PROJECT REPORT ON

FAKE NEWS DETECTION USING NLP

Subject is in the partial fulfilment for the requirements for the degree of

BACHELOR OF ENGINEERING

In

ELECTRONICS AND COMMUNICATION ENGINEERING

Under the guidance of

Mr.K.BALAJI

GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE,New Delhi Affiliated to

Anna University,Chennai

GRT Mahalakshmi nagar,Chennai – Tirupathi Highway ,Tiruttani-631 209

PROJECT REPORT SUBMITTED BY,
NAME: GOKUL NATH S
NM ID: au110321106013
Mail id: 12345.gokulnath.s@gmail.com
Year/sem/dept: III/V/ECE

Self Declaration

I **GOKUL NATH S** hereby declare that the project report entitled creating a **Fake News Detection Using NLP** is done by me under the guidance of **Mr BALAJI K** is submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering degree in Electronics and Communication Engineering.

Date of submission : 01/11/2023

Place: GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY, TIRUTTANI-631209

SIGNATURE OF THE CANDIDATE

NAAN MUDHALVAN PROJECT(IBM)

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

PROJECT:

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

Abstract:

Fake news is a real problem in today's world, and it has become more extensive and harder to identify. A major challenge in fake news detection is to detect it in the early phase. Another challenge in fake news detection is the unavailability or the shortage of labelled data for training the detection models. We propose a novel fake news detection framework that can address these challenges. Our proposed framework exploits the information from the news articles and the social contexts to detect fake news. The proposed model is based on a Transformer architecture, which has two parts: the encoder part to learn useful representations from the fake news data and the decoder part that predicts the future behaviour based on past observations. We also incorporate many features from the news content and social contexts into our model to help us classify the news better. In addition, we propose an effective labelling technique to address the label shortage problem. Experimental results on real-world data show that our model can detect fake news with higher accuracy within a few minutes after it propagates (early detection) than the baselines.

In today's digital era, the spread of information via social media and internet platforms has given people the power to access news from many different sources. The growth of fake news, meanwhile, is a drawback of this independence. Fake news is inaccurate information that has been purposefully spread to confuse the public and undermine confidence in reputable journalism. Maintaining an informed and united global community requires identifying and eliminating fake news. NLP, a subfield of artificial intelligence, gives computers the capacity to comprehend and interpret human language, making it a crucial tool for identifying deceptive information. This article examines how NLP can be used to identify fake news and gives examples of how it can be used to unearth misleading data.

Introduction:

Fake news detection is a subtask of text classification and is often defined as the task of classifying news as real or fake. The term 'fake news' refers to the false or misleading information that appears as real news. It aims to deceive or mislead people. Fake news comes in many forms, such as clickbait (misleading headlines), disinformation (with malicious intention to mislead the public), misinformation (false information regardless of the motive behind), hoax, parody, satire, rumour, deceptive news and other forms as discussed in the literature.

Fake news is not a new topic; however, it has become a hot topic since the 2016 US election. Traditionally, people get news from trusted sources, media outlets and editors, usually following a strict code of practice. In the late twentieth century, the internet has provided a new way to consume, publish and share information with little or no editorial standards. Lately, social media has become a significant source of news for many people. According to a report by Statistica, there are around 3.6 billion social media users (about half the population) in the world. There are obvious benefits of social media sites and networks in news dissemination, such as instantaneous access to information, free distribution, no time limit, and variety. However,

these platforms are largely unregulated. Therefore, it is often difficult to tell whether some news is real or fake.

Recent studies show that the speed at which fake news travels is unprecedented, and the outcome is its wide-scale proliferation. A clear example of this is the spread of anti-vaccination misinformationFootnote2 and the rumour that incorrectly compared the number of registered voters in 2018 to the number of votes cast in US Elections 2020. The implications of such news are seen during the anti-vaccine movements that prevented the global fight against COVID-19 or in post-election unrest. Therefore, it is critically important to stop the spread of fake news at an early stage.

A significant research gap in the current state-of-the-art is that it focuses primarily on fake news detection rather than early fake news detection. The seminal works on early detection of fake news usually detect the fake news after at least 12 h of news propagation, which may be too late . An effective model should be able to detect fake news early, which is the motivation of this research.

Another issue that we want to highlight here is the scarcity of labelled fake news data (news labelled as real or fake) in real-world scenarios. Existing state-of-the-art works [4, 7, 8] generally use fully labelled data to classify fake news. However, the real-world data is likely to be largely unlabelled. Considering the practical constraints, such as unavailability of the domain experts for labelling, cost of manual labelling, and difficulty of choosing a proper label for each news item, we need to find an effective way to train a large-scale model. One alternative approach is to leverage noisy, limited, or imprecise sources to supervise labelling of large amounts of training data. The idea is that the training labels may be imprecise and partial but can be used to create a strong predictive model. This scheme of training labels is the weak supervision technique.

Usually, the fake news detection methods are trained on the current data (available during that time), which may not generalize to future events. Many of the labelled samples from the verified fake news get outdated soon with the newly developed events. For example, a model trained on fake news data before the COVID-19 may not classify fake news properly during COVID-19. The problem of dealing with a target concept (e.g. news as ‘real’ or ‘fake’) when the underlying relationship between the input data and target variable changes over time is called concept drift. In this paper, we investigate whether concept drift affects the performance of our detection model, and if so, how we can mitigate them.

This paper addresses the challenges mentioned above (early fake news detection and scarcity of labelled data) to identify fake news. We propose a novel framework based on a deep neural network architecture for fake news detection. The existing works, in this regard, rely on the content of news, social contexts, or both. We include a broader set of news-related features and social context features compared to the previous works. We try to detect fake news early (i.e. after a few minutes of news propagation). We address the label shortage problem that happens in real-world scenarios. Furthermore, our model can combat concept drift.

Inspired by the bidirectional and autoregressive Transformer (BART) model from Facebook that is successfully used in language modelling tasks, we propose to apply a deep bidirectional encoder and a left-to-right decoder under the hood of one unified model for the task of fake news detection. We choose to work with the BART model over the state-of-the-art BERT model , which has demonstrated its abilities in NLP (natural language processing) tasks (e.g. question answering and language inference), as well as the GPT-2 model , which has impressive autoregressive (time-series) properties. The main reason is that the BART model combines the unique features (bidirectional and autoregressive) of both text generation and temporal modelling, which we require to meet our goals.

Though we take inspiration from BART, our model is different from the original BART in the following aspects: in comparison with the original BART, which takes a single sentence/document as input, we incorporate a rich set of features (from news content and social contexts) into the encoder part; we use a decoder to get predictions not only from previous text sequences (in this case, news articles) as in the original BART but also from previous user behaviour (how users respond to those articles) sequences, and we detect fake news early by temporally modelling user behaviour; on top of the original BART model, we add a single linear layer to classify news as fake or real.

Problem Statement:

News consumption is a double-edged sword. On the one hand, its low cost, easy access, and rapid dissemination of information lead people to seek out and consume news. It enables the wide spread of “fake news”, i.e., low quality news with intentionally false information. The extensive spread of fake news has the potential for extremely negative impacts on individuals and society. Therefore, fake news detection has recently become an emerging research that is attracting tremendous attention. First, fake news is intentionally written to mislead readers to believe false information, which makes it difficult and nontrivial to detect based on news content. To develop a FAKE NEWS DETECTION system using natural language processing and its accuracy will be tested using machine learning algorithms. The algorithm must be able to detect fake news in a given scenario.

Some challenges in fake news detection include:

1. Detecting it in the early phase
2. The unavailability or shortage of labeled data for training the detection models
3. The lack of a massive dataset and a labeled benchmark dataset with ground-truth labels

Some approaches to fake news detection include:

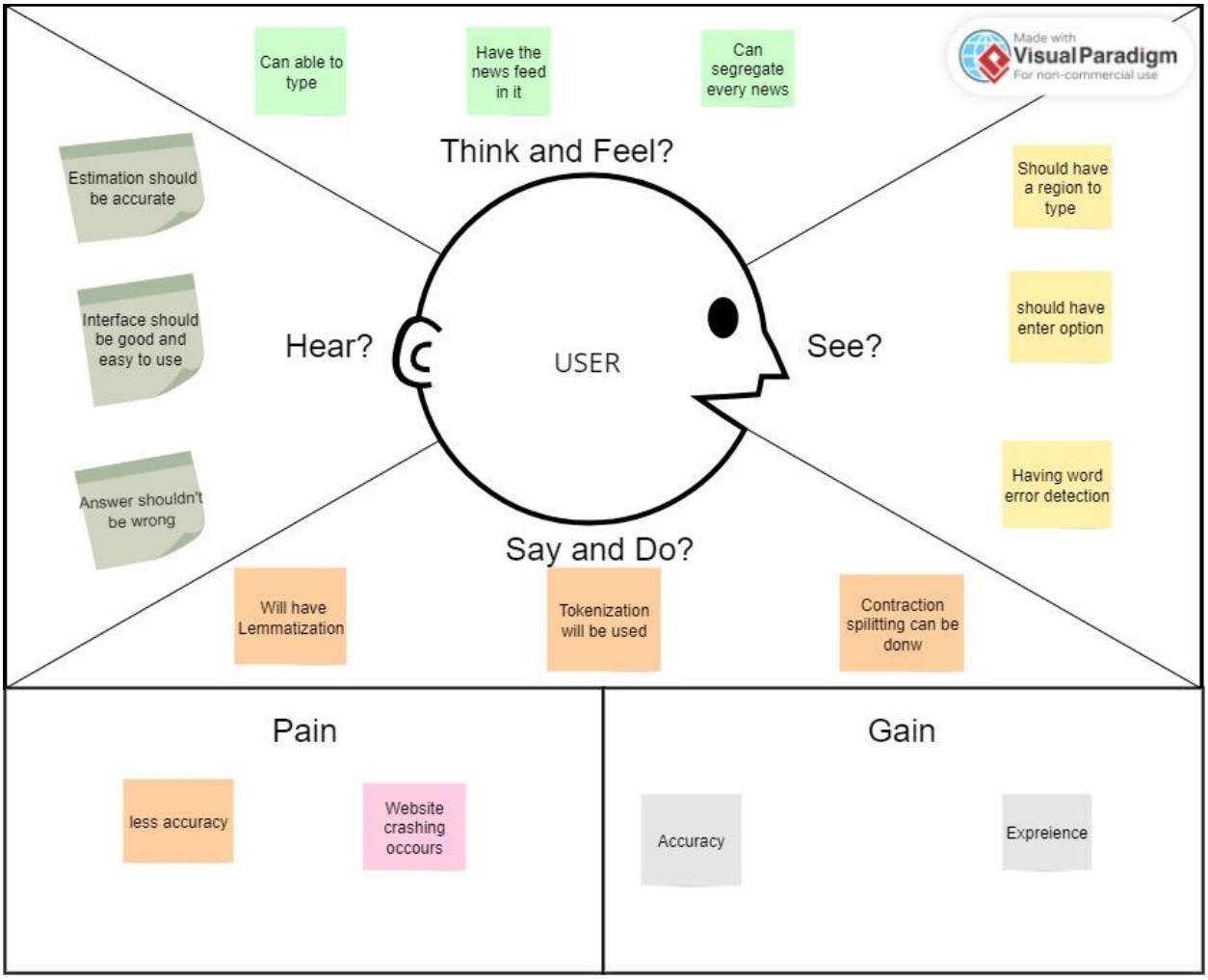
- We are using NLP based algorithm to build a model to predict whether a given news article is real or fake based on its text
- We will use Bag of Words to convert text to vector format
- BY using TF-IDF to extract the feature

- By using CNN for classification
- Using classification algorithms such as logistic regression, extreme gradient boosting, decision tree, and random forest classifier

Customer Problem Statement:

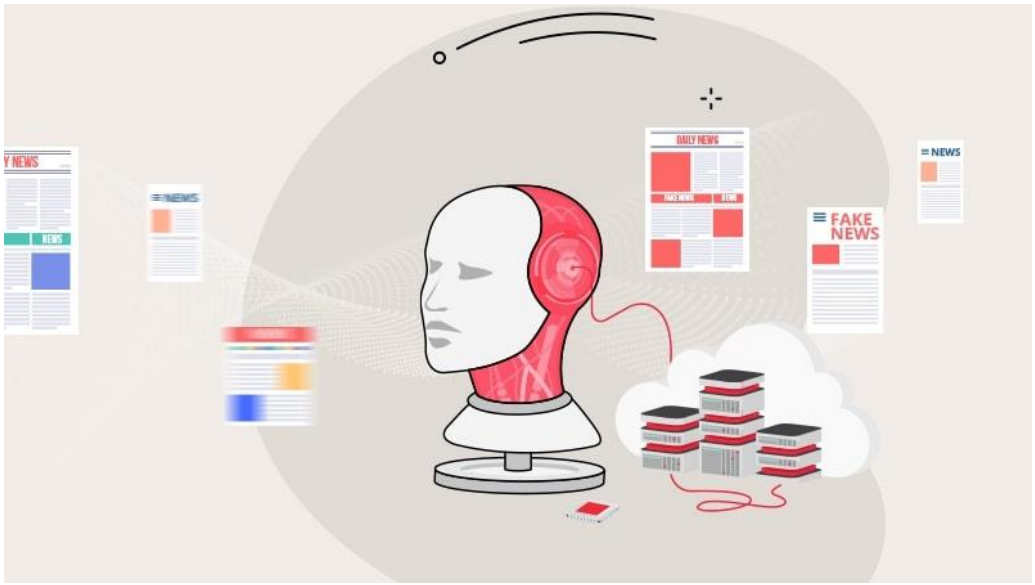
Nowadays, online fake news has become the main aspect of the growing interest in online media, social-networking sites, and online news portals (Bondielli and Marcelloni 2019). However, most people are generally incompetent for spending adequate time cross-checking the references and for ensuring the credibility of news (Zhou and Zafarani 2020; D'Ulizia 2021). Thus, more attention to fake news detection inspires the research community. In recent days, more research works regarding fake news detection have been implemented (Rama Krishna et al. 2021), though several studies only concentrated on news of specific categories like political or e-commerce reviews. Consequently, they have designed and developed certain features with some standard datasets with their topic of interest. These studies face poor performance in detecting news of another topic and also dataset bias (Beer and Matthee 2020). Therefore, it is necessary for studying whether these models are suitable for diverse classes of news propagated in social media through the evaluation of diverse datasets on different models and investigating their efficiency or performances (Ahmad et al. 2020). On the other hand, conventional studies on fake news detection techniques are focused on either a limited number of models or a particular category of the dataset (Dabbous et al. 2020a). Thus, there is a need of reviewing a fake news detection model.

Empathize & Discover:

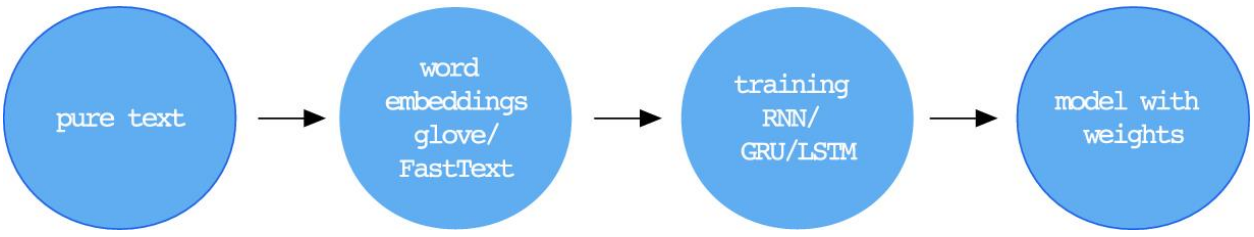


This is empathize map and discovery which is been implemented by our team and these are some of the ideologies that we want to implement in it.

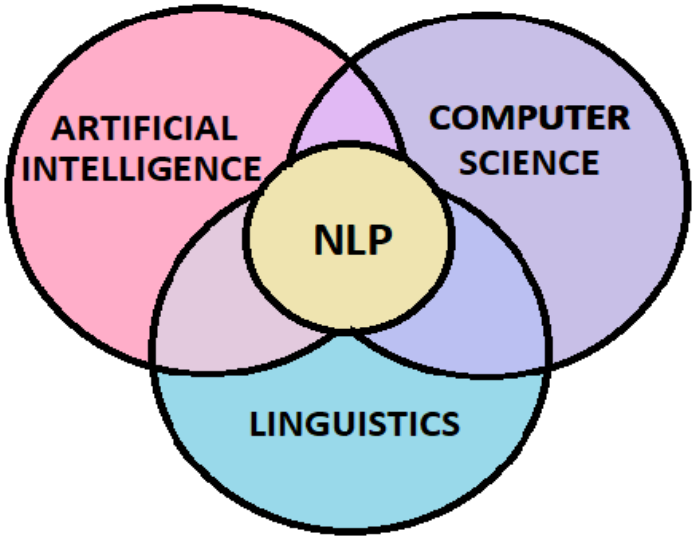
Innovation To Solve The Problem On Fake News Detection Using NLP



To tackle the challenge of fake news, the application of the Flair library is proposed, which offers outstanding features in terms of neural network design, includes many state-of-the-art methods, among them numerous methods based on the deep learning, also enabling GPU-based training. Flair is a Natural Language Processing library designed for all word embeddings as well as arbitrary combinations of embeddings. The crucial elements of creating the fake news detection model were carried out with the support of the Flair library. The training process was carried out based on deep learning methods afterword embeddings had been carried out using the modern and effective procedures in this area. In our work, we chose to use various types of neural networks to solve the problem of text-based fake news detection.

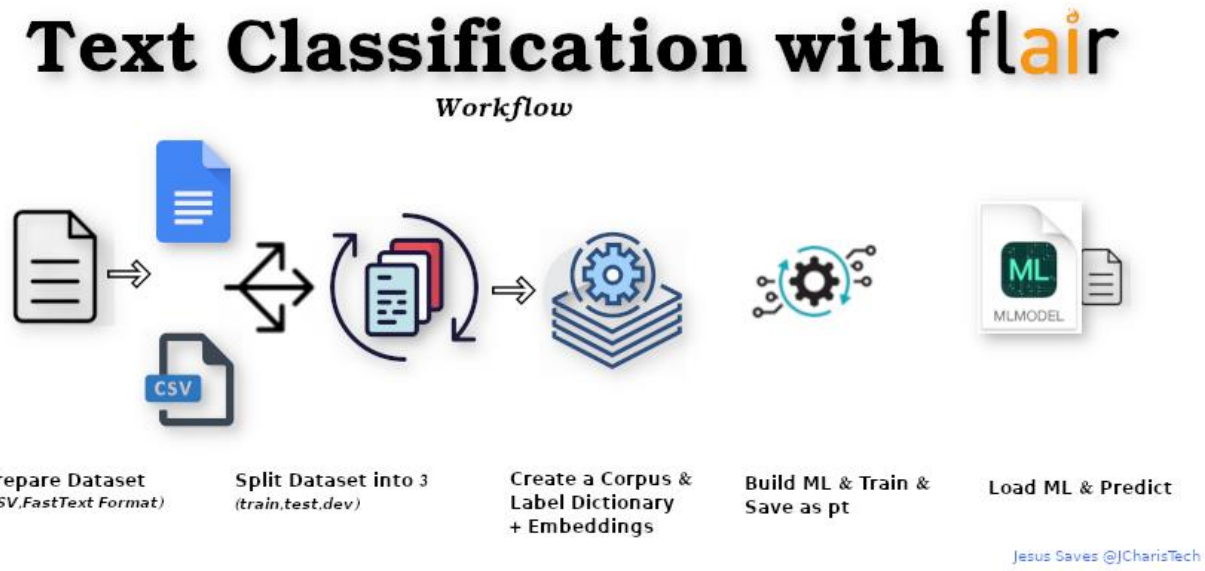


Text Pre-processing Using NLP



The goal of text pre-processing is to obtain the text, which is a reduced representation of the raw text. The reduced text enables the detection of specific patterns of the raw text simultaneously. A reduction strategy was adopted, consisting of the elimination of unnecessary elements and, through this step, achieving a higher generalization of the text. To detect unnecessary items and overrepresentation of words, statistical analysis of their occurrences in datasets was used. The clean text was obtained by creating a separate code, unrelated to the Flair library. Due to Flair use of embedding layers, it is not necessary to run the usual pre-processing steps such as constructing a vocabulary of words in the dataset or encoding words as one-hot vectors [7]. In the Flair, each embedding layer implements either the TokenEmbedding or the DocumentEmbedding interface for word and document embeddings respectively [7]. In our approach, we treated the content of articles as documents and we applied the DocumentEmbedding interface.

Word Embeddings in Flair



Neural networks used in NLP tasks do not operate directly on texts, sentences, or words, but on their representation in the numerical form. This process of converting them into numbers is called word embeddings and it is one of the key elements enabling sentiment analysis and fake news detection.

The main methods of word embeddings are ‘word2vec’, ‘glove’, and ‘FastText’, which are classified as canonical methods. In addition to the listed above, the Flair library supports a growing list of embeddings such as hierarchical character features, ELMo embeddings, ELMo transformer embeddings, BERT embeddings, byte pair embeddings, Flair embeddings and Pooled Flair embeddings .

In this work, the ‘glove’ method was used. For comparative purposes, the ‘twitter’ word embeddings, ‘news’ word embeddings and ‘crawl’ word embeddings were used as well. The synthesis of the methods used is summarized below.

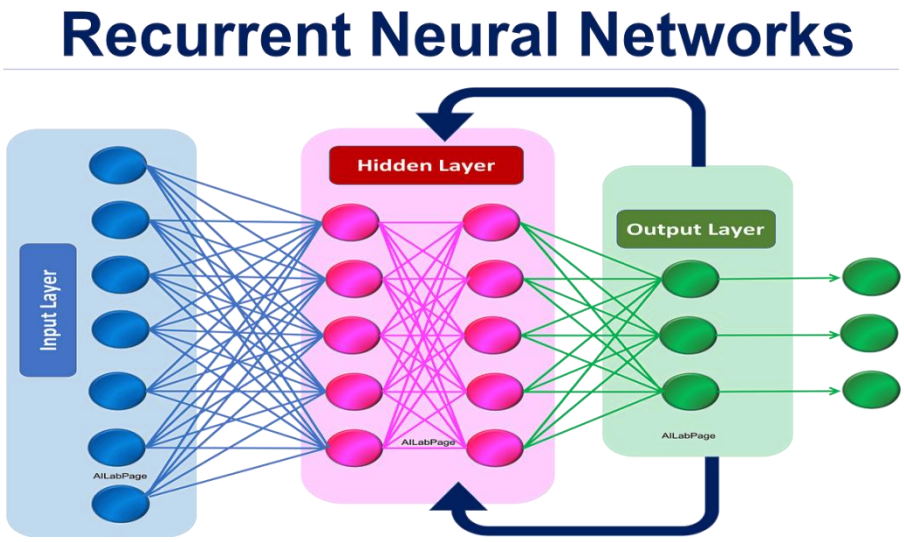
The ‘glove’ is an open-source project at Stanford University; its code is freely available [8]. The ‘glove’ overcomes the disadvantages of the models focusing only on local statistics and the models focusing only on global statistics. For example, methods like latent semantic analysis (LSA) efficiently leverage statistical information, but they do relatively poorly on the word analogy task. The other example, skip-gram methods, may do better on the analogy task, but they poorly utilize the statistics of the corpus. The ‘glove’ is a specific weighted least squares model that trains on global word-word co-occurrence counts and thus makes efficient use of statistics. The ‘glove’ is a global log-bilinear regression model for the unsupervised learning of word representations that outperforms other models on word analogy and word similarity.

To launch word embeddings in the Flair with the use of ‘glove’, the user enters the following WordEmbeddings (‘glove’) command in the code.

The FastText method was created by the Facebook AI Research lab based on the models contained in the article. The FastText method is based on the bag of n-grams and subword units. Each word is represented as a bag of character n-grams [10]. The method indicates better results than the state-of-the-art methods in word similarity and word analogy experiments.

Both methods are available in the Flair library as pre-trained databases of word embeddings. The ‘glove’ and the FastText methods were created based on data obtained from Wikipedia. Pre-trained models used in this paper, like ‘news’, were created using FastText embeddings over news and Wikipedia data; the ‘crawl’ was created using the FastText embeddings over web crawls; ‘twitter’ was created using two billion tweets.

Recurrent Neural Network



Currently, the text classification methods most often use methods based on Deep Neural Networks (DNN), which have better performance in Natural Language Processing (NLP) tasks solving than other neural networks. DNNs are characterized by high complexity and a large number of hidden layers, which is their distinguishing feature in comparison with standard Artificial Neural Networks (ANN).

Deep Neural Networks have already been extensively used in many areas of artificial intelligence, such as speech recognition, image recognition, text translation, sentiment analysis, and spam detection. There is a whole range of DNN methods used in NLP. In this article, we

focus on Recurrent Neural Network (RNN) as well as Gated Recurrent Unit (GRU) and Long-Short Term Memory (LSTM) methods that are classified as RNN methods (networks).

The feature distinguishing RNN networks from other ANN networks is their recurrency, referring to the flow of signals between input and output of the network. This type of networks has a kind of feedback loop, which means that the output is also the input for the next state and affects its output value. Such a network architecture results in the fact that the network has a kind of memory that theoretically allows for information storage. Apart from the difference mentioned above, the RNN network works like a regular, one-way ANN network, that is during the training weights and propagation errors are calculated.

Experimental Setup

Finding the right dataset is fundamental to create an efficient, reliable fake news detection model. Simultaneously, the access to such datasets is limited and creates a challenge to acquire current, ready-to-learn databases. In the article, we applied freely available datasets, which are accessible on the websites of Kaggle and the Information Security and Object Technology (ISOT) research lab. Two different sets of data were applied, one called “ISOT Fake News Dataset” [14], the other called “Getting real about fake news” (GRaFN).

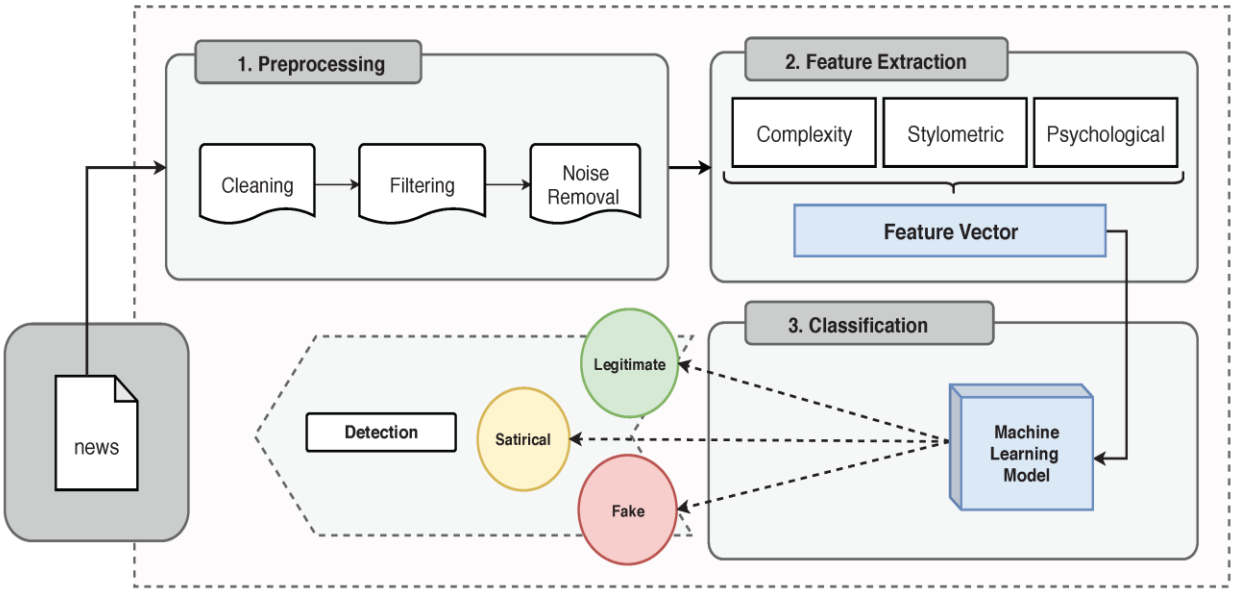
Two models were taught, the first one is based on the application of the ISOT dataset for training, and the second model, because the collection acquired from the Kaggle contains mostly fake news, was taught with the use of both collections, through attaching the real news collection from ISOT to the collection downloaded from the Kaggle webpage.

Information in both datasets contains news published on websites. The ISOT collection is dominated by the vast majority of political information and news from around the world. The dataset contains two files (true and fake) in csv file format. The real information database was created based on the websites of a reliable Reuters news agency, and the fake information was collected from the pages marked as unreliable by Politifact. The dataset contains a total of 44898 items, 21417 of which are real items and 23481 are fake items. Each file contains four columns: article title, text, article publication date and the subject which can relate to one of six types of information (world-news, politics-news, government-news, middle-east, US news, left-news) [14]. In order to prepare the data for pre-processing in a proper way, an analysis of the occurrence of e-mail addresses, social media addresses, website addresses (https and www) was conducted

BLOCKS TO BE ADDED

Creating a complete block diagram for a fake news detection system using NLP is a complex task, but I can provide a simplified representation of the key components and their connections. Please note that this is a high-level overview, and actual system architecture may vary based on specific requirements and technologies used.

Here's a simplified block diagram:



- The "Data Collection Block" collects news articles, social media posts, or other textual data from various sources.
- The "Preprocessing Block" cleans and standardizes the text data, performs language identification, and handles multimedia content if necessary.
- The "Feature Extraction Block" extracts relevant features from the preprocessed data, such as word embeddings, sentiment scores, and named entities.
- -The "Deep Learning Models Block" includes deep learning models for fake news detection, which take the extracted features as input.
- The "Explainability Block" provides explanations for the model's decisions, enhancing transparency.
- The "Real-Time Processing Block" handles incoming data streams and makes real-time predictions.
- The "Continuous Learning Block" updates and retrains the models to adapt to evolving misinformation tactics.
- The "Cross-Lingual Support Block" ensures the system can handle multiple languages.
- The "User Behavior Analysis Block" monitors user interactions and patterns.
- The "Ensemble Block" combines the outputs of multiple models for improved accuracy.
- The "Privacy-Preserving Block" protects user privacy.
- The "Human-in-the-Loop Block" involves human reviewers in the decision-making process.
- The "Fact-Checking Integration Block" integrates with external fact-checking resources.
- The "Blockchain Verification Block" verifies source authenticity using blockchain.
- The "Collaborative Filtering Block" recommends trustworthy sources based on user preferences and trust networks.
- Finally, the "Final Decision" is made based on the outputs of the various blocks, determining whether a piece of content is fake or not.

Please note that the actual implementation and connections between these blocks may vary depending on the system's complexity and specific goals. This diagram provides a high-level overview of the components involved in a fake news detection system using NLP.

PRE-PROCESSING THE DATASET

With the explosion of online fake news and disinformation, it is increasingly difficult to discern fact from fiction. And as natural language processing become more popular, Fake News detection serves as a great introduction to NLP.

Google Cloud Natural Language API is a great platform to use for this project. Simply upload a dataset, train the model, and use it to predict new articles.

But before we download a Kaggle dataset and get cracking on Google Cloud, it's in our best interest to pre-process the dataset.

What is pre-processing?

To preprocess your text simply means to bring your text into a form that is predictable and analyzable for your task. The goal of pre-processing is to remove noise. By removing unnecessary features from our text, we can reduce complexity and increase predictability (i.e. our model is faster and better). Removing punctuation, special characters, and 'filler' words (the, a, etc.) does not drastically change the meaning of a text.

Approach:

There are many types of text pre-processing and their approaches varied. We will cover the following:

1. Lowercase Text
2. URL Removal
3. Contraction Splitting
4. Tokenization
5. Stemming
6. Lemmatization
7. Stop Word Removal

We'll be using python due to the availability and power of its data analysis and NLP libraries.

Lowercase & URL Removal

Before we start any of the pre-processing heavy lifting, we want to convert our text to lowercase and remove any URLs in our text. A simple regex expression can handle this for us.

CODE:

```
import re

text = "http://www.google.com hello world"

text = re.sub(r'http\S+', '', text.lower())

print(text)
```

OUTPUT:

```
# hello world
```

Split Contractions

Similar to URLs, contractions can produce unintended results if left alone. The aptly named contractions python library to the rescue! It looks for contractions and splits them into root words.

CODE:

```
import contractions

def remove_contractions(text):

    return ' '.join([contractions.fix(word) for word in text.split()])

text = """can't won't shouldn't there's mustn't"""

print(remove_contractions(text))
```

OUTPUT:

```
# can not will not should not there is must not
```

Tokenization

At it's simplest, tokenization is splitting text into pieces.

Tokenization is breaking the raw text into small chunks. Tokenization breaks the raw text into words, sentences called tokens. These tokens help in understanding the context or developing the model for the NLP. The tokenization helps in interpreting the meaning of the text by analyzing the sequence of the words.

There are a multitude of ways to implement tokenization and their approaches varied. For our project we utilized RegexpTokenizer within the NLTK library. Using regular expressions, RegexpTokenizer will match either tokens or separators (i.e. include or exclude regex matches).

CODE:

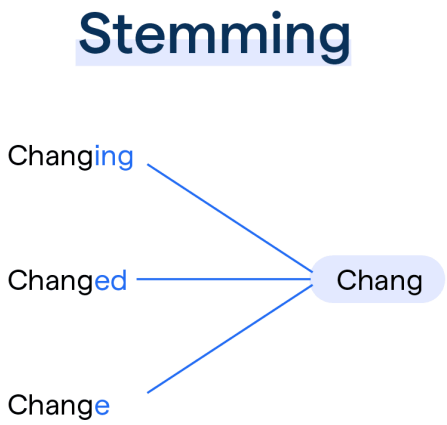
```
import nltk

from nltk.tokenize import RegexpTokenizer

# Create tokens out of alphanumeric characters

tokenizer = RegexpTokenizer(r'\w+')

tokens = tokenizer.tokenize("I think pineapple pizza is gross and not
worth $15!")

print(tokens)
```

OUTPUT:

```
# ['I', 'think', 'pineapple', 'pizza', 'is', 'gross', 'and', 'not',
'worth', '15']
```

Our string input is split by grouping alphanumeric characters. Notice the “$” and “!” characters do not appear in our tokenized list.

Stemming



Text normalization is the process of simplifying multiple variations or tenses of the same word. Stemming and lemmatization are two methods of text normalization, the former being the simpler of the two. To stem a word, we simply remove the suffix of a word to reduce it to its root.

CODE:

```
# Using Porter Stemmer implementation in nltk

from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

def stem(tokens):

    return [stemmer.stem(token) for token in tokens]
```

```
tokens = ['jumped', 'jumps', 'jumped']
print(stem(tokens))
```

OUTPUT:

```
# ['jump', 'jump', 'jump']
```

As an example, “jumping”, “jumps”, and “jumped” all are stemmed to “jump.”

Stemming is not without its faults, however. We can run into the issue of overstemming. Overstemming is when words with different meanings are stemmed to the same root — a false positive.

CODE:

```
tokens = ['universal', 'university', 'universe']
print(stem(tokens))
```

OUTPUT:

```
# ['univers', 'univers', 'univers']
```

Understemming is also a concern. See how words that should stem to the same root do not — a false negative.

CODE:

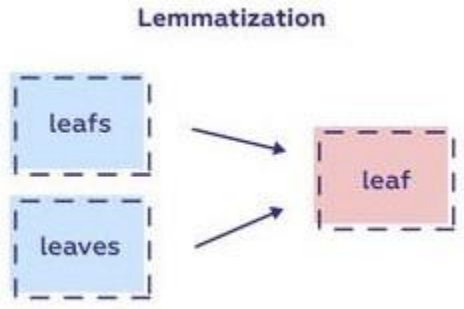
```
tokens = ['alumnus', 'alumni', 'alumnae']
print(stem(tokens))
```

OUTPUT:

```
# ['alumnu', 'alumni', 'alumna']
```

Let’s take a look at a more nuanced approach to text normalization, lemmatization.

Lemmatization



Lemmatization is the process of converting a word to its base form. The difference between stemming and lemmatization is, lemmatization considers the context and converts the word to its meaningful base form, whereas stemming just removes the last few characters, often leading to incorrect meanings and spelling errors.

Lemmatization differs from stemming in that it determines a words part of speech by looking at surrounding words for context. For this example we use nltk.pos_tag to assign parts of speech to tokens. We then pass the token and its assigned tag into WordNetLemmatizer, which decides how to lemmatize the token.

CODE:

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
lemmatizer = WordNetLemmatizer()
# Convert the nltk pos tags to tags that wordnet can recognize
def nltkToWordnet(nltk_tag):
```

```
  if nltk_tag.startswith('J'):
    return wordnet.ADJ
  elif nltk_tag.startswith('V'):
    return wordnet.VERB
  elif nltk_tag.startswith('N'):
    return wordnet.NOUN
  elif nltk_tag.startswith('R'):
    return wordnet.ADV
  else:
    return None
 # Lemmatize a list of words/tokens
def lemmatize(tokens):
  pos_tags = nltk.pos_tag(tokens)
  res_words = []
  for word, tag in pos_tags:
    tag = nltkToWordnet(tag)
    if tag is None:
      res_words.append(word)
    else:
      res_words.append(lemmatizer.lemmatize(word, tag))
  return res_words
```

Using the following text we can compare the results of our approaches to stemming and lemmatization.

CODE:

```
text = "it takes a great deal of bravery to stand up to our enemies,
but just as much to stand up to our friends"
```

```
tokens = tokenizer.tokenize(text)

# STEMMING RESULTS

print(stem(tokens))

#['it', 'take', 'a', 'great', 'deal', 'of', 'braveri', 'to', 'stand',
'up', 'to', 'our', 'enemi', 'but', 'just', 'as', 'much', 'to',
'stand', 'up', 'to', 'our', 'friend']

# LEMMATIZING RESULTS

print(lemmatize(tokens))
```

OUTPUT:

```
#['it', 'take', 'a', 'great', 'deal', 'of', 'bravery', 'to', 'stand',
'up', 'to', 'our', 'enemy', 'but', 'just', 'as', 'much', 'to',
'stand', 'up', 'to', 'our', 'friend']
```

Notice that 'enemies' was stemmed to 'enemi' but lemmatized to 'enemy'. Interestingly, 'bravery' was stemmed to 'braveri' but the lemmatizer did not change the original word. In general, lemmatization is more precise, but at the expense of complexity.

Stop Word Removal

Let's start this one off with an example. What comes to your mind when you read the following?

"quick brown fox lazy dog"

Hopefully you read that and thought of the common English pangram:

"the quick brown fox jumped over the lazy dog"

If you got it, you didn't need the missing words to know what I was referencing. "The" doesn't add any meaning to the sentence, and you had enough context that "jumped" and "over" weren't necessary. In essence, I removed the stop words.

Stop words are words in the text which do not add any meaning to the sentence and their removal will not affect the processing of text for the defined purpose. They are removed from the vocabulary to reduce noise and to reduce the dimension of the feature set.

CODE:

```
import nltk

nltk.download('words') #download list of english words

nltk.download('stopwords') #download list of stopwords

from nltk.corpus import stopwords

stopWords = stopwords.words('english')

englishWords = set(nltk.corpus.words.words())


def remove_stopWords(tokens):

    return [w for w in tokens if (w in englishWords and w not in
stopWords)]


tokens =
['the','quick','brown','fox','jumped','over','the','lazy','dog']

print(remove_stopWords(tokens))
```

OUTPUT:

```
# ['quick', 'brown', 'fox', 'lazy', 'dog']
```

Again the NLTK library comes in clutch here. We can download a set of English words and stop words and compare that against our input tokens (see tokenization). For the purpose of our fake news detector, we return tokens that are English but aren't stop words.

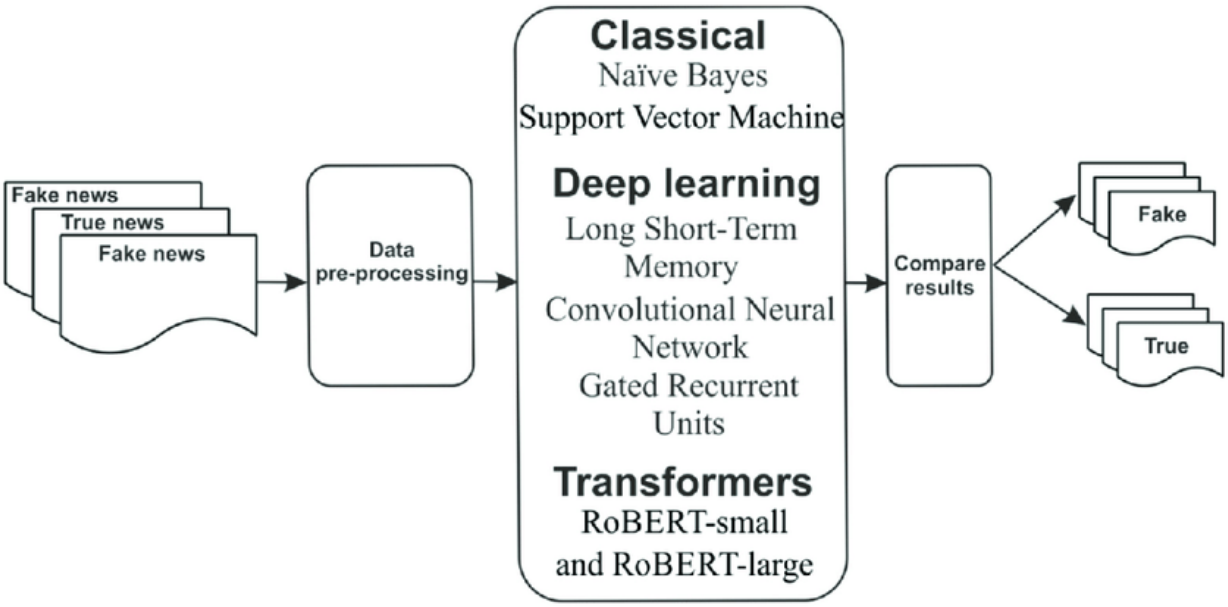
All Together Now

Combining all our steps together (minus stemming), let's compare the before and after. We'll use the following onion article as our article to pre-process. If you want to try it for yourself, check out this google collab.

LAKEWOOD, OH — Following a custom born out of cooperation and respect, local drivers reportedly pulled over to the side of the road Friday to let a pizza delivery guy through. "Gee, I hope it's nothing serious like a big, hungry party," said 48-year-old Rosanna Tuttle, who was just one of the dozens of drivers who quickly moved to the shoulder of the road after catching sight of the speeding pizza-delivery vehicle swerving through traffic in the rearview mirror. "It's honestly just a reflex. Sure, it slows everyone down, but wouldn't you want others to pull over for you if that was your pizza in there? I don't care if I'm late; I just hope that pizza is okay. Let's pray they get there safe." At press time, drivers at the scene had stopped their cars again to rubberneck as the delivery guy rushed into an apartment building carrying a large stack of pizzas and mozzarella sticks.

Returns the following:

oh follow custom bear respect local driver reportedly pull side road let pizza delivery guy gee hope nothing serious like big hungry party say year old one dozen driver quickly move shoulder road catch sight speed pizza delivery vehicle swerve traffic mirror honestly reflex sure slow everyone would want pull pizza care I late hope pizza let us pray get safe press time driver scene stop car rubberneck delivery guy rush apartment building carry large stack pizza stick


Fake news
True news
Fake news
Data pre-processing
Classical
Naïve Bayes
Support Vector Machine
Deep learning
Long Short-Term Memory
Convolutional Neural Network
Gated Recurrent Units
Transformers
RoBERT-small and RoBERT-large
Compare results
Fake
True

DATASET

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 G	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 Suicid	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
```

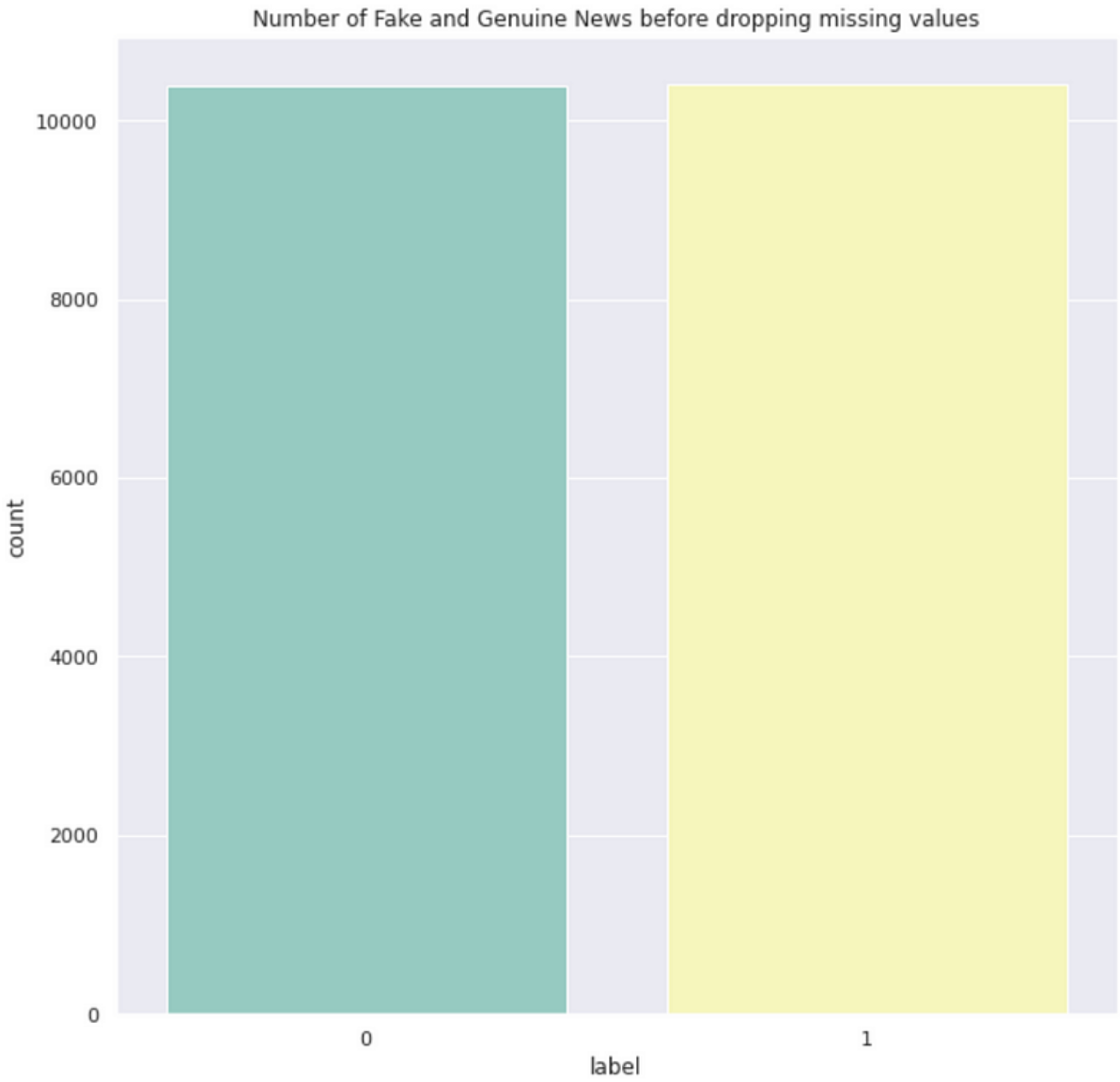
```
id           0
title      558
author    1957
text        39
label        0
dtype: int64
```

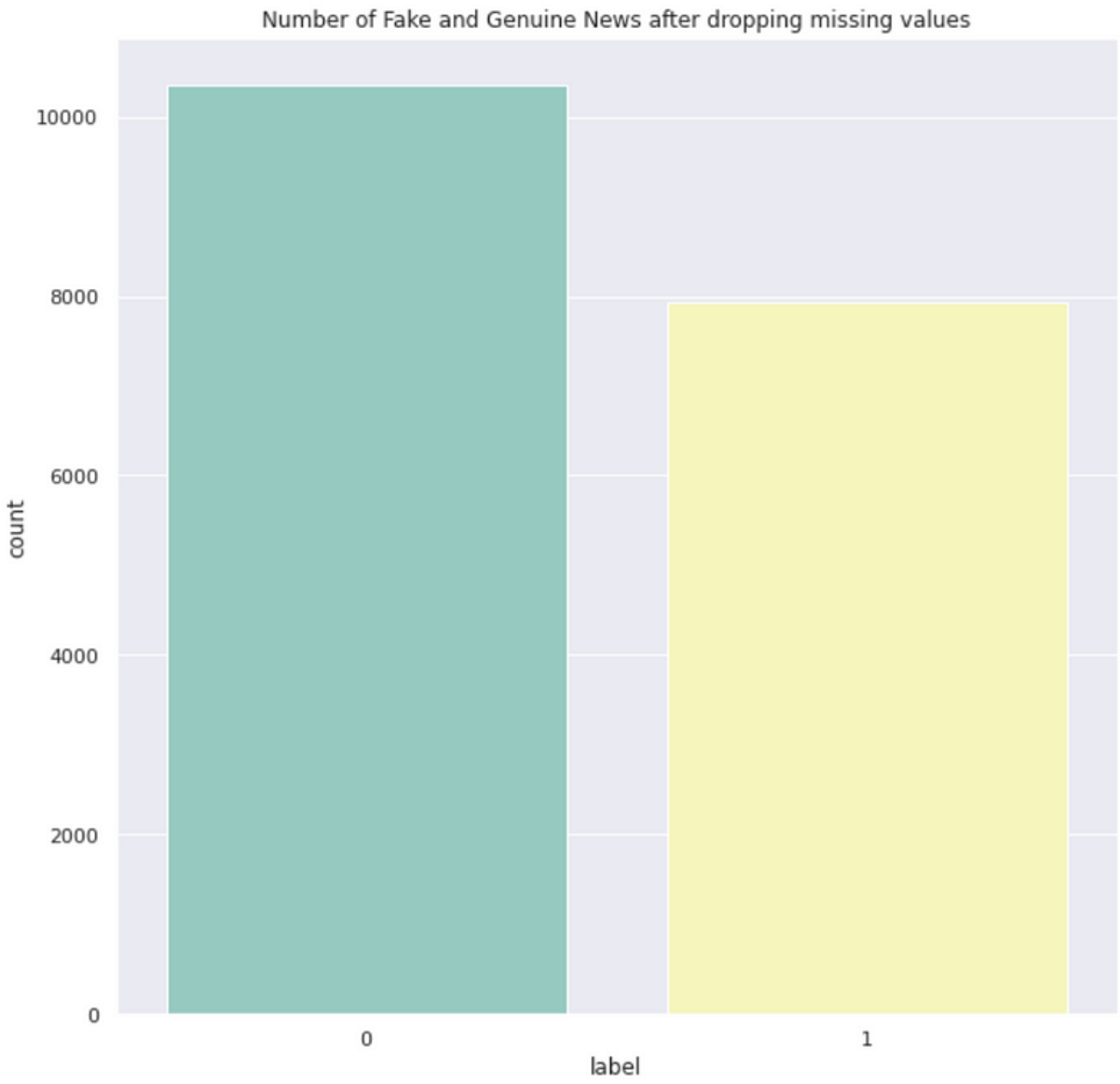
Number of Null Values in Train Data and Test Data, respectively

```
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

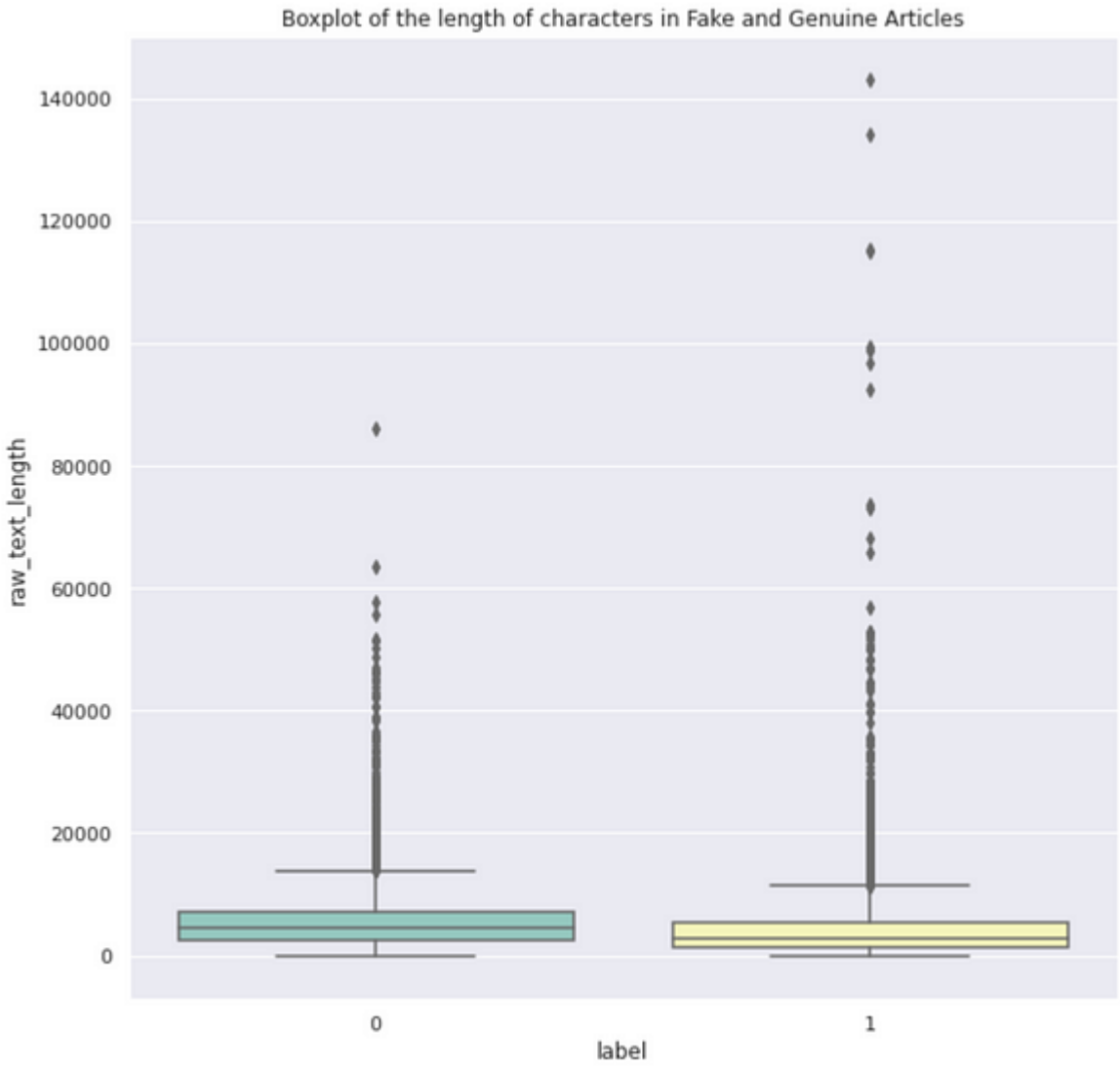


0 is Genuine News while 1 is Fake News

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()
```



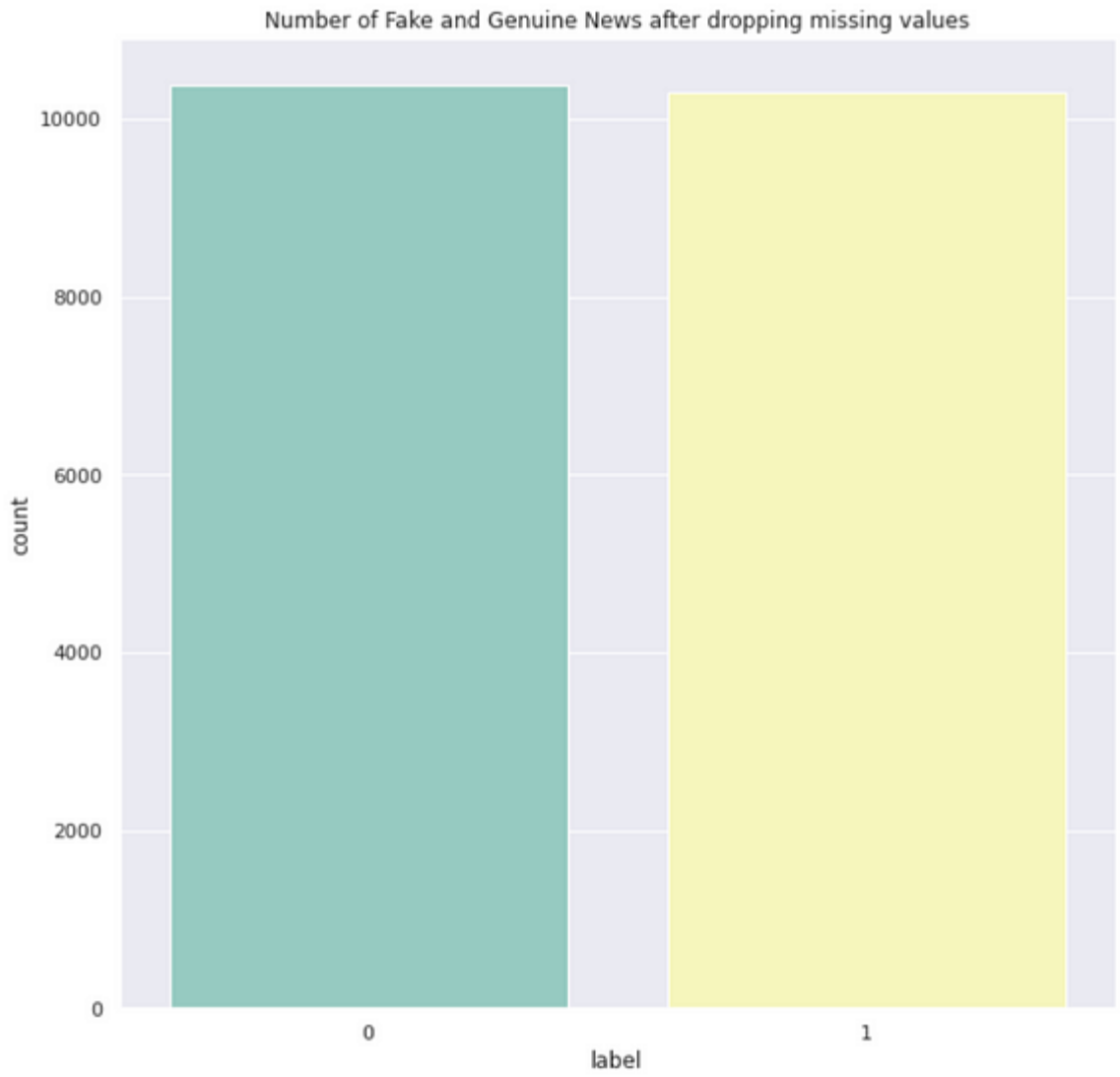
It is seen that the median length is lower for fake articles but it also has loads of outliers. Both have zero length.

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

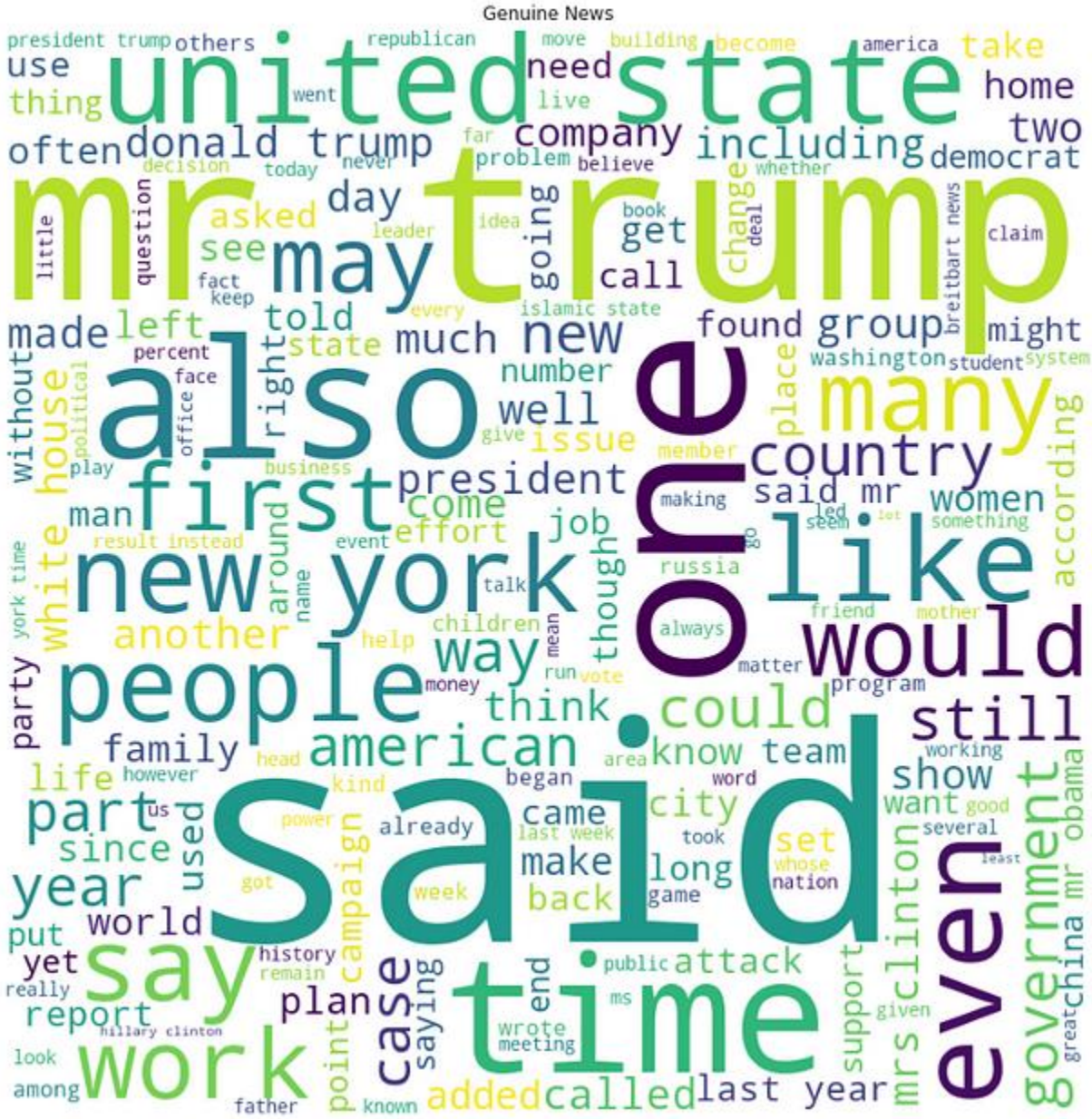
Now that the data is ready, I intend to look at frequent words using the wordcloud. In order to do that, I first joined all the tokenized texts into strings in separate columns since they will be used later while model training.

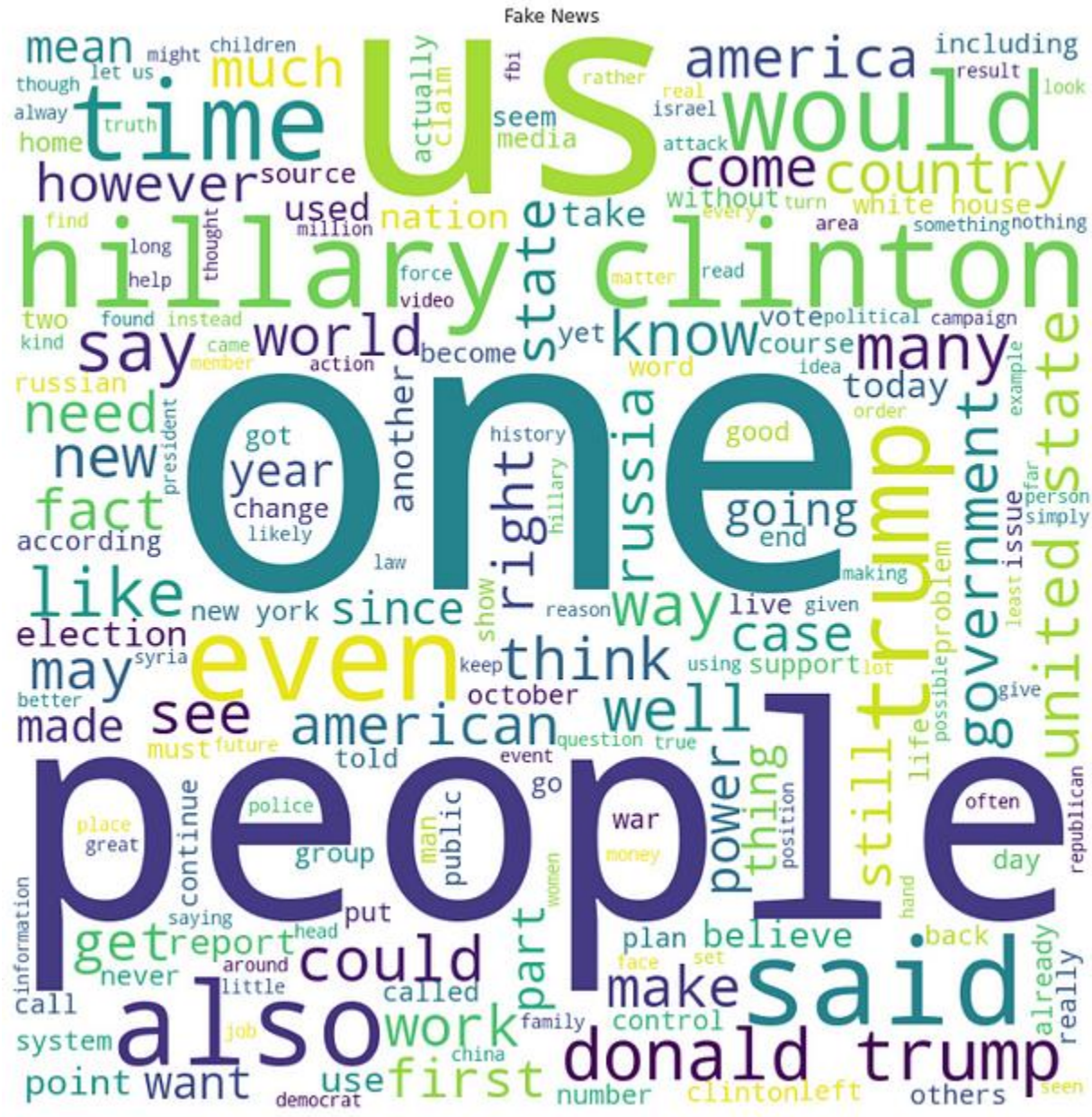
```
# since count vectorizer expects strings


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.

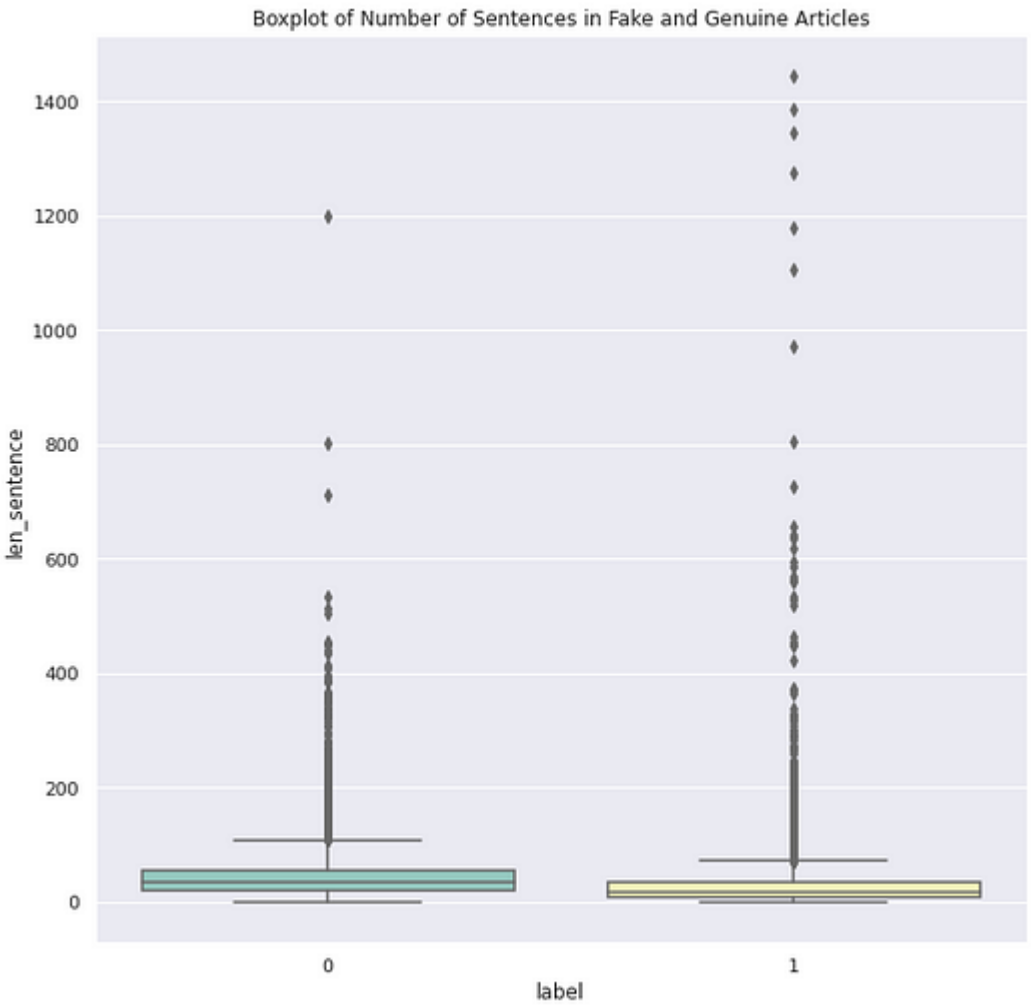
Number of Sentences per Article

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()
```

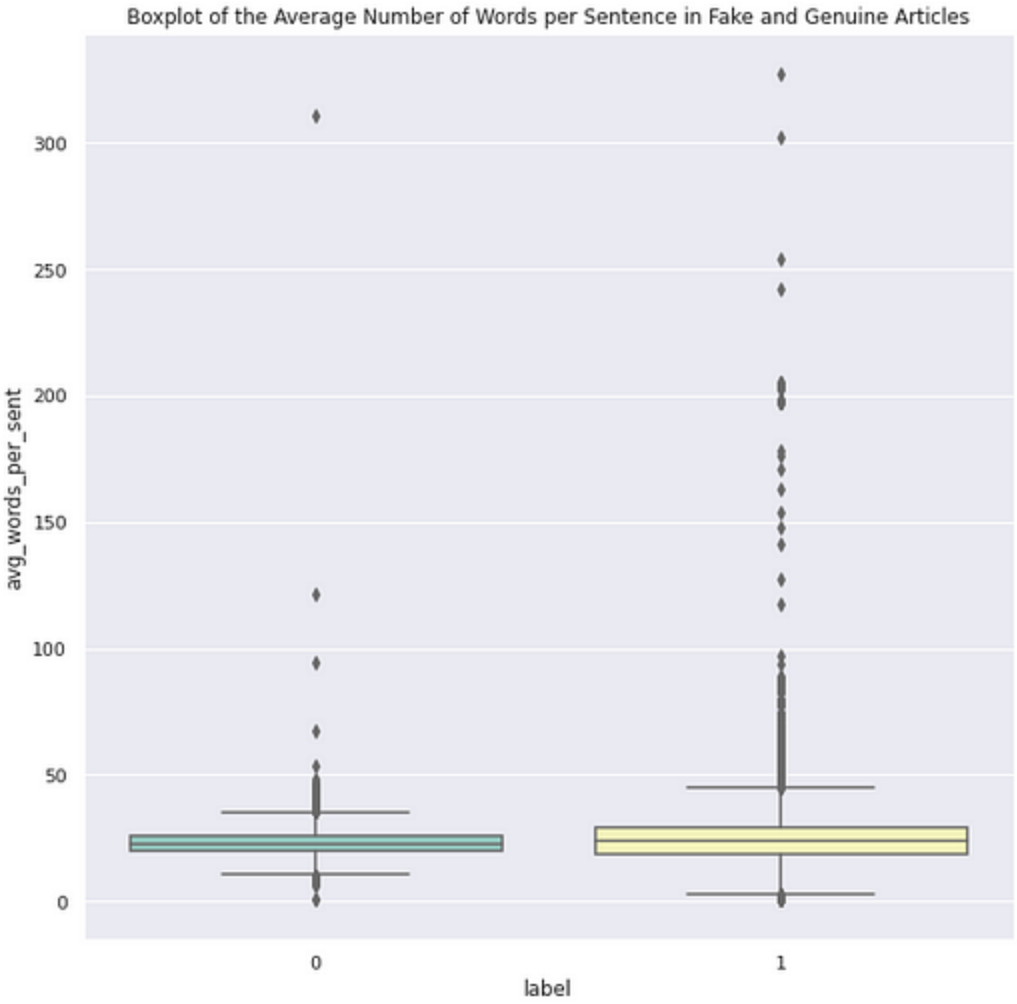


Evidently, fake articles have a lot of outliers but 75% of the fake articles have the number of sentences lower than the 50% of the genuine news articles.

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()
```

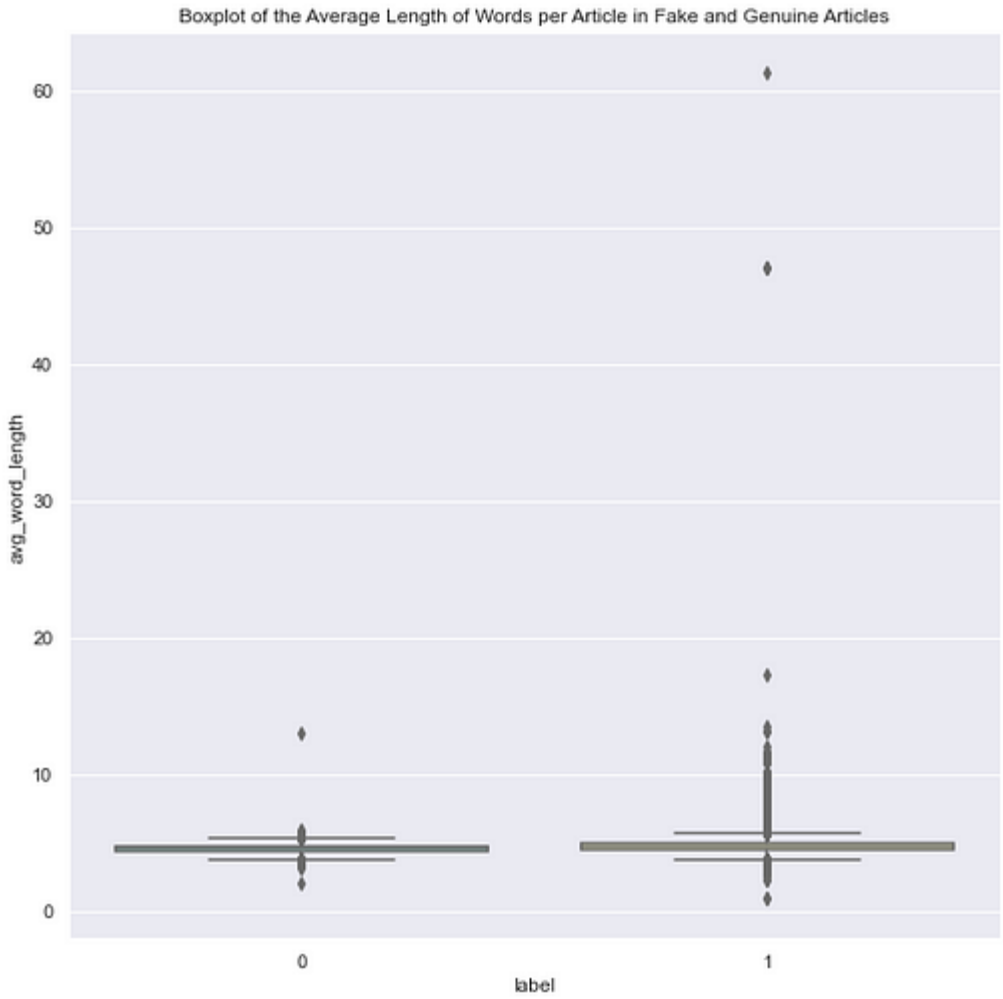


It is seen that, on average, fake articles are wordier than genuine ones.

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()
```

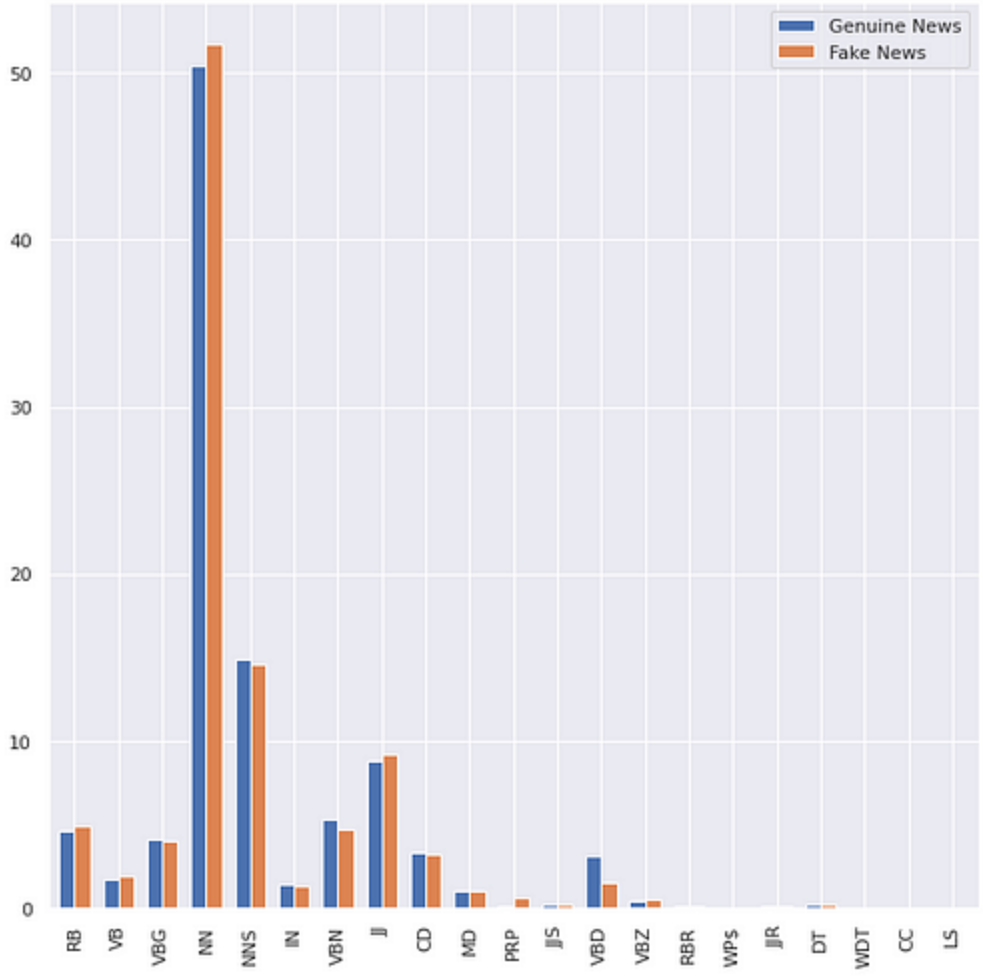


POS Tag Counts

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.

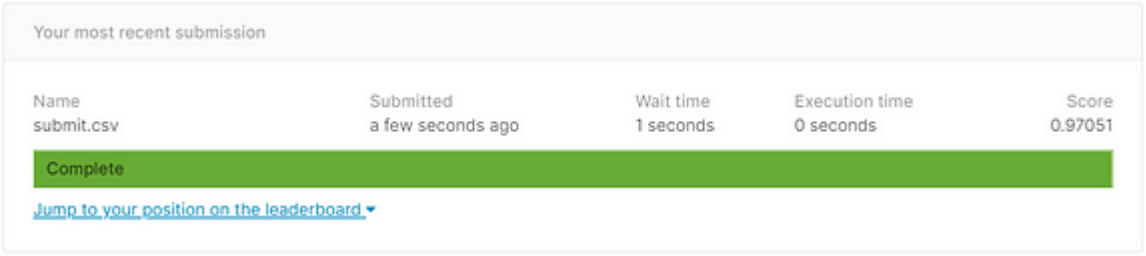
```
# Starting out with a range of values
max_iter = [100, 200, 500, 1000]
C = [0.1, 0.5, 1, 10, 50, 100]

# 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()])
```

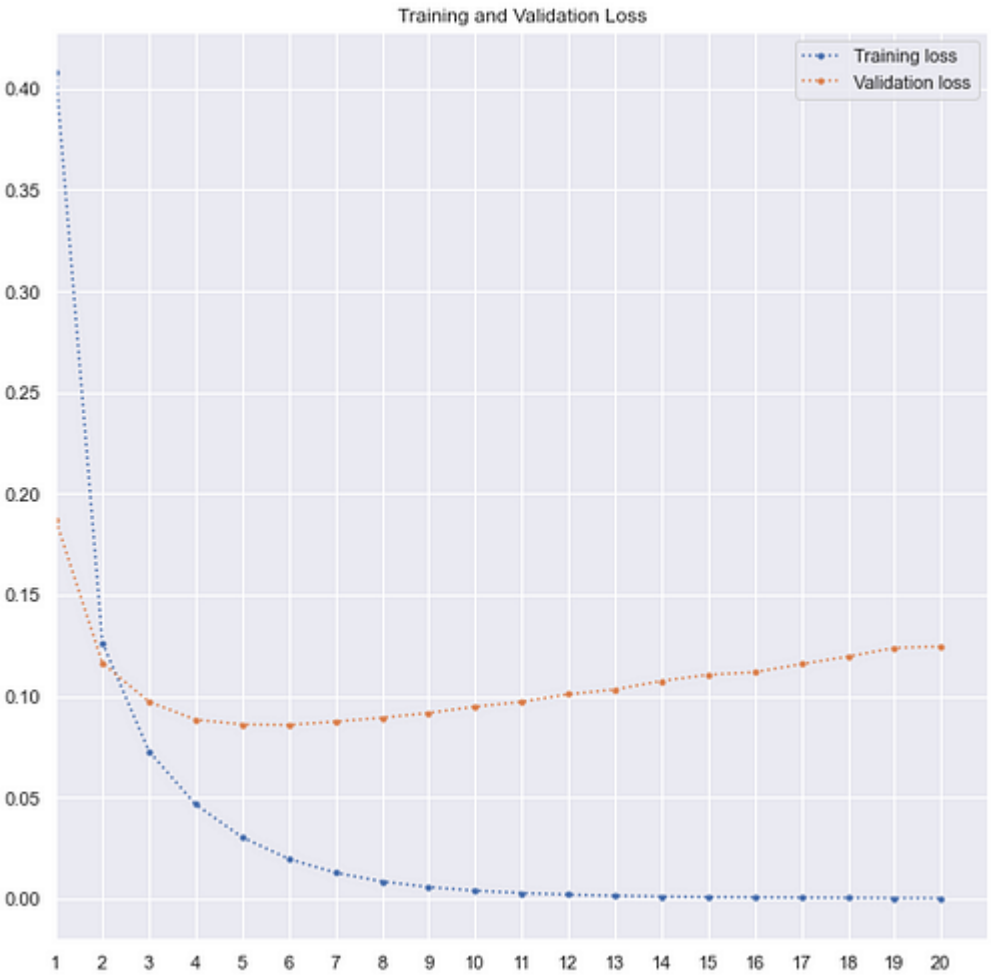
I also used early stopping to save time with patience=15 which indicates stopping if in the last 15 epochs there was no improvement in the model, and model checkpoint to store the best model with save_best_only=True. Added mode=min since I am monitoring loss here.

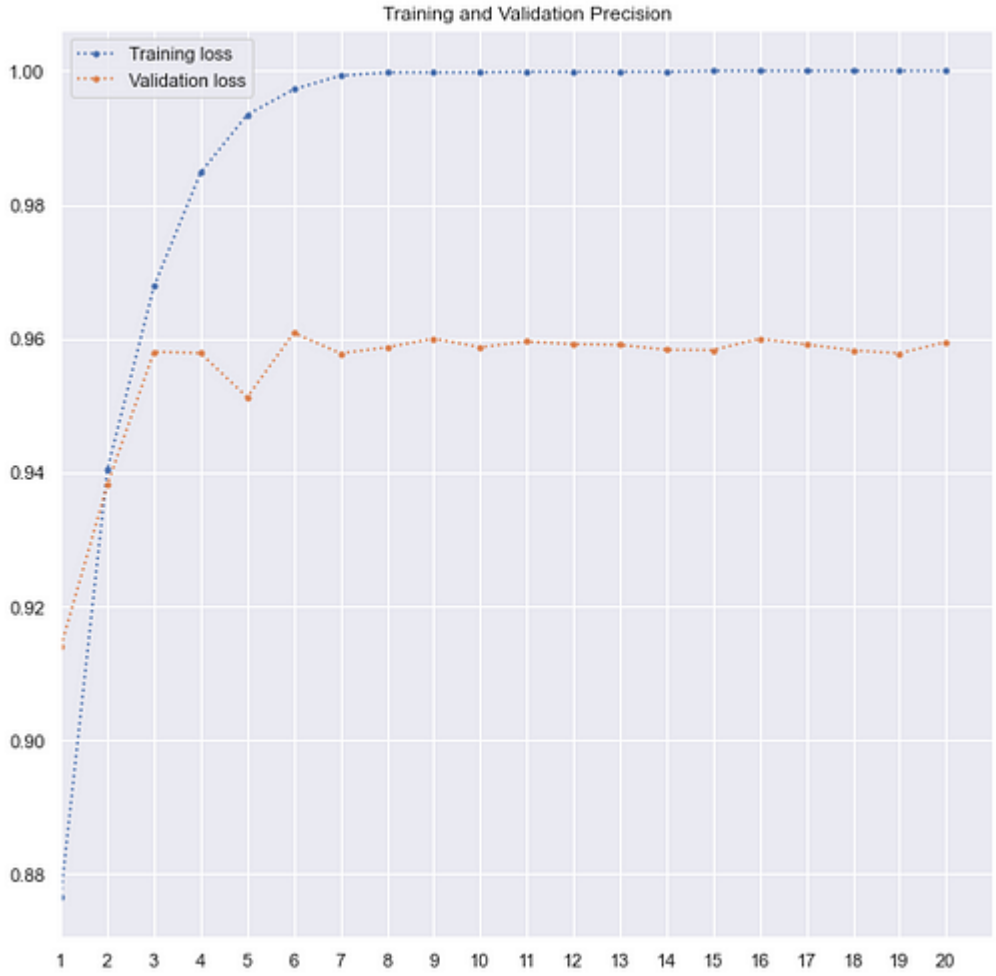
```
callbacks=[
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=15,
                                  verbose=1, mode="min",
restore_best_weights=True),
    keras.callbacks.ModelCheckpoint(filepath=best_model_file_name,
verbose=1, save_best_only=True)]
```

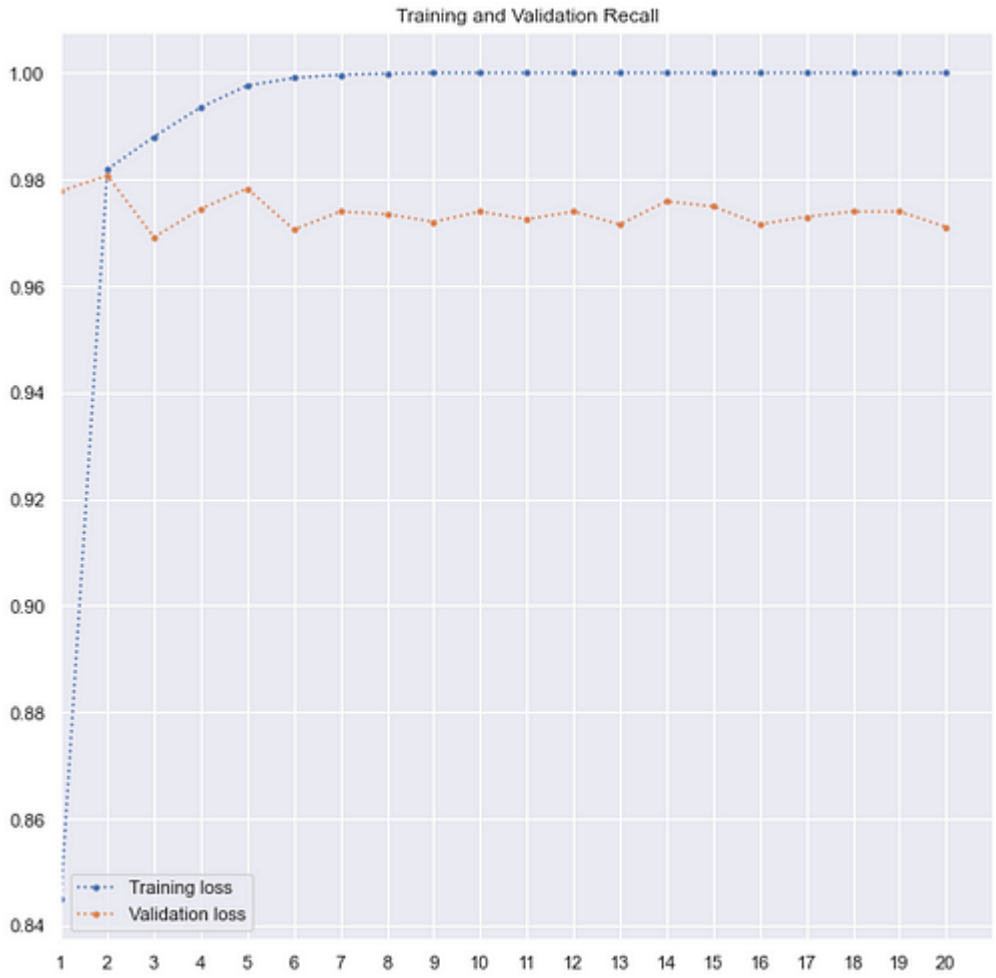
Time to fit the model now!

```
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



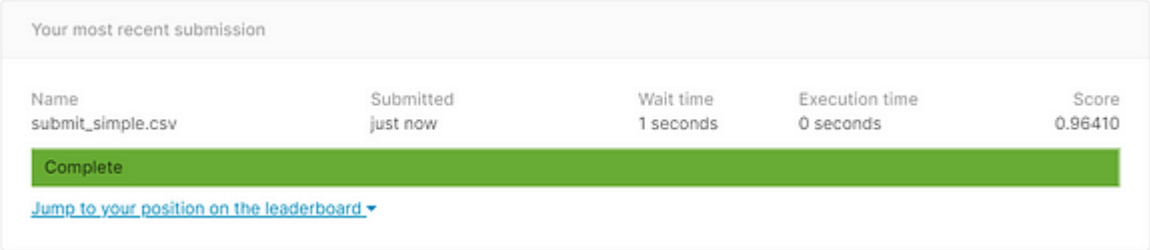
Below, is the code I used to plot the training and validation loss, precision, and recall. I used max(history.epoch) + 2 in the range function since history.epoch starts from 0. Hence, for 20 epochs, the maximum would be 19 and the range would generate a list from 1 to 18 for max(history.epoch).

```
# 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

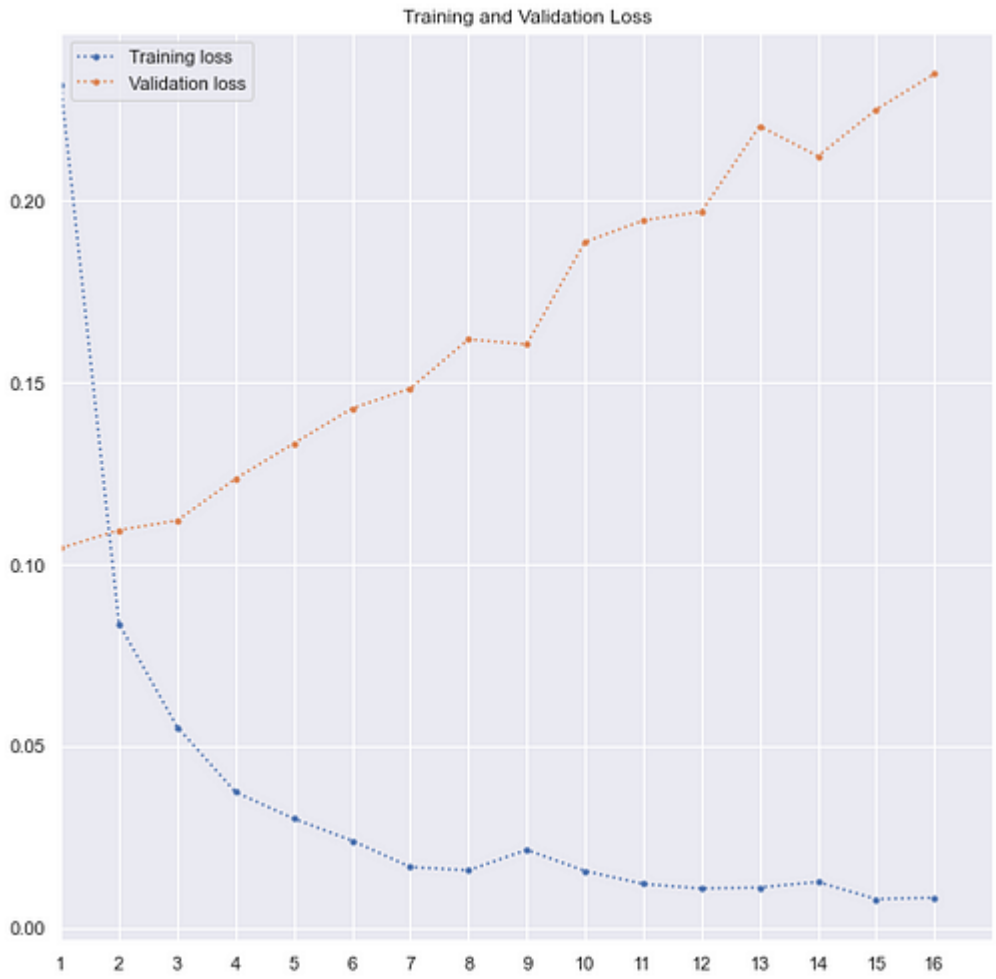
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 40, 10)            60000
_________________________________________________________________
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
```

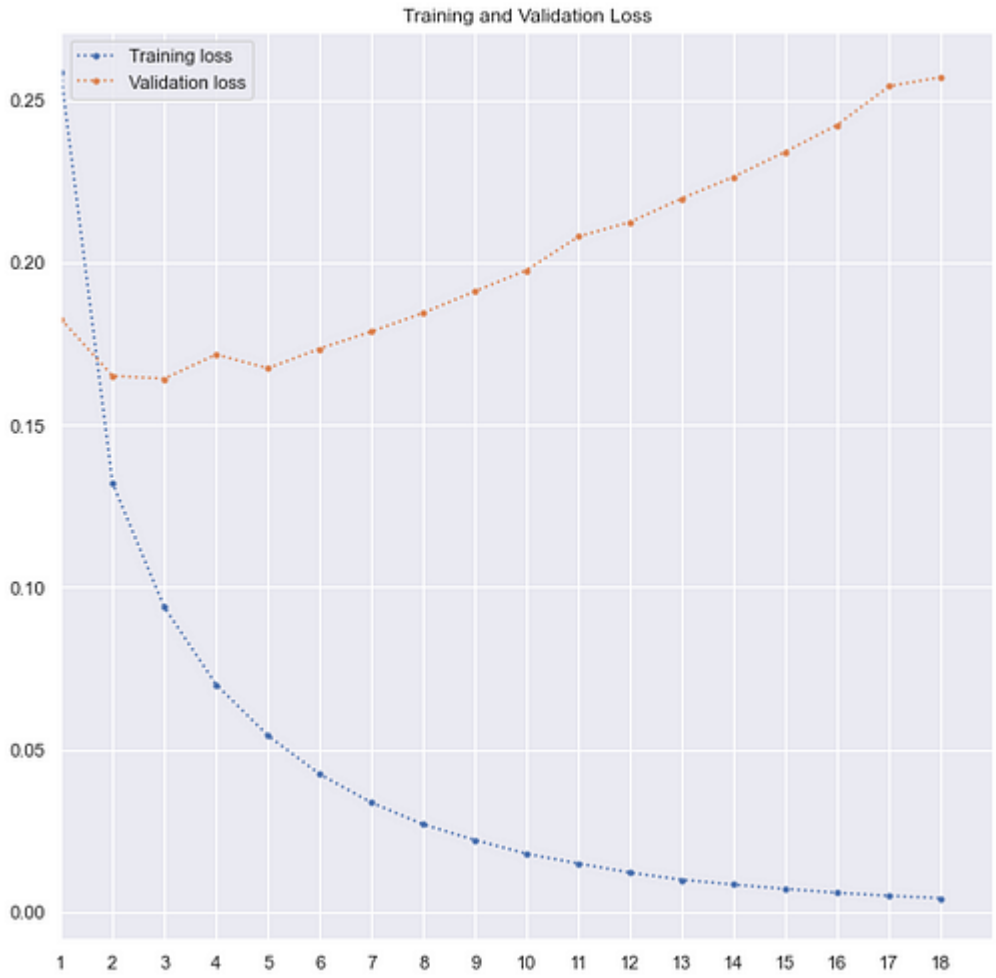
Simple Model with Glove

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

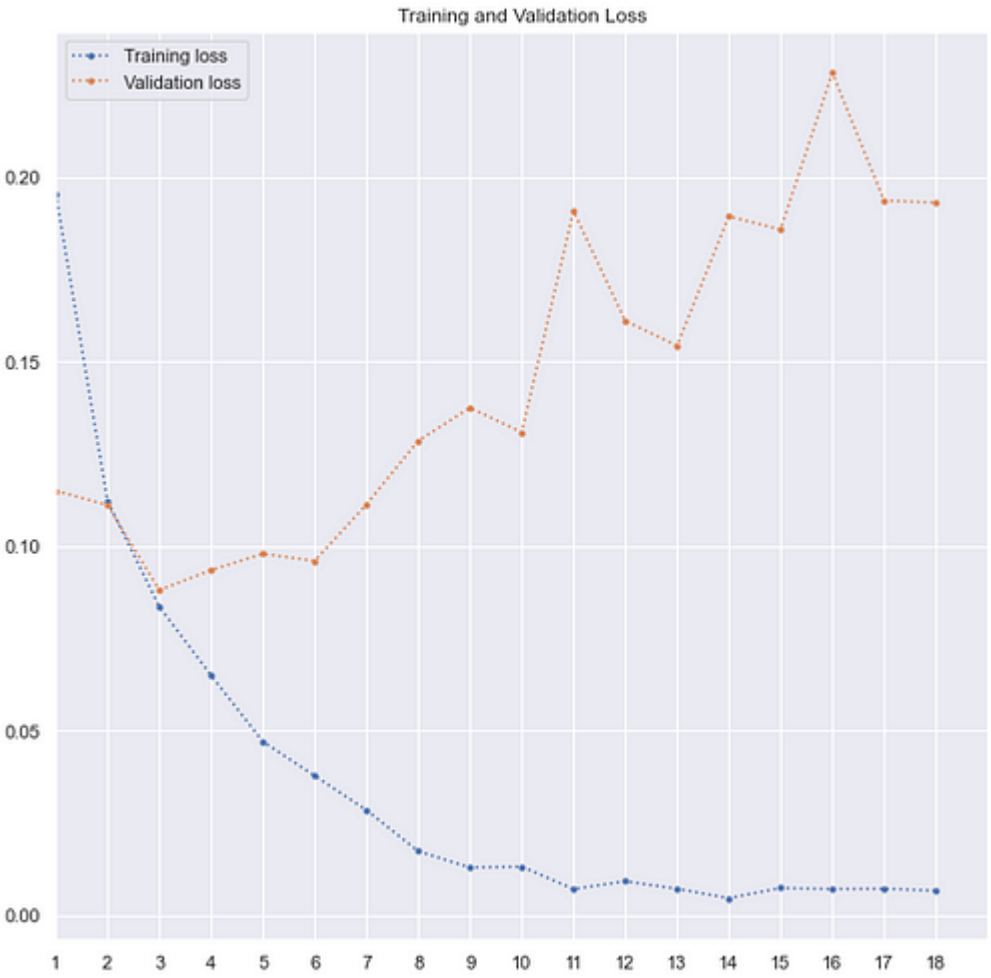
And.. finally, I used the GloVe embedding to train the LSTM model I used earlier to achieve better results. The complete code is below –

```
# 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,
```

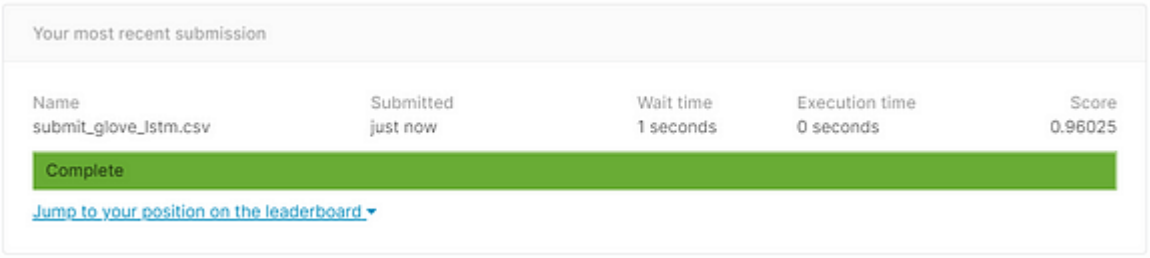
```
                        300,
                        weights=[embedding_matrix],
                        input_length=max_length,
                        trainable=False))
    model.add(LSTM(100))
    model.add(Dropout(0.3))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))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



SOURCE CODE OF THE PROJECT

```
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))

# 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()

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()

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]))

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()


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)

# 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)))

# 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))
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()])
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)
]
# 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
```

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

print(history.history.keys())
# 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()

# best_model_file_name = <the best model>

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))

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)

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))
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]

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

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(' ')
```

```
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()


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()


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()


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))

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])

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()


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))

from nltk import sent_tokenize
original_train_df = train_df.copy()
original_train_df['sent_tokens'] =
original_train_df['text'].apply(sent_tokenize)
models_list = [LogisticRegression(),
               MultinomialNB(),
```

```
                RandomForestClassifier(),
                DecisionTreeClassifier(),
                GradientBoostingClassifier(),
                AdaBoostClassifier()]

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

Conclusion

We have classified our news data using three classification models. We have analysed the performance of the models using accuracy and confusion matrix. But this is only a beginning point for the problem. There are advanced techniques like BERT, GloVe and ELMo which are popularly used in the field of NLP. If you are interested in NLP, you can work forward with these techniques.