Dhirubhai Ambani University
Technology

Formerly DA-IICT

Prof. Purbasha Das

Name : Nishil Patel

SID : 202201166

Name : Jainil Patel

SID : 202201030

Fake News Detector:

What We're Trying to Do

For this project, our main goal was to figure out if news articles are real or fake. We tackled this using a bunch of techniques from Natural Language Processing (NLP) and different types of Machine Learning (ML) and Deep Learning (DL). We didn't just stick to one approach; we built and tested several models –some classic machine learning ones, a deep learning model that understands words really well using GloVe, and even a super advanced one based on BERT (a transformer model). Finally, we put all our best models together into an "ensemble" to get even better and more reliable results.

How We Built It: A Step-by-Step Guide

Step 1: Gathering Our Tools (Libraries)

First we needed to import all the necessary libraries. These are basically the toolkits that let us do everything from handling data to building our models and even making nice visualisations.

- **For Data:** `pandas` and `numpy` were our go-to for organizing and crunching numbers.
- **For Visuals:** `matplotlib.pyplot` and `seaborn` helped us see what our data looked like.
- **For Text Prep:** `nltk` was crucial for breaking down sentences, getting rid of common filler words, and reducing words to their base forms (like changing "running" to "run").
- **For Models:** `sklearn`, `tensorflow.keras`, and `transformers` were essential for actually building and running our AI models.
- **Other Handy Bits:** We also used `Counter`, `re`, `string`, and `os` for various bits of text manipulation and file handling.

These libraries formed the backbone of our entire project pipeline, from reading raw data to getting our final predictions.

Step 2: Getting Our Data Ready

We worked with two main datasets:

- `DataSet_Misinfo_FAKE.csv`: We labeled all the articles in this one as `0` (fake).

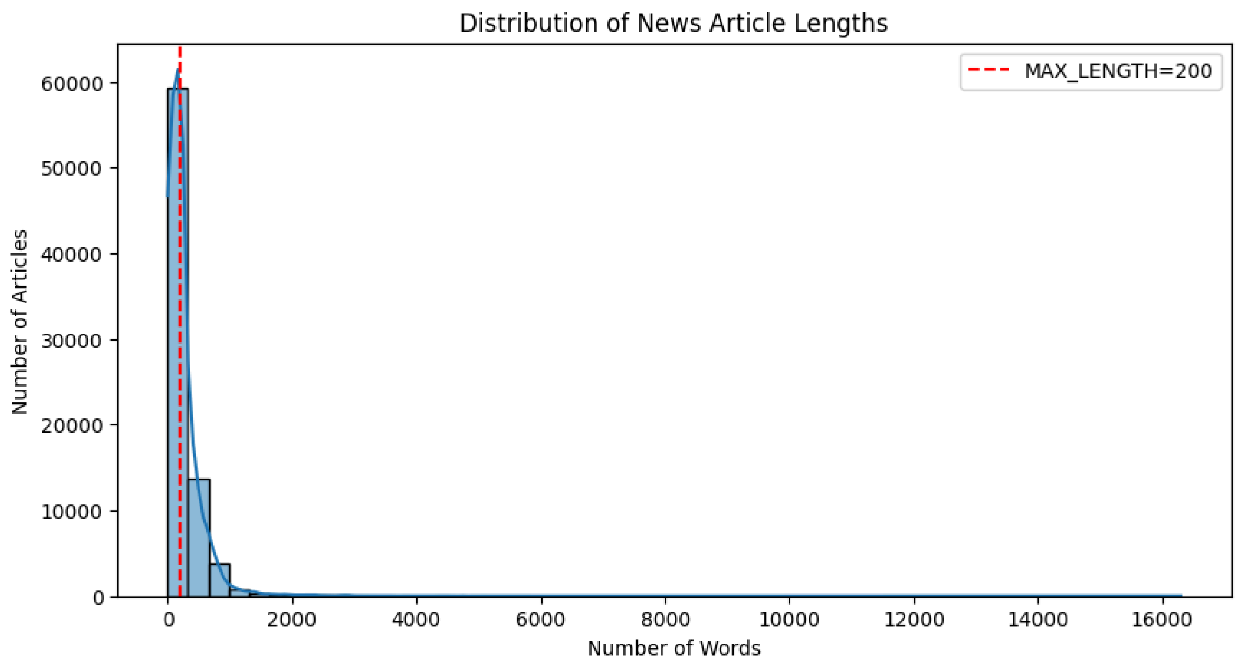
- DataSet_Misinfo_TRUE.csv: And these articles were labeled as 1 (real).

We then combined both into a single DataFrame (df) and gave it a good shuffle. Why shuffle? Because if all the fake news was at the top and real news at the bottom, our models might learn that order instead of the actual patterns. Shuffling makes sure our training and testing are fair and unbiased.

Step 3: Peeking at Our Data (EDA)

Before jumping into models, we spent some time doing Exploratory Data Analysis (EDA) to understand our data better.

- **Checking Fake vs. Real Balance:** We used sns.countplot to see how many fake versus real news articles we had. Luckily, our dataset was pretty balanced, which is great for building a classification model that tells one apart from the other.
- **Looking at Text Lengths:** We added a column called text_length to our data. This helped us understand how long, on average, our news articles were. This was super useful because it guided our decision to set maxlen=300 when preparing text for our deep learning models, ensuring all text inputs had a consistent length.



Step 4: Cleaning Up the Text Mess

Raw text is messy, so we needed to clean it up before feeding it to our models. We created a custom clean_text() function to do a few important things:

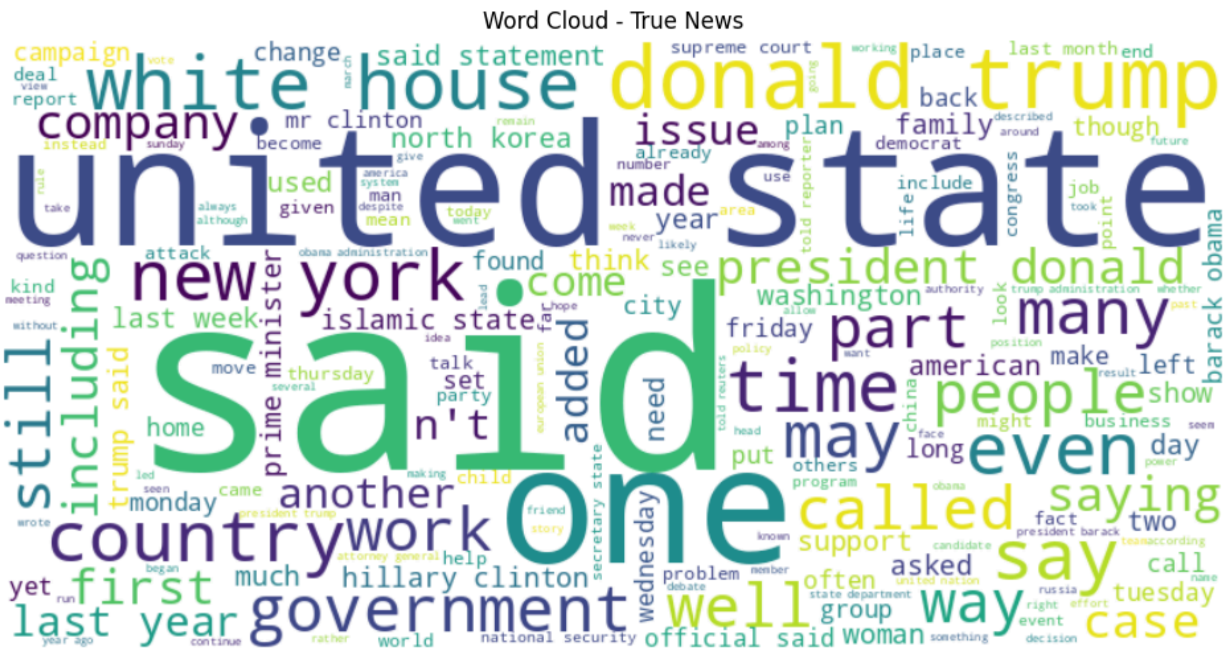
Transformation	**Why We Did It**
Lowercasing	To make sure "The" and "the" were treated as the same word.
Removing URLs/HTML Tags	To get rid of web addresses and code snippets that aren't relevant to news content.
Removing Punctuation	To simplify the vocabulary and focus on words, not symbols.
Stopword Removal	To ditch common words like "a," "the," "is" that don't add much meaning.
Lemmatization	To reduce words to their base form (e.g., "running," "ran," "runs" all become "run").

We applied this cleaning process to every article's text, storing the results in a new column called `clean_text`.

Step 5: Digging Deeper into the Words

Once the text was clean, we wanted to see what kind of words popped up most often in fake versus real news.

- **Word Frequencies:** We used `collections.Counter` to list the top 20 most common words in both fake and real articles. This gave us a quick look at their core vocabulary.
- **Word Clouds:** We generated separate word clouds for fake and real news. These visual representations were really cool because they showed us at a glance the thematic differences in the language used in each type of news.
- Here's an example of a Word cloud for true news where the size of a word indicates it's frequency.



Step 6: Splitting Our Data for Training & Testing

To properly train and evaluate our models, we split our cleaned data into three parts:

- **Training Set:** 70% of our data, used to teach the models.
- **Validation Set:** 15% of our data, used during training to catch overfitting (when a model learns the training data too well but struggles with new data).
- **Test Set:** The final 15% of our data, kept completely separate until the very end to give us an unbiased idea of how well our models would perform on unseen articles.

We used `train_test_split()` and made sure to use `stratify=y`. This little trick ensures that the balance of fake and real news labels is maintained in each of our splits, which is important for fair evaluation.

Step 7: Our First Shot: Traditional Machine Learning

Converting Text to Numbers (TF-IDF)

For our traditional machine learning models, we needed to turn our clean text into numbers. We used `TfidfVectorizer(max_features=5000)` for this. TF-IDF (Term Frequency-Inverse Document Frequency) is smart because it not only counts how often a word appears but also considers how unique that word is across all documents. This helps give more importance to rare but significant words.

The Models We Tried

We trained a few different classical machine learning models:

- **Logistic Regression:** A straightforward model, but often surprisingly effective for this kind of binary classification.
- **Multinomial Naive Bayes:** Great for text data because it makes a simple assumption about how words relate to each other.
- **Support Vector Machine (LinearSVC):** Known for performing well even with lots of features, which is common in text data.

How They Performed

We evaluated each model using `classification_report()`, which gave us detailed metrics like:

- **Accuracy:** Overall correct predictions.
- **Precision:** How many of the predicted "fake" were actually fake.
- **Recall:** How many of the actual fake news items our model caught.
- **F1-Score:** A balance between precision and recall.

All these models showed decent performance. Naive Bayes was the fastest to train, and Logistic Regression managed a good balance between precision and recall.

Step 8: Making Predictions with Our Traditional Models

We created a handy function, `predict_article(text)`, to quickly test any new piece of text. This function does all the prep work: it cleans the text, transforms it using our TF-IDF vectorizer, then gets predictions from all three classical models, showing us the predicted label and the probability.

Step 9: Level Up! Deep Learning with GloVe

Next, we moved into deep learning, starting with GloVe embeddings.

Getting Text Ready for Deep Learning

Deep learning models need numbers, so we used `Tokenizer(num_words=20000)` to build a vocabulary from our text. Then, we made sure all our text sequences were the same length (300 words) using `pad_sequences()`.

Using Pre-trained Word Meanings (GloVe)

Instead of teaching our model word meanings from scratch, we used `glove.6B.300d.txt`. This file contains "GloVe embeddings," which are pre-calculated 300-dimensional numerical representations of words that capture their semantic relationships. We created an `embedding_matrix` to align these GloVe vectors with the words in our own vocabulary.

Our Deep Learning Model's Brains

Here's how our GloVe-based deep learning model was structured:

```
Embedding Layer (using GloVe pre-trained weights)

 ↓

GlobalAveragePooling1D (to flatten the embeddings)

 ↓

Dense Layer (128 units, ReLU activation)

 ↓

Dropout (to prevent overfitting)

 ↓

Dense Layer (64 units, ReLU activation)

 ↓

Dropout (more overfitting prevention)

 ↓

Output Layer (single unit, sigmoid activation for binary classification)
```

- **Loss Function:** We used Binary Crossentropy, common for two-class problems.
- **Optimizer:** We chose Adam, a popular and effective optimizer.
- **Metrics:** We tracked Accuracy to see how well it was doing.

How Our GloVe Model Did

We kept a close eye on the validation set during training to make sure our model wasn't just memorizing the training data. Thanks to techniques like dropout layers and early stopping, we achieved good generalization – meaning it performed well on new, unseen data too.

Step 10: Predicting with Our GloVe Model

We also built a `predict_news_article(text)` function for our GloVe model. This function handles cleaning and tokenizing the input text, then uses our GloVe-based model to predict whether it's fake or real news, along with a confidence score.

Step 11: The Big Gun: BERT (Transformer Model)

For our most advanced approach, we used BERT, a powerful transformer-based model.

BERT's Special Text Prep

BERT has its own way of handling text. We used HuggingFace's `BertTokenizer` to tokenize our input and make sure it was padded to 512 tokens. A neat trick with BERT is that we can use the embedding from its `[CLS]` token (which is added at the beginning of every input) as a summary representation of the entire input text.

How We Used BERT

We took that 768-dimensional `[CLS]` vector from the pre-trained `BertModel` and fed it into our own small neural network:

```
Input Layer (768-dimensional CLS vector)

 ↓

Dense Layer (128 units)

 ↓

Dropout (to fight overfitting)

 ↓

Dense Layer (64 units)

 ↓

Output Layer (single unit, sigmoid activation)
```

Making Predictions with BERT

Similar to our other models, we made a `predict_news_article_bert(text)` function. This one uses BERT to tokenize the text, generate the `[CLS]` embedding, and then passes it through our custom dense layers to get the fake/real classification and confidence.

Step 12: The Dream Team: Our Ensemble Model

Finally, to get the absolute best performance, we decided to combine the strengths of our top models into an "ensemble."

How We Combined Them All

Our ensemble takes predictions from:

- Naive Bayes (our fastest classical model)
- Logistic Regression (a balanced classical model)
- GloVe Deep Model (our first deep learning success)

We gave each model a specific "weight" based on how well it performed individually and how diverse its predictions were:

- Naive Bayes: 50%
- Logistic Regression: 10%
- GloVe Model: 40%

The final_score was calculated like this:

final_score = (0.5 * prob_nb) + (0.10 * prob_logreg) + (0.40 * prob_glove)

If the `final_score` was above 0.47, we classified it as real; otherwise, it was fake.

Our Ensemble Prediction Tool

The `predict_ensemble(text)` function ties everything together. It takes new text, cleans it, runs it through all the selected models, then combines their probabilities using our set weights. The final output is the ensemble's predicted label and its overall confidence.

Stuff You Need to Run This Project

To get our project up and running, you'll need these files:

File	What It's For

DataSet_Misinfo_FAKE.csv	Contains the fake news articles we used.
DataSet_Misinfo_TRUE.csv	Contains the real news articles we used.
glove.6B.300d.txt	The GloVe word embeddings file, essential for our deep learning model.

What Our System Can Do (Final Deliverables)

Here's a quick look at the different prediction outputs our system can give you:

Model	**How Text Is Converted**	**Model Type**	**F1 Score**	**Accuracy(in %)**
Naive Bayes	TF-IDF	Classical	Fake : 0.87 Real : 0.83	85%
Logistic Reg.	TF-IDF	Classical	Fake : 0.94 Real : 0.92	93%
GloVe DL	GloVe 300d Embeddings	Deep Neural Network		90.55%
BERT DL	BERT (CLS Token)	Transformer		91.80%

SVM	TF-IDF	Hybrid	Fake : 0.95 Real : 0.94	94%

Deployment: Putting It All Online with Hugging Face

Once we had all our models ready and working locally, we wanted to make the project usable by anyone even non-technical users. So we deployed it on Hugging Face using Gradio.

Here’s how we did it:

1. We created a new Hugging Face Space and selected Gradio as the interface.
2. We kept our main logic in app.py — this included the UI and all prediction functions.
3. We added all the necessary files:
    - model.pkl for our classical models
    - model.h5 for the GloVe-based deep learning model
    - requirements.txt with all the libraries used
4. We uploaded everything to the Hugging Face web interface (you could also push via Git).
5. As soon as we uploaded the files, Hugging Face automatically built and hosted our app.

Once it was live, anyone could visit the link, enter a news article, and instantly get predictions from our models — no installations, no setup. Just pure, click-and-go fake news detection.

Quick Wrap-up

In short, our project built a robust fake news detection system.

- We used **Classical Models** for their speed and straightforwardness.
- We incorporated **GloVe Embeddings** to give our models a semantic understanding of words.
- We leveraged **BERT Transformers** for deep contextual understanding, tapping into the latest in AI.
- And finally, we created an **Ensemble** to combine all their strengths for the best possible results.

LINKS :

Kaggle Dataset Link :

https://www.kaggle.com/datasets/stevenpeutz/misinformation-fake-news-text-dataset-79k?select=DataSet_Misinfo_FAKE.csv

Website Link :

https://jainilp30-fake-news-detector.hf.space/