

Fake News Detection Using NLP Techniques

Edward Montoya, Joshna Devi Vadapalli, Nghi Nguyen, Sangamithra Murugesan, Vidushi Bhati

Department of Applied Data Science, San Jose State University

DATA 255: Deep Learning

Professor: Dr. Mohammad Masum

2023, Dec 10

Table of Contents

Member Contributions	4
Abstract	5
1. Introduction	6
2. Related Work	8
3. Methodology	11
3.1 Dataset Description	11
3.1.1 Dataset Description & Merging	11
3.1.2 Data Cleaning and Data Augmentation	17
3.2 Data Preprocessing for Embeddings	23
3.2.1 DNN	23
3.2.2 Bi-LSTM	26
3.2.3 LSTM	32
3.2.4 Logistic Regression	38
3.2.5 CNN+Bi-LSTM	43
3.2.5 Meta Llama 2 7B	44
3.2 Proposed Model (Framework) Architecture	45
3.2.1 DNN	46
3.2.2 LSTM	47
3.2.3 Bi LSTM	49
3.2.4 Logistic regression	50
3.2.5 CNN+Bi-LSTM	51
3.2.6 Llama2	53
4. Experimental Setup	56
4.1 DNN	57
4.2 LSTM	58
4.3 Bi-LSTM	59
4.4 Logistic Regression	63
4.5 CNN+Bi-LSTM	64
4.6 Llama 2	65
5. Results and Analysis:	69
5.1 DNN	69
5.2 LSTM	71
5.3 Bi-LSTM	72
5.4 Logistic regression	75
5.5 CNN+Bi-LSTM	77
5.6 Llama 2	79

6. Conclusion	81
6.1.1 Project Summary	81
6.1.2 Result discussion	81
6.1.3 Future work	82
References	84

Abstract

This project addresses the critical issue of fake news detection by employing a multifaceted deep learning strategy. The escalating prevalence of misinformation in the digital age necessitates accurate discernment of genuine news from deceptive content. The project's central inquiry revolves around the effective detection and classification of fake news through the integration of advanced transformer technology, specifically the Llama-2 7B model, and a mix of active and standard deep learning models. Existing research often exhibits bias and limitations by relying on singular datasets or machine learning models in isolation. To overcome this, the project integrates diverse datasets to enhance model robustness and equity, fostering a more generalized analytical capability. The methodology combines state-of-the-art transformer technology with active learning strategies, addressing the challenge of limited labeled data. The Llama-2 7B transformer model, trained on a vast corpus, brings deep contextual understanding to discern linguistic patterns in fake news content. Active learning techniques, deployed in models like CNN+Bi-LSTM and DNN, selectively focus on informative data points, optimizing model training efficiency. Standard deep learning approaches, including Bi-LSTM, and LSTM, contribute diverse analytical perspectives accompanied by a machine learning model- Logistic Regression. Results demonstrate the Llama-2 7B model's exceptional accuracy (99%) in fake news detection. The study emphasizes the importance of embedding and hyperparameter choices in achieving a balance between accuracy and computational feasibility. Active learning models excel with minimal data points, offering cost-effective labeling solutions. Future work involves incorporating phrases in vocabulary building to enhance contextual understanding, increasing the training corpus, cross-checking data labels with fact-check applications, and optimizing models for real-time applications. The positive impact of this multifaceted approach is expected to

enhance society's resilience against misinformation, safeguard public discourse, and bolster the reliability of information sources, ultimately contributing to a more informed and discerning public.

1. Introduction

This project adopts a multifaceted deep learning strategy to tackle the increasingly critical challenge of fake news detection. In the current era, the rapid dissemination of information through social media and various online channels underscores the urgency of accurately discerning authentic news from misleading content. The ramifications of this issue are profound, touching on public security against misinformation that can incite panic, tarnish reputations, sway electoral outcomes, and undermine confidence in journalism. This project seeks to answer the question: How can we effectively detect and classify fake news using a multifaceted deep learning strategy? This strategy includes the state-of-the-art Llama-2 7B transformer model and incorporates both active and standard deep learning techniques, employing models such as CNN+Bi-LSTM, DNN, Bi-LSTM, and LSTM also ML model Logistic Regression with various embeddings.

While existing research often relies on a singular dataset or multiple datasets examined in isolation across different machine learning models, such approaches can introduce model bias, reflecting the limitations and particularities of the source data. By integrating datasets from multiple origins, this project aims to enhance the robustness and equity of the model, promoting a more generalized and unbiased analytical capability. Current methodologies in fake news detection predominantly employ conventional machine learning techniques, including Logistic Regression, ensemble methods like Random Forest, XGBoost and deep learning techniques such

as Bi-LSTM. However, recent advances have seen the adoption of transformer-based architectures such as BERT, which have demonstrated promising results.

We deploy a multifaceted approach to the pressing challenge of fake news detection, employing the latest transformer technology and innovative active learning strategies. The Llama-2 7B transformer, a state-of-the-art model developed by Meta, is a central component of our methodology. Trained on a vast corpus of one trillion tokens, the Llama-2 7B is poised to bring a deep contextual understanding to the task, enabling it to discern subtle linguistic patterns and nuances characteristic of fake news content. This model operates independently, leveraging its extensive training to offer robust fake news detection capabilities.

Parallel to deploying the Llama-2 7B, the project also explores the application of both active and standard deep learning strategies. The active learning aspect, utilized in models like CNN+Bi-LSTM and DNN, is particularly beneficial in situations with limited labeled data. It allows these models to learn from the most informative data points selectively. In the context of fake news detection, where acquiring a diverse and extensive set of labeled examples can be challenging, active learning provides an efficient pathway to model improvement. By focusing on uncertain or ambiguous samples, these models can achieve higher accuracy and better generalization with less labeled data. This strategic approach not only optimizes the training process but also enhances the overall effectiveness of the models in identifying and classifying fake news, complementing the extensive capabilities of the Llama-2 7B transformer.

Additionally, we incorporate standard deep learning approaches using Logistic Regression, Bi-LSTM, and LSTM models with various embeddings. These models, devoid of active learning techniques, contribute to the project by providing a diverse range of analytical perspectives and techniques, enriching our methodology.

In summary, this project is focused on a crucial research question: How can the integration of advanced transformer technology, such as the Llama-2 7B, combined with a mix of active and standard deep learning models effectively improve the detection and classification of fake news in a landscape fraught with information overload and limited labeled data? Our efforts aim to impact society positively by curbing the spread of misinformation, safeguarding public discourse, boosting the reliability of information, supporting journalistic integrity, and much more.

2. Related Work

Bharadwaj, et al. (2023) used the false and real news dataset, which includes more than 40,000 articles with fake and real news, to address the growing problem of fabricated news, particularly in the context of social media and the internet's widespread effect. In order to achieve the highest level of accuracy in false news identification, the authors conducted an in-depth evaluation and contrast of many different deep learning models, including Lightning Module, Logistic Regression, LSTM, Word Embedding, RNN, and Bag of Ngrams. The study concluded that RNN emerged as the more efficient model due to its ability to handle sequential input data.

Jaiswal et al. (2023) implemented RoBerta and Firefly optimization for feature extraction and selection. They used ISOT and FakeNewsNet datasets for their research. The optimized feature vector was fed into - Bi-LSTM, VGGNet, and CNN-supervised deep-learning-based models for classification. Bi-LSTM performed the best on both datasets with 75.90% accuracy and 76.77% F1-score on ISOT, and 86.30% accuracy and 87.60% F1-score on FakeNewsNet.

In the 2023 paper by T. Mahara et al. (2023), an in-depth analysis was conducted using the Fake News Healthcare dataset, which consists of 9,581 articles. The study implemented a

Decision Tree, Random Forest, Support Vector Machine, AdaBoost-Decision Tree, and AdaBoost-Random Forest, Convolutional Neural Network-Long Short-Term Memory (CNN-LSTM) and Convolutional Neural Network-Bidirectional Long Short-Term Memory (CNN-BiLSTM). The results indicated that the AdaBoost-Random Forest model achieved the highest F1 score of 98.9%, closely followed by the CNN-LSTM model, which garnered an F1 score of 97.09%.

Rana et al. (2023) investigated the efficacy of pre-trained distilled BERT model versions to separate online rumors from fake news using the Fake News Challenge dataset. To determine how effectively the distilled BERT-tiny and BERT-small models can learn the main properties necessary to differentiate between fake and authentic news, researchers examined the outcomes obtained using different methods before and after integrating two smaller pre-trained BERT models into one framework. The findings reveal that BERT-small achieved 90.11% accuracy and demonstrated that these more compact variations were equally accurate to the related research while remaining effective, compact, and simple to train.

In their 2022 paper, Mahara and Gangele (2022) conducted an extensive study utilizing a publicly available dataset from Kaggle, which comprised 37,000 instances. The researchers employed two types of recurrent neural network models for their analysis: Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (Bi-LSTM). Their empirical evaluation revealed that the Bi-LSTM model outperformed the traditional LSTM model in accuracy with a score of 94%.

The study led by Chen et al. (2021) focused on the COVID-19 fake news dataset to investigate the effectiveness of various machine learning models in identifying misinformation. The models evaluated in the study included BERT (Bidirectional Encoder Representations from

Transformers), RoBERTa (Robustly Optimized BERT), ALBERT (A Lite BERT), COVID-TWITTER-BERT (CT-BERT), and a specialized model termed Robust-COVID-Twitter-BERT (Ro-CT-BERT). Ro-CT-BERT emerged as the most effective model, achieving an accuracy rate of 99%.

Xavier et al. (2021) categorized, classified, and detected Fake News on Online Social Media (OSM) space. They identified seven types of fake news in OSM networks but worked on false connections between title and content and fabricated content. They employed the Stance model and trained Logistic Regression, Decision Tree, Random Forest, Multinomial Naive Bayes, and SVM Classifier to find false connections. Logistic Regression performed the best with 90.3% accuracy. For the fabricated content classifier, LSTM and Bi-LSTM were implemented where Bi-LSTM outperforms all other models by yielding 93.4% accuracy. The solution can be strengthened in the future by creating more machine-learning models or methods for identifying other sorts of fake news.

Shu et al. (2020) introduces a comprehensive data repository called FakeNewsNet with three features: news content which is the textual characteristics of the news, social context which is the social reactions to the news, and spatiotemporal information which provides details about when and where the articles were posted or shared (Shu et al., 2020). Support vector machine, Naïve Bayes, logistic regression, CNN, and social article fusion models (SAF, SAF/A, SAF/S) are developed to classify news based on their content and for social context evaluation, the paper uses Social article fusion model. For PolitiFact, the best model is Social Article Fusion with 0.691 accuracy score and for GossipCop, the highest accuracy score is 0.723 with CNN model (Shu et al., 2020).

In their paper, Hiramath & Deshpande (2019) proposed a fake news detection system based on classification algorithms such as Logistic regression, Support vector machine, Naïve Bayes algorithm, Random Forest algorithm, and Deep neural networks. Stemming and stop word removal were used on the News dataset as part of data pre-processing and utilized the Backpropagation algorithm to get the training data. As a result, they conclude that DNN performs better in terms of execution time and accuracy of 91% but requires more memory than other methods.

Thota et al. (2018) propose neural network architecture to predict fake news by classifying the relationship between articles' title and content into 'agree', 'disagree', 'discuss' or 'unrelated'. The project uses Fake News Challenge dataset and implements Tf-Idf Vectors with Dense Neural Network (DNN), Bag of Words (BOW) Vector with Dense Neural Network, and Pre-trained word embeddings with Neural Networks models. As a result, Tf-IDF - DNN outperforms other models with an accuracy score of 0.94 after being tuned and BOW - DNN is the second-best model (Thota et al., 2018).

3. Methodology

3.1 Dataset Description

3.1.1 Dataset Description & Merging

The data has been collected from diverse sources such as ISOT dataset (Ahmed et al., 2017a),(Ahmed et al., 2017b) and Fake News Real Dataset (GeorgeMcIntire, n.d.).

ISOT Dataset. ISOT Fake News dataset is provided by University of Victoria. There are two types of articles present in this dataset - Real news and Fake News. The Real news had been scrapped from Reuters.com and Fake News had been taken from various websites that were tagged as unreliable by a fact checking organization of USA, Politico. The Real data consists of

21417 rows with subjects such as ‘World-News’ and ‘Politics-News’. The Fake news consists of 23481 rows with subjects such as ‘Government-News’, ‘Middle-East’, ‘US News’, ‘left-News’, ‘politics’, and ‘News’. The dataset features include ‘Title’, ‘Text’, ‘Subject’, and ‘Date’.

Figure 1 shows the distribution of ISOT data with respect to its subjects. Figure 2 shows the merging of Real and Fake sets. Figure 3 shows that there was an increase in the Fake news in 2016. Figure 4 shows the Fake News distribution in the months of 2016 and 2017. It is clearly seen that in 2016 the Fake News was on the rise. This may be attributed to the US presidential elections in 2016, giving rise to Fake News.

Figure 1

ISOT Real and Fake News Merged

```
data_isot = real_isot.append(fake_isot).sample(frac=1).reset_index().drop(columns=['index'])
data_isot.shape
```

Figure 2

ISOT Data Subject Distribution With Respect to Real and Fake

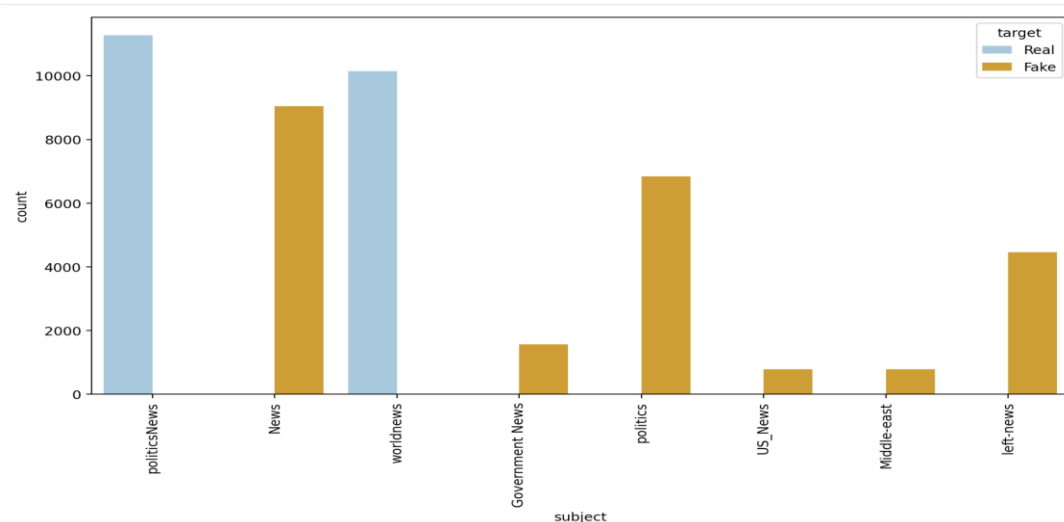
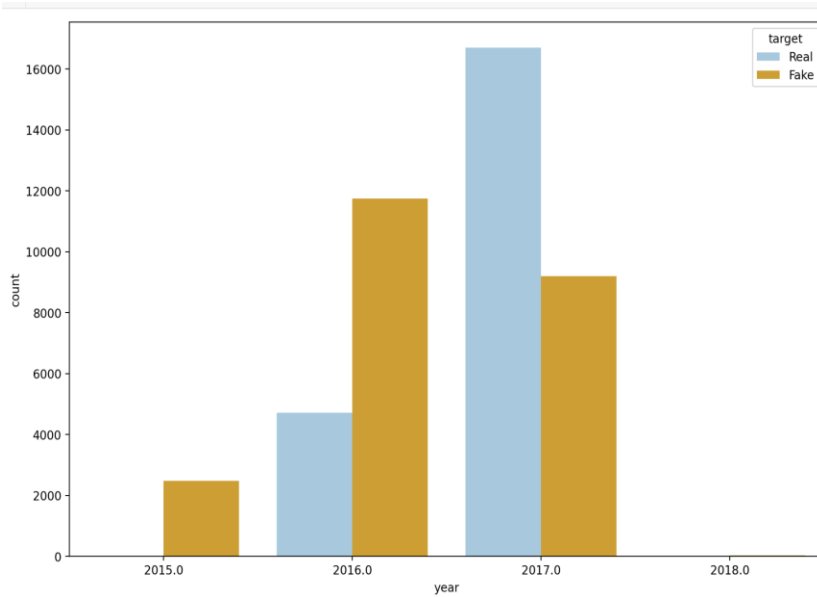
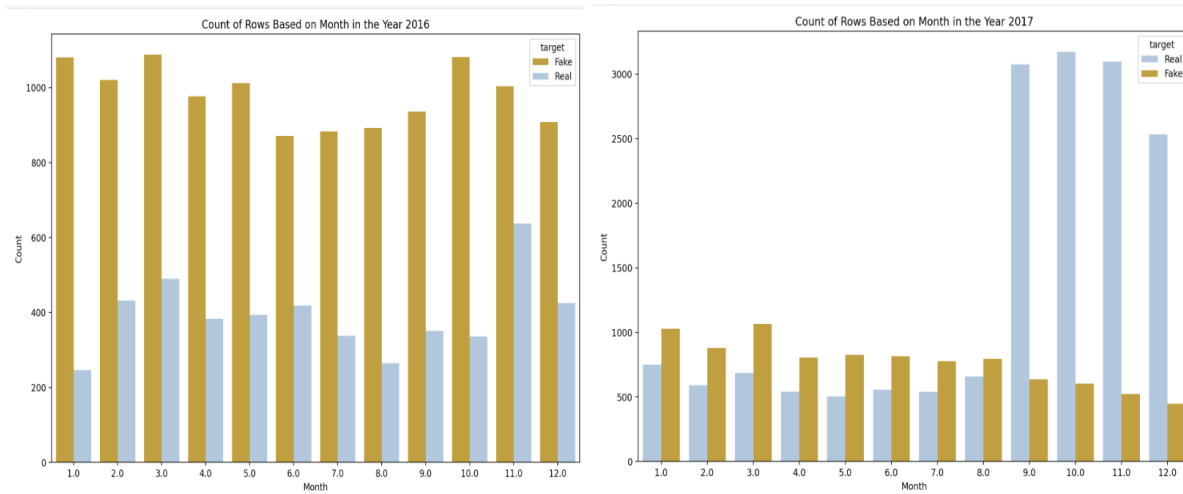


Figure 3*Fake News Increased in Year 2016***Figure 4***Comparison of Fake News(Orange) in Across all the Months of 2016 and 2017*

Redundant rows are checked for the dataset and only one row with the majority of the target label is kept. Out of 44898 rows , 6252 rows were redundant. The final ISOT data is shown in Figure 5.

Figure 5

ISOT Data After Cleaning

```
data_isot.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 38646 entries, 0 to 38645
Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype
---  ---
0    title   38646 non-null    object
1    text    38646 non-null    object
2   target  38646 non-null    object
dtypes: object(3)
memory usage: 905.9+ KB
```

Fake Real News Dataset. This dataset has been taken from the Github repository (GeorgeMcIntire, n.d.). This dataset is created as a part of an Open Data science blog post (Odsc et al., 2018). The dataset has been created using Fake News from the Kaggle competition held in 2016 and the Real news had been scrapped from various media sources such as Wall Street Journal, The New York Times, the Guardian, and the NPR from the time period 2015 to 2016. The dataset has four features - 'idd', 'title', 'text', and 'label'. Figure 6 shows the initial distribution of the dataset.

Figure 6

Fake Real News Dataset

```
getting_real.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4594 entries, 0 to 4593
Data columns (total 4 columns):
#   Column  Non-Null Count  Dtype
---  ---
0   idd     4594 non-null    object
1   title   4593 non-null    object
2   text    4594 non-null    object
3   label   4594 non-null    object
dtypes: object(4)
memory usage: 143.7+ KB
```

The irrelevant column `idd` is removed from the data and the 'label' column is changed to 'target'. 'Target' column values are renamed to Real and Fake for consistency. The number of redundant rows are checked and if there are multiple instances of the same text, then the one with the highest number of target variables is retained. Out of 4594, 185 redundant rows are found.

Figure 7 shows the Fake Real News Dataset final distribution.

Figure 7

Fake Real News Final Distribution

```
getting_real.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4409 entries, 0 to 4408
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  -
 0   title   4409 non-null    object
 1   text    4409 non-null    object
 2   target  4409 non-null    object
dtypes: object(3)
memory usage: 103.5+ KB
```

Both the datasets are merged as seen in Figure 8 and the final distribution can be seen in Figure 9 and Figure 10

Figure 8

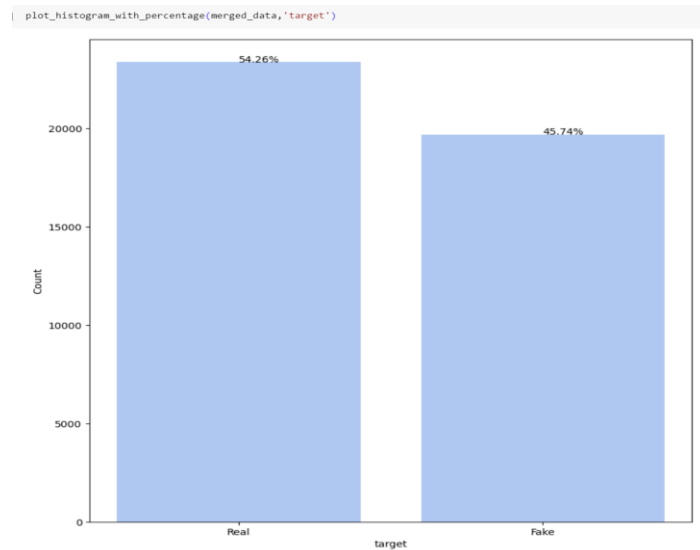
Merging of ISOT and Fake Real News Dataset

```
merged_data = pd.concat([data_isot, getting_real], axis=0)
merged_data.head(2)
```

Figure 9*Merged Dataframe*

```
merged_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 43055 entries, 0 to 4408
Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype
---  ---
0   title   43055 non-null    object
1   text    43055 non-null    object
2   target  43055 non-null    object
dtypes: object(3)
memory usage: 1.3+ MB
```

Figure 10*Merged Dataset Target Class Distribution Check*

The Figure 11 and Figure 12 shows the word cloud for Real News and Fake News. As can be seen from Figure 12, persuasive words such as ‘report’, ‘think’, ‘watch’, ‘will’, ‘now’, and ‘support’ are used for Fake News.

"https" and "www." This analysis revealed that 2,824 rows, as illustrated in Figure 13 below, contained text with URLs. These URLs were subsequently extracted from the text using Python's "re" module, which effectively matched and removed the specified patterns in each row.

Figure 13

Count of Rows Containing URLs

```
[ ] count_rows_with_urls = df_with_urls.shape[0]
    print(f"Number of rows containing URLs: {count_rows_with_urls}")
```

Number of rows containing URLs: 2824

The second step in our data preprocessing involved checking for and removing email addresses within the text. This was accomplished using the `re` module in Python, which effectively identified and eliminated email addresses. Additionally, the preprocessing phase included the removal of HTML tags, punctuations, and special characters. This was achieved using a custom-defined function, which also leveraged the `re` module. The process and its results are depicted in Figure 14 below.

Figure 14

User Defined Function for Preprocessing

```
#### Removing HTML tags,punctuations,special characters etc using regular expressions.
TAG_RE = re.compile(r'<[>]+>') # match anything in the tag <...>

def remove_tags(text):
    #print(text)
    return TAG_RE.sub('', text) # replace that tag with a null string

def process_text(sen):
    #print(sen)
    # Removing html tags
    sentence = remove_tags(sen)

    # Remove punctuations and numbers
    sentence = re.sub('[^a-zA-Z]', ' ', sentence) # look for any character except a to z or A to Z and replace with space

    # Single character removal
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence) # Look for one or more space with any a-z or a-Z letter followed by multiple space and replace with a space
    sentence = re.sub(r"\s+[a-zA-Z]\s+", ' ', sentence)

    # Removing multiple spaces
    sentence = re.sub(r'\s+', ' ', sentence) # finally check for continuous space and replace them with a single space

    return sentence
```

After completing the initial cleaning, the team proceeded with NLP-based preprocessing on the data. This involved several steps: tokenizing the 'text' column into individual words, converting all words to lowercase, removing stopwords to filter out irrelevant words, and finally, performing lemmatization. Lemmatization was chosen over stemming as it takes into account the context of a word, transforming it into its more meaningful base form. All these tasks were carried out using the NLTK library in Python, a powerful tool for natural language processing. These processes can be seen in Figure 15. Figure 16 shows before and after data processing.

Figure 15

Tokenization, Lowercase, Stopwords Removal, and Lemmatization

```
# Define a function for preprocessing
def preprocess_text(text):
    # Tokenize
    words = word_tokenize(text)

    # Lowercasing
    words = [word.lower() for word in words]

    # Stopwords removal
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(word) for word in words]

    return ' '.join(words)

# Apply the preprocessing function to the 'text' column
df['text'] = df['text'].apply(preprocess_text)
print(df)
```

Figure 16*Before and After Text Processing*

	text	cleaned_text	target
0	ANKARA (Reuters) - Turkish and Sudanese intell...	ankara reuters turkish sudanese intelligence a...	Real
1	(Reuters) - The U.S. Supreme Court on Friday r...	reuters supreme court friday refused stay ruli...	Real
2	If there s one presidential candidate who is t...	one presidential candidate king dog whistle po...	Fake
3	No wonder she didn t want anyone to see her sp...	wonder want anyone see speech singing praise t...	Fake
4	Feel the Bern of a lot of gullible young peopl...	feel bern lot gullible young people fed big li...	Fake
...
43050	WASHINGTON (Reuters) - Democratic presidential...	washington reuters democratic presidential nom...	Real
43051	VATICAN CITY/WASHINGTON (Reuters) - U.S. Democ...	vatican city washington reuters democratic pre...	Real
43052	Last week, after the New York Giants player Od...	last week new york giant player odell beckham ...	Fake
43053	ALMATY (Reuters) - Kazakhstan is to change its...	almaty reuters kazakhstan change official alph...	Real
43054	President Obama had scathing words for the law...	president obama scathing word lawmaker voted o...	Fake

43055 rows × 3 columns

The ‘target’ is then changed from Fake to 1 and Real to 0 as seen in Figure 17. Figure 18 shows the final dataset that is used for embeddings.

Figure 17*Conversion of Fake and Real to Numerals*

```
df2['target'] = df2['target'].apply(lambda x: 0 if x == 'Real' else 1)
```

Figure 18*Final Cleaned and Processed Dataset Ready for Embeddings*

	text	target
0	ankara reuters turkish sudanese intelligence a...	0
1	reuters supreme court friday refused stay ruli...	0
2	one presidential candidate king dog whistle po...	1
3	wonder want anyone see speech singing praise t...	1

Train, Validation, and Test Splits. The data needs to be split before augmentation.

Figure 19 shows the train test splits done for Data Augmentation using stratified sampling to incorporate the same ratio of target values in the splits. The training set was kept at 90% and the test set was 10%. Validation set was 10% of the 90% training set. The augmentation was done on training data only as seen in Figure 20.

Figure 19

Train Test Splits

```
# Split the dataset into train, validation, and test sets with stratified sampling
train_df, test_df = train_test_split(df, test_size=0.1, random_state=42, stratify=df['Fake_rating']) # 10% test, 90% train
train_df, val_df = train_test_split(train_df, test_size=0.1, random_state=42, stratify=train_df['Fake_rating']) # 0.1x0.9 = 0.09 (validation set)
```

Figure 20

Real and Fake Training Data

```
# Separate "Real" and "Fake" samples
real_samples = train_df[train_df['Fake_rating'] == 0]
fake_samples = train_df[train_df['Fake_rating'] == 1]
```

Data Augmentation - Oversampling. Although the dataset was fairly balanced, we employed Oversampling and Undersampling techniques to enhance the performance of the model in case the models were not performing well.

For Oversampling, only the deficit in the number of Real Samples and Fake Samples were utilized to generate samples of Fake news using Synonym Augmentation method of NLP Aug Library. This method generates the synthetic samples by replacing synonyms of the words of the original samples as seen in Figure 21. The combined dataset was then shuffled.

Figure 21*Oversampling Using SynonymAug*

```
# Calculate the difference in sample count between "Real" and "Fake" classes
sample_diff = len(real_samples) - len(fake_samples)

# If there is a deficit of "Fake" samples, augment them to match the "Real" sample count
if sample_diff > 0:

    # Oversample the "Fake" class
    oversampled_fake_samples = fake_samples.sample(n=sample_diff, replace=False)

    print(oversampled_fake_samples.shape)
    print(oversampled_fake_samples.head(2))

    # Was using BERT but it was taking a lot of time
    #aug = naw.ContextualWordEmbsAug(model_path='bert-base-uncased', action="substitute")

    # Perform text augmentation on the "Fake" samples (synonym replacement)
    aug = naw.SynonymAug(aug_src='wordnet')

    augmented_fake_samples = oversampled_fake_samples.apply(lambda x: aug.augment(x['text']), axis=1)
    augmented_fake_samples = pd.DataFrame({'text': augmented_fake_samples, 'Fake_rating': 1})

    # Concatenate the augmented "Fake" samples with the original "Fake" samples
    fake_oversamples = pd.concat([fake_samples, augmented_fake_samples])

# Combine the "Real" and "Fake" samples
augmented_df = pd.concat([real_samples, fake_oversamples])
```

Data Augmentation - Undersampling. The difference between the number of Real samples and Fake Samples were just 2973 samples. Therefore, if undersampling was used then there would be the loss of only these 2973 data points and there would not be any bias with respect to synthetic samples. So, undersampling was also tried upon as seen in Figure 22. The number of Real News sampled were equal to the number of Fake News samples.

Figure 22*Undersampling of Real News*

```
real_samples.shape, fake_samples.shape

((18922, 2), (15949, 2))

# merge the balanced data with undersampled Real Samples
df_train_undersample = pd.concat([fake_samples, real_samples.sample(n = len(fake_samples))],axis = 0)

print(df_train_undersample.shape)

# shuffle the order of training samples
df_train_undersample = df_train_undersample.sample(frac=1).reset_index(drop=True)

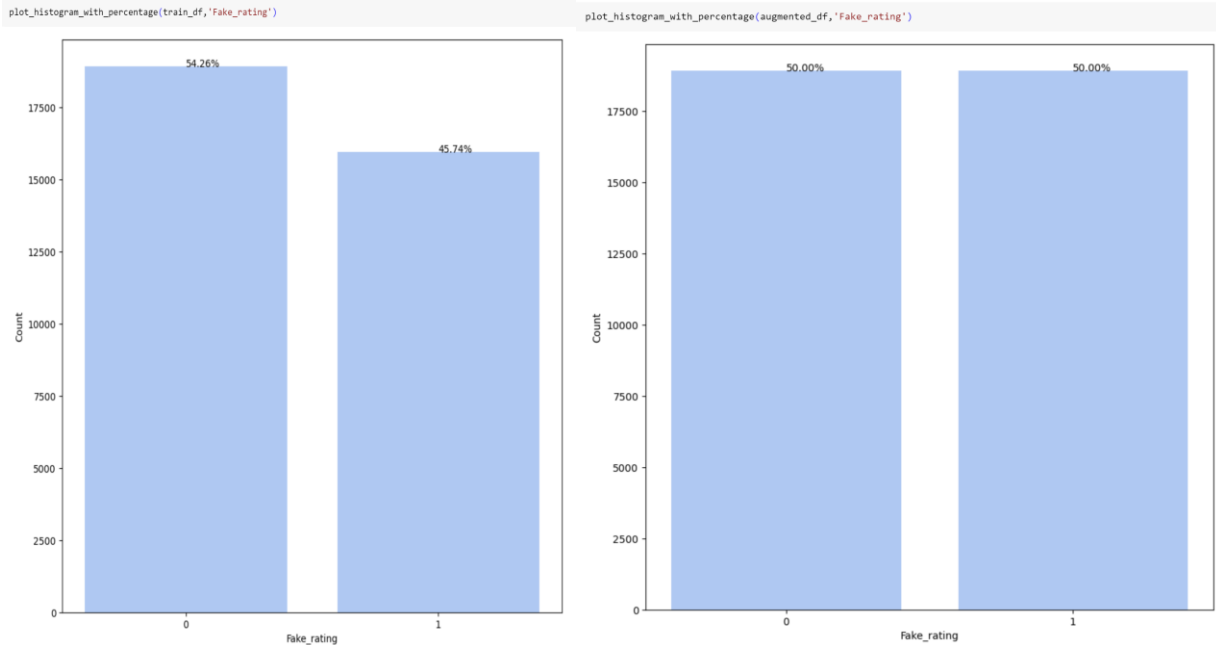
print(df_train_undersample.head(5))

(31898, 2)
      text  Fake_rating
0  SPRINGFIELD Mo Reuters President Donald Trump ...      0
1  WASHINGTON Reuters S President Barack Obama sa...      0
```

Oversampling and Undersampling both generated a dataset of exact equal target classes as shown in Figure 23. These oversampled and undersampled datasets were not used later on as the models were performing satisfactorily well without the use of these augmentation techniques.

Figure 23

Before and After Data Augmentation



3.2 Data Preprocessing for Embeddings

3.2.1 DNN

Five different embedding approaches are applied to the cleaned data to test the compatibility between the techniques and our DNN models, including Doc2Vec, Word2Vec, Glove, FastText, and BERT. Besides the article text, we extracted a length attribute that captures the length of each sentence to experiment if the feature would improve the performance.

The cleaned data was split into 70% training and 30% testing at the beginning with random state equal 42 to make sure that different embedding methods were applied on the same

training and testing datasets. Figure 24 and 25 below show how the training dataset looked after we split.

Figure 24

Training Datasets.

	text	length_info
22049	oops mistake mistake one time understandable d...	1151.0
36793	ankara reuters president tayyip erdogan dismis...	1047.0
4220	scared death prospect donald trump president u...	2192.0
2545	thank alabama protecting history good bad impo...	1063.0
1482	whiskey tango hotel worry brooke clarified lud...	2285.0
...
6265	sochi russia reuters critical decision taken s...	341.0
11284	beijing reuters china defended ally pakistan t...	1465.0
38158	sydney reuters australia prime minister malco...	777.0
860	sean spicer former communication man republica...	2747.0
15795	barack obama planning use taxpayer money trave...	2525.0

30098 rows x 2 columns

Figure 25

Training Labels.

y_train	
22049	1
36793	0
4220	1
2545	1
1482	1
...	..
6265	0
11284	0
38158	0
860	1
15795	1

Doc2Vec and Word2Vec. Doc2Vec and Word2Vec embedding processes were quite similar. We loaded functions Doc2Vec and Word2Vec from the gensim library using the Python platform and trained the models on the text portion of our cleaned data. We chose the vector setting of 300 dimensions and we didn't have to conduct padding for Doc2Vec and Word2Vec.

The word vectors of Doc2Vec and Word2Vec training data are shown below. The vector dimensions became 301 when we added the length attribute to the vector arrays. Below is the Doc2Vec vectors after embedding.

Figure 26

Doc2Vec Embedding Training Vectors.

```
array([[ -0.92420196, -0.39491269, -0.06119602, ..., -1.21163249,
         0.41028154, -0.11778655],
       [ 0.40591747, 0.28663436, 0.07339168, ..., 0.38373429,
         0.65907335, 0.85271257],
       [-0.48524573, 0.36244523, -0.89432037, ..., -0.03247582,
         0.85231566, -0.20682625],
       ...,
       [ 0.02652752, 0.05386706, -0.03928535, ..., -0.00751448,
        -0.08940426, 0.01674267],
       [ 0.13865076, 0.30280882, 1.10059142, ..., 0.01373419,
        -0.16091536, 0.08778293],
       [ 0.41091472, 1.0942167 , -0.87838149, ..., 1.32015073,
        -0.7358095 , 0.57479232]])
```

Glove. The second embedding technique is Glove. The intention was to use a high-dimensional pre-trained Glove model; however, our code crashed when running the deep learning model due to the computational limitation. As a result, we downloaded a 50—dimensional pre-trained model from Stanford University’s website. The max sequence length for padding is 512, which is the common choice for NLP projects. The training dataset’s shape is (30098, 512) without the length attribute and (30098, 513) with the length attribute.

FastText. Another embedding technique is FastText. We loaded a pre-trained model wiki-news-300d-1M.vec which included 1-million-word vectors trained on Wikipedia 2017. We conducted transfer learning to train the model on our article text data and padded the sentences with the maximum length of 512. Figure 27 shows how the training dataset looked after FastText embedding.

Figure 27*FastText Embedding Training Vectors.*

```
array([[ -1.37089844e-03, -8.12890625e-04, -1.55267578e-02, ...,
        3.41292969e-02,  1.01056641e-02, -3.67617188e-03],
       [ 1.06121094e-02, -5.02402344e-03,  4.43671875e-03, ...,
        2.76525391e-02,  2.11894531e-03, -1.15917969e-03],
       [ 7.18339844e-03, -4.78398437e-03, -5.26757812e-04, ...,
        7.42353516e-02,  6.97148437e-03, -1.79335937e-03],
       ...,
       [ 5.29355469e-03,  1.11210937e-03,  7.13085938e-04, ...,
        2.08904297e-02,  1.37519531e-03,  6.55273437e-04],
       [-1.79433594e-03, -9.07792969e-03, -3.82363281e-03, ...,
        1.02952930e-01,  1.54152344e-02, -1.02156250e-02],
       [ 1.20419922e-02,  4.29687500e-06,  1.39777344e-02, ...,
        6.65011719e-02,  8.84687500e-03, -1.84472656e-03]])
```

BERT. Last but not least, BERT is quite different compared to the other four models we applied in this project as it is designed to capture bidirectional context information by considering both the left and right context of each word in a sentence. On the other hand, Word2Vec, Glove, and FastText typically generate embeddings in a unidirectional manner, considering either the left or right context of a word. They did not explicitly capture bidirectional context. BERT embedding required padding and the extraction of input ids and attention masks. The padding length was set as 128 and the remaining was truncated. A larger number caused crashes when running the deep learning model. Figure 28 shows how the training and testing dataset looked like after preprocessing:

Figure 28*Train Dataset for BERT Embedding.*

```
X_train_ids
array([[ 101, 1051, 11923, ..., 2175, 2361, 102],
       [ 101, 20312, 26665, ..., 3038, 12478, 102],
       [ 101, 6015, 2331, ..., 4071, 4613, 102],
       ...,
       [ 101, 3994, 26665, ..., 0, 0, 0],
       [ 101, 5977, 17688, ..., 2812, 4078, 102],
       [ 101, 13857, 8112, ..., 4895, 3432, 102]])

X_train_mask
array([[1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       ...,
       [1, 1, 1, ..., 0, 0, 0],
       [1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1]])
```

3.2.2 Bi-LSTM

Two new features were added to the cleaned dataset - number of words and number of characters to evaluate if the performance of the model increases. This can be seen as in Figure 29. The strange thing that was noticed was that the number of words would be either very high or very low for Fake News. It was observed that for Fake news the number of words were outliers as seen in Figure 30. On the opposite spectrum, if the number of words would be very less then also, the Fakeness of the article increases. This can be seen in Figure 31. So, it can be noticed that both for extremely high numbers of words or very low numbers of words, one has to be cautious about the article being Fake.

Figure 29

Addition of Number of Words to the Dataset

	text	target	num_chars	num_words
	ankara reuters turkish sudanese intelligence a...	0	1028	138
	reuters supreme court friday refused stay ruli...	0	1145	154
	one presidential candidate king dog whistle po...	1	1582	210
	wonder want anyone see speech singing praise t...	1	633	89

Figure 30

Fake News has Unusually High Numbers of Words

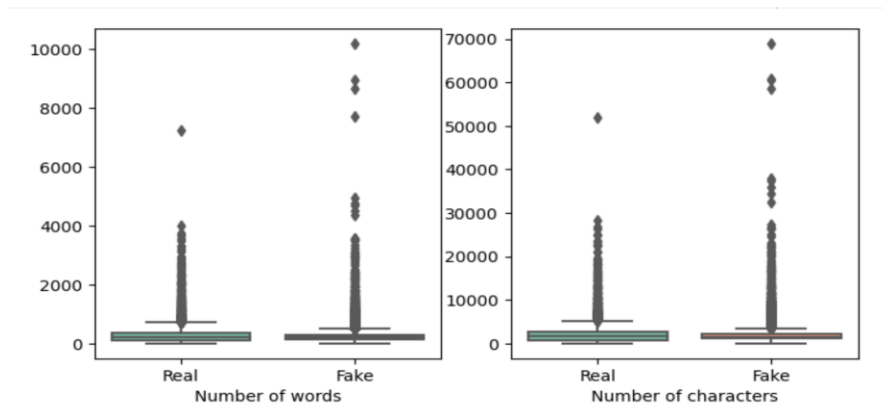
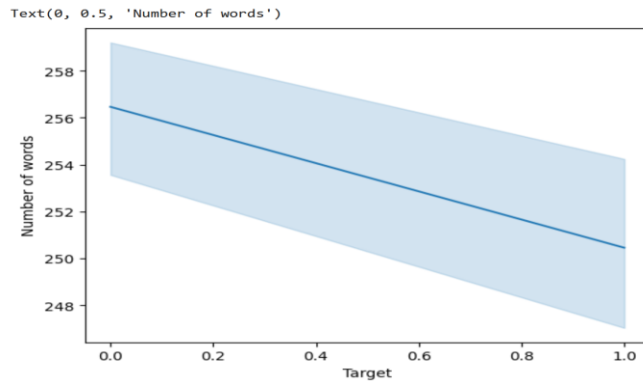


Figure 31

Fakeness Increases as the Numbers of Words Decreases



The data is split into 80% train and 20% test splits. Training data is fit on the text preprocessing Tokenizer by Keras, which builds the vocabulary of the unique words along with their index. Next 'text_to_sequence' is used to convert all the words into their corresponding vocabulary index as generated by the tokenizer. Maximum length is calculated as the maximum length in the training and test sets. Sequences are pre-padded, so that all the sequences have the same length when fed into the model. Number of words are scaled using standard scalar and converted and reshaped into numpy arrays to concatenate with the train and test data. Both samples with number of words and without number of words are maintained for feeding into the model.

Word2Vec. For embeddings, a Pre-trained Word2Vec model was used that is provided by Google via gensim.downloader api. This model is trained on Google News Dataset with 100 billion words. The model consists of 300-dimensional vectors that cover three million words and phrases. The embedding matrix of the number of vocabulary words of the training data with 300 dimensions for each is built using the pre-trained vectors and fed to the embedding layer with 300 output dimension, meaning each word will be a 300 dimensional vector. Figure 32 shows the embedding matrix built using the word2vec pretrained model.

Figure 32*Embedding Matrix Using Word2Vec Vectors*

```
embedding matrix of embedding Word2Vec : [[ 0.          0.          0.          ...  0.          0.
 0.          ]
 [-0.07910156  0.12158203 -0.00842285 ... -0.39257812  0.07763672
 0.27148438]
 [-0.00909424 -0.04418945  0.09960938 ...  0.14453125  0.18066406
 -0.08691406]
 ...
 [ 0.          0.          0.          ...  0.          0.
 0.          ]
 [ 0.          0.          0.          ...  0.          0.
 0.          ]
 [ 0.          0.          0.          ...  0.          0.
 0.          ]]
```

FastText. FastText is provided by Facebook. Words are represented by bags of character n-grams (subword units) in FastText. For instance, in addition to the entire word <care>, the character n-grams <ca, car, are, re> might be used to represent the word "care". This method works well for managing uncommon or invisible words and aids in the acquisition of morphological information. We have used Pre-trained FastText embeddings (*Wiki Word Vectors* · *fastText*, n.d.) that is trained on Wikipedia. It gives 300 dimensional vectors, trained using a skip-gram model. The same logic as Word2Vec is used to extract embedding vectors for the training data. The embedding matrix using FastText embedding vectors is seen in Figure 33

Figure 33*Embedding Matrix sing Fasttext Vectors*

```
embedding matrix of embedding FastText : [[ 0.          0.          0.          ...  0.          0.
 0.          ]
 [-0.24036215 -0.22908141 -0.31712487 ...  0.28751108 -0.0689316
 0.04915212]
 [-0.14996129 -0.18912345 -0.31939328 ...  0.00385566  0.13015246
 -0.04950039]
 ...
 [ 0.01499472  0.07942831 -0.1759029 ... -0.03000036  0.1012929
 0.30486816]
 [ 0.43211842 -0.69099122  0.0371074 ...  0.38718542  0.00426797
 -0.41213489]
 [-0.50899577 -0.39953855 -0.13307911 ... -0.16635975  0.18100037
 0.1541521 ]]
```

Glove. Glove Vector embeddings are provided by Stanford. It is a technique for unsupervised learning that generates vector representations of words. The training process is

carried out using combined global word-word co-occurrence metrics from a corpus. The outputs demonstrate intriguing linear substructures inside the word vector space (Pennington, n.d.). The team has used vectors of dimension 300, generated from a pre-trained model on six billion tokens and 400K vocabulary words. The logic used is similar to the Word2Vec and FastText embeddings for creating embedding matrices using these pre-trained vectors according to our train data vocabulary. Figure 34 shows the embedding matrix by Glove.

Figure 34

Embedding Matrix Using Glove Vectors

```
embedding matrix of embedding GloVe : [[ 0.          0.          0.          ...  0.          0.
 0.          ]
 [ 0.049177    0.056631    0.36761999 ... -0.41918999  0.37843001
 0.41108      ]
 [-0.24135     0.15132     0.016839    ...  0.44316     -0.93458998
 0.40801999]
 ...
 [ 0.53665     -0.27088001 -0.52245998 ...  0.35896999  0.34038001
 0.057998     ]
 [ 0.          0.          0.          ...  0.          0.
 0.          ]
 [ 0.          0.          0.          ...  0.          0.
 0.          ]]
```

Doc2Vec. Doc2Vec embeddings for the document are generated using the dataset.

Doc2Vec generates vectors for the document instead of words. Each document in the text column is split into words, and a TaggedDocument object is created for each document. Each document is identified using a unique tag. An instance of the Doc2Vec model is created with specific parameters, including the vector_size (representing the dimensionality of the document vectors), window (indicating the maximum context distance between current and predicted words), min_count (setting the threshold for word frequency), workers (specifying the number of CPU cores for training), and epochs (determining the number of iterations over the entire dataset during training). Subsequently, the model's vocabulary is built using the build_vocab method, preparing it for training on the provided data. The train method is then invoked to train the model

on the tagged_data, with total_examples and epoch parameters specifying the corpus size and the number of training epochs, respectively. Embeddings for each document are then inferred such as embedding for Document with index 0 can be seen in Figure 35. These are further used to feed in the models.

Figure 35

Doc2Vec Embedding for the First Document

```
array([ 0.48105335,  0.20427379, -0.5676249 , -0.5674991 ,  0.07129045,
        -0.53334564, -0.05005675, -0.10610711, -0.22882861,  0.42358488,
        -0.677329 ,  0.22499332,  0.0147077 ,  0.1174981 , -0.3165813 ,
        -0.5843365 ,  0.73238456, -0.04254897,  0.3010163 , -0.1053777 ,
        -0.29642573,  0.01444227,  0.56794864, -0.5145634 ,  0.31312847,
        -0.63974965, -1.0846188 , -0.16577165, -0.06669813,  0.14207442,
        0.0276394 ,  0.00331636,  0.67168814,  0.67173195, -0.08996778,
        -0.577362 , -0.8251031 ,  0.14834487,  0.2702316 , -0.19959986,
        -0.15070975,  0.5197014 ,  0.2043997 , -0.1322713 ,  0.09291244,
        -0.6780481 , -0.17847456,  0.14414515, -0.3695894 , -0.21372855,
        -0.40953597, -0.1485551 , -0.16917096,  0.49597225, -0.3213066 ,
        0.17127079, -0.44175535, -0.4204545 , -0.33650056,  1.0790533 ,
        0.85572296, -0.5494825 , -0.34189636,  0.33708885,  0.19414128,
        0.49685958, -0.3710284 ,  0.34849742,  0.01205301,  0.53447336,
        -0.33408943, -0.67779577, -0.06436718,  0.308493 ,  0.2927857 ,
        0.6654432 ,  0.12054957, -0.53868276, -0.60830617,  0.26627102,
        0.23001315, -0.05853191, -0.32366833,  0.68686944, -0.34767514,
        -0.00281852,  0.737455 ,  0.9540949 ,  0.22222348, -0.14281493,
        -0.4308951 , -0.04817377, -0.5726045 , -0.828794 ,  0.77660084,
        -0.22299644,  0.37123674, -0.65138763,  0.09807827,  0.7158876 ],
      dtype=float32)
```

BERT. BERT is a self-supervised 12-layer deep neural network model that has been taught to comprehend language. BERT architecture provides both word-level and sentence-level representations after training. BookCorpus, a dataset comprising 11,038 unpublished books and English Wikipedia (excluding lists, tables, and headings) was used to pretrain the BERT model. Pre-trained BERT uncased model with 110 million parameters is used via AutoTokenizer from the transformers library. Based on BERT's specifications, the tokenizer function tokenizes, truncates, and pads the input texts of Real/Fake dataset. This results in the input encodings that point to the indices in the BERT vocabulary and attention masks that tell which tokens to give attention to and which ones to ignore. The encodings can be seen in Figure 36. Maximum length

is kept at 128 for faster computation. Figure 36 shows the input ids of Bert vocabulary with each row restricted to 128 length and in total of 34398 rows. Figure 37 shows the corresponding attention masks. When input ids with attention masks are fed into the input layer of Bert Model, it gives for a sequence of length 128, with each token represented by a 768 dimensional vector.

Figure 36

Input Encodings for Indexing Vocabulary

```
train_input_ids

<tf.Tensor: shape=(34398, 128), dtype=int32, numpy=
array([[ 101,  2047,  2259, ...,  3433,  9870,  102],
       [ 101,  5396, 13307, ...,  7207,  8103,  102],
       [ 101,  4924, 26665, ...,  6313,  3466,  102],
       ...,
       [ 101,  3994, 26665, ...,    0,    0,    0],
       [ 101,  5977, 17688, ...,  2812,  4078,  102],
       [ 101, 13857,  8112, ...,  4895,  3432,  102]], dtype=int32)>
```

Figure 37

Attention Mask for Which Tokens to Focus on

```
train_attention_mask

<tf.Tensor: shape=(34398, 128), dtype=int32, numpy=
array([[1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       ...,
       [1, 1, 1, ..., 0, 0, 0],
       [1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1]], dtype=int32)>
```

3.2.3 LSTM

Word embeddings are numerical representations of words in multiple dimensions that capture the syntactic and semantic meanings of words according to the context in which they appear in a corpus. A range of word embedding methods including Doc2Vec, Word2Vec, Glove,

FastText, and BERT was chosen to preprocess the data for evaluation with LSTM (Long Short-Term Memory) models.

Doc2Vec.For word embedding, we utilized the Doc2Vec algorithm. With Doc2Vec, a Word2Vec model is extended to learn how to represent every document in a corpus as a vector. This method works especially well for tasks involving natural language processing where it is important to have context provided by the entire document rather than just specific words.

In order to implement Doc2Vec, certain parameters must be set for it to define the model's capabilities and behavior. The Doc2Vec model was started with a vector size of 20 (vec_size = 20). This implies that every document in the dataset is represented as a 20-dimensional vector, striking a compromise between computational efficiency and semantic information granularity. The model's initial learning rate is determined by the alpha parameter, which is set at 0.025. This parameter is essential for the training process to converge. The learning rate (alpha) is progressively reduced by 0.0002 after each epoch until it reaches the minimum value specified by min_alpha (0.00025), which prevents overfitting and guarantees a robust learning trajectory. We are using the 'distributed memory' version of Doc2Vec, as indicated by the dm parameter. The training process iterates over the dataset for a total of 10 epochs (max_epochs = 10), allowing the model to refine its understanding of the document semantics incrementally m=1 parameter. The sample of the embedded data is as shown in Figure 38.

Figure 38

Sample of the Embedded Data of Doc2Vec

```
Vector for document 0:
[ 0.12429012  2.0400977 -0.18884096 -6.0575266  1.3443866  6.0986567
 -2.478748   1.3457749 -0.35385522  3.9434366  1.0950987  5.839357
 2.378692   6.6034727  4.747325   0.4930044 -1.4477793 -1.7025168
 -1.4777813 -1.640216 ]
```


Word2Vec. The Word2Vec embedding technique was utilized in the code provided to convert unprocessed text data into numerical representations. The pre-trained model utilized for this purpose was the Google News vectors. A well-liked and effective technique for learning vector representations of words in a high-dimensional space is called Word2Vec. It maps terms with similar meanings near to each other in the vector space by capturing the contextual relationships between words in big corpora.

The 3 million words and phrases in the Google News Word2Vec model that we used comprise 300-dimensional vectors. With this model, every word in our text data is transformed into a 300-dimensional vector. A zero vector is used to indicate words from our dataset that are absent from the Google News lexicon. Each text sample is converted into a series of 300-dimensional vectors as a result of this process. We set a maximum sequence length ($\text{maxlen} = 100$) in order to handle texts of different lengths and to standardize the input size for the next LSTM layers. To fit this predetermined length, each text sequence is either padded with zero vectors or truncated. Figure 39 shows the sample of embedded data that will be used as the input to the LSTM model.

Figure 39

Sample of the Embedded Data of Word2Vec

```
Sample training vector at index 0 :
[[ 0.0112915  0.02893066  0.08349609 ...  0.08154297 -0.04589844
 -0.04638672]
 [-0.5625    0.10107422 -0.02648926 ...  0.03393555 -0.31445312
 -0.09277344]
 [-0.05200195 0.06176758 -0.13671875 ... -0.0703125  -0.17285156
  0.10546875]
 ...
 [-0.06591797 0.2734375  -0.08740234 ... -0.03295898  0.16894531
  0.06347656]
 [-0.08349609 0.01782227  0.04492188 ... -0.09179688  0.15917969
  0.01904297]
 [-0.15917969 0.29492188 -0.08203125 ... -0.22070312 -0.20019531
  0.12597656]]
```

FastText. Our natural language processing project's data preprocessing stage involved using the Gensim library's FastText algorithm to convert unprocessed text data into a numerical format that captures semantic meaning. Using NLTK's 'punkt' tokenizer, the corpus was first tokenized into words, making sure that every word was changed to lowercase to preserve consistency. FastText model is set up with a vector size of 50 (vector_size=50). With this configuration, every word in the dataset is represented by a 50-dimensional vector that encompasses different facets of the semantic and syntactic properties of the word. In other words, during training, FastText looks at three words before and after the target word to learn its representation. The window parameter is set to 3, which determines the context size for predicting the target word. Since the min_count parameter is set to 1, all words in the dataset—even those that appear only once—are included in the model. With the workers=4 setting, four cores can be used for parallel processing, which speeds up model training. Last but not least, choosing sg=1 results in the skip-gram architecture, where the model infers context words from the target words. For larger datasets, this option usually produces word vectors of higher quality. The fastText embedded data sample that will be fed into the LSTM model is displayed in Figure 40.

Figure 40

Sample of the Embedded Data of FastText

```
Sample of embedded training data: [[-0.48358622 -0.40328792 -0.10426242 ... -0.11479074 -0.21172674
 1.0290862 ]
 [-0.11375611 -0.08153183 -0.1856575 ... -0.20400308 -0.32832545
 1.4437412 ]
 [-0.7789925 -0.09605259 -0.8327457 ... 0.17617305 -0.35042125
 0.77452344]
 ...
 [-0.81573457 -0.8493368 0.15205169 ... 0.05653673 -0.3606683
 0.43250728]
 [-0.26416048 -0.80577016 -0.15016021 ... -0.1162755 -0.0704464
 0.9248837 ]
 [-0.3890996 -0.3551575 0.35788307 ... -0.42514047 0.37838382
 0.67696685]]
```

BERT. The development of BERT by Google is a significant advancement in the field of natural language processing, mainly because of its deep bidirectional nature, which enables it to comprehend sentence context more accurately than previous models.

First, a BERT tokenizer and model were initialized using the bert-base-uncased version. The task of transforming text data into a BERT model-compatible format falls to the tokenizer `BertTokenizer.from_pretrained(bert_model_name)`. Tokenization, or the division of text into tokens, is carried out, and after that, the tokens are translated into the appropriate IDs in the BERT vocabulary and any necessary padding or truncation is applied to guarantee consistent length throughout all text samples. Because BERT needs input data in a specific format, including special tokens like [CLS] for the beginning of each sequence and [SEP] for the end of a sentence, this preparation is essential. The BERT model, a pre-trained neural network with a deep architecture based on transformers, is then loaded by the `TFBertModel.from_pretrained` function. After this model runs over each input token ID, each token is represented as a 768-dimensional vector. Because BERT was trained on a vast corpus of text, it developed a deep understanding of language context, which allowed it to capture complex syntactic and semantic information in these embeddings as shown in Figure 41.

Figure 41

Sample of the Embedded Data of BERT

```

Sample training embedding vector: tf.Tensor(
[ 101 2899 26665 4989 3066 2437 4905 4787 6108 11811 2132 3036
 3863 3222 2343 11322 6221 8398 14828 4034 3046 5547 7816 6232
 2156 10859 14045 17666 7999 5971 3930 8398 2623 9317 18754 23388
 11811 4256 2047 2259 2436 2375 3813 7624 16759 2599 4034 2610
 26773 2813 2395 11811 16997 2270 2797 7660 7654 3007 6274 3947
 5546 2499 3988 2270 5378 4862 3676 3676 2177 3173 2194 14208
 2036 2393 2194 22149 10738 7285 2895 2164 2193 2553 2920 14344
 3036 2560 2048 7396 11811 3421 4862 3676 3676 9698 3361 21362
 3522 2095 10847 10819 3154 3251 2593 3319 5531 6108 11811 3811
 10904 6739 2116 7814 3361 10738 2375 5676 3361 5145 25220 3443
 3105 2652 3627 2051 8398 2056 4861 102], shape=(128,), dtype=int32)

```

GloVe.GloVe (Global Vectors for Word Representation), an unsupervised learning algorithm for generating vector representations for words, was used to implement the process of embedding text data. With its development from Stanford University, GloVe leverages aggregated global word-word co-occurrence statistics from a corpus to capture word meanings, both syntactic and semantic.

To create a numerical representation of the text, the text data is first tokenized using the Keras Tokenizer. This process turns the text data into sequences of integers, each of which corresponds to the index of a distinct word. The longest sequence's length (max_len) is then used to pad these sequences to a uniform length, guaranteeing consistent input dimensions that are crucial for neural network models. Next, GloVe embeddings from a pre-trained model (glove.6B.100d.txt) are loaded. These embeddings represent rich syntactic and semantic information of words by assigning a 100-dimensional vector to each word. The dimensions of an embedding matrix are built with respect to the GloVe vector dimensions (100 in this case) and the vocabulary size (vocab_size). This matrix maps every word in the tokenizer's index to either a zero vector or, if one is not available, the word's GloVe vector. Figure 42 shows a sample of GloVe embedded data that will be fed into the LSTM model.

Figure 42

Sample of the embedded data of GloVe

```
Vector for example 0:
[ 0  0  0 ... 544 224 55]
GloVe vectors for example 0:
[[ 0.      0.      0.      ...  0.      0.      0.      ]
 [ 0.      0.      0.      ...  0.      0.      0.      ]
 [ 0.      0.      0.      ...  0.      0.      0.      ]
 ...
 [-0.78142 -0.11488 -0.26742 ... -0.10594 -0.28093  0.42064]
 [ 0.80336  0.51195 -0.58513 ... -0.12729  0.64313  0.13859]
 [ 0.85932 -0.59981  0.5806  ... -0.14398  0.61525  0.02249]]
```

3.2.4 Logistic Regression

We will incorporate advanced word embedding techniques like Doc2Vec, GloVe, Word2Vec, FastText, and BERT as essential data preprocessing steps to improve the predictive accuracy of our logistic regression model. Because these embeddings efficiently capture the semantic relationships and contextual details of words, they are crucial for converting unstructured text into a machine-understandable format and enhancing the model's capacity to interpret and analyze textual data.

GloVe. After data splitting, we used the GloVe (Global Vectors for Word Representation) model to convert textual data into numerical vectors during the word embedding stage. The Stanford-developed GloVe project includes the pre-trained GloVe model used in this procedure, glove.6B.100d.txt. Six billion token embeddings are available, with each token being represented as a 100-dimensional vector.

Tokenization was done using the NLTK library, which divided the text into individual words. Pre-trained GloVe embeddings that were loaded from a specified file path using the `load_glove_model` function. The sentence vector function ensured that each sentence was represented by a fixed-size vector by calculating the average of the embeddings.. In order to transform raw text into a format appropriate for logistic regression analysis, the embedding procedure as a whole was essential. A sample of the embedded vector is shown in Figure 43. The shape of the training data that has been embedded is (34398,100).The same

Figure 43*Sample of Embedded Vector of GloVe*

```

Shape of the training data matrix: (34398, 100)
Sample of the embedded vector: [ 0.09572037 -0.15186262  0.2268855  -0.07922118  0.1352916
-0.22688734  0.17228936 -0.13954501  0.05449103 -0.14062561 -0.07441506
 0.07125036 -0.07701838 -0.09189867 -0.23030091  0.20506982  0.03897813
-0.48723557  0.05704563  0.2702852  -0.12804495  0.3153616  -0.00304305
-0.0037042  -0.03888341  0.01852928 -0.46213905 -0.05924136  0.04225479
 0.21003521  0.27157475 -0.04058625  0.082451  -0.13836061  0.0751769
 0.01267753  0.03935781 -0.10420885 -0.0241023  -0.5047357  -0.18383645
 0.28156485 -0.29124509  0.01574947 -0.26021757  0.09223899 -0.29652799
-0.0125523  -0.50675624  0.27083795 -0.22014932  0.17240179  0.64866079
-0.18435233 -1.61898276  0.00412878 -0.16129779  1.46923166  0.40742776
-0.23771755  0.16861064  0.06417657 -0.06294742  0.56630016 -0.1391983
-0.00455964  0.51595523 -0.12778919  0.06143666  0.07697499 -0.12623367
-0.18096629 -0.34501608  0.07887452  0.06800215 -0.18427988 -0.03306428
-0.72127455 -0.02583014  0.42249255 -0.23967631 -0.14737332 -0.03586467
-0.87867321 -0.21561202  0.01153857  0.02239433  0.04620268 -0.32693359
 0.07324983 -0.32317669 -0.20829336  0.13173592 -0.17863403  0.18897541
 0.07671475 -0.11504755  0.36040863  0.01622313]

```

Word2Vec. The pre-trained 'GoogleNews-vectors-negative300.bin' model from Word2Vec, a potent tool created by a group of researchers at Google, was used in the Word2Vec embedding stage of the text processing. The robustness and accuracy of this model are especially well-known because it was trained on a corpus of Google News articles, which contains approximately 100 billion words. The resulting embeddings capture complex syntactic and semantic relationships, providing 300-dimensional vectors for every word as shown in Figure 44.

Each text entry from the dataset was preprocessed to lower case and free of punctuation upon loading this pre-trained model, producing a list of words. The corresponding 300-dimensional embeddings of these words were then obtained by running them through the Word2Vec model. Significantly, words that were absent from the Word2Vec model were accommodated amicably by designating a 300-dimensional zero vector, guaranteeing uniform vector representation throughout the text corpus. The information was essentially condensed into a single 300-dimensional vector for each text entry by averaging the embeddings of every word in the text to create an aggregate sentence vector. After all of the dataset's entries went through

this procedure, the text data was uniformly encoded with numbers and prepared for further machine learning tasks.

Figure 44

Sample of Embedded Vector of Word2Vec

```
Shape of the training data matrix: (34398, 300)
Sample of an embedded vector: [-0.00796877  0.05072187  0.04398667  0.05012347 -0.00762139 -
 0.00239006 -0.05016745  0.08255722  0.02419301 -0.00285684 -0.06007168
-0.0291473  0.03106626 -0.13845171  0.07421228  0.08865909  0.07210567
-0.02474175 -0.02376613  0.02870377  0.03132188  0.01994059 -0.00393786
 0.03764854 -0.06559703 -0.0772699  0.02464583  0.00532317  0.0276999
 0.05635189 -0.05115252 -0.019728  0.04467453  0.01845026 -0.0188729
 0.03174484  0.03664696  0.02415061  0.04982439  0.00717447 -0.04592518
 0.11140791  0.02247319 -0.05838768 -0.10385156 -0.02036604  0.01210072
-0.05844248  0.06919613  0.03313464 -0.00315089 -0.0159366  0.02436926
-0.03906746  0.02921973 -0.10206798 -0.08274098 -0.02155562 -0.09924397
-0.05459991 -0.0060982  -0.03525338 -0.04064096 -0.05359823  0.0003294
-0.01330557  0.09888638  0.01254367  0.02714187  0.02582297  0.02138107
 0.08963854  0.05824522 -0.06669012 -0.04570624  0.07034469  0.05057497
 0.07892519  0.04795555  0.04144422 -0.02306666  0.01401854 -0.00378195
 0.02548149 -0.05317485 -0.10243506  0.11353631 -0.0028184  0.0101428
 0.09566578 -0.0252292  -0.01924725 -0.04880008 -0.03031617 -0.02730961
-0.00812597 -0.00792571  0.01208347 -0.02815088  0.00082133  0.0271076
 0.0377056  0.03205262  0.02750829 -0.04512293 -0.03344037 -0.06454302
 0.02660762  0.06606828  0.02060132  0.02544216  0.05067201  0.04050226]
```

Doc2Vec. The goal of the Doc2Vec embedding phase was to use the Doc2Vec model, a sophisticated Word2Vec extension created especially for document-level embeddings, to convert the textual data into meaningful vector representations. Each document in the dataset was assigned a unique identifier following the tokenization of the text data using the `word_tokenize` function from NLTK. The Doc2Vec model's subsequent training required this preparation. With a vector size of 20, we started the model, meaning that every document would be represented as a 20-dimensional vector. The learning rate was regulated by the parameters `alpha` and `min_alpha`, which were set at 0.025 and 0.00025, respectively. To guarantee that even the rarest words were taken into account, `min_count` was set to 1.

In order to ensure that the Doc2Vec model received enough exposure to the dataset to acquire reliable representations, the training process was spread across ten epochs. Following training, all the documents in the training dataset were converted into a 20-

dimensional vector, which served as a dense numerical representation of the semantic content of each document. Subsequent machine learning tasks employed these vectors. A sample of the embedded vector of Doc2Vec is as shown in Figure 45.

Figure 45

Sample of Embedded Vector of Doc2Vec

```
Shape of the training data matrix: (34398, 20)
Sample of an embedded vector: [ 0.12362926  0.8099524 -4.3534555  2.4906297 -0.40779796
-3.620176  6.0181046 -2.0479743  5.2972665  2.192476  2.1807516
 3.8522294 -1.182576  0.6350732  3.358171 -0.5030569  2.4326248
-1.5031992 -4.246361 ]
```

FastText. The FastText model, a sophisticated word embedding method created by Facebook's AI Research lab, was utilized in the FastText embedding stage of the text processing workflow. FastText differs from conventional word embedding techniques in that it considers not only entire words but also sub-word units (n-grams), which allows it to handle words that are not commonly used. The vector size was set to 30 when the FastText model was initiated for this task, meaning that every word (and sub-word unit) would be represented in a 30-dimensional vector space. Additionally, the window size was set to 3, enabling the model to take context into account for the three words that come before and after the target word. To ensure that even rare words were included and to enable parallel processing for quicker training, the parameters `min_count` and `workers` were set to 1 and 4, respectively

The FastText model was trained using tokens after tokenizing the text data with NLTK's `word_tokenize` function. Interestingly, the trained FastText model averaged the vectors of the words that made up each sentence in the dataset to create a vector representation. By the time the embedding phase was over, the unprocessed text data had been converted into a format that could be analyzed with logistic regression. Each sentence was represented as a 30-dimensional

vector as shown in Figure 46. Through this process, the synergy between traditional machine learning algorithms and advanced NLP techniques is highlighted, opening the door to insightful modeling and data analysis.

Figure 46

Sample of Embedded Vector of FastText

```
Shape of the training data matrix: (34398, 30)
Sample of an embedded vector: [ 0.8142319  0.5137732  0.1853654  0.38205424  0.5701379
-0.12639292  0.7448162 -0.4544826 -0.11175265  1.0422825  0.3744904
 0.0681086 -0.06971049  0.4066194 -0.23998886 -0.13298012  0.62968147
-0.20677012 -0.3283587  0.36432835 -0.0978518 -0.18715201 -0.05692126
 0.2909598  0.56506455  0.1471379 -0.26456004 -0.01089138 -0.5144665 ]
```

Sentence BERT. The Sentence Transformers library's advanced all-MiniLM-L6-v2 model, a sentence-specific variant of BERT (Bidirectional Encoder Representations from Transformers), was used for the SBERT (Sentence-BERT) embedding step. Recognized for its efficacy and efficiency, the all-MiniLM-L6-v2 model reduces, without appreciable performance loss, the power of larger transformer models into a more compact, computationally manageable package. This model is perfect for tasks requiring deep linguistic understanding because it excels at producing semantically meaningful representations of sentences.

Using the SBERT model, the process started with encoding the textual data into embeddings. In this stage, every text entry was fed through the model, which produced a high-dimensional vector representation of every sentence. Much more than what conventional word-level embeddings can offer, these embeddings capture the subtle semantic information present in the text. Next, it was verified that every sentence in the dataset had been successfully converted into a vector in the embedding space of the model by looking at the shape of the resulting embeddings as shown in Figure 47.

Figure 47

Sample of Embedded Vector of FastText

```
Shape of the training data embeddings: (34398, 384)
Sample of an embedded vector: [ 2.64656846e-03 -6.73711440e-03  6.15236536e-02 -5.93257044e-
 2.03378499e-02  3.36759128e-02 -3.77623215e-02 -5.52685708e-02
-1.85850430e-02  9.25048161e-03  3.02210427e-03  2.60598361e-02
-9.96930525e-03  4.78734914e-03  3.82177979e-02  9.17598382e-02
 8.07040706e-02  2.64981370e-02 -7.79232308e-02  4.69952226e-02
 2.97297761e-02  4.04751394e-03 -1.46104336e-01 -1.90294255e-02
-6.28259927e-02  1.84501521e-02 -8.36584419e-02 -3.81218791e-02
-1.28665771e-02 -3.45067568e-02 -1.02177067e-02  2.47575492e-02
-4.15604301e-02  9.66365449e-03 -2.30151843e-02 -3.78244445e-02
-1.92759279e-03  3.46489511e-02  6.78765923e-02 -4.95426171e-02
 3.49560915e-03  1.01284795e-02  2.56049801e-02 -6.85669258e-02
 2.40350179e-02 -5.88780083e-02 -5.50588556e-02 -6.32130206e-02
-1.11414278e-02  1.59829687e-02 -6.51592687e-02 -1.89666748e-02
 3.59033011e-02  3.23565677e-02 -6.75051659e-02 -3.67102623e-02
 3.44099961e-02  4.16590869e-02  1.66841503e-02 -6.90678060e-02
```

3.2.5 CNN+Bi-LSTM

The preprocessing stage of the CNN+Bi-LSTM active learning pipeline begins with transforming the target variable into a numerical format using Label Encoding, a common practice in handling categorical data for classification tasks. This is followed by a strategic division of the dataset into 70% training, 15% validation, and 15% test sets, ensuring that the model is trained, validated, and tested on distinct subsets of data to avoid data leakage. A crucial step in processing the textual data involves tokenizing the text using Keras's Tokenizer and padding the sequences to a uniform length. This standard procedure allows for efficient batch processing in neural networks. Additionally, the implementation utilizes pre-trained Word2Vec embeddings from Google (google-news-300) to convert text tokens into dense vectors. This technique effectively captures the semantic meanings of words, which is particularly beneficial in NLP.

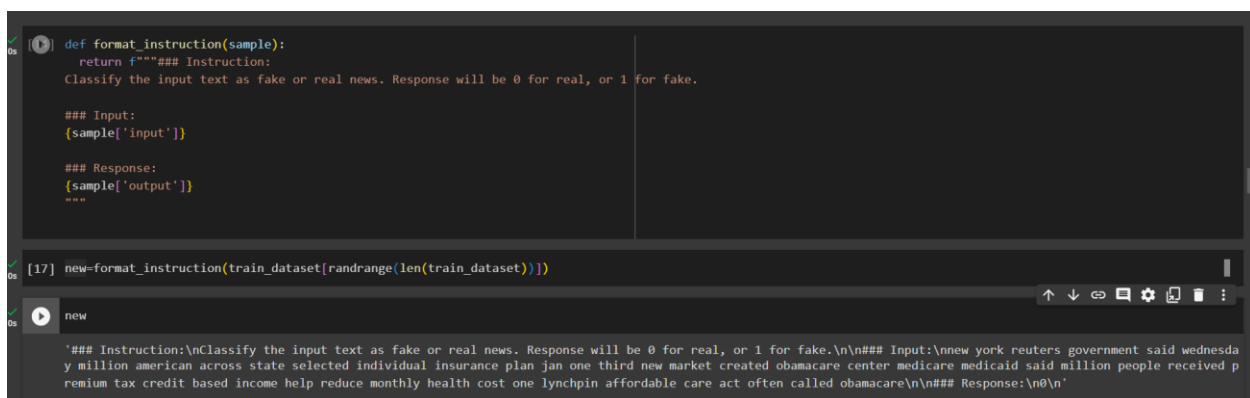
3.2.5 Meta Llama 2 7B

Although the dataset underwent NLP-based preprocessing, further transformation was necessary to make it compatible with the LLaMA2 - 7B model. The dataset's 'text' column was renamed to 'input', and the 'target' column to 'output'. It was then split into 80% training and

20% testing subsets. A key aspect of the transformation process was the development of a prompt template. This template integrated an instructional statement that clearly defined the task, along with the corresponding value from the 'input' column. The expected model response was articulated, correlating with the data in the 'output' column. A sample prompt, generated from a random training dataset instance, is showcased in Figure 48.

Figure 48

Prompt Template Sample



```
def format_instruction(sample):
    return f"""### Instruction:
Classify the input text as fake or real news. Response will be 0 for real, or 1 for fake.

### Input:
{sample['input']}

### Response:
{sample['output']}
"""

[17] new=format_instruction(train_dataset[randrange(len(train_dataset))])

new

'### Instruction:\nClassify the input text as fake or real news. Response will be 0 for real, or 1 for fake.\n\n### Input:\nnew york reuters government said wednesda
y million american across state selected individual insurance plan jan one third new market created obamacare center medicare medicaid said million people received p
remium tax credit based income help reduce monthly health cost one lynchpin affordable care act often called obamacare\n\n### Response:\n0\n'
```

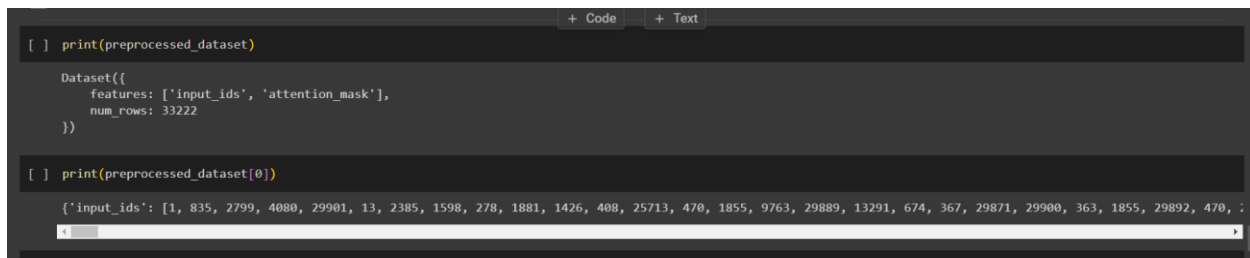
While the preprocessing steps for both training and test sets are largely similar, there is a subtle difference in the prompt format. Specifically, the test set does not include the Response value in its prompts. This is because the Response value represents the target output that the model is expected to predict, and is thus omitted in the test scenarios.

Once the training set's prompt template was established, the resultant formatted text was saved into a newly created 'formatted_text' column. Subsequently, all other columns were removed from the dataset, leaving a singular column for batch processing. The dataset's batches are set with a maximum length of 1024 tokens, despite LLaMA2's capability of handling 4096, to accommodate the memory limitations of Colab. A custom preprocessing function was crafted to carry out these tasks, producing 'input_ids' and 'attention_masks' as outputs. The preprocessed dataset, detailed in Figure 49 below, consists of 'input_ids' — indices representing tokens

transformed into a numerical format for model training, using a tokenizer from the Hugging Face Transformers library. The tokenizer is initialized with the configuration from the `bitsandbytes` model. The `bitsandbytes` library facilitates model quantization, a process that shrinks the size of deep learning models by decreasing the bit-width of the weights and activations. By employing quantization, models benefit from quicker inference times and lower memory usage, which is especially advantageous for running the models on edge devices with constrained computational capacity. Meanwhile, 'attention_mask' is a binary sequence that discerns significant tokens from padding, ensuring the model focuses on the relevant data.

Figure 49

Preprocessed Dataset Structure



```
[ ] print(preprocessed_dataset)

Dataset({
  features: ['input_ids', 'attention_mask'],
  num_rows: 33222
})

[ ] print(preprocessed_dataset[0])

{'input_ids': [1, 835, 2799, 4080, 29901, 13, 2385, 1598, 278, 1881, 1426, 408, 25713, 470, 1855, 9763, 29889, 13291, 674, 367, 29871, 29900, 363, 1855, 29892, 470, ;
```

3.2 Proposed Model (Framework) Architecture

3.2.1 DNN

We developed DNN models on five different embedded datasets mentioned in previous sections: Doc2Vec, Glove, Word2Vec, FastText, and BERT. For each embedding approach, we conducted three separate experiments: with length attribute, without length attribute, and with the active learning technique. Figure 50 and Table 1 show the architecture and parameters of our DNN models. The architecture started with an embedding layer followed by a fully connected layer. Our model consisted of two fully connected or dense layers with the numbers of neurons being 128 and 64 respectively for the first and second layers. Rectified Linear Unit (ReLU) activation was a popular choice for the activation function in dense layers of neural networks and

they were utilized for our two dense layers to introduce non-linearity to the model. Besides, dropout was known as a regularization technique, and it worked by randomly setting a fraction of input units to zero at each update during training time. We had two dropout layers (50% dropout) between each of the dense layers to prevent overfitting. Last but not least, as this is a binary classification problem, the output activation function was sigmoid to categorize the output to Real or Fake.

Figure 50

DNN Architecture

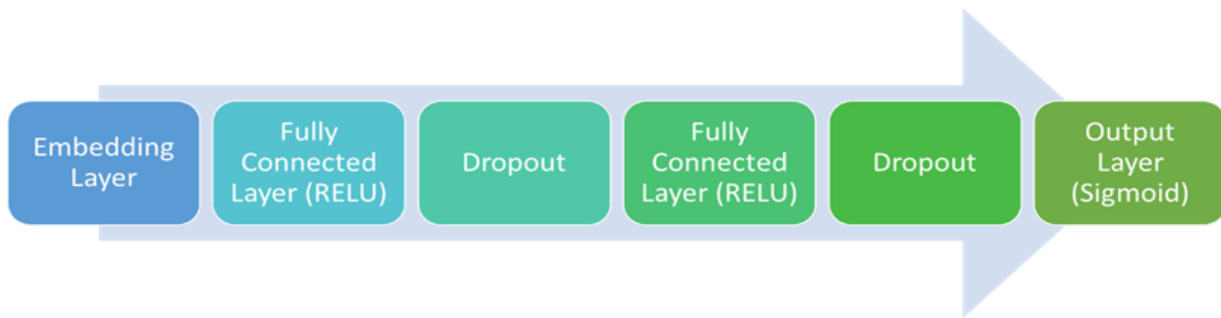


Table 1

Parameters of the DNN Model Architecture

Model Architecture Parameter	Value
activation function (two fully connected layer)	ReLU
activation function (output convolution layer)	sigmoid
number of hidden layers	2 fully connected layers
neurons in hidden layers (first fully connected layer)	6272
neurons in hidden layers (two fully connected)	128, 64

3.2.2 LSTM

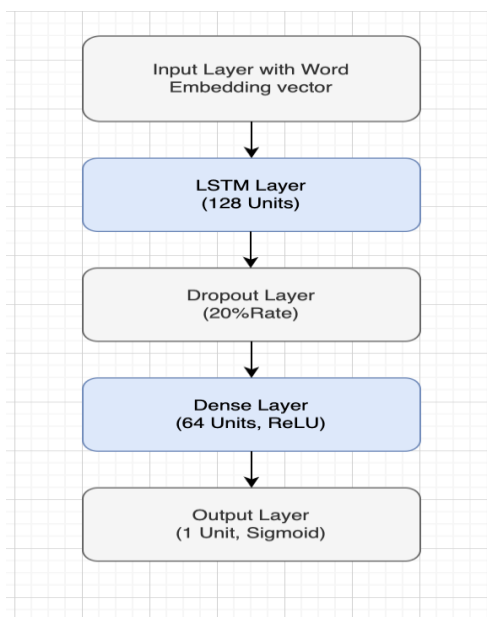
Our model architecture combines a custom neural network for binary text classification with advanced word embeddings. Language processing is facilitated by an LSTM (Long Short-Term Memory) layer with 128 units, which fundamentally captures the sequential nature and temporal dependencies present in textual data.

After the LSTM, a 0.2 rate Dropout layer is used to reduce overfitting by randomly deactivating a portion of the neurons during training, improving the generalization capacity of the model. The network then has a 64-unit Dense layer with a "relu" activation function to process and interpret the output of the LSTM.

Deciding that the model is optimized for the given binary classification task, consisting of a single unit Dense output layer with 'sigmoid' activation function. The likelihood that the input text belongs to one of the two classes is reflected in the probability score that this layer generates. Figure 51 illustrates the architecture diagram of the LSTM model that was incorporated. These layers work together to form a strong architecture that effectively classifies text data while utilizing the advantages of LSTM in comprehending language context and sequence. Each layer plays a distinct role in this architecture. The parameters of the model architecture of LSTM model is mentioned in Table 2.

Figure 51

Model Architecture of LSTM

**Table 2***Model Architecture Parameters**Architecture Parameters Used in LSTM*

Architecture parameters	values	Function
LSTM Units	128	Number of units in the LSTM layer
Dropout Rate	0.2	Fraction of the input units to drop
Dense Layer Units	64	Number of neurons in the dense layer
Activation Function (Dense)	'relu'	Activation function for the dense layer
Output Activation Function	'sigmoid'	Activation function for the output layer

3.2.3 Bi LSTM

Bi-LSTM is an extension of LSTM and RNN neural networks in which it processes sequences of text in both forward and reverse directions. This helps the model to capture the nuances from the past and future data points effectively capturing the long-term dependencies that more accurately represent input sequences. A very simple model was employed to evaluate the performance across all the embeddings discussed above. For all the embeddings a model with embedding layer, fully connected layer with 64 LSTM units each corresponding to 128 Bi-LSTM layer, and output layer of using sigmoid activation function (binary classification) was used. For Doc2Vec, each LSTM layer consisting of 128 units was used, resulting in 256 units for Bi-LSTM. Figure 52 shows the overall flow of the input data to the classified output.

Figure 53 shows the summary of the Fasttext model without the addition of the number of words. The model has a sequence length of 10208 as calculated by the maximum train and test sequence length. The output vector dimension is 300. The number of parameters in the first embedding layer is $98281 \times 300 = 29484300$. These are non-trainable due to pre-defined weights. The number of parameters in the Bi-directional layer = $8(64 \times (64 + 300) + 64) = 186880$ owing to the eight Feed Forward Neural Networks in case of Bi-LSTM. The number of parameters in the output layer is 128 from the 128 units plus one bias, totalling to 129 parameters. There are no dropout layers. The output layer has one dense unit with a 'sigmoid' activation function for classification.

Figure 52

Bi-LSTM Model Flow

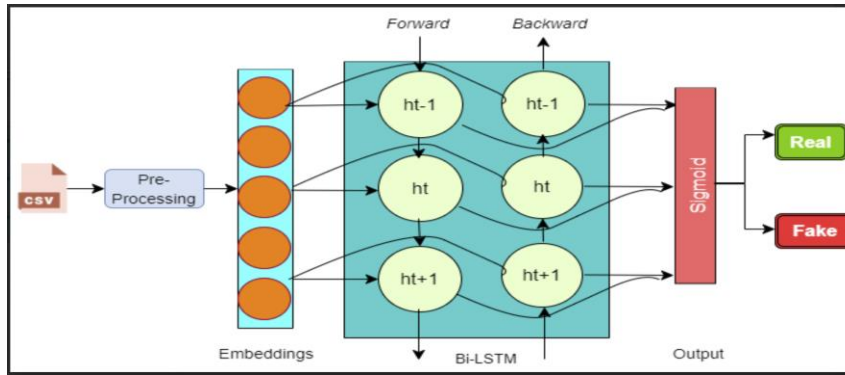


Figure 53

Model Summary of Bi-LSTM Using Fasttext Embeddings

Model: "sequential_3"

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, 10208, 300)	29484300
bidirectional_1 (Bidirectional)	(None, 128)	186880
dense_3 (Dense)	(None, 1)	129
Total params: 29671309 (113.19 MB)		
Trainable params: 187009 (730.50 KB)		
Non-trainable params: 29484300 (112.47 MB)		

3.2.4 Logistic regression

A simple yet efficient method to implement the logistic regression machine learning model, excluding the use of deep learning architecture like recurrent or convolutional layers is introduced. By estimating probabilities using a logistic function, the basic classification algorithm known as logistic regression performs exceptionally well in binary classification tasks. The model functions by first combining features in a linear fashion to change the input into probabilities, which are then output via the application of a sigmoid function. Its straightforward design makes it very interpretable and computationally efficient for some kinds of data,

particularly when working with features that are linearly correlated with the target variable's log odds.

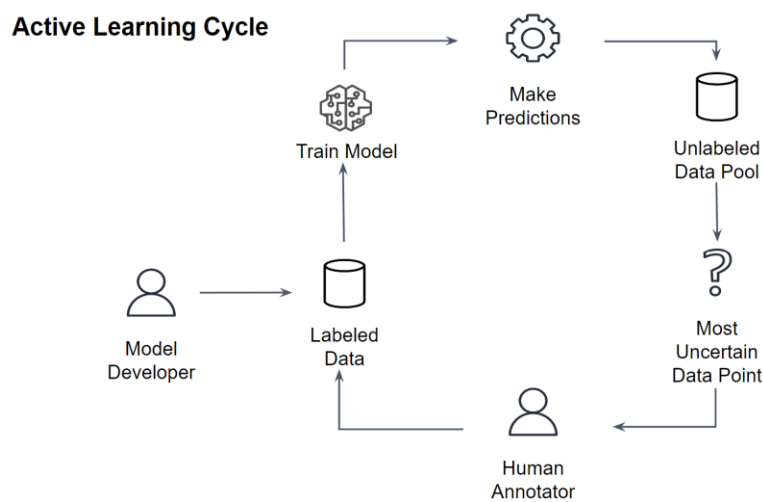
The logistic regression model was set up for our particular application to manage the high-dimensional feature vectors produced by text data embeddings. In order to guarantee that the optimization algorithm would converge, the model's main hyperparameter, `max_iter`, was set to 1000. This was necessary because of the high-dimensional embedding space's added complexity.

3.2.5 CNN+Bi-LSTM

In the framework for the CNN+Bi-LSTM, the model employs an active learning approach, a semi-supervised technique where the model iteratively queries for labels on new data points. Initially, the model trains on a small set of data with labels, in this case, 20 samples. It progressively queries the most informative samples from the unlabelled pool, determined by their uncertainty levels. This process involves training the model in each iteration and making predictions on the unlabeled pool, using entropy as a criterion for sample selection. The most uncertain samples are added to the training set one at a time with each query and provided their respective label. A visual representation of this process can be found in figure 54. An early stopping mechanism based on validation accuracy is incorporated to prevent overfitting, ensuring that the model training halts when no significant improvement in accuracy is observed after three epochs (Settles, 2009).

Figure 54

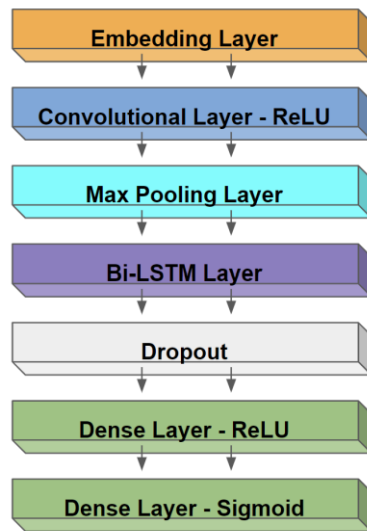
Diagram of Active Learning Cycle



The model's architecture is a hybrid of Convolutional Neural Network (CNN) and Bidirectional Long Short-Term Memory (Bi-LSTM), a combination that offers comprehensive feature extraction for text classification tasks such as Fake News detection. The CNN layer is adept at extracting features, such as n-grams, from the text, while the Bi-LSTM layer excels in capturing long-range dependencies by processing data in both forward and backward directions. This blend allows the model to grasp the textual features and the broader context effectively. The architecture is further enhanced with Dropout and Dense layers, providing the necessary regularization and classification capabilities, making it a suitable choice for complex text-based binary classification tasks. The Figure 55 below depicts the model architecture design.

Figure 55

CNN+Bi-LSTM Architecture

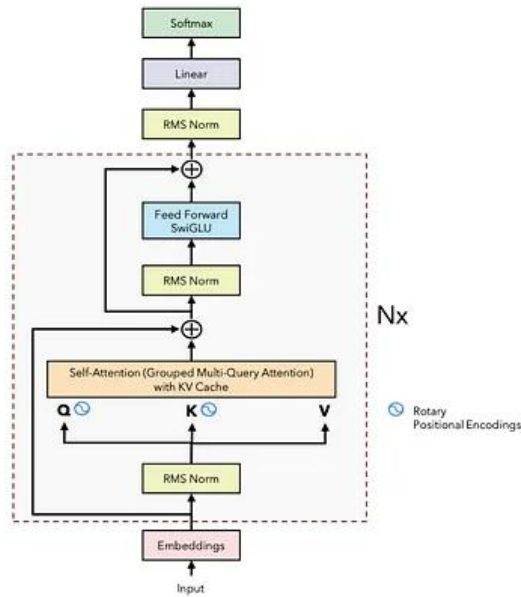


3.2.6 Llama2

LLaMA 2 represents a series of advanced pre-trained and fine-tuned large language models (LLMs), with sizes ranging from 7 billion to 70 billion parameters. Building on the foundation of the original LLaMA, LLaMA 2 utilizes the Google transformer architecture and introduces several enhancements. As shown in Figure 56 below (CheeKean, 2023), these enhancements include RMSNorm pre-normalization, akin to GPT-3's approach; the SwiGLU activation function, derived from Google's PaLM; the adoption of multi-query attention over the conventional multi-head attention; and the implementation of rotary positional embeddings (RoPE), similar to GPT Neo. The training of LLaMA utilized the AdamW optimizer for improved weight updates. Distinct from its predecessor, LLaMA 2 features a longer context window of 4096 tokens compared to 2048 and employs grouped-query attention (GQA) in its larger models, replacing the multi-query attention (MQA) mechanism (Heller, 2023).

Figure 56

LLama2 Architecture



To fine-tune the model, this project employed QLORA, an effective technique that quantizes a pre trained large language model (LLM) to 4 bits and integrates "Low-Rank Adapters," allowing for fine-tuning on a single GPU. This method is facilitated by the PEFT library. Users are required to log in to Hugging Face using the command `!huggingface-cli login` and an access token. A function `create_bnb_config` was defined to establish the `bitsandbytes` configuration before loading the model. The `bitsandbytes` library supports model quantization, and the `BitsAndBytesConfig` class from the `transformers` library is used to set up the quantization method. The pre-trained model "meta-llama/Llama-2-7b-hf" was loaded with this configuration. Table 3 below outlines the parameters selected for model loading via the bitsandbytes configuration.

Table 3

BitsandBytes configuration parameters

BitsAndBytesConfig() parameters		
load_in_4bit	True	Activate 4-bit precision base model loading
bnb_4bit_use_double_quant	True	Activate nested quantization for 4-bit base models (double quantization)
bnb_4bit_quant_type	"nf4"	Quantization type (fp4 or nf4)
bnb_4bit_compute_dtype	torch.bfloat16	Compute data type for 4-bit base models

The Fine-tuned model summary is shown below in Figure 57 and includes an embedding layer designed to process 32,000 different tokens, each mapped to a 4096-dimensional space. It comprises 32 stacked decoder layers, indicating a deep network structure conducive to complex language tasks. Each layer features a custom attention mechanism with linear projections for queries, keys, values, and outputs, but without biases for these projections, which is slightly unusual as biases are often included to help with learning offsets. Rotary Positional Embeddings (RoPE) are utilized within the attention mechanism, providing a way to encode the sequence order of tokens. The model also includes a feedforward neural network block (LlamaMLP) with a gated linear unit and a SiLU (Sigmoid Linear Unit) activation function. The architecture uses Root Mean Square Layer Normalization (LlamaRMSNorm) before the attention and after the attention mechanism, which is a normalization technique that often helps stabilize the training of

deep networks. Finally, the linear layer named `lm_head` suggests the presence of a language modeling head on top of the transformer, with 4096 input features and 32,000 output features, mapping back to the vocabulary size for the prediction of the next token in a sequence.

Figure 57

Model Summary

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(32000, 4096, padding_idx=0)
    (layers): ModuleList(
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (v_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (o_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (up_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (down_proj): Linear(in_features=11008, out_features=4096, bias=False)
          (act_fn): SiLUActivation()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=32000, bias=False)
)
```

4. Experimental Setup

4.1 DNN

The model architecture mentioned in the previous section (3.2.1 Proposed Model DNN) already discussed the model layers and activation functions attached with these layers. In general we have one embedding layer followed by a fully connected layer of 128 neurons and ReLU activation, a dropout layer with 50% drop, another dense layer of 64 neurons and ReLU activation, another 50% dropout layer, and finally an output layer with sigmoid activation. This section will focus on other hyperparameters.

Table 4 shows the hyperparameters that we used for all embedding layers and DNN combinations. `binary_crossentropy` was chosen because it was designed for binary classification problems and was well-suited for optimizing models that output probabilities. Smaller batch sizes, like 32, could sometimes lead to better generalization performance. Training with smaller batches introduced more noise into the optimization process, which could help the model generalize better to unseen data. The learning rate is default and number of epochs are set as 10 and the same for the first 4 embedding datasets, but for BERT we reduced the epoch number to 5 due to computational limitations. On the other hand, accuracy was a commonly used metric in classification tasks, including fake news detection. It was the ratio of correctly predicted instances to the total number of instances in the dataset. The metric was picked as it is a straightforward and easy-to-understand metric.

Table 4

Hyperparameters of DNN combination models

Hyperparameter	Value
loss function	<code>binary_crossentropy</code>
optimizer	<code>adam</code>
batch_size	32
number of epochs	10
Learning rate	default
evaluation metric	accuracy score

To conduct the training process, we used Google Collab Pro platform with a variety of memory choices, including High-RAM CPU, T4 GPU, A100 GPU, V100 GPU, and TPU. For analysis softwares, we used TensorFlow and Keras both at version 2.14.0, which had excellent GPU support and were crucial for handling the computational demands of training large models on large datasets. This also allowed for accelerated training of deep neural networks.

4.2 LSTM

In our experiments with five different word embeddings leading up to the LSTM model, we maintained consistent hyperparameters across the first four models to ensure comparability. The model's 128-unit sequence-processing long short-term memory (LSTM) layer was thoughtfully designed. The choice of this layer size was made in order to capture contextual dependencies with a significant capacity without overly complicating the model. In addition, each model had an output layer for the binary classification task that was activated by a sigmoid and contained 64 neurons in its Dense layer, which was triggered by the 'relu' function. These models were trained for 10 epochs with a batch size of 16, optimizing for binary cross-entropy loss and accuracy metric. They were then assembled using an Adam optimizer with a learning rate of 0.01.

The number of epochs for the model with BERT embeddings was lowered to five because of its computational complexity. In order to account for the complexities of the BERT model, the batch size was raised to 64 and learning rate to $1e-5$. Enhancing training effectiveness and model performance was the goal of this fine-tuning. Table 5 mentions the hyperparameters used in the LSTM model.

Table 5*Hyperparameters Used in LSTM*

Hyperparameter	Description	Values
Learning Rate	Step size at each iteration during optimization	0.01, 1e-5
Batch Size	Number of samples per gradient update	16, 64
Epochs	Number of complete passes through the training dataset	5, 10
Loss Function	Loss function used for binary classification.	'binary_crossentropy'
Optimizer	Optimization algorithm used for training.	Adam

For the training process, Google Colab Pro was utilized, offering diverse memory options such as High-RAM, V100 A100 GPU. The analysis was conducted using TensorFlow and Keras, both at version 2.14.0. These versions are particularly efficient in GPU utilization, essential for managing the computational requirements of training extensive models on large datasets. This setup significantly expedited the training of deep neural networks, including LSTM models

4.3 Bi-LSTM

The idea was to start with developing a very simple Bi-LSTM model without any added complexities and evaluate its performance. If the performance were not at par then increase the complexity of the model gradually. Bi-LSTM performed reasonably well giving 93% accuracy at the first epoch. Increasing the epochs to five increased the accuracy to 95%. It was observed that validation loss was consistently reduced and accuracy was consistently increased. When epochs were assigned as 10, then for most of the embeddings the learning stopped at the seventh epoch only, yielding more than 98% accuracy. Bi-LSTM model was trained using 10 epochs with batch_size equal to 32. Batch Adam optimizer was used with 0.001 learning rate. Model checkpoint was used to save the model progress at every epoch by monitoring maximum validation accuracy and Early Stop callbacks was used to avoid overfitting by monitoring the validation loss with the patience of five respectively. Dynamic reduction of learning rate was also programmed by the factor of two when the validation loss did not decrease for consecutive three epochs. Due to the model performing reasonably well and not getting stuck in local minima, this callback was not used for faster execution. Doc2Vec modeling started with 64 units for each LSTM layer, totalling to 128 for Bi-LSTM but it gave 62% accuracy. Increasing the Units to 256 for Bi-LSTM and epochs to 20, the models performance increased to 90%. For BERT, the five epochs with learning rate $1e-5$ gave an accuracy of 97%. Table 6 shows the hyperparameters used for training the models.

Table 6

Hyperparameters Used in Bi-LSTM

Hyperparameters	Values	Description
-----------------	--------	-------------

Number of layers	3 (Embedding, Bi-LSTM, Output)	Only three layers were there in the model
Units(neurons) Unit (Doc2Vec)	64 128	Glove, word2Vec, Fasttext has 64 units of LSTM accounting to 128 Bi-LSTM. Doc2Vec has 128->256 units for Bi-LSTM

Hyperparameters	Values	Description
Optimizer	Adam Learning rate =0.001 Learning_rate=1e-5(BERT)	The learning rate 0.001 was used as default with Adam optimizer. For BERT it was reduced to 1e-5
metrics	Accuracy	Metrics used for training were Accuracy and Accuracy and AUC for BERT.
epochs	10 20((Doc2Vec) 5(BERT)	Epochs started with 1, increased to 10 for Glove, word2Vec, Fasttext. For Doc2Vec it was increased to 20. It was kept at 5 for BERT for faster computation.
loss_function	binary_crossentropy	Since binary classification loss function used was binary cross entropy
batch_size	32	Batch size was 32 for all, decreased to 16 for BERT(it took longer to train). After moving to Collab Pro, increased batch_size to 32
early_stop	Patience level=5 monitor=validation loss	Early stop criteria to monitor if validation loss remained same for 5 consecutive epochs then stop to prevent overfitting.
ModelCheckpoint	Filepath (check & only save the best model after every epoch), monitor=validation accuracy mode=max	Checked the performance of the model at every epoch, and only the best model is saved based on best validation accuracy till that epoch.

ReduceLROnPlateau	factor=0.2 patience=3 min_lr=1e-6	If there is no decrease in the validation loss for three consecutive epochs then reduce the learning rate by the factor of 0.2. Don't go less than the minimum at 1e-6
Max_length	10208(without number) 10209 (with_num_of_words) 128 (BERT)	The maximum length fetched from the x_train and x_test data. Since BERT was computationally expensive, 128 was chosen as the max_length
Activation function	sigmoid	Output dense layer has activation function of Sigmoid for binary classification into Fake or Real

Google Collab and Later Google Collab Pro was utilized for the course of this project.

Apart from the freely available compute instances, A100 GPU was used for executing the final streamlined code. The details can be seen in Figure 58.

Figure 58

A100 GPU Runtime Accelerator for Training Bi-LSTM

+-----+ NVIDIA-SMI 525.105.17 Driver Version: 525.105.17 CUDA Version: 12.0 +-----+									
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute	M.		
						MIG	M.		
+-----+									
0	NVIDIA A100-SXM...	Off	00000000:00:04.0	Off			0		
N/A	36C	P0	46W / 400W	0MiB / 40960MiB	0%	Default	Disabled		
+-----+									
+-----+ Processes: GPU GI CI PID Type Process name GPU Memory ID ID Usage +-----+									
No running processes found									
+-----+									

There are a plethora of Softwares and Frameworks used for this project. Frameworks such as Tensorflow, Keras are used for model building and Tokenization. transformers are used

for Pre-trained BERT models. Gensim library is used for pre-trained embedding models. Scikit-Learn was used for train, test, and validation splits, metrics, confusion matrix, classification report, 'roc_curve', 'auc'. Matplotlib and Seaborn are used for visualizations. Time is used for monitoring time. Regular Expression package is used for grepping different patterns for identifying emails,URLS, single characters, etc. NLTK library is used for tokenization, stopwords, lemmatization. NLPAug is used for augmentation. Some of the versions used are as shown in Figure 59

Figure 59

Some of the Libraries Used with Their Versions

gensim	4.3.2
keras	2.14.0
matplotlib	3.7.1
matplotlib-inline	0.1.6
nltk	3.8.1
notebook	6.5.5
numpy	1.23.5
pandas	1.5.3
scikit-learn	1.2.2
seaborn	0.12.2
tensorflow	2.14.0

4.4 Logistic Regression

Throughout all the word embedding the max iteration was set as 1000 as a hyperparameter. Additionally for the logistic regression model that was performed using the sentence BERT word embedding went through additional hyperparameter tuning as follows. With the help of cross-validation, GridSearchCV is a potent tool that methodically explores various parameter tune combinations to identify the optimal tune. Here, the particular hyperparameters that have been adjusted are C, max iteration and solver. Table 7 provides a detailed description of the various hyperparameters used on this model.

C. In logistic regression, this hyperparameter regulates how strong regularization is. Regularization is a technique that penalizes large coefficients in the model in order to prevent overfitting. Smaller values in logistic regression indicate stronger regularization. C is the inverse of regularization strength. C is varied across five values in your grid search: [0.01, 0.1, 1, 10, 100]. From stronger to weaker regularization, this range encompasses it.

solver. The algorithm to apply to the optimization problem is specified by this hyperparameter. For certain kinds of datasets, different solvers may perform better. we are using ['liblinear', 'lbfgs'] in this instance. For smaller datasets, the liblinear solver works well; for larger datasets, lbfgs performs better.

max_iter. The max_iter parameter is important since it directly affects the solver's convergence. A higher number of iterations may be required when working with complex models or high-dimensional data to make sure the algorithm has enough time to find the best solution.

Table 7

Hyperparameters used by Logistic Regression Model

Hyperparameter	Description	Values
C	Inverse of regularization strength	0.01, 0.1, 1, 10, 100
solver	Algorithm to use in the optimization problem	'liblinear', 'lbfgs'
max_iter	Maximum number of iterations	1000

Hyperparameter	Description	Values
	for solver convergence	

Comparing the computational demands of the model to the deep learning models used in the project, logistic regression is significantly less demanding. However, since transformer word embedding is being used, more computational time is needed. For the project, Google Colab Pro was used, which provided a variety of memory options, including High-RAM and V100 for training and evaluating the model.

4.5 CNN+Bi-LSTM

The experimental design for the CNN+Bi-LSTM model employs hyperparameter tuning on the training data. Optimal parameters were established with a learning rate of 0.001 and a batch size of 32. The Adam optimizer was selected for its capacity to efficiently manage sparse gradients and automatically adjust the learning rate, thus removing manual tuning. Training is conducted for up to 10 epochs within each active learning iteration; a balance is struck to allow adequate learning while avoiding overfitting, further augmented by early stopping mechanisms using the validation set.

For activation functions, the model utilizes the ReLU function in the Convolutional and initial Dense layers due to their ability to process non-linear data and computational speed. For the binary classification task—distinguishing between real and fake news—the output layer employs a Sigmoid function, mapping the final decision to a probability score.

The model comprises seven layers, including a dropout layer implemented for regularization by reducing overfitting. The neuron count stands at 128 for the Convolutional

layer, 128 total for the Bidirectional LSTM layer (with 64 in each direction), and 64 neurons in the first Dense layer, leading up to a single neuron for the output Dense layer. Overall, the model consists of 3,259,265 parameters; 2,929,265 are trainable and updated during training, while the remaining 3,260,000 are non-trainable, fixed to the pre-trained Word2Vec embeddings.

An A-100 GPU provided the computational backend for the model's training through the Google Colab Pro + service. The execution of 30 random_state tests was completed in approximately 4 hours. For development, TensorFlow and its high-level API counterpart, Keras, were employed, both at version 2.14.0. This setup ensures a robust and consistent modeling environment, crucial for reproducibility and maintenance of performance standards in the machine learning workflow.

4.6 Llama 2

After loading the model, as previously mentioned, we establish the Parameter-Efficient Fine-Tuning (PEFT) configuration specifically for the LoRA modules. This approach focuses on updating only a select subset of the model's parameters, greatly enhancing efficiency. With this PEFT configuration applied, the model is now primed and ready to commence training. Table 8 below outlines a range of hyperparameters utilized in the fine-tuning process.

Table 8

Hyperparameters Used for Fine-tuning

QLoRA and trainable arguments parameters		
lora_r	8	LoRA attention dimension
lora_alpha	64	Alpha parameter for LoRA scaling

lora_dropout	0.1	Dropout probability for LoRA layers
task_type	"CAUSAL_LM"	Task type
per_device_train_batch_size	1	Batch size per GPU for training
learning_rate	5e-4	Initial learning rate (AdamW optimizer)
optim	"paged_adamw_32bit"	Optimizer to use
gradient_accumulation_steps	8	Number of update steps to accumulate the gradients for

The Figure 60 below is the summary of a neural network's training parameters, specifically mentioning LoRA (Low-Rank Adaptation) module names which are parts of the network architecture likely used to adapt pre-trained models with fewer trainable parameters. These modules include 'q_proj', 'k_proj', 'v_proj', 'gate_proj', 'o_proj', and 'down_proj', which are typically related to the query, key, value, and output projections in the attention mechanism of a transformer model. The total number of parameters in the model is 3,520,401,408, but only a small fraction, 19,988,480 parameters, are trainable which is about 0.567% of the total. This suggests that most of the model's parameters are frozen and only a small, targeted subset is being updated during training. The use of LoRA and such a small percentage of trainable parameters

are techniques to efficiently fine-tune large models without the need to update all parameters, thereby saving computational resources and time.

Figure 60

Trainable Parameters

```
LoRA module names: ['q_proj', 'k_proj', 'up_proj', 'v_proj', 'gate_proj', 'o_proj', 'down_proj']
All Parameters: 3,520,401,408 || Trainable Parameters: 19,988,480 || Trainable Parameters %: 0.5677897967708119
Training...
```

After completing 200 max_steps in the training process, there is a notable decline in the training loss, indicating effective learning by the model. Figure 61 below illustrates the performance metrics for the refined model which shows the training loss as 2.8. The model has been uploaded to Hugging Face for subsequent inference processes.

Figure 61

Training Metrics

```
***** train metrics *****
epoch                =      0.05
total_flos           = 112315296F
train_loss           =      2.8015
train_runtime        = 0:15:19.49
train_samples_per_second =      1.74
train_steps_per_second  =      0.218
{'train_runtime': 919.4928, 'train_samples_per_second': 1.74, 'train_steps_per_second': 0.218, 'total_flos': 1.2059762769051648e+16, 'train_loss': 2.8014697}
Saving last checkpoint of the model...
```

The model fine-tuning process was carried out using the high-performance A100 GPU available on Google Colab Pro Plus. Additionally, a personal computer configuration was employed, featuring an Intel(R) Core(TM) i5-8250U CPU, capable of speeds from 1.60GHz to 1.80 GHz, complemented by 8.00 GB of RAM. This PC runs a 64-bit operating system on an x64-based processor and includes an Intel UHD Graphics 620 GPU. The software libraries used for this model are detailed in Table 9 below

Table 9

Software Libraries Used for Llama2

Library	version	Description
---------	---------	-------------

accelerate	0.21.0	A library from Hugging Face that simplifies running machine learning models on distributed hardware
peft	0.4.0	The library streamlines adapting large pre-trained models to specific tasks with minimal parameter tuning, reducing both computational and storage demands.
bitsandbytes	0.40.2	A library for training neural networks with 8-bit optimizers, which can help reduce memory usage and increase the training speed.
transformers	4.31.0	A popular library by Hugging Face that provides general-purpose architectures for Natural Language Processing.

5. Results and Analysis:

5.1 DNN

The table 10 below shows the performance of the five embedding and DNN model combinations. The metric that we used for the experiments is accuracy score as it is a straightforward and easy-to-understand metric. Observe that BERT performed poorly compared to other models, which could relate to the small padding length limitation. BERT has achieved SOTA in many NLP downstream tasks, therefore having a machine with plentiful computational resources may potentially improve its performance. All combinations except for BERT performed well and their accuracy scores were not very different from each other. Another outstanding point is that the length attribute didn't significantly affect the results even when all indexes did slightly increase when the attribute existed. The best combination was Word2Vec + DNN as it achieved the highest accuracy score among all. For possible improvements in the future, investing in GPU memory can facilitate the increment of iterations, epochs, embedding dimensions, hyperparameters search, and so forth. Another possible way to boost models is to reconstruct the DNN architecture with a denser structure but it might lead to overfitting.

Table 10

Embedding Layer + DNN Combination Performance

Combination	Accuracy score on test dataset	Accuracy score on test dataset
	(without the length attribute)	(with the length attribute)
Doc2Vec + DNN	0.9460	0.9471
Glove + DNN	0.9324	0.9257
Word2Vec + DNN	0.9600	0.9648
FastText + DNN	0.9502	0.9508
BERT + DNN	0.5481	0.5481

Next, we implemented active learning to see how good the models performed with a small portion of the training dataset. We set up 10 iterations and 10 epochs for each iteration. We started with 100 random data points in the training set, then in each iteration, we continued to train the model on the 100 most uncertain points in each loop. As a result, the models gained 1100 points in total. The table 11 below presents the active learning performance of each model. With a remarkably small amount of training points (1100 rows compared with the training dataset of than 30,098 rows), three models were able to obtain impressive results of more than 82% accuracy score. Similar to the normal training process, BERT+DNN underperformed with only a 45% accuracy score, and Word2Vec+DNN once again was the best combination with the best performance among all combinations. We still have room to train more points to reach a specific level of accuracy.

Table 11

Embedding Layer + DNN Combination Applying Active Learning Performance

Combination	Accuracy score
Doc2Vec + DNN	0.8721
Glove + DNN	0.8203
Word2Vec + DNN	0.8974
FastText + DNN	0.7608
BERT + DNN	0.4519

5.2 LSTM

Following a thorough process of hyperparameter tuning with different word embeddings for our LSTM model, we noticed clear differences in performance. As a comparative benchmark, the baseline model with no pre-trained embeddings obtained an accuracy of 95.8%. With an accuracy of 98.1%, the GloVe-embedded model outperformed the others, because it could take advantage of the combined global word-word co-occurrence statistics from a corpus. Word2Vec embeddings produced a strong accuracy of 95.9% by taking into account the context of individual words. On the other hand, the accuracy of the Doc2Vec embeddings, which take into consideration the document's semantics and word order, was only 93.46%. The accuracy of the FastText embeddings, which were created to comprehend subword information, was 96.2%.

Notably, the BERT embeddings achieved a nearly competitive accuracy of 97.64%, capturing deep contextual relationships between words. These findings demonstrate the

effectiveness of GloVe embeddings within the framework of our model and emphasize how crucial it is to select the appropriate embeddings in order to improve model performance.

Table 12 consolidates the results obtained from different word embedding techniques combined with the LSTM model.

Table 12

Performance comparison of the Different Word Embeddings combined with LSTM Model

Embeddings	Accuracy
Baseline	95.8%
GloVe +LSTM	98.1%
Word2Vec + LSTM	95.9%
Doc2Vec + LSTM	93.46%
FastText + LSTM	95.33%
BERT + LSTM	97.64%

5.3 Bi-LSTM

The results of five embedding models trained on Bi-LSTM is shown in Figure 62. All the embeddings performed considerably well. The strength of the model lies in its simplicity with very less number of units and layers, which helps with less number of learnable parameters and faster execution. The models with number of words performed similarly without the number of word features as the number of words doesn't affect the dimensions much because in the embedding layer the number of trainable parameters correspond to the tokenizer's vocabulary multiplied with the output size of the vector that is a hyperparameter. Therefore, adding a number to the sequence of the text will only increase the `max_length` parameter by one and won't increase the size of the vocabulary. Due to this observation, the `number_of_word` feature was not added and executed for BERT and Doc2Vec embeddings. Fasttext without the addition of words performed the best amongst all the models. It works on breaking the words into many small parts, leading to better morphological results. Confusion Matrix, Classification Reports and Accuracies of all the models are generated Figure 63 shows the Confusion Matrix of FastText Model without addition of the number of words. Figure 64 shows the Classification report and Figure 65 shows the plot of the Roc-AUC curve. AUC is 99%. As per classification, recall is 98% which is performing well for Fake classification. As per confusion matrix, only 39 instances have been tagged as Real instead of Fake (False Negative). The model can be made better by including data from more sources. Also, increasing the computational budget may help with better searching of the hyper-parametric space. The BERT model was only trained on five epochs. Increasing GPU units might have increased its performance. We could have tried to see the model's performance by increasing complexity to verify if the overfitting occurs or not.

Figure 62

Results in Descending Order of Accuracy

Embedding	Model	Accuracy
FastText_without_num	Bi-LSTM	98.651165
Word2Vec_num	Bi-LSTM	98.418605
GloVe_num	Bi-LSTM	98.360467
Word2Vec_without_num	Bi-LSTM	98.302323
FastText_num	Bi-LSTM	98.232555
GloVe_without_num	Bi-LSTM	98.197675
Bert	Bi-LSTM	97.653341
Doc2Vec	Bi-LSTM	90.046512

Figure 63

Confusion Matrix of Fasttext with Bi-LSTM

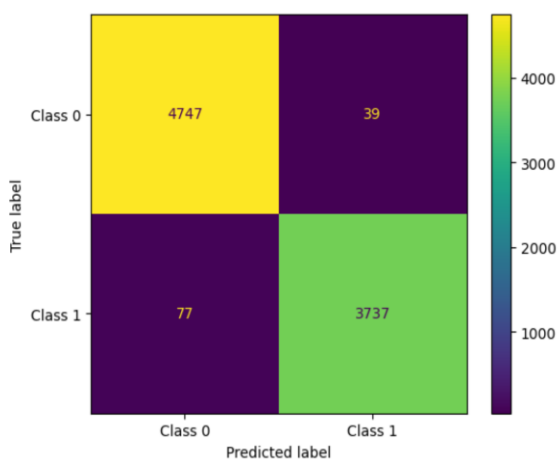


Figure 64

Classification Report of Fasttext with Bi-LSTM

```

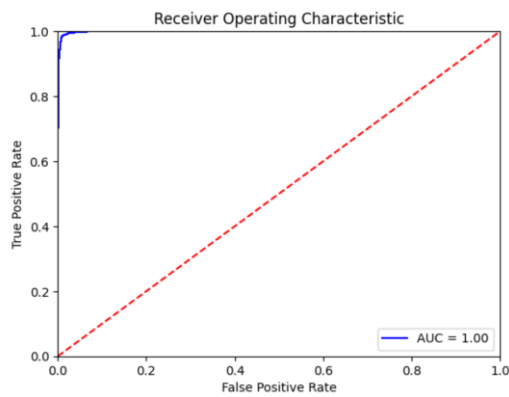
model_name Bi-LSTM with FastText has accuracy of 0.9865116477012634
269/269 [=====] - 60s 222ms/step
Classification Report for Bi-LSTM with FastText:

```

	precision	recall	f1-score	support
0	0.98	0.99	0.99	4786
1	0.99	0.98	0.98	3814
accuracy			0.99	8600
macro avg	0.99	0.99	0.99	8600
weighted avg	0.99	0.99	0.99	8600

Figure 65

Roc-AUC of Fasttext with Bi-LSTM



5.4 Logistic regression

This section consolidates the performance metrics for different word embedding techniques that are used as classifiers, then followed by logistic regression. Word2Vec attained a 92.94% accuracy rate. A marginal gain from incorporating broader textual context is suggested by the slightly higher accuracy of 93.09% displayed by the Doc2Vec algorithm, which takes the document-level context into account. The accuracy of FastText, which adds subword information to Word2Vec, was 90.01%.

Following a thorough grid search for optimizing hyperparameters on a logistic regression model that used the sentence bert word embedding with C: 100 and solver: liblinear were found to be the ideal parameters. With a cross-validated score of roughly 0.9355, this combination produced the best results, suggesting that the model performed well during training. This model

configuration's effectiveness was further validated by its accuracy of approximately 93.84% when tested on the test dataset. The model's performance is thoroughly explained in the classification report as shown in Figure 66 , which displays excellent recall, precision, and F1-score for both classes 0 and 1 making . For classes 0 and 1, in particular, the model showed precision values of 0.94 and 0.93, respectively, along with matching recall rates. Which makes sentence BERT word embedding used with logistic regression proving to be the best performing model. The classification report of the alone with the best chosen parameters for sentence BERT is as mentioned in the Figure 66

Figure 66

Results of the Sentence BERT and Logistic Regression Model

```
Best Parameters: {'C': 100, 'solver': 'liblinear'}
Best Score: 0.9354903405037914
Test Accuracy: 0.9383720930232559
Classification Report:
```

	precision	recall	f1-score	support
0	0.94	0.95	0.94	4786
1	0.93	0.93	0.93	3814
accuracy			0.94	8600
macro avg	0.94	0.94	0.94	8600
weighted avg	0.94	0.94	0.94	8600

The macro and weighted averages demonstrate how these metrics are balanced between the two classes, highlighting the model's strong generalization capabilities and ability to perform consistently across the dataset's various categories. The logistic regression model, with the identified hyperparameters, appears to be highly effective for the classification task at hand, based on its high accuracy and balanced precision and recall. The Table 13 below provides the

overall comparison of performance between each word embedding combined with the Logistic regression model

Table 13

Evaluation Comparison of the Word Embeddings with the Logistic Regression Model

Embeddings	Accuracy
Sentence BERT: Before hyperparameter:	93%
After Hyperparameter tuning:	93.83%
Word2Vec	92.94%
Doc2Vec	93.09%
FastText	90.01%
GloVe	92.22%

5.5 CNN+Bi-LSTM

Accuracy is the chosen metric for the performance evaluation of the CNN+Bi-LSTM active learning model. The model undergoes 30 runs with varying random states, a strategy that helps assess its stability and performance under different conditions, similar to cross-validation. The active learning component includes specific parameters like the number of queries, initial training size, patience (early stopping), and tolerance (stopping active learning due to accuracy not improving on validation data for five queries), all of which contribute to the model's learning

process. Finally, the model's effectiveness is evaluated on a test set post the active learning loop, providing insights into its generalization capabilities.

In evaluating the CNN+Bi-LSTM model's performance across 30 iterations with varying `random_state` values, several key insights emerge that reflect the model's effectiveness and reliability. The mean accuracy achieved by the model is 85.34%, which is quite impressive, indicating a high level of proficiency in classifying the text data on just 42.8 samples with labels. This level of accuracy, especially in a complex task like text classification, demonstrates the model's robustness and the efficacy of the combined CNN+Bi-LSTM architecture along with the active learning approach. However, the standard deviation of 04.35% in accuracy across different iterations suggests some variability in performance. This variability could be attributed to the tolerance threshold on the validation accuracy to prevent the model from overfitting as it continues performing queries. The table 14 below shows the model performance on the test data over the span of 30 iterations of the random_state variable.

Table 14

CNN+Bi-LSTM Results on Test Data for 30 Iterations

Embeddings	Accuracy
Initial Training Set Size	20 labeled samples
Initial Pool Size	30,078 unlabeled samples
Iterations of Random_State	30
Mean Accuracy	85.34%
Mean Accuracy SD	4.35%
Mean Training Set Size	42.8 labeled samples

Embeddings	Accuracy
Mean Training Set Size SD	12.39 labeled samples

Another important aspect is the mean training set size, which stands at 42.8 samples with a standard deviation of 12.39 samples. This result is particularly interesting as it starts from an initial size of just 20 samples. The increased training set size due to active learning illustrates the model's capability to query and learn from the most informative samples selectively. The standard deviation in training set size indicates variability in the number of queries needed across different iterations to reach optimal performance. This is a natural outcome of the active learning process where each iteration might converge at different rates based on the initial data subset.

Overall, these results underscore the model's success in adapting and learning from a dynamically changing training set, achieved through the active learning framework. The high mean accuracy points to the model's effectiveness in Fake News text classification, while the standard deviation in both accuracy and training set size highlights the impact of initial data selection and the inherent uncertainty in the learning process. This analysis suggests that while the model is generally effective, there could be room for further optimization, particularly in reducing performance variability across iterations.

5.6 Llama 2

As previously discussed, the test dataset utilizes a unique prompt template which excludes the response, as the response itself is the intended target value. An example of this test dataset is illustrated in Figure 67. For the purpose of model evaluation, the initial step involves loading the model and tokenizer from Hugging Face, using a write access token. The specific model used for this task is available on Hugging Face under the name "jua2020/fakenews-binaryclassification-version7-llama-2-7b".

Figure 67*Test Dataset Sample*

```

print(formatted_dataset_sample)

### Instruction:
Classify the Input text as fake or real news. Response will be 0 for real, or 1 for fake.

### Input:
washington reuters republican led senate tuesday ruled taking action nominee put forth president barack obama supreme court political power move intended thwart ability change court ideological ba

### Response:

```

Due to the high computational demands of testing on a large dataset, this project utilized a subset of 1000 rows from the test dataset for evaluation. Initially, a single sample was examined to assess the model's response. The response of this sample is illustrated in Figure 68.

Figure 68*Sample Prediction*

```

print("Input Text:\n(formatted_dataset_sample)\n")
print("Generated Prediction:\n(tokenizer.batch_decode(outputs.detach().cpu().numpy(), skip_special_tokens=True)[0][len(formatted_dataset_sample):])")

Input Text:
### Instruction:
Classify the Input text as fake or real news. Response will be 0 for real, or 1 for fake.

### Input:
washington reuters republican led senate tuesday ruled taking action nominee put forth president barack obama supreme court political power move intended thwart ability change court ideological ba

### Response:

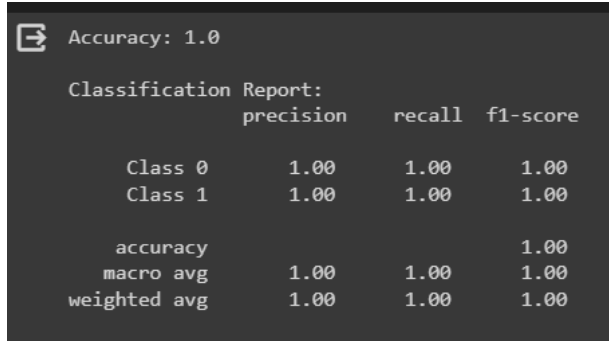
Generated Prediction:
0

```

The process begins by expanding the number of rows to 12, followed by an increase to 1000, to evaluate the predictions. Outputs are produced using the model's `generate` method, which requires `input_ids` as input. These outputs are then passed through tokenizer's `batch_decode` method to get the final predictions. A loop iterates over the test set, consisting of 1000 rows. Considering the high cost of utilizing an A100 GPU due to limited computational resources, this project effectively uses 1000 rows, achieving 100% accuracy on the test set. It's noted that the performance remains consistent regardless of the number of rows processed. The accuracy and classification report are displayed in Figure 69.

Figure 69

Classification Report for LLama2



```

Accuracy: 1.00

Classification Report:
      precision    recall  f1-score   support

   Class 0       1.00      1.00      1.00        10
   Class 1       1.00      1.00      1.00        10

   accuracy               1.00
  macro avg              1.00
 weighted avg              1.00
  
```

One of the key strengths of the llama2 model is its pre-training on 7 billion parameters, which has shown to yield good performance, especially notable when trained with a maximum of 200 steps. However, a significant limitation of this model is its exclusive compatibility with the A100 GPU, which is difficult to access on platforms like Colab, even with the Colab Pro Plus subscription. Additionally, the model demands high RAM and memory resources during fine-tuning, a requirement that persists even when using a quantized version of the model.

6. Conclusion

6.1.1 Project Summary

This project is focused on developing an advanced approach for detecting fake news. The team employed three sequential models — DNN, Bi-LSTM, and LSTM — alongside a traditional machine learning technique, logistic regression. In addition, the large language model LLaMA-2 was fine-tuned for performance comparison. The project also incorporated an innovative active learning strategy using a CNN+Bi-LSTM model. Various word embeddings, including Word2Vec, Doc2Vec, GloVe, FastText, and BERT, were utilized for the sequential models, while Sentence BERT was applied to logistic regression. The active learning model used a pre-trained Word2Vec model. LLaMA-2, leveraging dynamic, context-dependent embeddings from its architecture, achieved the highest accuracy of 99%, with Bi-LSTM following closely

behind. This outcome highlights the superior performance of fine-tuned large language models, which are also efficiently implementable on single GPUs using QLoRA and PEFT libraries from the Hugging Face Transformers library. The high accuracy achieved, especially with the LLaMA-2 model, demonstrates the potential of AI in enhancing our ability to discern truth in the vast sea of digital information.

6.1.2 Result discussion

Our Llama2 model has been able to achieve a 99% accuracy score, which is remarkable compared with the current SOTA results on the fake news detection task. Other models such as DNN, Logistic Regression, LSTM, and Bi-LSTM also perform well with the right combination, indicating the importance of embedding and hyperparameter choices.

For the case of BERT embedding and DNN combination where the model underperforms, the result may highlight the trade-off between model performance and computational feasibility. Choosing an embedding approach that strikes a balance between accuracy and efficiency is crucial for real-world applications.

On the other hand, active learning models are able to achieve outstanding performance with only few data points, which is meaningful for cost-effective labeling. However, it often requires human feedback in the loop to guide the model's selection of informative instances. This collaboration between human annotators and the model ensures that domain expertise is leveraged effectively.

6.1.3 Future work

Incorporating phrases and multi-word expressions into the vocabulary can enhance the language model's ability to understand and generate coherent and contextually relevant text, such as contextual understanding, handling slang and informal language, reducing ambiguity,

language variations, etc. As a result, we may include phrases along with just single word tokens for building vocabulary in future.

Even when our data is a combination of two popular datasets, we could increase the training corpus by scraping more data and include spelling checks in pre-processing steps if news data is scrapped from not-so-popular sites. Obtaining labeled datasets for fake news detection can be challenging due to the subjective nature of labeling and the constantly changing landscape of misinformation. Therefore, cross-check data labels with reliable fact-check applications is an option for future work.

Another add-ons to consider is to improve the speed and efficiency of fake news detection systems for real-time applications, especially in the context of social media platforms where information spreads rapidly. Future work could focus on optimizing models for faster inference without compromising accuracy. In the meantime, developing fake news detection models that can adapt and learn from new information over time is important. Incremental learning or lifelong learning approaches can enable models to continuously update their knowledge and adapt to evolving patterns of misinformation.

References

- Ahmed, H., Traoré, I., & Saad, S. (2017). Detecting opinion spams and fake news using text classification. *Security and Privacy*, 1(1). <https://doi.org/10.1002/spy2.9>
- Ahmed, H., Traoré, I., & Saad, S. (2017a). Detection of online fake news using N-Gram analysis and machine learning techniques. In *Lecture Notes in Computer Science* (pp. 127–138). https://doi.org/10.1007/978-3-319-69155-8_9
- Bharadwaj, G., Manikanta, N. S., Bishi, M. R., Teja, P., Rao, G. R. K., & Srinivas, P. V. V. S. (2023). Uncovering the Truth: Exploring Traditional Deep Learning Techniques for Fabricated News Detection. 2023 2nd International Conference on Edge Computing and Applications (ICECAA)(pp. 714-723).IEEE.
<https://doi.org/10.1109/icecaa58104.2023.10212337>
- CheeKean. (2023, November 6). Understanding Llama2: KV cache, grouped query attention, rotary embedding and more. Medium. <https://ai.plainenglish.io/understanding-llama2-kv-cache-grouped-query-attention-rotary-embedding-and-more-c17e5f49a6d7>

Chen, B., Chen, B., Gao, D., Chen, Q., Huo, C., Meng, X., Ren, W., & Zhou, Y. (2021).

Transformer-Based Language Model Fine-Tuning Methods for COVID-19 Fake news Detection. In *Communications in computer and information science* (pp. 83–92).

https://doi.org/10.1007/978-3-030-73696-5_9

Heller, M. (2023, September 12). *What is Llama 2? Meta's large language model explained.*

InfoWorld. <https://www.infoworld.com/article/3706470/what-is-llama-2-metas-large-language-model->

[explained.html#:~:text=Llama%20%20is%20a%20family,fine%2Dtuned%20for%20programming%20tasks](https://www.infoworld.com/article/3706470/what-is-llama-2-metas-large-language-model-explained.html#:~:text=Llama%20%20is%20a%20family,fine%2Dtuned%20for%20programming%20tasks).

Hiramath, C. K., & Deshpande, G. C. (2019, July). Fake news detection using deep learning techniques. In *2019 1st International Conference on Advances in Information Technology (ICAIT)* (pp. 411-415). IEEE.

<https://doi.org/10.1109/ICAIT47043.2019.8987258>

GeorgeMcIntire. (n.d.). *GitHub - GeorgeMcIntire/fake_real_news_dataset*. GitHub.

https://github.com/GeorgeMcIntire/fake_real_news_dataset

Jaiswal, A., Verma, H., & Sachdeva, N. (2023). Swarm optimized Fake News Detection on Social-media textual content using Deep Learning. *2023 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI)*.

<https://doi.org/10.1109/accai58221.2023.10201229>

Kanchana, M., Kumar, V. M., Anish, T. P., & Gopirajan, P. (2023, April). Deep Fake BERT: Efficient Online Fake News Detection System. In *2023 International Conference on Networking and Communications (ICNWC)* (pp. 1-6). IEEE.

<https://doi.org/10.1109/ICNWC57852.2023.10127560>

- Mahara, G. S., & Gangele, S. (2022). Fake news detection: A RNN-LSTM, Bi-LSTM based deep learning approach. *2022 IEEE 1st International Conference on Data, Decision and Systems (ICDDS)*. <https://doi.org/10.1109/icdds56399.2022.10037403>
- Mahara, T., Josephine, V. L. H., Srinivasan, R., P. P., Algarni, A. D., & Verma, O. P. (2023). Deep vs. Shallow: A Comparative Study of Machine Learning and Deep Learning Approaches for Fake Health News Detection. *IEEE Access*, 11, 79330–79340. <https://doi.org/10.1109/access.2023.3298441>
- Odsc, G. M., Odsc, G. M., & Odsc, G. M. (2018, April 18). *How to build a “Fake news” classification model*. Open Data Science - Your News Source for AI, Machine Learning & More -. <https://opendatascience.com/how-to-build-a-fake-news-classification-model/>
- Pennington, J. (n.d.). *GLOVE: Global Vectors for Word Representation*. <https://nlp.stanford.edu/projects/glove/>
- Rana, V. K., Garg, V., Mago, Y., & Bhat, A. (2023). Compact BERT-Based Multi-Models for Efficient Fake News Detection. *2023 3rd International Conference on Intelligent Technologies (CONIT)*. <https://doi.org/10.1109/conit59222.2023.10205773>
- Settles, B. (2009). Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison. Retrieved from <http://burrsettles.com/pub/settles.activelearning.pdf>
- Shu, K., Mahudeswaran, D., Wang, S., Lee, D., & Liu, H. (2020). FakeNewsNet: A data repository with news content, social context, and spatiotemporal information for studying fake news on social media. *Big Data*, 8(3), 171–188. <https://doi.org/10.1089/big.2020.0062>

- Shu, K., Mahudeswaran, D., Wang, S., Lee, D., & Liu, H. (2019). Fakenewsnet: A data repository with news content, social context and dynamic information for studying fake news on social media. arXiv preprint arXiv:1809.01286. <https://arxiv.org/abs/1809.01286>
- Shu, K., Sliva, A., Wang, S., Tang, J., & Liu, H. (2017). Fake news detection on social media: A data mining perspective. ACM SIGKDD explorations newsletter, 19(1), 22-36. <https://arxiv.org/abs/1708.01967>
- Shu, K., Wang, S., & Liu, H. (2017). Exploiting tri-relationship for fake news detection. arXiv preprint arXiv:1712.07709, 8. <https://arxiv.org/abs/1712.07709>
- Thota, A., Tilak, P., Ahluwalia, S., & Lohia, N. (2018) "Fake News Detection: A Deep Learning Approach," SMU Data Science Review: Vol. 1: No. 3, Article 10. <https://scholar.smu.edu/cgi/viewcontent.cgi?article=1036&context=datasciencereview>
- Wiki word vectors · fastText*. (n.d.). <https://fasttext.cc/docs/en/pretrained-vectors.html>
- Xavier, J., Kumar, S., & Chandran, P. (2021). Characterization, classification and detection of fake news in online social media networks. 2021 IEEE Mysore Sub Section International Conference (MysuruCon). <https://doi.org/10.1109/mysurucon52639.2021.9641517>