features to an LSTM to preserve the long-term dependencies in the data. The CNN and LSTM are typically connected in a sequential manner, with the output of the CNN being passed as the input to the LSTM. CNN-LSTM networks are particularly useful for tasks that involve both spatial and temporal information, such as video classification and action recognition, and speech recognition.

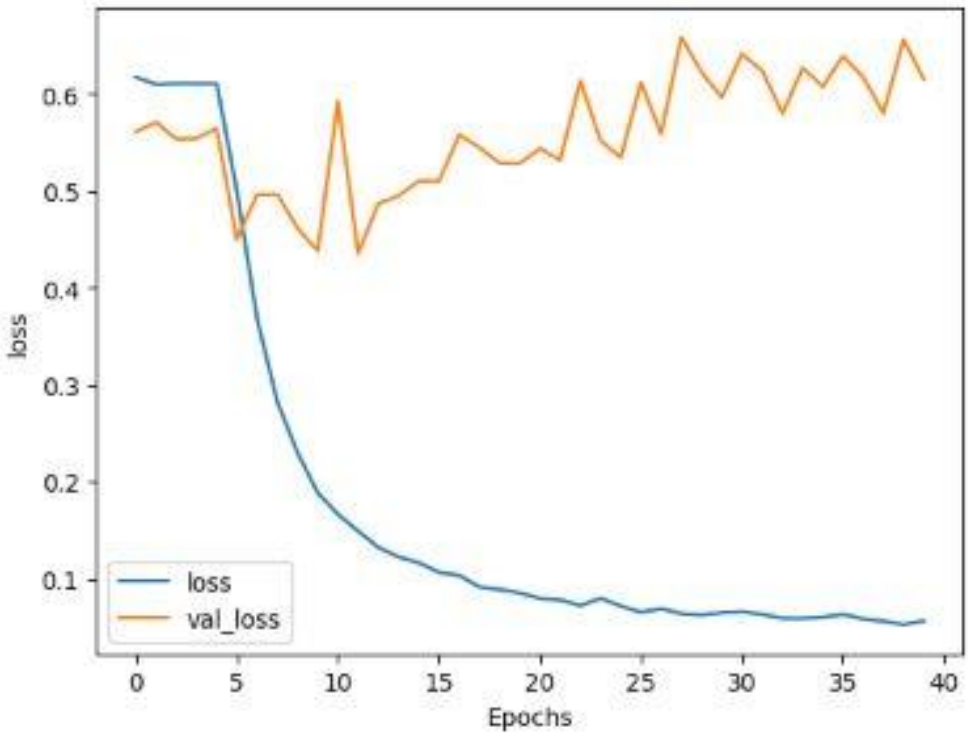
Implementation: Here are the general steps for implementing a CNN-LSTM for text classification:

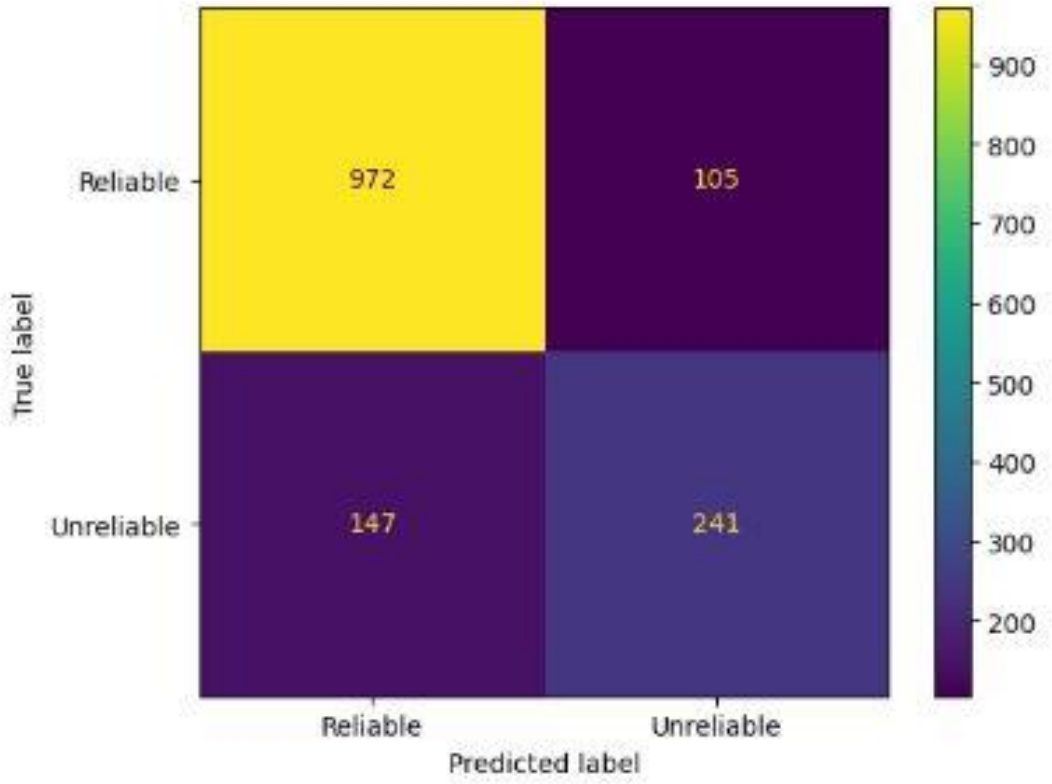
1. **Data preprocessing:** Clean and preprocess the text data, such as removing stop words, stemming, or lemmatizing the words, and tokenizing the text.
2. **Word embedding:** Train a word embedding model, such as word2vec or GloVe, on a large corpus of text, and use the trained model to obtain the embedding of the words in the text data.
3. **Define the CNN-LSTM model:** Initialize the CNN-LSTM model and define the architecture, including the number of convolutional layers, the number of filters, the filter size, the pooling layers, the LSTM layers, and the fully connected layers.
4. **Train the model:** Train the model on the text data and the corresponding labels, using a loss function such as cross-entropy loss and an optimizer such as Adam.
5. **Evaluate the model:** Evaluate the model on a validation or test dataset and measure its performance using metrics such as accuracy or F1-score.

CNN Model Summary:

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 331, 16)           224000

bidirectional_2 (Bidirectio  (None, 331, 12)           1104
nal)

conv1d_1 (Conv1D)            (None, 330, 24)           600

dropout (Dropout)            (None, 330, 24)           0

conv1d_2 (Conv1D)            (None, 328, 24)           1752

dropout_1 (Dropout)          (None, 328, 24)           0

conv1d_3 (Conv1D)            (None, 324, 24)           2904

dropout_2 (Dropout)          (None, 324, 24)           0

conv1d_4 (Conv1D)            (None, 319, 24)           3480

dropout_3 (Dropout)          (None, 319, 24)           0

conv1d_5 (Conv1D)            (None, 312, 12)           2316

dropout_4 (Dropout)          (None, 312, 12)           0

conv1d_6 (Conv1D)            (None, 309, 12)           588

dropout_5 (Dropout)          (None, 309, 12)           0

conv1d_7 (Conv1D)            (None, 306, 12)           588

dropout_6 (Dropout)          (None, 306, 12)           0

conv1d_8 (Conv1D)            (None, 304, 12)           444

dropout_7 (Dropout)          (None, 304, 12)           0

conv1d_9 (Conv1D)            (None, 302, 12)           444

dropout_8 (Dropout)          (None, 302, 12)           0

conv1d_10 (Conv1D)           (None, 301, 12)           300

dropout_9 (Dropout)          (None, 301, 12)           0

conv1d_11 (Conv1D)           (None, 300, 12)           300

dropout_10 (Dropout)         (None, 300, 12)           0

bidirectional_3 (Bidirectio  (None, 300, 24)           2400
nal)

max_pooling1d (MaxPooling1D  (None, 75, 24)            0
)

dropout_11 (Dropout)         (None, 75, 24)            0

flatten (Flatten)            (None, 1800)              0

dense_6 (Dense)              (None, 26)                46826

dense_7 (Dense)              (None, 8)                 216

dense_8 (Dense)              (None, 1)                 9

=================================================================
Total params: 288,271
Trainable params: 288,271
Non-trainable params: 0
_________________________________________________________________
```


accuracy
val_accuracy
accuracy
Epochs
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
5
10
15
20
25
30
35
40


loss
val_loss
loss
Epochs
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30
35
40



	precision	recall	f1-score	support
Reliable	0.90	0.87	0.89	1119
UnReliable	0.62	0.70	0.66	346
accuracy			0.83	1465
macro avg	0.76	0.78	0.77	1465
weighted avg	0.84	0.83	0.83	1465

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained transformer-based neural network model for natural language processing (NLP) tasks such as text classification, question answering, and named entity recognition. BERT is trained on a large corpus of text using a technique called unsupervised pre-training, which means it is trained on a massive amount of text data without any specific task in mind. This allows it to learn the general patterns and features of the language, which can then be fine-tuned for specific NLP tasks.

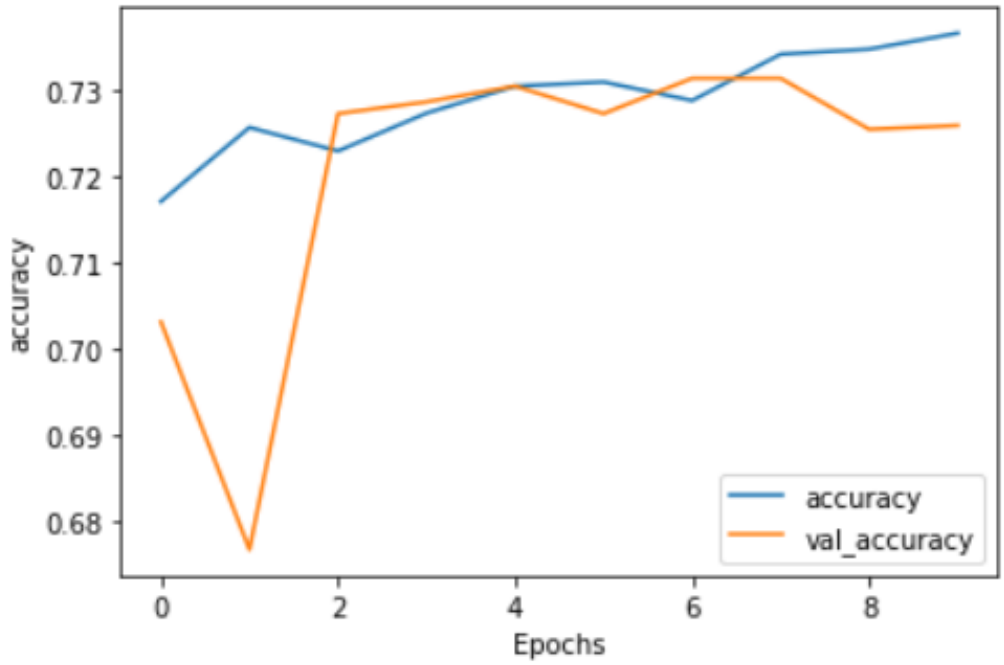
BERT is unique because it is bidirectional, meaning it considers the context of the words both to the left and to the right of the current word, rather than only the context to the left as in traditional models. This allows BERT to better understand the meaning of the words in the sentence and improve the performance on NLP tasks. BERT has achieved state-of-the-art results on a wide range of NLP tasks and has become a popular choice for natural language understanding and text classification tasks. It is available in several pre-trained versions, such as BERT-base and BERT-large, which can be fine-tuned to specific tasks and datasets.

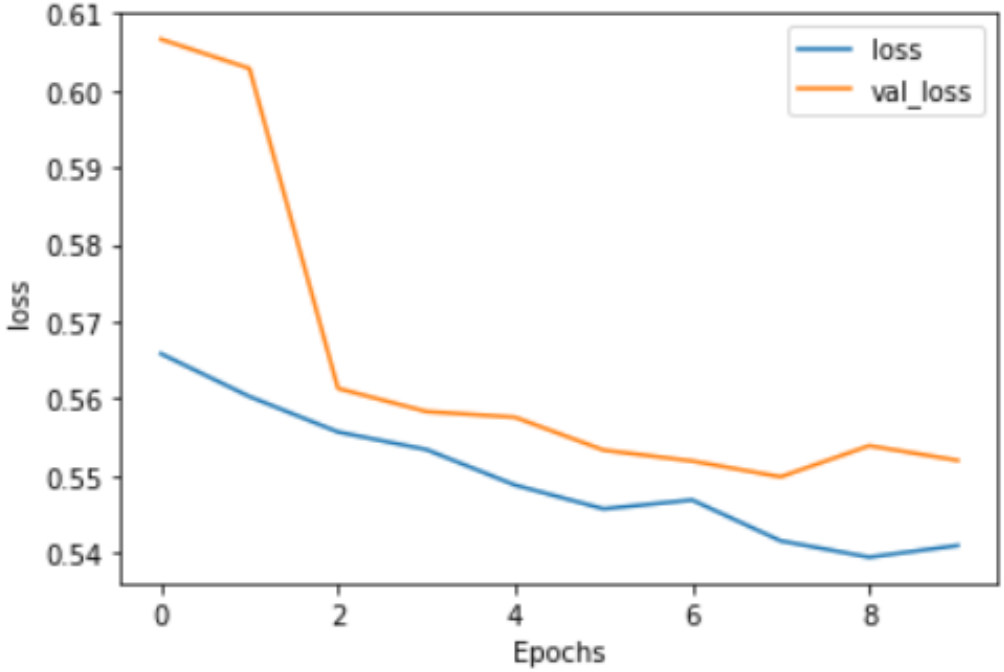
Implementation: Here are the general steps for implementing BERT for text classification:

1. **Data preprocessing:** Clean and preprocess the text data, such as removing stop words, stemming or lemmatizing the words, and tokenizing the text.
2. **Download a pre-trained BERT model:** Download a pre-trained BERT model from the HuggingFace library or TensorFlow Hub, such as BERT-base or BERT-large.
3. **Tokenize the text data:** Use the BERT tokenizer to tokenize the text data, which will convert the text into a format that the BERT model can understand.
4. **Create a data loader:** Create a data loader that loads the tokenized text data and labels and creates batches for training and evaluation.
5. **Define the model architecture:** Define the architecture of the model, including the pre-trained BERT model and the additional layers for the classification task.
6. **Fine-tune the pre-trained model:** Fine-tune the pre-trained BERT model on the text classification task using a loss function such as cross-entropy loss and an optimizer such as Adam.
7. **Evaluate the model:** Evaluate the model on a validation or test dataset and measure its performance using metrics such as accuracy or F1-score.

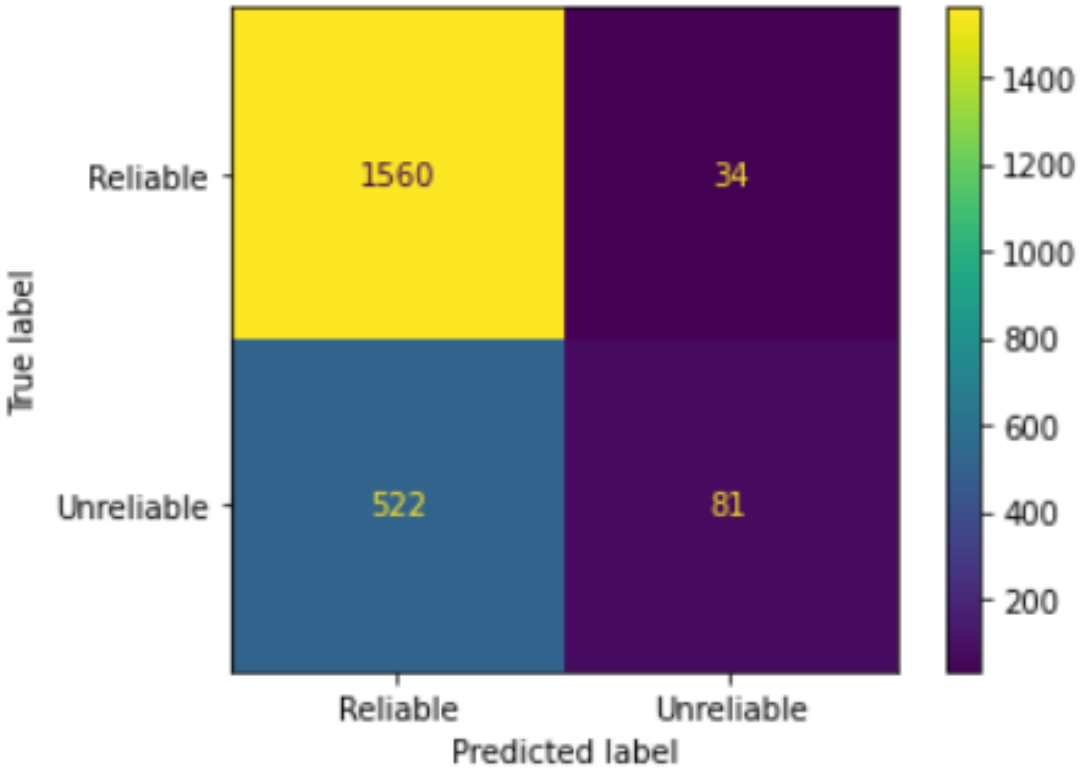
BERT Model Summary:

```
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 text (InputLayer)           [(None,)]                 0           []

 keras_layer (KerasLayer)    {'input_type_ids':        0           ['text[0][0]']
                             (None, 128),
                              'input_word_ids':
                             (None, 128),
                              'input_mask': (Non
                             e, 128)}

 keras_layer_1 (KerasLayer)  {'sequence_output':       109482241   ['keras_layer[0][0]',
                              (None, 128, 768),                     'keras_layer[0][1]',
                              'encoder_outputs':                    'keras_layer[0][2]']
                              [(None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768)],
                              'pooled_output': (
                             None, 768),
                              'default': (None,
                             768)}

 output (Dense)              (None, 1)                 769         ['keras_layer_1[0][13]']

==================================================================================================
Total params: 109,483,010
Trainable params: 769
Non-trainable params: 109,482,241
__________________________________________________________________________________________________
```


0.73
0.72
0.71
0.70
0.69
0.68
accuracy
0
2
4
6
8
Epochs
accuracy
val_accuracy


0.61
0.60
0.59
0.58
0.57
0.56
0.55
0.54
loss
0
2
4
6
8
Epochs
loss
val_loss

	precision	recall	f1-score	support
0	0.75	0.98	0.85	1594
1	0.70	0.13	0.23	603
accuracy			0.75	2197
macro avg	0.73	0.56	0.54	2197
weighted avg	0.74	0.75	0.68	2197



ALBERT

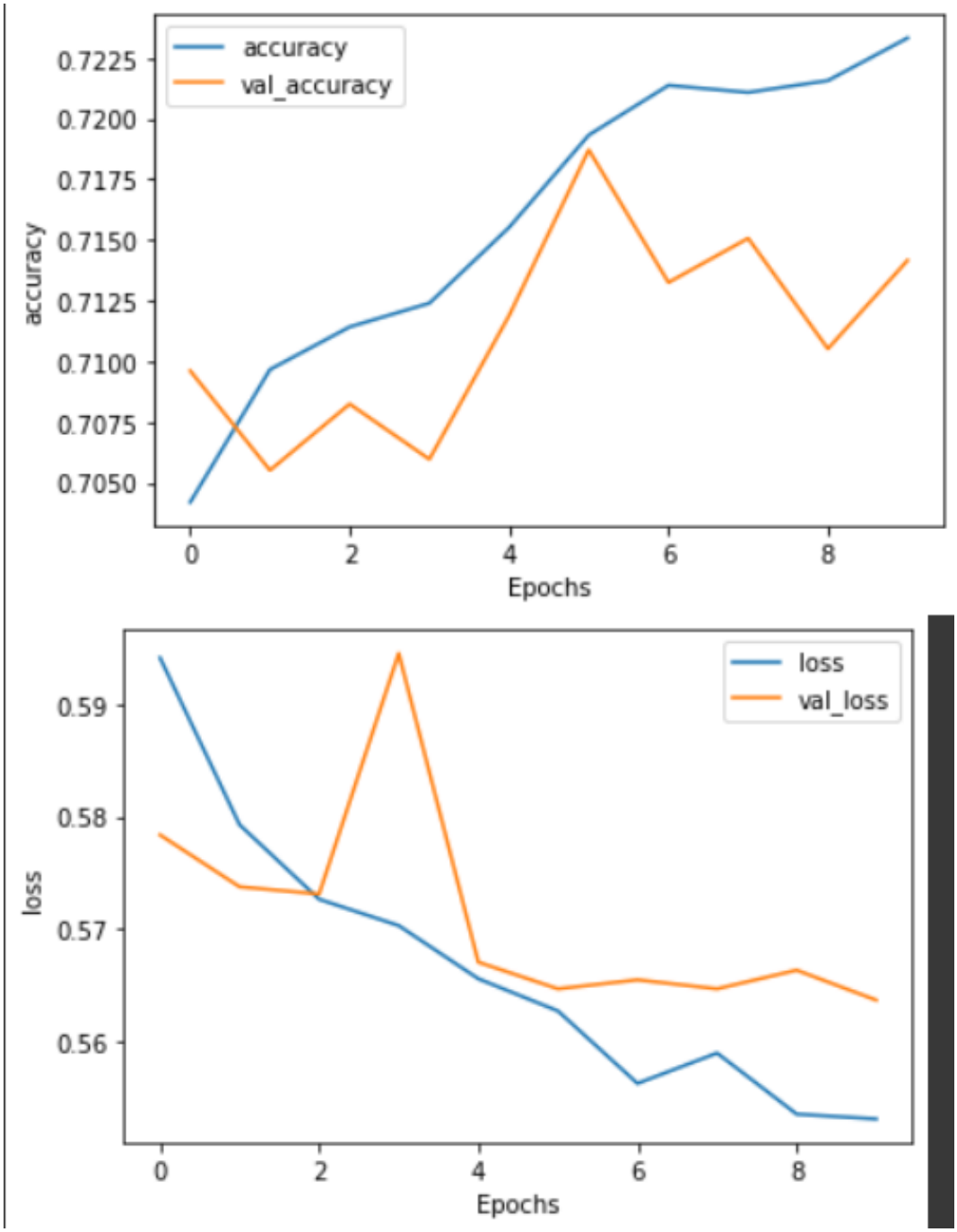
ALBERT (A Lite BERT) is a light version of BERT (Bidirectional Encoder Representations from Transformers), a pre-trained transformer-based neural network model for natural language processing (NLP) tasks such as text classification, question answering, and named entity recognition. ALBERT is similar to BERT but it is designed to be more lightweight and efficient, while still maintaining similar or even better performance on NLP tasks. It was achieved by applying two main techniques: factorization and cross-layer parameter sharing.

Factorization of the transformer-based architectures was used to reduce the number of parameters in the model, which makes the training and inference faster and more memory-efficient. Cross-layer parameter sharing was used to share the parameters across the layers, which also reduces the number of parameters. ALBERT is available in different versions such as ALBERT-base and ALBERT-xxlarge, which can be fine-tuned to specific tasks and datasets. It has been proven to have better performance on many NLP tasks compared to BERT and it is widely used in natural language understanding and text classification tasks.

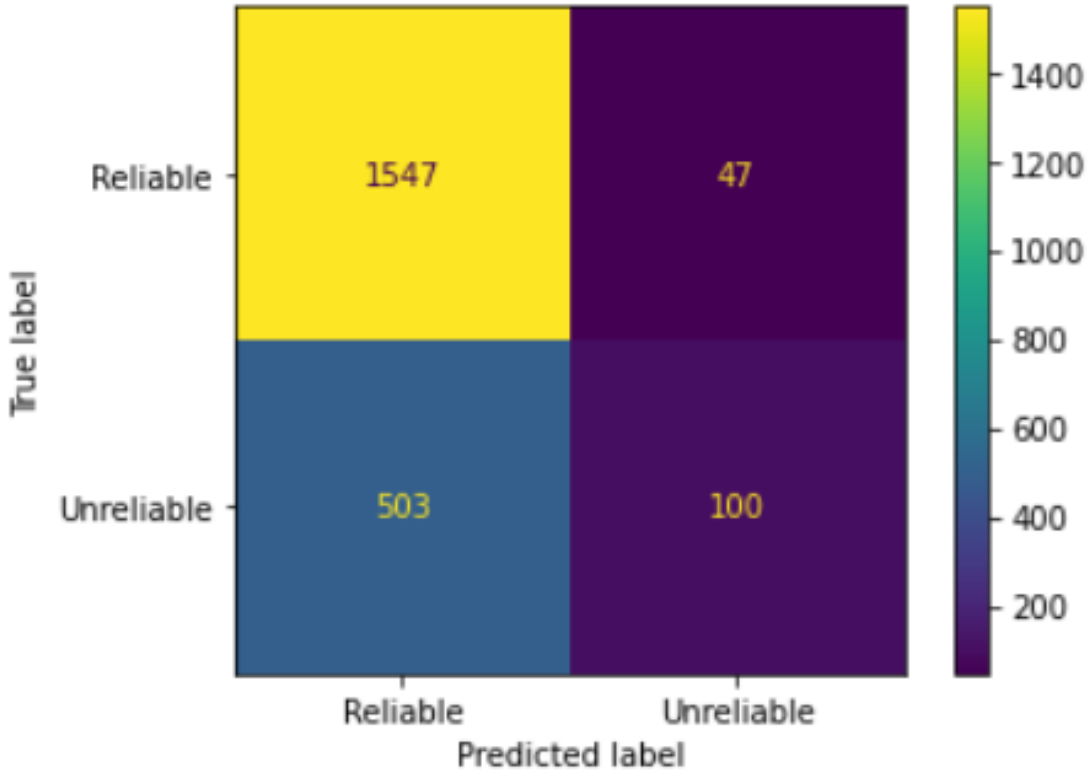
In summary, ALBERT is a light version of BERT, a pre-trained transformer-based neural network model for natural language processing tasks that is more lightweight and efficient while still maintaining similar or even better performance on NLP tasks. It was achieved by applying factorization and cross-layer parameter-sharing techniques. It is widely used in natural language understanding and text classification tasks.

ALBERT Model Summary:

```
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 text (InputLayer)           [(None,)]                 0           []

 keras_layer (KerasLayer)    {'input_mask': (Non       0           ['text[0][0]']
                             e, 128),
                              'input_word_ids':
                             (None, 128),
                              'input_type_ids':
                             (None, 128)}

 keras_layer_1 (KerasLayer)  {'pooled_output': (       11683584    ['keras_layer[0][0]',
                             None, 768),                            'keras_layer[0][1]',
                              'default': (None,                     'keras_layer[0][2]']
                             768),
                              'sequence_output':
                              (None, 128, 768),
                              'encoder_outputs':
                              [(None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768),
                              (None, 128, 768)]}

 output (Dense)              (None, 1)                 769         ['keras_layer_1[0][13]']

==================================================================================================
Total params: 11,684,353
Trainable params: 769
Non-trainable params: 11,683,584
__________________________________________________________________________________________________
```


accuracy
val_accuracy
0.7225
0.7200
0.7175
0.7150
0.7125
0.7100
0.7075
0.7050
accuracy
0
2
4
6
8
Epochs
loss
val_loss
0.59
0.58
0.57
0.56
loss
0
2
4
6
8
Epochs

	precision	recall	f1-score	support
0	0.75	0.97	0.85	1594
1	0.68	0.17	0.27	603
accuracy			0.75	2197
macro avg	0.72	0.57	0.56	2197
weighted avg	0.73	0.75	0.69	2197



Comparison Table

Models	Accuracy (Training Dataset)	Accuracy (Testing Dataset)
LSTM	99.75%	81.30%
GRU	98.54%	78.98%
CNN	98.59%	81.36%
CNN + LSTM	98.59%	82.80%
BERT	73.67%	75%
ALBERT	72.33%	75%