GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY, TIRUTTANI - 631209
Approved by AICTE, New Delhi Affiliated to Anna University, Chennai

NAAN MUDHALVAN PROJECT(IBM)

IBM AI 101 ARTIFICIAL INTELLIGENCE-GROUP 1

DONE BY
GOKUL NATH S
(Email: 12345.gokulnath.s@gmail.com)
(NM ID: au110321106013)
ECE 3Rd Year
From the Department of
ELECTRONICS AND COMMUNICATION ENGINEERING

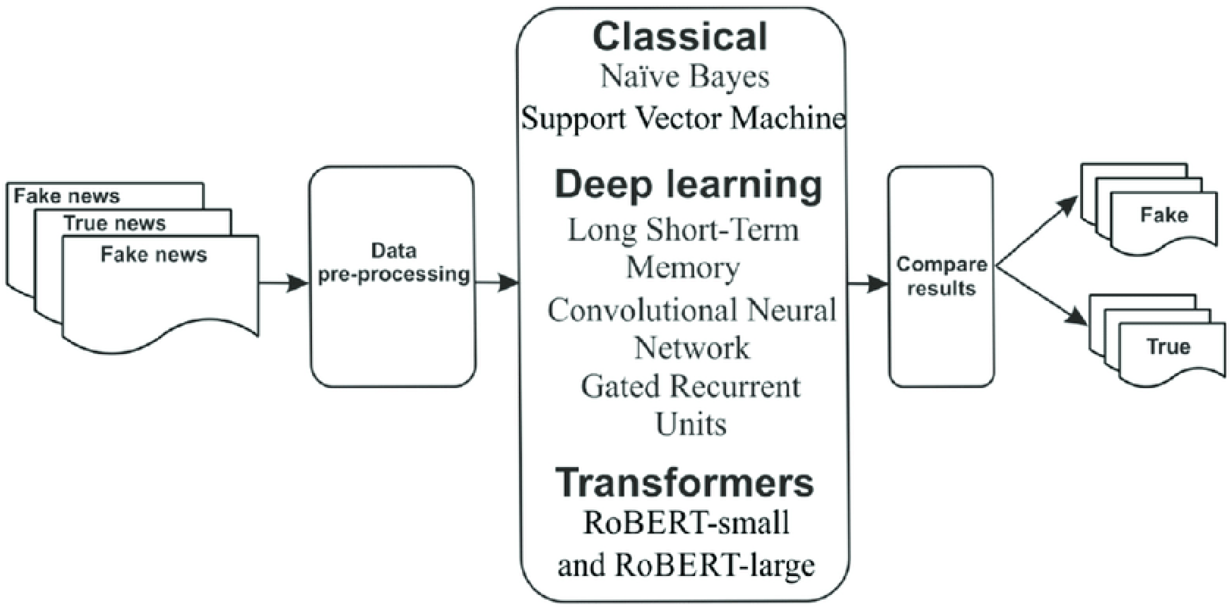
NAAN MUDHALVAN PROJECT(IBM)

IBM AI 101 **ARTIFICIAL INTELLIGENCE-GROUP 1**

PROJECT:

TEAM-6 **FAKE NEWS DETECTION USING NLP**

PHASE III : DEVELOPMENT PART-II



<u>DATASET</u>

title	text	subject	date
Donald Trump Sends Out Embarra	Donald Trump just couldn t wish all Ameri	News	December 31, 2017
Drunk Bragging Trump Staffer Star	House Intelligence Committee Chairman	News	December 31, 2017
Sheriff David Clarke Becomes An I	On Friday, it was revealed that former Mi	News	December 30, 2017
Trump Is So Obsessed He Even Ha	On Christmas day, Donald Trump announ	News	December 29, 2017
Pope Francis Just Called Out Dona	Pope Francis used his annual Christmas D	News	December 25, 2017
Racist Alabama Cops Brutalize Bla	The number of cases of cops brutalizing a	News	December 25, 2017
Fresh Off The Golf Course, Trump	Donald Trump spent a good portion of his	News	December 23, 2017
Trump Said Some INSANELY Racis	In the wake of yet another court decision	News	December 23, 2017
Former CIA Director Slams Trump	Many people have raised the alarm regar	News	December 22, 2017
WATCH: Brand-New Pro-Trump A	Just when you might have thought we d g	News	December 21, 2017
Papa Johnâ€™s Founder Retires, F	A centerpiece of Donald Trump s campaig	News	December 21, 2017
WATCH: Paul Ryan Just Told Us He	Republicans are working overtime trying t	News	December 21, 2017
Bad News For Trump â€” Mitch M	Republicans have had seven years to com	News	December 21, 2017
WATCH: Lindsey Graham Trashes I	The media has been talking all day about	News	December 20, 2017
Heiress To Disney Empire Knows C	Abigail Disney is an heiress with brass ova	News	December 20, 2017
Tone Deaf Trump: Congrats Rep. S	Donald Trump just signed the GOP tax sca	News	December 20, 2017
The Internet Brutally Mocks Disne	A new animatronic figure in the Hall of Pr	News	December 19, 2017
Mueller Spokesman Just F-cked U	Trump supporters and the so-called presi	News	December 17, 2017
SNL Hilariously Mocks Accused Ch	Right now, the whole world is looking at t	News	December 17, 2017
Republican Senator Gets Dragged	Senate Majority Whip John Cornyn (R-TX)	News	December 16, 2017
In A Heartless Rebuke To Victims,	It almost seems like Donald Trump is troll	News	December 16, 2017
KY GOP State Rep. Commits Suicic	In this #METOO moment, many powerful	News	December 13, 2017
Meghan McCain Tweets The Most	As a Democrat won a Senate seat in deep	News	December 12, 2017

The fake news dataset is one of the classic text analytics datasets available on Kaggle. It consists of genuine and fake articles' titles and text from different authors. In this article, I have walked through the entire text classification process using traditional machine learning approaches as well as deep learning.

Getting Started

I started with downloading the dataset from Kaggle on Google Colab.

CODE

```
# Upload Kaggle json


!pip install -q kaggle

!pip install -q kaggle-cli

!mkdir -p ~/.kaggle

!cp "/content/drive/My Drive/Kaggle/kaggle.json" ~/.kaggle/ # Mount GDrive

!cat ~/.kaggle/kaggle.json

!chmod 600 ~/.kaggle/kaggle.json

!kaggle competitions download -c fake-news -p dataset

!unzip /content/dataset/train.csv.zip

!unzip /content/dataset/test.csv.zip
```

Next, I read the DataFrame and checked the null values in it. There are 7 null values in the text articles, 122 in title and 503 in author out of a total of 20800 rows, I decided to drop the rowsFor the test data, I filled them up with a blank.

```
id          0
title     122
author    503
text        7
dtype: int64
```

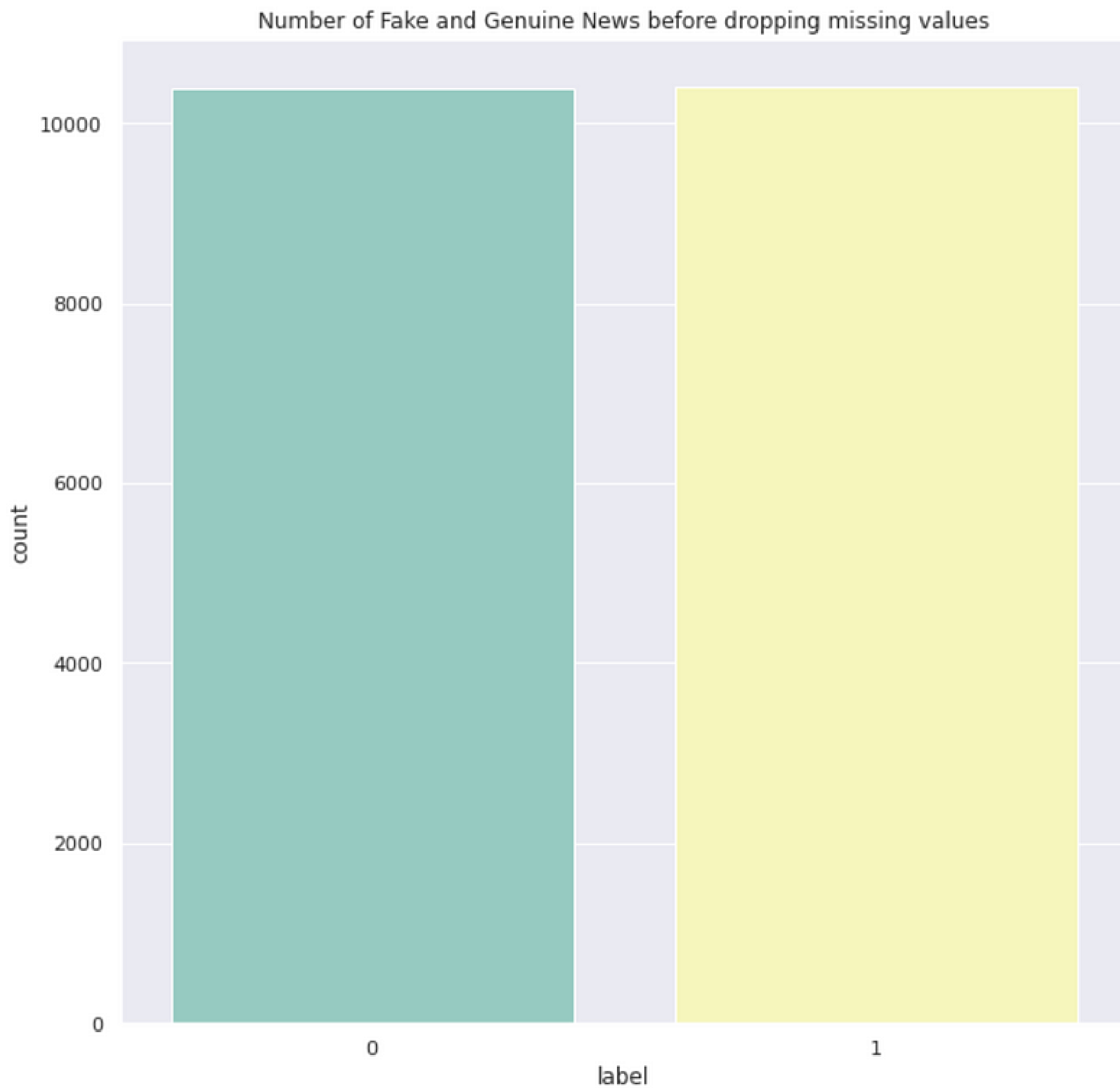
```
id          0
title     558
author   1957
text       39
label       0
dtype: int64
```

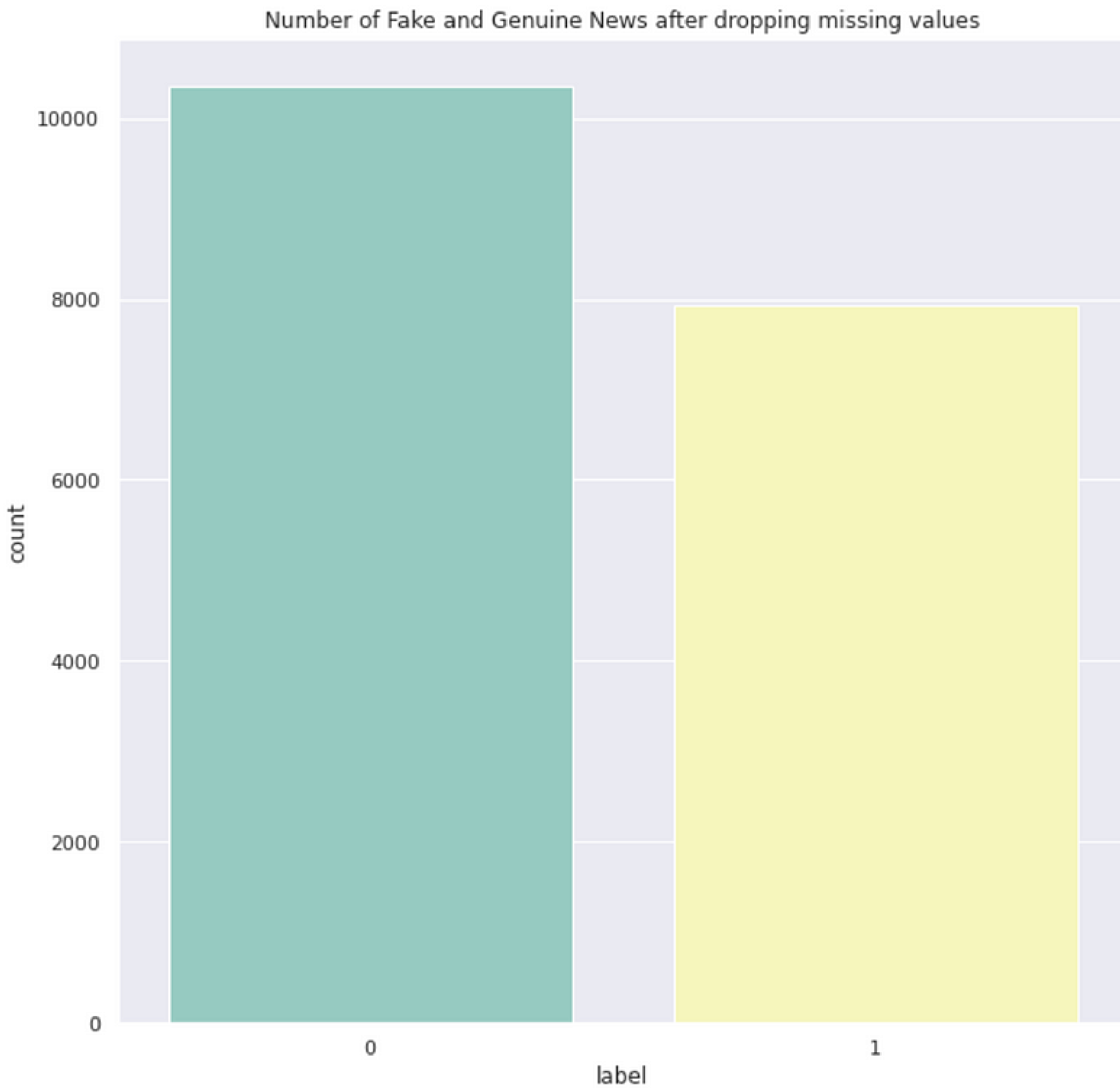
Number of Null Values in Train Data and Test Data, respectively

```
train_df = pd.read_csv('/content/train.csv', header=0)
test_df = pd.read_csv('/content/test.csv', header=0)
print(train_df.isna().sum())
print(test_df.isna().sum())
train_df.dropna(axis=0, how='any',inplace=True)
test_df = test_df.fillna(' ')
```

Additionally, I also check the distribution of 'Fake' and 'Genuine' news in the dataset. Usually, I set the rcParams for all plots on the notebook while importing matplotlib.

```
import matplotlib.pyplot as plt
from matplotlib import rcParams
plt.rcParams['figure.figsize'] = [10, 10]
import seaborn as sns
sns.set_theme(style="darkgrid")

sns.countplot(x='label', data=train_df, palette='Set3')
plt.show()
```


Number of Fake and Genuine News before dropping missing values
10000
8000
6000
4000
2000
0
count
0
1
label

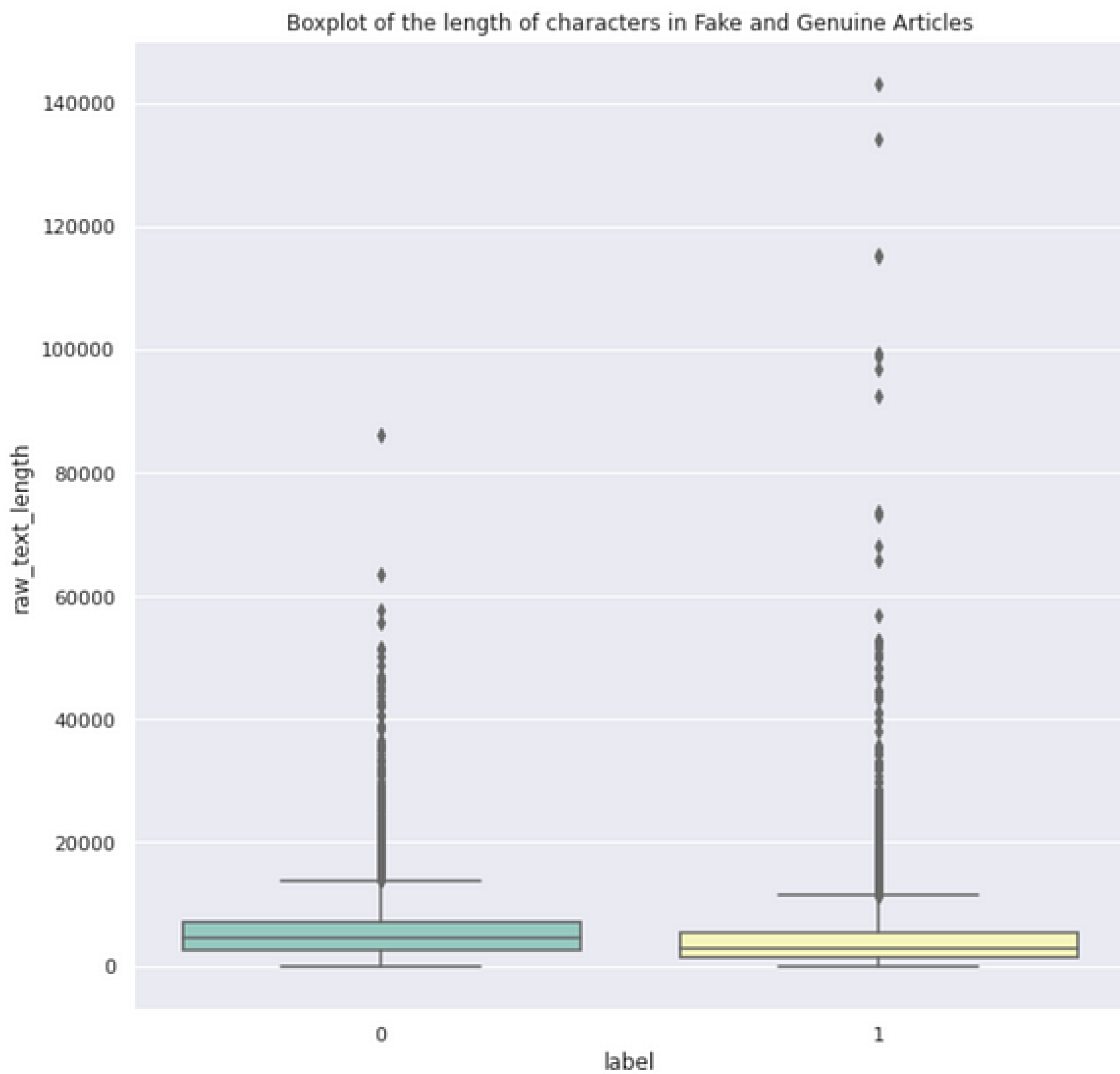


0 is Genuine News while 1 is Fake News

The ratio is disturbed from being 1:1 to 4:5 for genuine to fake news.

Next, I decided to look at the article length like below —

```
train_df['raw_text_length'] = train_df['text'].apply(lambda x: len(x))

sns.boxplot(y='raw_text_length', x='label', data=train_df,
palette="Set3")

plt.show()
```



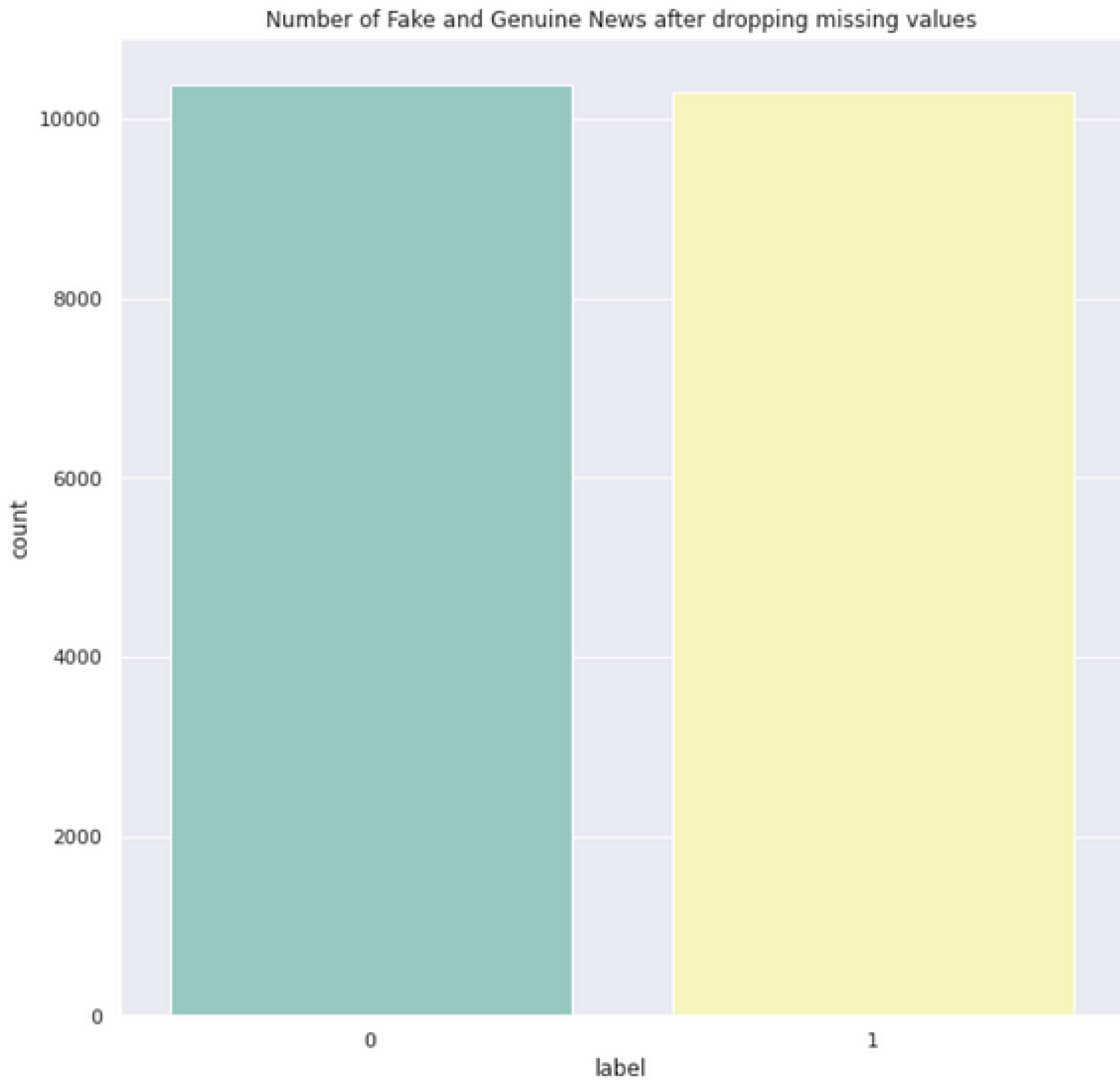
It is seen that the median length is lower for fake articles but it also has loads of outliers. Both have zero length.

It is seen that they start from 0 which is concerning. It actually starts from 1 when I used .describe() to see the numbers. So I took a look at these texts and found that they are blank. The obvious answer to this is strip and drop length zero. I checked the total number of zero-length texts is 74.

```
train_df['text'] = train_df['text'].str.strip()

# Recalculate the length

train_df['raw_text_length'] = train_df['text'].apply(lambda x: len(x))

print(len(train_df[train_df['raw_text_length']==0]))
```

I decided to start over again. So, I would fill all nans with a blank and strip them next, then, remove the zero-length texts and that should be good to start the preprocessing. Following is the new code that handles missing values essentially. The final shape of the data is (20684, 6), that is, it contains 20684 rows, only 116 less than 20800.

```
train_df = pd.read_csv('/content/train.csv', header=0)

train_df = train_df.fillna(' ')

train_df['text'] = train_df['text'].str.strip()

train_df['raw_text_length'] = train_df['text'].apply(lambda x: len(x))

print(len(train_df[train_df['raw_text_length']==0]))

print(train_df.isna().sum())

train_df = train_df[train_df['raw_text_length'] > 0]

print(train_df.shape)

print(train_df.isna().sum())


# Visualize the target's distribution

sns.countplot(x='label', data=train_df, palette='Set3')

plt.title("Number of Fake and Genuine News after dropping missing
values")

plt.show()
```



It so appeared after that there are more texts that have single-digit lengths or as low as 10. They seemed more like comments than proper texts. I will keep them for the time being as it is and move on to the next step.

Text Preprocessing

So before I began with text preprocessing, I actually looked at the overlapping number of authors that have fake and genuine articles. In other words, would having the author's information be helpful in any way? I found out that there are 3838 authors, out of which 2225 are genuine and 1618 are fake news' authors. 5 authors among them are both genuine and fake news' authors.

```
gen_news_authors =
set(list(train_df[train_df['label']==0]['author'].unique()))

fake_news_authors =
set(list(train_df[train_df['label']==1]['author'].unique()))

overlapped_authors = gen_news_authors.intersection(fake_news_authors)

print("Number of distinct authors with genuine articles: {}",
len(gen_news_authors))

print("Number of distinct authors with fake articles: {}",
len(fake_news_authors))

print("Number of distinct authors with both genuine and fake: {}",
len(overlapped_authors))
```

To start with pre-processing I initially had chosen to directly split by blank and expand contractions. However, that has yielded errors due to some (I suppose Slavic) other language texts. So, in the first step, I used regex to preserve only the Latin character, digits, and spaces. Then, expand contractions and then convert to lower-case. This is because contractions such as i've is converted to I have. Therefore, conversion to lower-case comes after expanding contractions. The full code is below:

```
def preprocess_text(x):

  cleaned_text = re.sub(r'[^a-zA-Z\d\s\']+', '', x)

  word_list = []

  for each_word in cleaned_text.split(' '):

    try:

      word_list.append(contractions.fix(each_word).lower())

    except:

      print(x)

  return " ".join(word_list)


text_cols = ['text', 'title', 'author']

for col in text_cols:
```

```
  print("Processing column: {}".format(col))
  train_df[col] = train_df[col].apply(lambda x: preprocess_text(x))
  test_df[col] = test_df[col].apply(lambda x: preprocess_text(x))
```

Once, this is done, the regular word tokenization is done followed by stopword removal.

```
for col in text_cols:
    print("Processing column: {}".format(col))
    train_df[col] = train_df[col].apply(word_tokenize)
    test_df[col] = test_df[col].apply(word_tokenize)

for col in text_cols:
    print("Processing column: {}".format(col))
    train_df[col] = train_df[col].apply(
        lambda x: [each_word for each_word in x if each_word not in
stopwords])
    test_df[col] = test_df[col].apply(
        lambda x: [each_word for each_word in x if each_word not in
stopwords])
```

Text Analysis

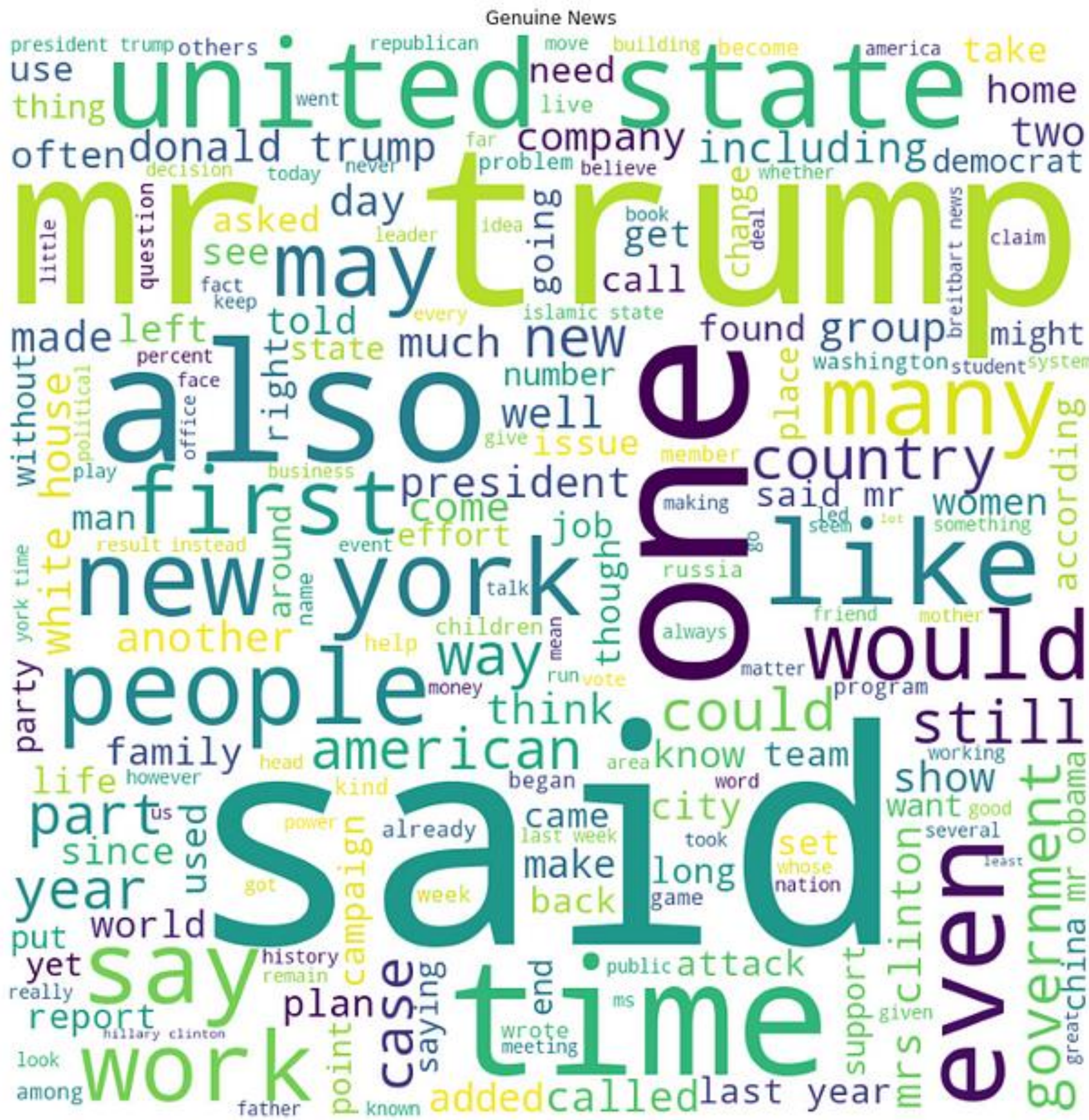
Now that the data is ready, I intend to look at frequent words using the wordcloud. In order to do that, I first joined all the tokenized texts into strings in separate columns since they will be used later while model training.

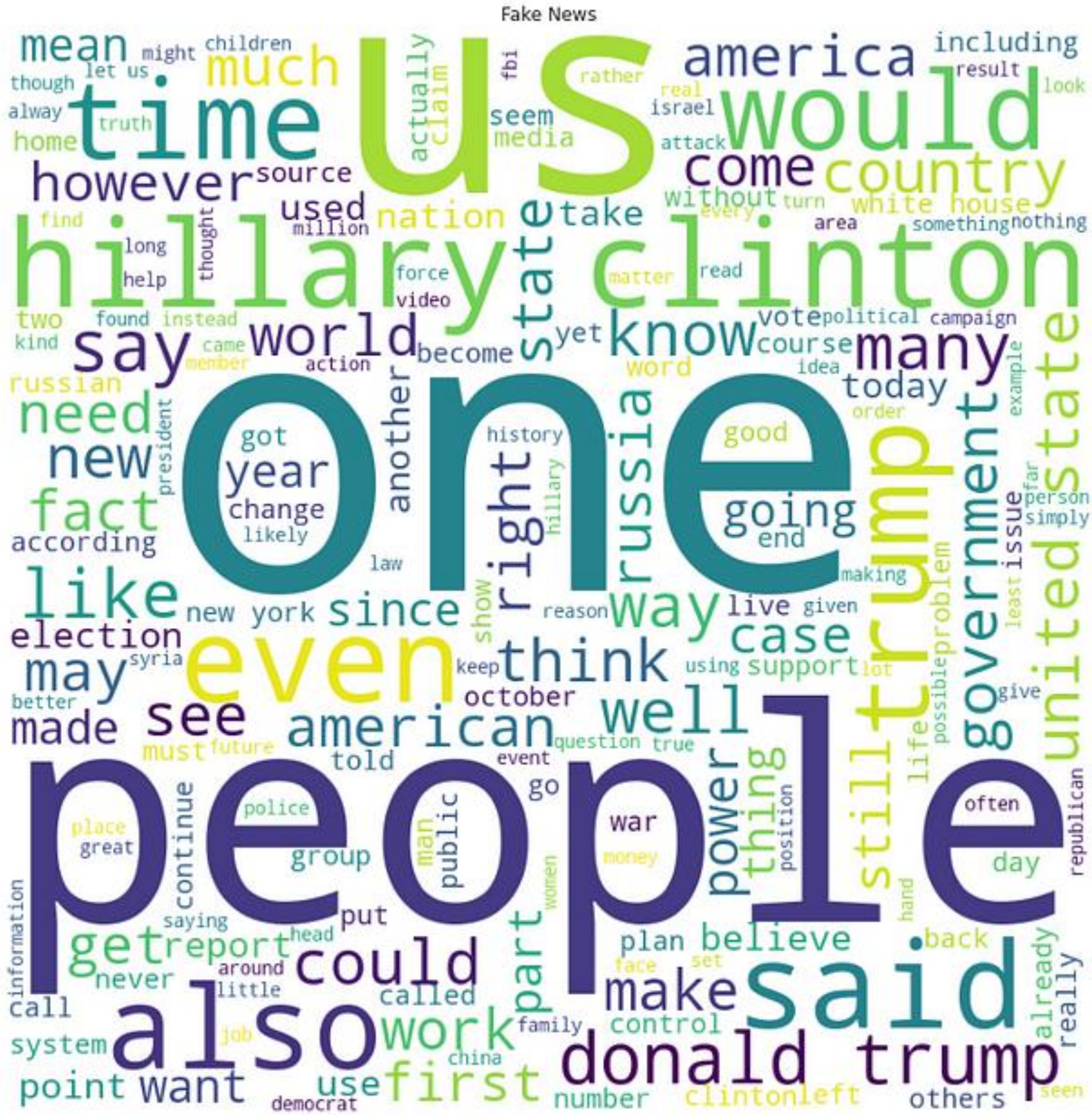
```
# since count vectorizer expects strings

train_df['text_joined'] = train_df['text'].apply(lambda x: "
".join(x))
test_df['text_joined'] = test_df['text'].apply(lambda x: " ".join(x))
```

Next, per label, create a string of all texts and created the wordcloud as below:

```
# join all texts in resective labels
all_texts_gen = "
".join(train_df[train_df['label']==0]['text_joined'])
all_texts_fake = "
".join(train_df[train_df['label']==1]['text_joined'])

# Wordcloud for Genuine News
wordcloud = WordCloud(width = 800, height = 800,
                background_color ='white',
                stopwords = stopwords,
                min_font_size = 10).generate(all_texts_gen)
plt.imshow(wordcloud)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()

# Worldcloud for Fake News
wordcloud = WordCloud(width = 800, height = 800,
                background_color ='white',
                stopwords = stopwords,
                min_font_size = 10).generate(all_texts_fake)
```

```
plt.imshow(wordcloud)
plt.axis("off")
plt.tight_layout(pad = 0)

plt.show()
```





In the fake news wordcloud, the frequency of some words is strikingly higher than the others. On the genuine news' wordcloud, there is a mix of different font sizes. On the contrary, in the fake news dataset, the smaller texts are in the background and some of the words are used much more frequently. There are fewer medium-sized words in the fake news wordcloud or, in other words, there is a disconnect in progressively diminishing frequency of appearance. The frequency is either high or low.

Stylometric Analysis

The stylometric analysis is often referred to as the analysis of the author's style. I will look into a few of the stylometric features such as the number of sentences per article, the average words per sentence in an article, the average length of words per article, and the POS tag counts.

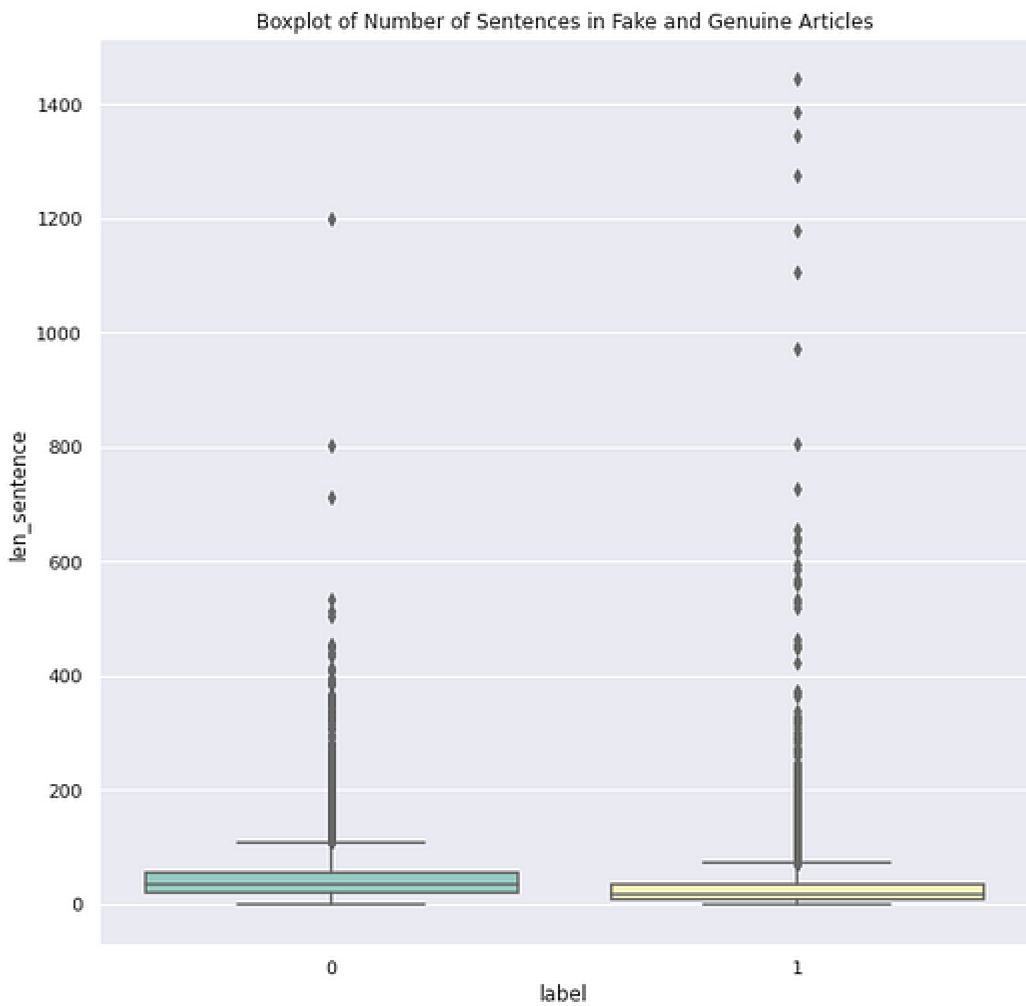
Number of Sentences per Article

To get this I needed the original dataset since I have lost the sentence information in train_df. So, I saved a copy of the actual data in orginal_train_df which I used to convert the sentences to sequences.

```
from nltk import sent_tokenize

original_train_df = train_df.copy()

original_train_df['sent_tokens'] =
original_train_df['text'].apply(sent_tokenize)
```

Next, I looked at the count of the sentences by each target category as follows:

```
original_train_df['len_sentence'] =
original_train_df['sent_tokens'].apply(len)

sns.boxplot(y='len_sentence', x='label', data=original_train_df,
palette="Set3")

plt.title("Boxplot of Number of Sentences in Fake and Genuine
Articles")

plt.show()
```

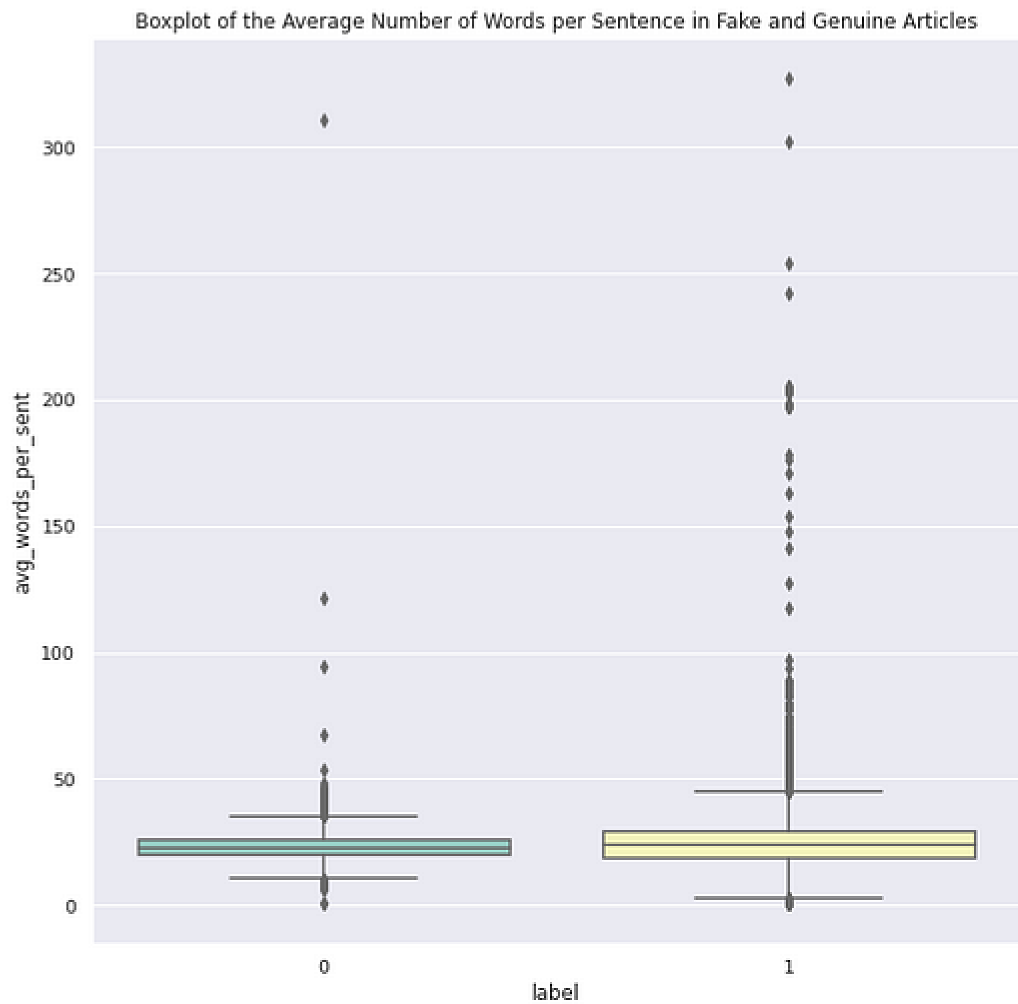


Evidently, fake articles have a lot of outliers but 75% of the fake articles have the number of sentences lower than the 50% of the genuine news articles.

Average Number of Words per Sentence in an Article

Here, I counted the total number of words per sentence in each article and returned the average. Then I plotted the counts in a boxplot to visualize them.

```
# tokenize words within the sequences

original_train_df['sent_word_tokens'] =
original_train_df['sent_tokens'].apply(lambda x:
[word_tokenize(each_sentence) for each_sentence in x])


# Clean the punctuations

def get_seq_tokens_cleaned(seq_tokens):

  no_punc_seq = [each_seq.translate(str.maketrans('', '',
string.punctuation)) for each_seq in seq_tokens]

  sent_word_tokens = [word_tokenize(each_sentence) for each_sentence
in no_punc_seq]

  return sent_word_tokens


# Count the avg number of words in each sentence

def get_average_words_in_sent(seq_word_tokens):

  return np.mean([len(seq) for seq in seq_word_tokens])


original_train_df['sent_word_tokens'] =
original_train_df['sent_tokens'].apply(lambda x:
get_seq_tokens_cleaned(x))

original_train_df['avg_words_per_sent'] =
original_train_df['sent_word_tokens'].apply(lambda x:
get_average_words_in_sent(x))


sns.boxplot(y='avg_words_per_sent', x='label', data=original_train_df,
palette="Set3")

plt.title("Boxplot of the Average Number of Words per Sentence in Fake
and Genuine Articles")

plt.show()
```

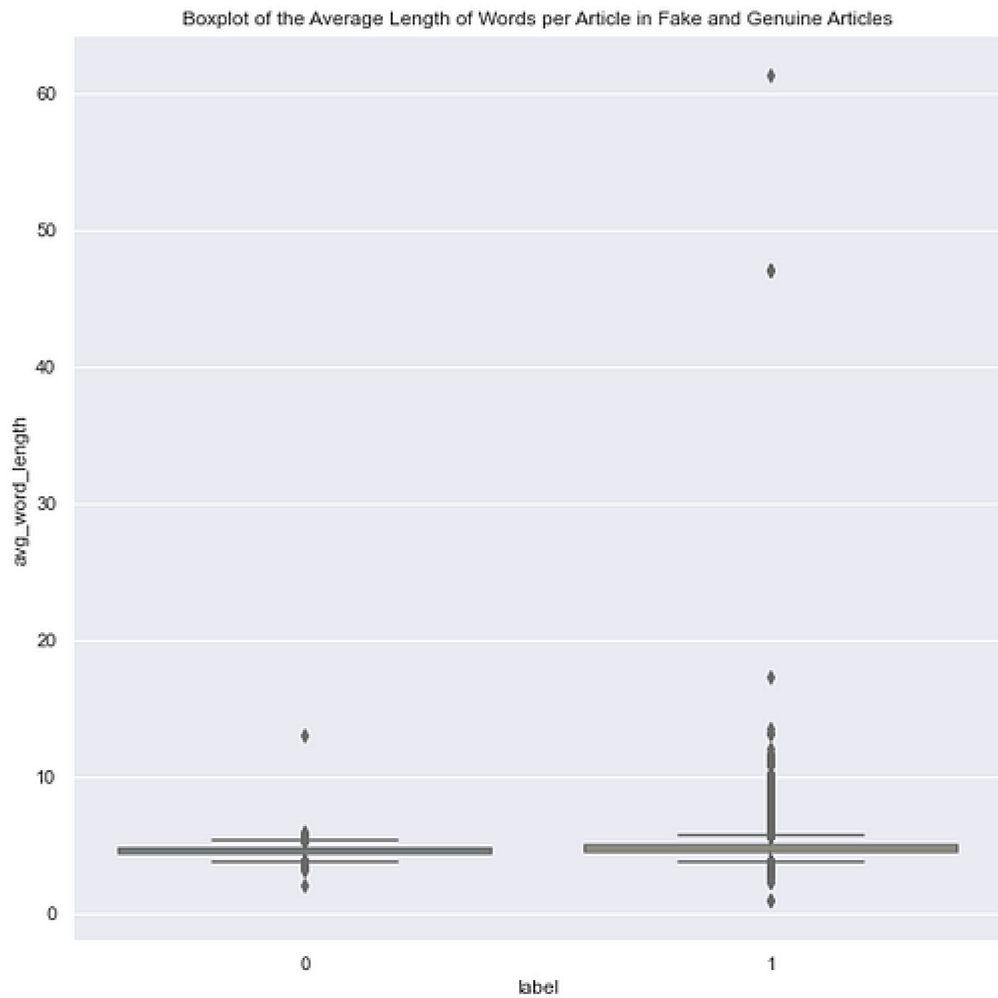


It is seen that, on average, fake articles are wordier than genuine ones.

Average Word Length per Article

This is the average word length in one article. In the box plot, it is evident that the average word length is higher in the fake articles.

```
def get_average_word_length(seq_word_tokens):
  return np.mean([len(word) for seq in seq_word_tokens for word in seq])


original_train_df['avg_word_length'] = original_train_df['sent_word_tokens'].apply(lambda x: get_average_word_length(x))


sns.boxplot(y='avg_word_length', x='label', data=original_train_df, palette="Set3")
plt.title("Boxplot of the Average Length of Words per Article in Fake and Genuine Articles")
plt.show()
```

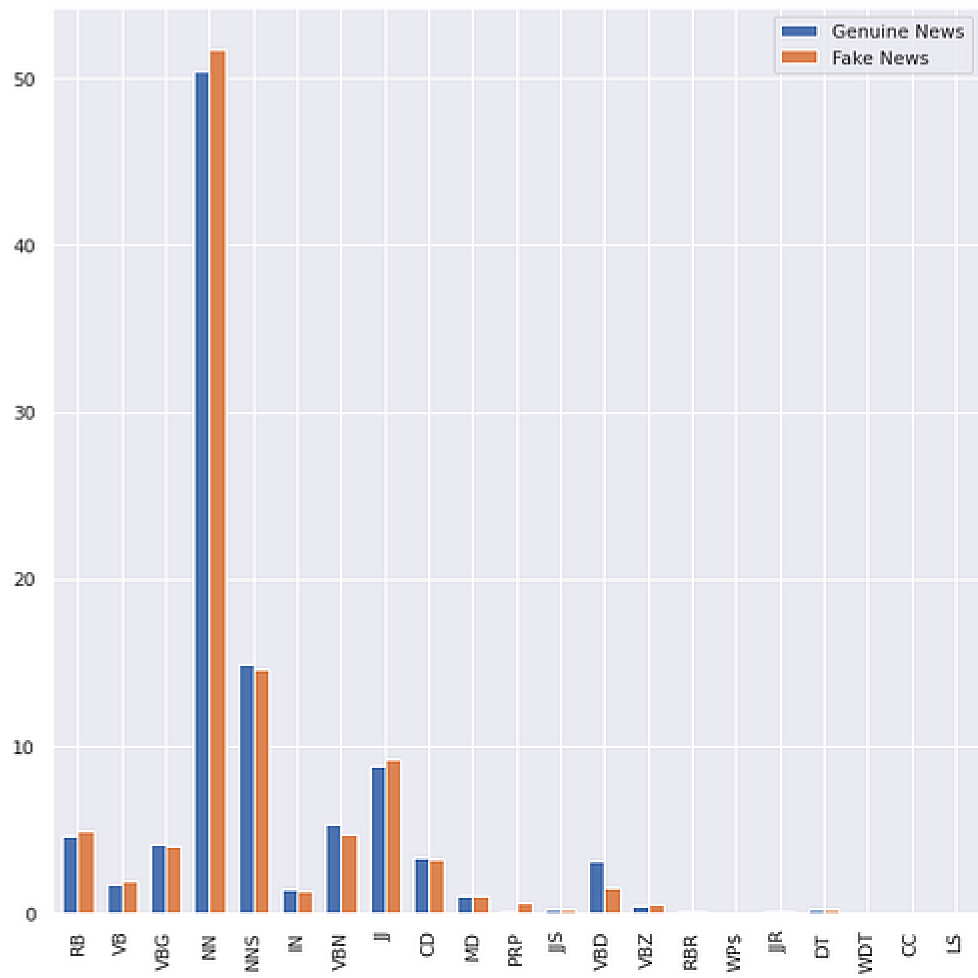


POS Tag Counts

Next, I tried to look at the part-of-speech (POS) combinations in Fake vs Genuine articles. I only stored the POS of the words into a list while iterating through each article, put the respective POS count in one DataFrame, and used a bar plot to show the percentage combination of the POS tags in Fake and News articles. The Nouns are much higher in both the articles. In general, there is no distinct pattern except for the percentage of past-tense verbs in fake news is half of that in the genuine ones. Apart from that, all other POS types are almost equal in fake and genuine articles.

```
all_tokenized_gen = [a for b in
train_df[train_df['label']==0]['text'].tolist() for a in b]

all_tokenized_fake = [a for b in
train_df[train_df['label']==1]['text'].tolist() for a in b]


def get_post_tags_list(tokenized_articles):

  all_pos_tags = []

  for word in tokenized_articles:

    pos_tag = nltk.pos_tag([word])[0][1]

    all_pos_tags.append(pos_tag)

  return all_pos_tags


all_pos_tagged_word_gen = get_post_tags_list(all_tokenized_gen)

all_pos_tagged_word_fake = get_post_tags_list(all_tokenized_fake)


pritn(all_pos_tagged_word_gen[:5])

print(all_pos_tagged_word_fake[:5])


gen_pos_df =
pd.DataFrame(dict(Counter(all_pos_tagged_word_gen)).items(),
columns=['Pos_tag', 'Genuine News'])

fake_pos_df =
pd.DataFrame(dict(Counter(all_pos_tagged_word_fake)).items(),
columns=['Pos_tag', 'Fake News'])

pos_df = gen_pos_df.merge(fake_pos_df, on='Pos_tag')


# Make percentage for comparison

pos_df['Genuine News'] = pos_df['Genuine News'] * 100 /
pos_df['Genuine News'].sum()

pos_df['Fake News'] = pos_df['Fake News'] * 100 / pos_df['Fake
News'].sum()
```

```
pos_df.head()


# plot a multiple bar chart
pos_df.plot.bar(width=0.7)
plt.xticks(range(0,len(pos_df['Pos_tag'])), pos_df['Pos_tag'])
plt.show()
```



Text Classification using Machine Learning

Tf-idf and Count Vectorizer

Once the analysis is complete, I took first the conventional way of using the Count Vectorizer and term frequency-inverse document frequency or Tf-idf. The Count Vectorizer, as configured in the code, generates bigrams as well. The counts of their occurrences are obtained in the form of a matrix using the CountVectorizer() and this word-count matrix is then transformed into the normalized term-frequency (tf-idf) representation. Here, I have used smooth=False, to avoid zero division error. By providing smooth=False, I am basically adding one to the document frequency since it is the denominator in the formula for idf calculation, as shown below —

```
idf(t) = log [ n / (df(t) + 1) ]
```

```
train_df['text_joined'] = train_df['text'].apply(lambda x: " ".join(x))

test_df['text_joined'] = test_df['text'].apply(lambda x: " ".join(x))


count_vectorizer = CountVectorizer(ngram_range=(1, 2))

tf_idf_transformer = TfidfTransformer(smooth_idf=False)


# fit and transform train data to count vectorizer

count_vectorizer.fit(train_df['text_joined'].values)

count_vect_train = count_vectorizer.transform(train_df['text_joined'].values)


# fit the counts vector to tfidf transformer

tf_idf_transformer.fit(count_vect_train)

tf_idf_train = tf_idf_transformer.transform(count_vect_train)


# Transform the test data as well
```

```
count_vect_test =
count_vectorizer.transform(test_df['text_joined'].values)

tf_idf_test = tf_idf_transformer.transform(count_vect_test)


# Train test split

X_train, X_test, y_train, y_test = train_test_split(tf_idf_train,
target, random_state=0)
```

Benchmarking with Default Configurations

Next, I intended to train the models with the default configurations and pick out the best-performing model to tune later. For this, I looped through a list and saved all the performance metrics into another DataFrame and the models in a list.

```
df_perf_metrics = pd.DataFrame(columns=['Model',
'Accuracy_Training_Set', 'Accuracy_Test_Set', 'Precision', 'Recall',
'f1_score'])

df_perf_metrics = pd.DataFrame(columns=[

    'Model', 'Accuracy_Training_Set', 'Accuracy_Test_Set',
'Precision',

    'Recall', 'f1_score', 'Training Time (secs'

])


# list to retain the models to use later for test set predictions

models_trained_list = []


def get_perf_metrics(model, i):

    # model name

    model_name = type(model).__name__

    # time keeping

    start_time = time.time()
```

```
    print("Training {} model...".format(model_name))
    # Fitting of model
    model.fit(X_train, y_train)
    print("Completed {} model training.".format(model_name))
    elapsed_time = time.time() - start_time
    # Time Elapsed
    print("Time elapsed: {:.2f} s.".format(elapsed_time))
    # Predictions
    y_pred = model.predict(X_test)
    # Add to ith row of dataframe - metrics
    df_perf_metrics.loc[i] = [
        model_name,
        model.score(X_train, y_train),
        model.score(X_test, y_test),
        precision_score(y_test, y_pred),
        recall_score(y_test, y_pred),
        f1_score(y_test, y_pred), "{:.2f}".format(elapsed_time)
    ]
    # keep a track of trained models
    models_trained_list.append(model)
    print("Completed {} model's performance
assessment.".format(model_name))
```

I used Logistic Regression, Multinomial Naive Bayes, Decision Trees, Random Forest, Gradient Boost, and Ada Boost classifiers. The precision of MultinomialNB is the best among all, but f1-score falters because of the poor recall score. In fact, recall is the worst at 68%. The best models in the results were Logistic Regression and AdaBoost whose results are similar. I chose to go with Logistic Regression to save training time.

```
models_list = [LogisticRegression(),
               MultinomialNB(),
               RandomForestClassifier(),
               DecisionTreeClassifier(),
               GradientBoostingClassifier(),
               AdaBoostClassifier()]

for n, model in enumerate(models_list):
    get_perf_metrics(model, n)
```

	Model	Accuracy_Training_Set	Accuracy_Test_Set	Precision	Recall	f1_score	Training Time (secs
0	LogisticRegression	0.982660	0.946432	0.927520	0.966562	0.946638	54.51
1	MultinomialNB	0.949333	0.844131	0.997136	0.684894	0.812034	0.59
2	RandomForestClassifier	0.999936	0.905047	0.926049	0.876869	0.900788	897.86
3	DecisionTreeClassifier	0.999936	0.903114	0.899101	0.904406	0.901745	404.55
4	GradientBoostingClassifier	0.945723	0.938697	0.924780	0.952793	0.938578	6715.15
5	AdaBoostClassifier	0.939019	0.941597	0.934109	0.948072	0.941039	2965.27

GridSearchCV for Tuning Logistic Regression Classifier

So, time to tune my chosen classifier. I started out with a wider range for max_iter and C. Then used GirdSearchCV with cv=r, i.e. 5 folds for cross-validation since label distribution is fairly distributed. I have used f1-score for scoring and used refit to return the trained model with the best f1-score.

```
model = LogisticRegression()

max_iter = [100, 200, 500, 1000]
C = [0.1, 0.5, 1, 10, 50, 100]

param_grid = dict(max_iter=max_iter, C=C)

grid = GridSearchCV(estimator=model,
                    param_grid=param_grid,
                    cv=5,
                    scoring=['f1'],
                    refit='f1',
                    verbose=2)

grid_result = grid.fit(X_train, y_train)
print('Best params: ', grid_result.best_params_)

model = grid_result.best_estimator_

y_pred = model.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, y_pred))
print('Precision: ', precision_score(y_test, y_pred))
print('Recall: ', recall_score(y_test, y_pred))
print('f1-score: ', f1_score(y_test, y_pred))
```

The best resulting model had an accuracy of 97.62% and an f1-score of 97.60%. For both, we have achieved 4% improvement. Now, I noticed that the max_iter's best value was 100, which was the lower boundary of the range, and for C, it was also 100, but it was the upper boundary of the range. So, to accommodate parameter search, I used max_iter = 50, 70 , 100 and C = 75, 100, 125. There was a marginal improvement with max_iter=100 and C=125. So, I decided to keep that constant and scaled up the parameter search for C from 120 to 150, with step size 10. All performance metrics were equal for this run to the starting grid's results. However, the value of C=140 for this run.

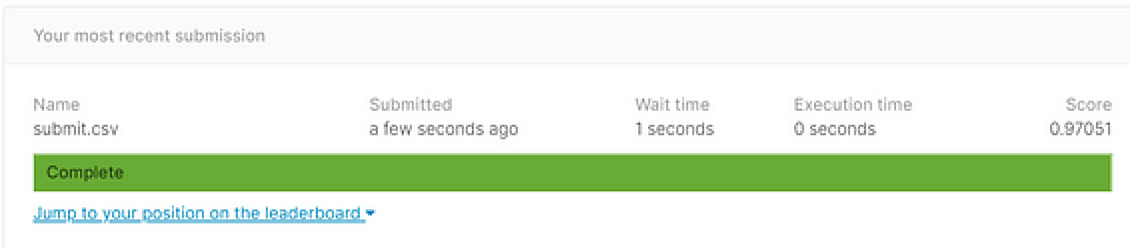
```
# Starting out with a range of values
max_iter = [100, 200, 500, 1000]
C = [0.1, 0.5, 1, 10, 50, 100]

# Attempt 2
max_iter = [50, 75, 100]
C = [75, 100, 125]

# Attempt 3
max_iter = [100]
C = [120, 130, 140, 150]

# Final Attempt - Attempt 4
max_iter = [100]
C = [100, 125, 140]
```

One final time, I ran a grid search on max_iter=100 and C = [100, 125, 140] where C had the best parameters from all the runs. The best one was max_iter=100 and C=140, which I eventually saved as the best model.

```
Accuracy:  0.9762134983562174
Precision:  0.9678916827852998
Recall:  0.98426435877262
f1-score:  0.976009362200117
```

One of the potential future works here is to test with GradientBoost and AdaBoost Classifier since their performances were also good. In some cases, the performances after tuning could be much better but in the interest of time, I would conclude here as Logistic Regression is the best performing model with max_iter=100 and C=140.

I finally uploaded the results on Kaggle. This challenge is 3 years old but I was interested in testing the score on the test data of this model.



Text Classification using GloVe and LSTM

Data Preparation

For using deep learning techniques, the text data had to re-loaded in the original format since the embedding would be a little different. In the following code, I have handled the missing values and appended the title and author of the articles to the article's text.

```
import pandas as pd

train_df = pd.read_csv('fake-news/train.csv', header=0)
test_df = pd.read_csv('fake-news/test.csv', header=0)

train_df = train_df.fillna(' ')
test_df = test_df.fillna(' ')

train_df['all_info'] = train_df['text'] + train_df['title'] +
train_df['author']
```

```
test_df['all_info'] = test_df['text'] + test_df['title'] +
test_df['author']

target = train_df['label'].values
```

Next, I used Keras API's Tokenizer class to tokenized the texts and replaced the out of vocabulary token using oov_token = "<OOV>", which actually creates a vocabulary index based on word frequency. I then fit the tokenizer on the texts and converted them into sequences of integers which uses the vocabulary index created by fitting the tokenizer. Finally, since the sequences could be of different lengths, I used pad_sequences to pad them with zeros at the end using padding=post. Each of the sequences is expected to, hence, have a length of 40, according to the code. Finally, I have split them into train and test sets.

```
from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences


tokenizer = Tokenizer(oov_token = "<OOV>", num_words=6000)

tokenizer.fit_on_texts(train_df['all_info'])


max_length = 40

vocab_size = 6000


sequences_train = tokenizer.texts_to_sequences(train_df['all_info'])

sequences_test = tokenizer.texts_to_sequences(test_df['all_info'])


padded_train = pad_sequences(sequences_train, padding = 'post',
maxlen=max_length)

padded_test = pad_sequences(sequences_test, padding = 'post',
maxlen=max_length)
```

```
X_train, X_test, y_train, y_test = train_test_split(padded_train,
target, test_size=0.2)


print(X_train.shape)

print(y_train.shape)
```

Binary Classification Model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 40, 10)            60000
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 401
=================================================================
Total params: 60,401
Trainable params: 60,401
Non-trainable params: 0
_________________________________________________________________
```

To create a model for text classification, I started with the simplest form of binary classification model structure where the first layer is an Embedding layer with expects an embedding of texts of 6000 vocab size (specified in vocab_size), each sequence of length 40 (thus, input_length=max_length) and gives an output of 40 vectors of 10 dimensions for each input sequence. Next, I used Flatten layer to flatten the matrix of shape (40, 10) into a single array of shapes (400,). Then, this array was passed through a Dense layer to produce a one-dimensional output and used sigmoid activation function to produce binary classifications. I initially thought of experimenting more with this model so created a function for it and I also like the grouping of the layers into a function as a practice. It is not really needed for this work. Finally, I compiled the model using precision and recall for metrics to monitor while training and validation.

```
def get_simple_model():
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  return model


model = get_simple_model()
print(model.summary())


model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=[tf.keras.metrics.Precision(),
tf.keras.metrics.Recall()])
```

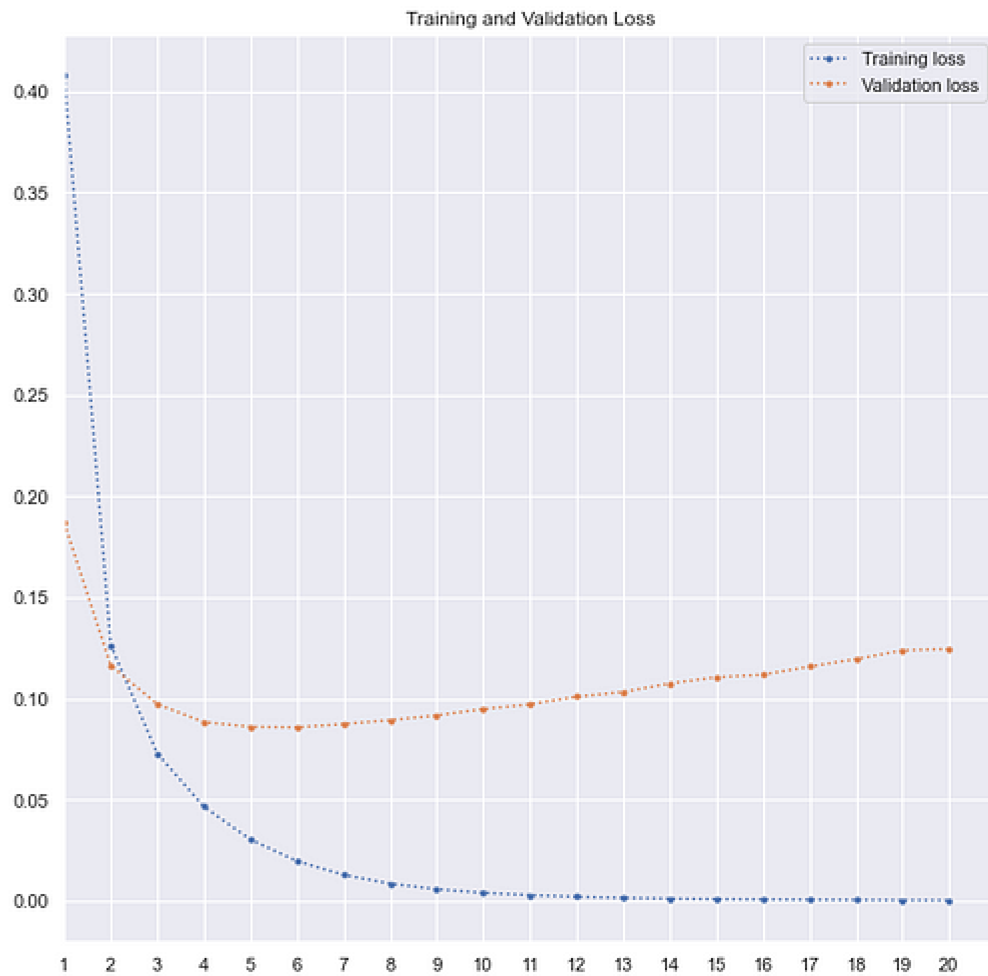
I also used early stopping to save time with patience=15 which indicates stopping if in the last 15 epochs there was no improvement in the model, and model checkpoint to store the best model with save_best_only=True. Added mode=min since I am monitoring loss here.

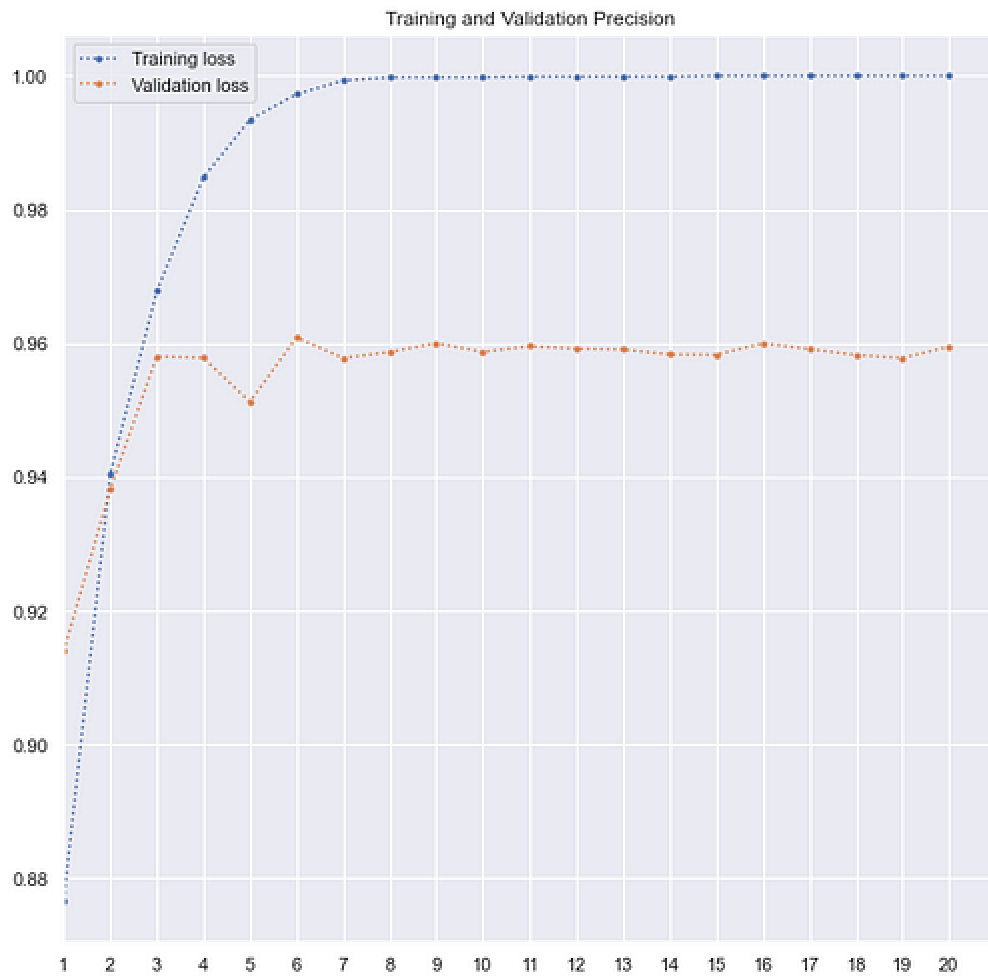
```
callbacks=[
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=15,
                                  verbose=1, mode="min",
restore_best_weights=True),
    keras.callbacks.ModelCheckpoint(filepath=best_model_file_name,
verbose=1, save_best_only=True)]
```

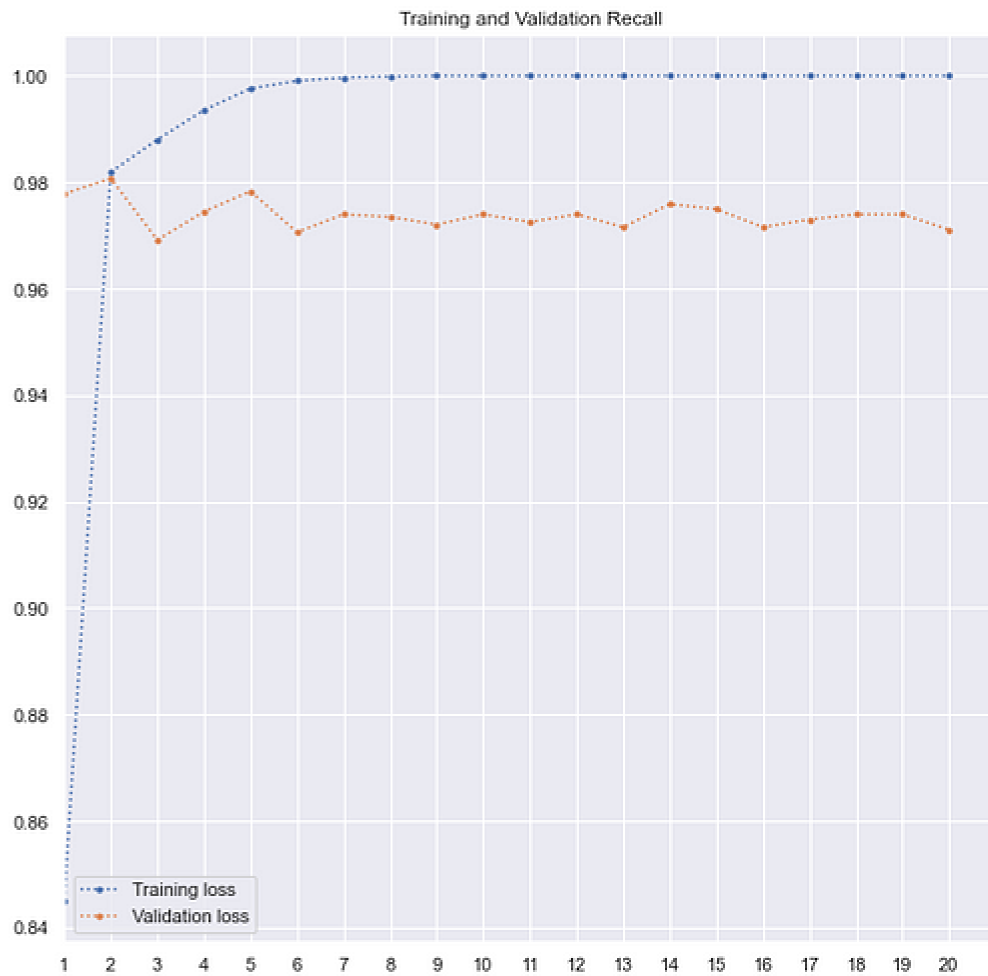
Time to fit the model now!

```
history = model.fit(X_train,
                   y_train,
                    epochs=20,
                    validation_data=(X_test, y_test),
                    callbacks=callbacks)

print(history.history.keys())
```

Since I used precision and recall, along with loss, I can also track the precision and recall values here. As in the graph below, the validation loss was the lowest at the 6th epoch and then the loss was either stagnant or increasing. Hence, the best model was saved after the 6th epoch of training. It is evident how the model was overfitting with training loss improving while the validation loss is increasing after the 6th epoch.


Training and Validation Loss
Training loss
Validation loss
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


Training and Validation Precision
Training loss
Validation loss
1.00
0.98
0.96
0.94
0.92
0.90
0.88
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20



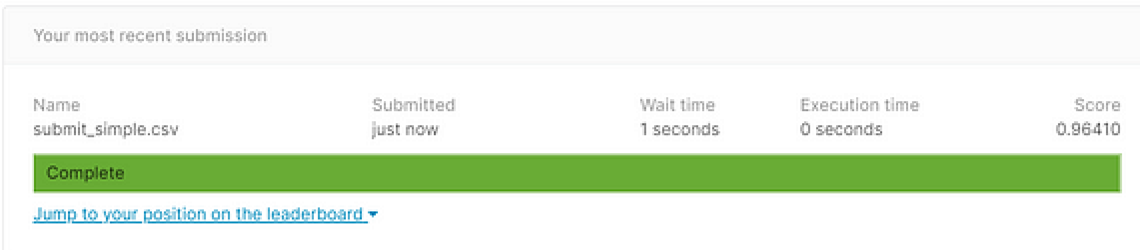
Below, is the code I used to plot the training and validation loss, precision, and recall. I used max(history.epoch) + 2 in the range function since history.epoch starts from 0. Hence, for 20 epochs, the maximum would be 19 and the range would generate a list from 1 to 18 for max(history.epoch).

```
# plot training and validation loss

metric_to_plot = "loss"

plt.plot(range(1, max(history.epoch) + 2),
history.history[metric_to_plot], ".:", label="Training loss")

plt.plot(range(1, max(history.epoch) + 2), history.history["val_" +
metric_to_plot], ".:", label="Validation loss")
```

```
plt.title('Training and Validation Loss')

plt.xlim([1,max(history.epoch) + 2])

plt.xticks(range(1, max(history.epoch) + 2))

plt.legend()

plt.show()


# plot training and validation precision


metric_to_plot = "precision"

plt.plot(range(1, max(history.epoch) + 2),
history.history[metric_to_plot], ".:", label="Training loss")

plt.plot(range(1, max(history.epoch) + 2), history.history["val_" +
metric_to_plot], ".:", label="Validation loss")

plt.title('Training and Validation Precision')

plt.xlim([1,max(history.epoch) + 2])

plt.xticks(range(1, max(history.epoch) + 2))

plt.legend()

plt.show()


# plot training and validation recall


metric_to_plot = "recall"

plt.plot(range(1, max(history.epoch) + 2),
history.history[metric_to_plot], ".:", label="Training loss")

plt.plot(range(1, max(history.epoch) + 2), history.history["val_" +
metric_to_plot], ".:", label="Validation loss")

plt.title('Training and Validation Recall')

plt.xlim([1,max(history.epoch) + 2])

plt.xticks(range(1, max(history.epoch) + 2))
```

```
plt.legend()

plt.show()
```

This model had an accuracy value of 96.6% and an f1-score of 96.6%. I also tested the performance of this model on the Kaggle test data and it was not bad, but not better than the Logistic Regression I trained earlier.



LSTM

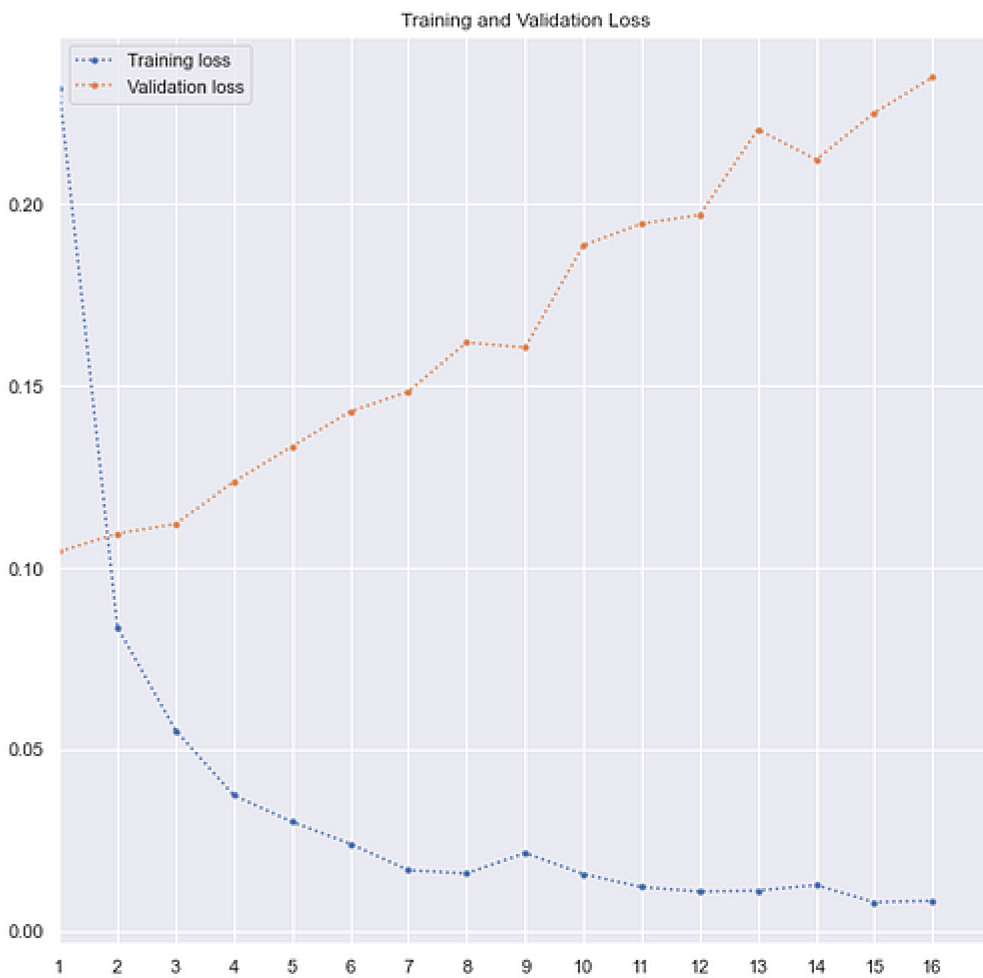
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 40, 10)            60000
_________________________________________________________________
dropout (Dropout)            (None, 40, 10)            0
_________________________________________________________________
lstm (LSTM)                  (None, 100)               44400
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                6464
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 110,929
Trainable params: 110,929
Non-trainable params: 0
```

Phew! Now let's fit an LSTM model to the text data. The first and the last layer are the same since the input and the outputs are the same. In between, I have used a Dropout layer to filter out 30% of units and then go to the LSTM layer of 100 units. Long Short Term Memory (LSTM), is a special kind of RNN, capable of learning long-term dependencies. Their specialty lies in remembering information for a longer period of time. After using LSTM, I used another Dropout layer, then a fully-connected layer with 64 hidden units, then another Dropout layer, and finally another fully-connected layer of one unit with 'Sigmoid' activation function for binary classification.

```
def get_simple_LSTM_model():
  model = Sequential()
  model.add(Embedding(vocab_size, 10, input_length=max_length))
  model.add(Dropout(0.3))
  model.add(LSTM(100))
  model.add(Dropout(0.3))
  model.add(Dense(64, activation='relu'))
  model.add(Dropout(0.3))
  model.add(Dense(1, activation='sigmoid'))
  return model

model = get_simple_LSTM_model()
print(model.summary())
```

Once done, I followed the same process outlined in the previous section to compile, use callback, and fit the model. The number of epochs I provided was 20. But in this case, the model trained for only 16 epochs because for 15 consecutive iterations after the first epoch there was no improvement in the validation loss. It is clear from the plot below as well. The validation loss has been only increasing while the training loss was going down due to over-fitting. Recall the callback settings where I had encoded the model to wait for an improvement in validation loss for 15 consecutive epochs before stopping.

```
callbacks=[
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=15,
                                  verbose=1, mode="min",
restore_best_weights=True),
    keras.callbacks.ModelCheckpoint(filepath=best_model_file_name,
verbose=1, save_best_only=True)
]
```



There was no significant improvement in this model although there are potential improvements that could be made to this model. It has an accuracy of 96.1% and an f1-score of 96.14%.

Using pre-trained Word Embedding — GloVe

Now, we can also use pre-trained word-embeddings, like GloVe. GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. [4]

I have used the one which was trained on 6 billion tokens with 400k vocabulary, represented in 300-dimensional vector format.

In the following code, I have a code to load GloVe on Google Colab since I was partly working on Colab.

```
# Load GloVe on Colab
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove*.zip

f = open('/content/glove.6B.300d.txt')

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded {} word vectors.'.format(len(embeddings_index)))



# Load local GloVe weights - Download the file and store it
```

```
embeddings_index = dict()
f = open('your/path/glove.6B/glove.6B.300d.txt', encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded {} word vectors.'.format(len(embeddings_index)))
```

Next, our objective is to find the tokens in the fake news data in the GloVe embedding and get the corresponding weights.

```
# create a weight matrix for words in training docs
print('Get vocab_size')
vocab_size = len(tokenizer.word_index) + 1

print('Create the embedding matrix')
embedding_matrix = np.zeros((vocab_size, 300))
for word, i in tokenizer.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

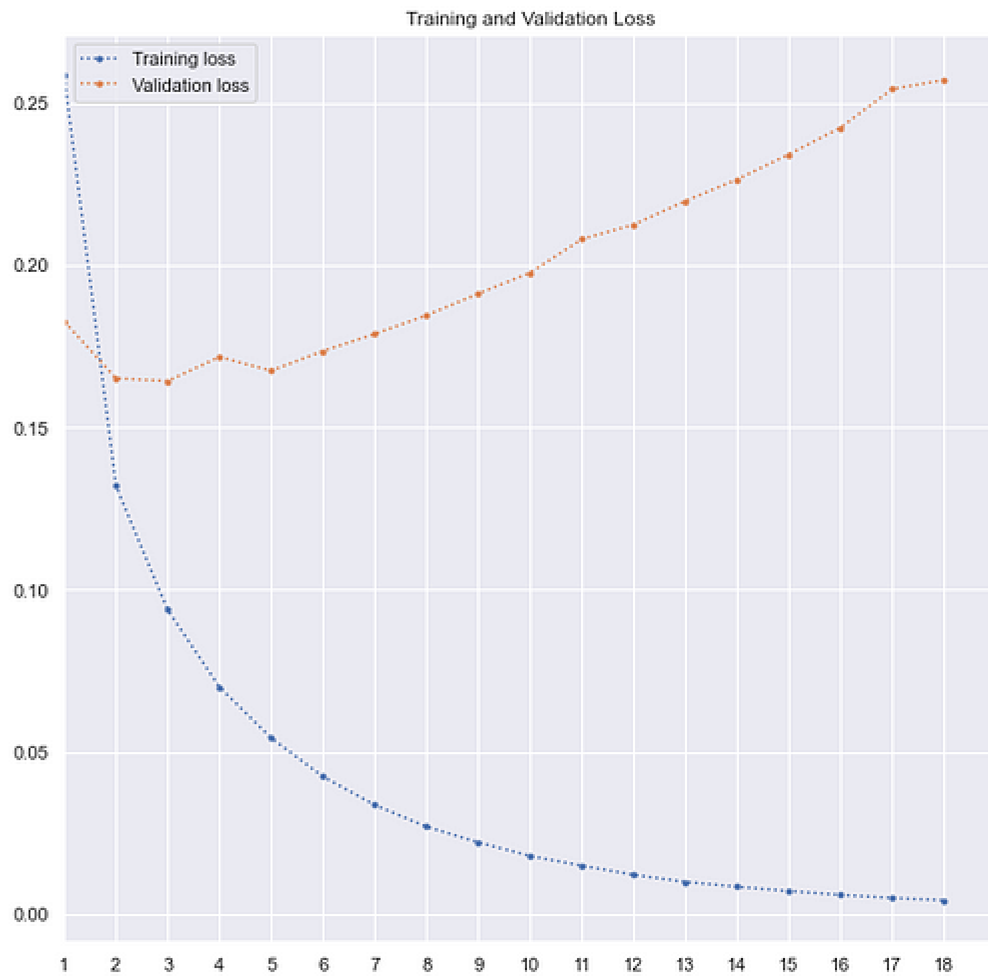
Simple Model with Glove

Now, that I have the GloVe embedding for our training data, I used the Embedding layer with output_dim=300, which is the GloVe vector representation shape. Also, I used trainable=False, since I am using pre-trained weights, I should not update them while training. They hold relationships with other words so it is best not to disturb that.

```
# The best model file name for uniformity
best_model_file_name = "models/best_model_simple_with_GloVe.hdf5"

# the model
def get_simple_GloVe_model():
    model = Sequential()
    model.add(Embedding(vocab_size,
                        300,
                        weights=[embedding_matrix],
                        input_length=max_length,
                        trainable=False))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    return model

callbacks=[
    keras.callbacks.EarlyStopping(monitor="val_loss",
                                  patience=15,
                                  verbose=1,
                                  mode="min",
                                  restore_best_weights=True),
    keras.callbacks.ModelCheckpoint(filepath=best_model_file_name,
                                    verbose=1,
                                    save_best_only=True)
]

model = get_simple_GloVe_model()
print(model.summary())
```

```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=[tf.keras.metrics.Precision(),
tf.keras.metrics.Recall()])

history = model.fit(X_train,
                    y_train,
                    epochs=50,
                    validation_data=(X_test, y_test),
                    callbacks=callbacks)

model = keras.models.load_model(best_model_file_name)
y_pred = (model.predict(X_test) > 0.5).astype("int32")
print('Accuracy: ', accuracy_score(y_test, y_pred))
print('Precision: ', precision_score(y_test, y_pred))
print('Recall: ', recall_score(y_test, y_pred))
print('F1 Score: ', f1_score(y_test, y_pred))
```

Finally, with the same process as I used earlier, I trained the model with 50 epochs. However, since there was no improvement after the 3rd epoch, the model stopped training after the 18th epoch. The scores were lower than the previous two models. The accuracy and f1-score were both ~93%.



GloVe with LSTM

And.. finally, I used the GloVe embedding to train the LSTM model I used earlier to achieve better results. The complete code is below –

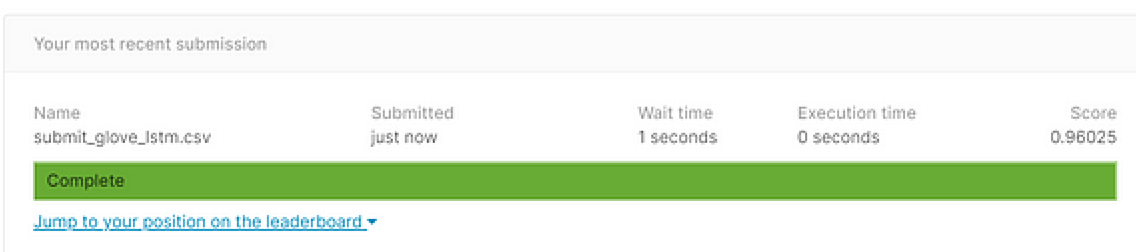
```
# The best model file name for uniformity
best_model_file_name = "models/best_model_LSTM_with_GloVe.hdf5"

# the model
def get_simple_GloVe_model():
    model = Sequential()
    model.add(Embedding(vocab_size,
```

```
                    300,
                    weights=[embedding_matrix],
                    input_length=max_length,
                    trainable=False))
    model.add(LSTM(100))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    return model

callbacks=[
    keras.callbacks.EarlyStopping(monitor="val_loss",
                                  patience=15,
                                  verbose=1,
                                  mode="min",
                                  restore_best_weights=True),
    keras.callbacks.ModelCheckpoint(filepath=best_model_file_name,
                                    verbose=1,
                                    save_best_only=True)
]

model = get_simple_GloVe_model()
print(model.summary())

model.compile(loss='binary_crossentropy',
              optimizer='adam',
```

```
              metrics=[tf.keras.metrics.Precision(),
tf.keras.metrics.Recall()])


history = model.fit(X_train,
                    y_train,
                    epochs=50,
                    validation_data=(X_test, y_test),
                    callbacks=callbacks)


model = keras.models.load_model(best_model_file_name)
y_pred = (model.predict(X_test) > 0.5).astype("int32")
print('Accuracy: ', accuracy_score(y_test, y_pred))
print('Precision: ', precision_score(y_test, y_pred))
print('Recall: ', recall_score(y_test, y_pred))
print('F1 Score: ', f1_score(y_test, y_pred))
```

Again, I used 50 epochs and the model did not improve after the third epoch. Therefore, the training process stopped after the 18th epoch. The accuracy and f1-score both improved to 96.5% which is close to the first Keras model.

So, I tried the predictions for this model on Kaggle’s test data, and here is my result —



Conclusion

In this exercise, the best model was the tuned Logistic Regression model. There are loads of scopes for further improvement on this use case, especially designing better deep learning models. Also, in the interest of time, I did not tune the Random Forest and AdaBoost classifier which could result in improved performance than the Logistic Regression.