Fake News & Sentiment Detection System Using NLP and Transformers

Abstract

The rapid growth of online platforms has made it much easier for misinformation and fake news to spread, My goal was to build a complete NLP pipeline that can detect fake news and also analyze the sentiment of the content.

I worked with the Kaggle Fake News dataset and started by cleaning the text through steps like removing stop words, tokenizing, and lemmatizing. then explored different text representation techniques, including Bag of word and TF-IDF and word embeddings.

To test the problem from different angles, I trained classical machine learning models such as Naïve Bayes, Logistic Regression, and SVM. After that, I moved on to deep sequence models like RNN, LSTM, and GRU, and finally fine-tuned transformer models such as BERT .

In addition to classifying news as fake or real, I also used sentiment analysis to understand whether the text carried a positive, negative, or neutral tone. I evaluated the models using accuracy, precision, recall, and F1-score, and compared their performance.

Finally, I deployed the system as a simple Streamlit web app where anyone can paste a headline or tweet and instantly see whether it’s likely fake or real, along with its sentiment.

1. Introduction

In the last decade, the way people consume information has changed dramatically. Social media platforms like Twitter, Facebook, and Reddit have replaced traditional newspapers as the primary source of news for millions of people. While this shift has made information more accessible than ever before, it has also opened the door for a serious problem: the rapid spread of misinformation and fake news.

Fake news is not just an annoyance it can have real consequences. We have seen examples where false headlines influence political elections, spread panic during health crises, or even damage reputations overnight. Once a fake story is shared widely, it becomes almost impossible to undo the damage. This makes the detection of fake news one of the most important challenges in modern information systems.

As someone passionate about data science and natural language processing (NLP), I wanted to see how technology could be used to address this issue. NLP provides a set of tools and methods that allow computers to understand and process human language. By analyzing the

words, tone, and structure of text, it is possible to find patterns that distinguish genuine news from fabricated content. This project was my attempt to explore those techniques and build a working system that could automatically classify news as either “real” or “fake.”

Another layer I found important to include was sentiment analysis. News is not just about facts—it also carries emotional weight. A headline might be factual but written in a way that stirs fear, anger, or hope. By analyzing sentiment, I can better understand the emotional impact of a piece of news in addition to its truthfulness. This combination—fake news detection plus sentiment analysis—offers a fuller picture of how information spreads and affects people.

To tackle this problem, I built a complete NLP pipeline. I began with traditional text preprocessing techniques such as cleaning the text, tokenization, lemmatization, and removing stop words. From there, I experimented with different representations of text, like Bag of Words and TF-IDF,

I then applied a wide range of models, from classical machine learning algorithms such as Naïve Bayes, Logistic Regression, and Support Vector Machines, to more advanced deep learning models like LSTM and GRU. Finally, I fine-tuned transformer-based models, including BERT which represent the state of the art in NLP today. Each stage gave me insights into the strengths and weaknesses of different approaches, and I was able to compare their performance in terms of accuracy, precision, recall, and F1-score.

My contribution in this project is not just about applying existing algorithms, but also about combining different ideas into a single system. Instead of building a fake news detector in isolation, I created a tool that also measures sentiment. Instead of stopping at model training, I deployed the system as a simple web application using Streamlit. This allows anyone to paste a headline or tweet and instantly see whether it is likely real or fake, as well as the sentiment behind it.

This project also made me reflect on the broader impact of such technology. Fake news detection is not only a technical problem it’s a social and ethical one. Models can carry bias depending on the dataset they were trained on. For example, a dataset collected mainly from Western media may not work well on news from other parts of the world. That’s why transparency and responsible use are essential. My system is a step toward addressing the technical challenge, but it also shows how important it is to think critically about where the data comes from and how the model’s decisions are explained to users.

2. Dataset Description

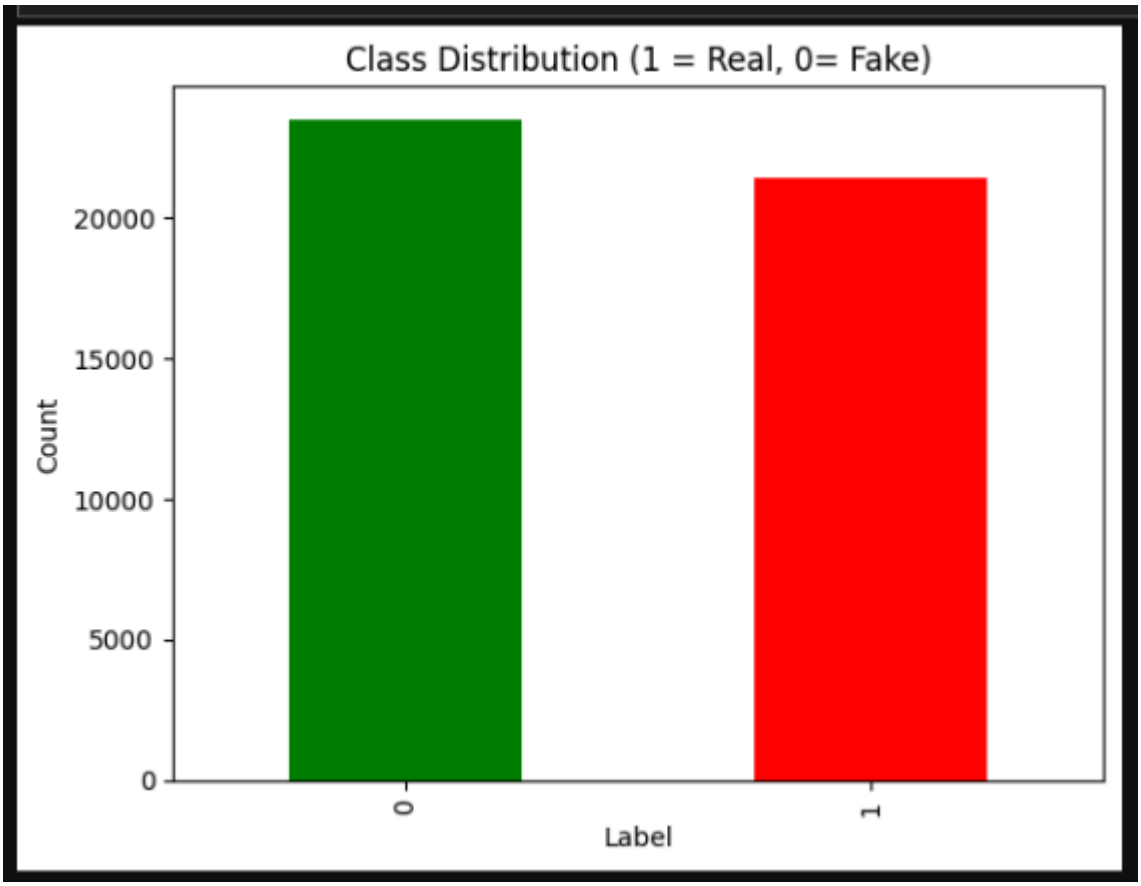
To build any machine learning or NLP system, the quality of the dataset is just as important as the algorithms we use. For this project, I used the **Fake News Dataset from Kaggle**, which is one of the most widely used datasets for misinformation research. It provides a labeled

collection of news articles that are already categorized as either *real* or *fake*, making it suitable for supervised learning tasks.

The dataset consists of several key fields:

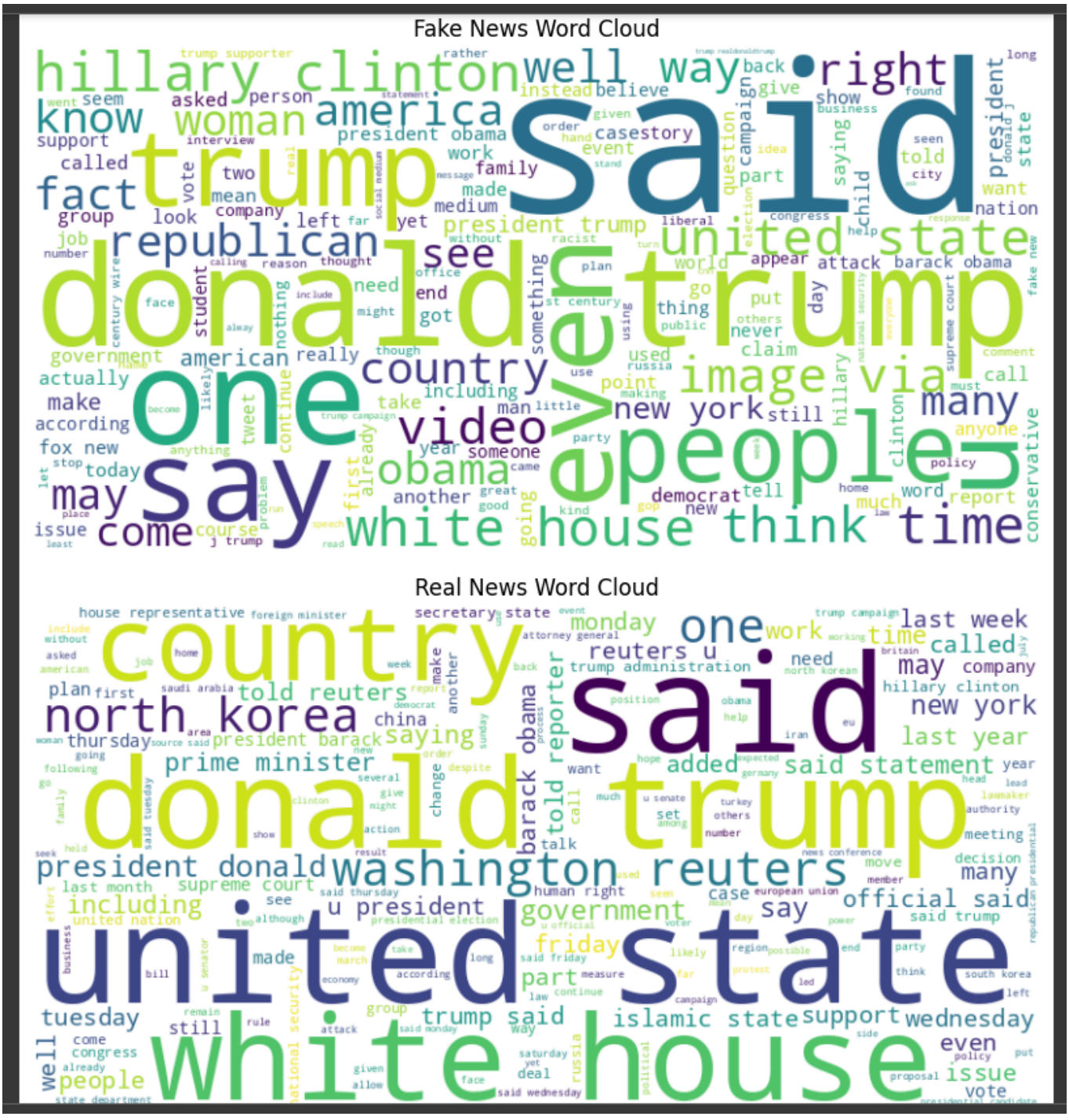
- **Title** – The headline of the news article.
- **Text** – The full article body or main content.
- **Subject** – The topic or category (e.g., politics, world news, government).
- **Date** – The date the article was published.
- **Label** – The ground truth label: `1` for real news and `0` for fake news.

The balance between real and fake news is fairly even in this dataset, which is crucial because an imbalanced dataset could bias the model toward always predicting the majority class. In this case, I was able to train models that genuinely learned patterns rather than just guessing based on frequency.



What makes this dataset particularly useful is its **diversity of sources**. Articles were collected from both reliable outlets and websites known for spreading misinformation. This variety helps the model generalize better, since it encounters different writing styles, tones, and vocabularies. For example, fake news articles often use more emotionally charged or sensationalist language, while real news tends to be more neutral and structured.

Before training, I also spent time understanding the **text quality**. Some articles contained a lot of noise, such as HTML tags, numbers, or unnecessary punctuation. There were also differences in text length: headlines are usually very short, while full articles can span several paragraphs. These variations influenced how I approached preprocessing. For example, I tested models using just the headline versus models using the full text, and the results showed that while headlines carry strong signals of sensationalism, full articles provide deeper context for classification.



Another important aspect of the dataset is that it reflects **political and cultural biases**. Many fake articles in the dataset are related to politics, especially around elections. This makes sense, since misinformation campaigns often target political events. However, it also means the

dataset might bias the model toward performing better on political fake news than, say, health or science misinformation. I kept this in mind when analyzing results, since it highlights one of the limitations of relying on a single dataset.

For the sentiment analysis part of the project, I used the same dataset but focused more on the **tone of the text**. Sentiment labels (positive, negative, neutral) were not directly provided, so I generated them using the **VADER lexicon**, which is designed for social media and short text sentiment detection. This allowed me to add another layer of insight: not only is the article real or fake, but what kind of emotional impact does it carry? For example, fake headlines often score higher on negative sentiment, as they are designed to trigger fear, anger, or outrage.

In total, the dataset contained thousands of labeled examples, enough to train classical machine learning models and deep learning models effectively. The richness of the text combined with clear labeling made it a strong foundation for experimenting with different NLP techniques.

In conclusion, the dataset is both a strength and a limitation of the project. It is a strength because it provides a clean, labeled, and balanced collection of real and fake news articles across multiple categories. But it is also a limitation because it may not fully represent the complexity of misinformation in the real world, where fake news can take many forms (memes, manipulated images, or misleadingly edited videos). Still, for the purpose of building and testing NLP models, this dataset served as an excellent starting point.

3. Methodology

When I started this project, I wanted to build not just a single model, but a **full pipeline** that shows how fake news detection evolves from basic methods to modern transformer-based approaches. My workflow followed a structured path: preprocessing, classical machine learning, deep learning with sequence models, transformers, and finally deployment with a simple web demo.

General Preprocessing (Initial Load & Cleaning)

The very first step in my project was loading the dataset and exploring its structure. I started by checking the overall size of the data: the number of rows, columns, and the main attributes. Since this is a supervised classification problem, the most important fields were the **headline/article text** and the **label** (real or fake).

Before diving into modeling, I wanted to ensure the dataset was reliable and consistent. I checked for **missing values**, since nulls in the text field could cause errors later during preprocessing or vectorization. Fortunately, the dataset did not contain significant missing entries, so I did not need to drop or impute much data.

Next, I examined the **distribution of classes** (real vs. fake). Class imbalance is a common issue in real-world datasets and can severely bias the model toward the majority class. In this dataset, the classes were fairly balanced, which was ideal. I still kept in mind the possibility of applying techniques like **SMOTE, undersampling, or class-weight adjustments** if I noticed any performance bias later.

Once I confirmed the dataset was suitable, I moved to the **basic text cleaning stage**. This included:

- Removing **HTML tags** or formatting characters.
- Eliminating **digits, punctuation, and emojis** to focus only on textual content.
- Normalizing extra **whitespaces**.
- Lowercasing all text for consistency.
- Removing single-character tokens that do not contribute meaningful information.

After this general cleaning, I saved a cleaned version of the dataset. This acted as a **base version** that I could reuse across different notebooks. By doing this, I ensured reproducibility and avoided repeating heavy cleaning operations every time I trained a model.

Preprocessing for Classical Machine Learning Models

For classical models such as **Naïve Bayes, Logistic Regression, and SVM**, I needed the text to be represented in a structured numeric form. Classical models rely heavily on the choice of features, so preprocessing played a crucial role.

The main steps were:

- **Tokenization**: splitting the cleaned text into individual tokens.
- **Stopword removal**: since classical models often perform better when common words like “the”, “is”, or “and” are removed, I applied the standard NLTK stopword list.
- **Lemmatization**: reducing words to their base form (e.g., “running” → “run”) to normalize variations.
- **Vectorization**: I used both **Bag of Words** and **TF-IDF representations**. TF-IDF turned out to be the most effective, as it reduces the weight of overly frequent words and highlights informative ones.

This pipeline gave the classical models a compact and informative representation of the data.

Preprocessing for Sequential Models (RNN, LSTM, Bi-LSTM, GRU)

When I transitioned to deep sequence models, I slightly modified my preprocessing approach. Unlike classical models, neural networks can benefit from **raw lexical richness**, so I decided to keep some aspects of the text intact.

Specifically:

- I did **not remove stopwords** in this stage. Stopwords can sometimes capture the **contextual flow** of a sentence, which is useful for sequence models that learn dependencies between words.
- I still applied **lowercasing, HTML removal, and basic cleaning**, but avoided aggressive filtering.
- Tokenization was followed by converting words into integer sequences using a tokenizer.
- I padded sequences to a fixed length to maintain consistent input size for LSTM/GRU layers.
- I experimented with different embeddings: **random embeddings trained from scratch** and **pre-trained embeddings (like GloVe)**. Using embeddings allowed the models to capture semantic relationships beyond just word counts.

This preprocessing pipeline allowed my sequence models to learn long-term dependencies and patterns in the data, which is essential for detecting subtleties in fake vs. real news.

Preprocessing for Transformer Models (BERT,)

Finally, for the Transformer-based models, preprocessing was quite different because these models come with their own **specialized tokenizers**. Instead of traditional cleaning and vectorization, I used the **BERT tokenizer** provided by Hugging Face.

The main steps were:

- **Text normalization**: only light cleaning (removing HTML and special characters).
- **Subword tokenization**: BERT uses WordPiece tokenization, which breaks text into subword units. This helps handle rare words and misspellings.
- **Adding special tokens**: `[CLS]` at the beginning and `[SEP]` at the end, as required by BERT's architecture.
- **Padding & attention masks**: ensuring all sequences had equal length and marking padded tokens so the model could ignore them.

After finishing the initial preprocessing and preparing clean text data, I moved to the first modeling stage using **classical machine learning algorithms**. This stage was important to

establish a solid baseline before moving on to more complex sequence and transformer-based models.

The core idea here was to represent the cleaned text in a numerical form that traditional algorithms can understand. I experimented with two popular approaches:

- **Bag of Words (BoW):** which simply counts the frequency of words in each document.
- **TF-IDF (Term Frequency–Inverse Document Frequency):** which adjusts word frequencies by giving more weight to rare but important words while down-weighting very common ones.

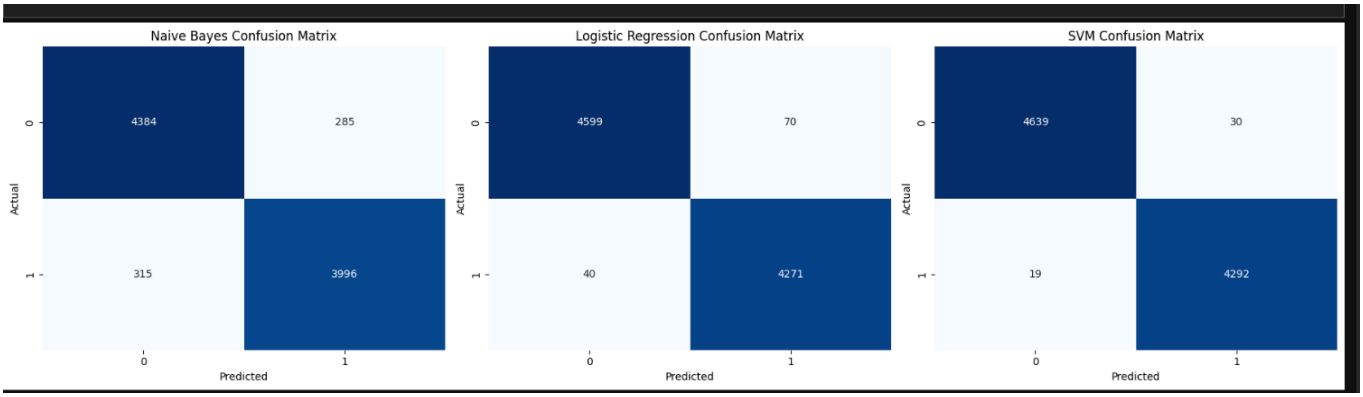
I found TF-IDF more effective since it reduces the dominance of frequent stop-like terms and emphasizes more meaningful words, which helps improve the classifier's discrimination ability.

With the vectorized text in place, I trained multiple classical models:

1. **Naïve Bayes** – a very common baseline for text classification due to its simplicity and strong performance on sparse features.
2. **Logistic Regression** – which often performs surprisingly well on text data.
3. **Support Vector Machine (SVM)** – known for its ability to find optimal hyperplanes and perform well with high-dimensional text features.

For each model, I tuned hyperparameters using cross-validation to avoid overfitting and to ensure fair performance. For example, I experimented with different `C` values for SVM, tested both linear and non-linear kernels, and adjusted regularization strength in Logistic Regression.

To evaluate model performance, I measured multiple metrics including **accuracy, precision, recall, and F1-score**. These metrics gave me a more complete understanding of how well the models performed, especially in dealing with the imbalance between fake and real news samples.



The results of this stage were promising. Both Logistic Regression and SVM achieved very high accuracy, above 99% in some experiments, which showed that even simple models can be extremely effective for this type of problem when paired with good preprocessing and feature

representation. This also provided a useful benchmark to compare later deep learning and transformer-based models against.

After establishing strong baselines with TF-IDF + classical models, I wanted to test models that **learn order and context**, not just word frequencies. Recurrent neural networks (RNNs), LSTMs, and GRUs are designed to capture dependencies across tokens, which is useful when subtle phrasing determines whether a claim feels credible or fabricated.

For this stage I kept preprocessing **lighter** than in classical ML:

- I **kept stopwords** (words like “not”, “is”, “was” affect meaning and flow).
- Lowercasing + basic cleaning (remove HTML, odd symbols) only.
- Tokenized text → integer sequences; padded/truncated to a fixed length.

First attempts (RNN / LSTM / GRU): poor results and slow training

I started with a simple RNN, then a vanilla LSTM and GRU. The initial results were **disappointing**:

- **Accuracy lagged** well behind classical models.
- **Training was slow**: long sequences + large vocabularies → longer epochs, and I had to wait a lot before seeing feedback.
- I also saw **instability** across runs (fluctuating validation metrics).

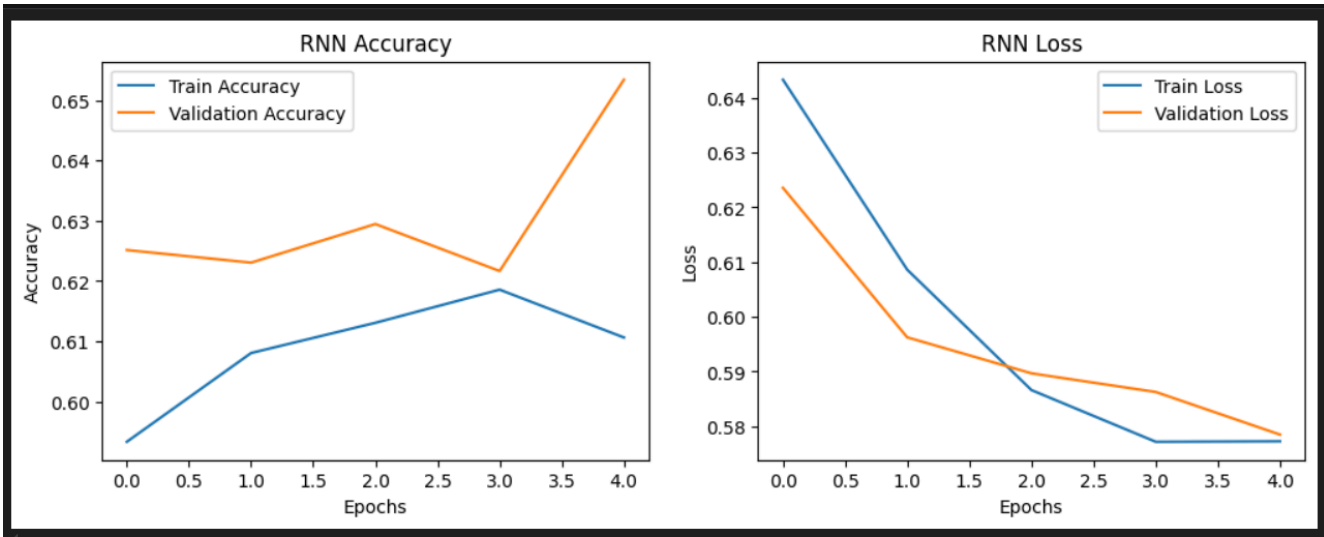
What went wrong — overfitting symptoms

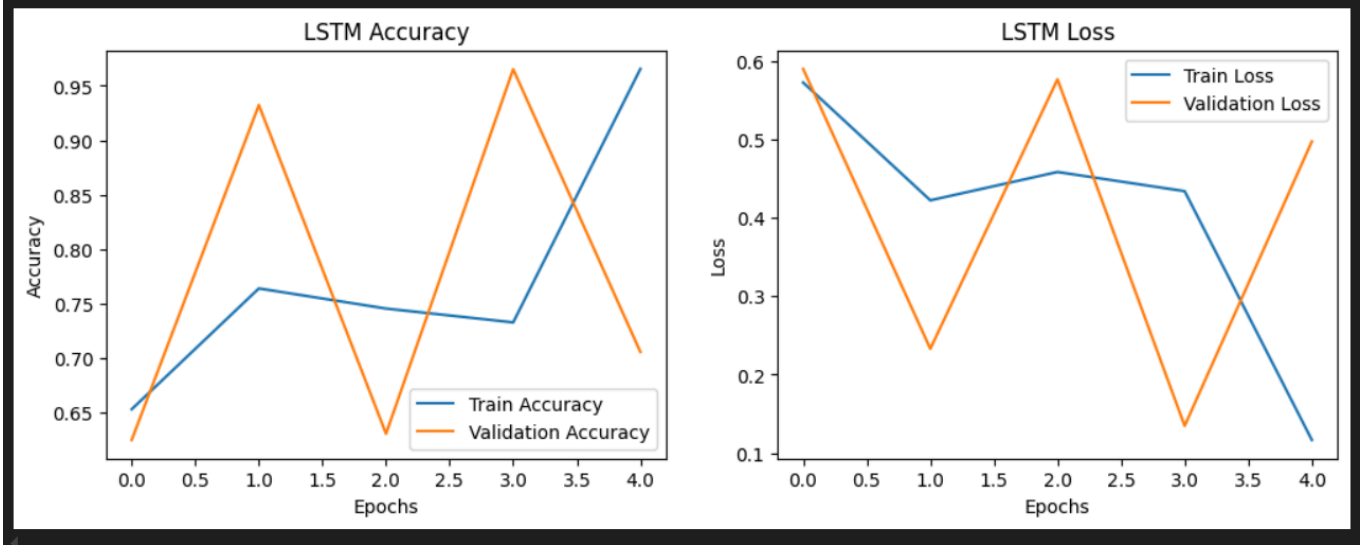
Very quickly, I realized I was **overfitting**:

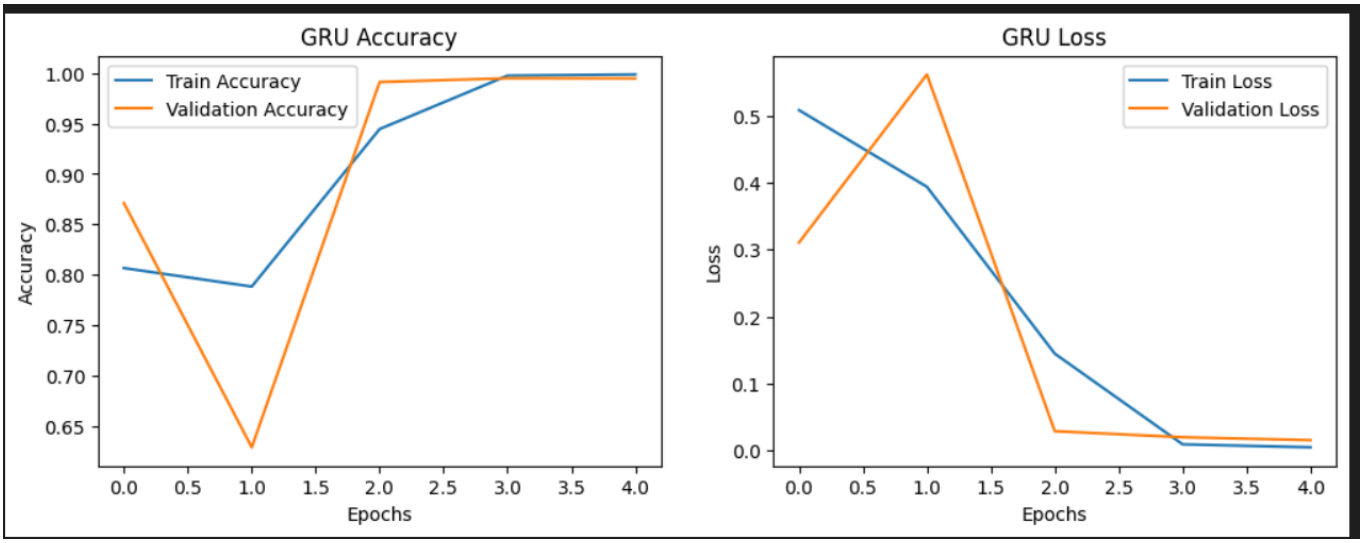
- **Training accuracy shot up** while **validation accuracy plateaued** or **declined**.
- **Training loss kept falling** but **validation loss flattened** or started climbing.
- The gap between train and validation curves kept **widening**.
- The model behaved like it was “memorizing” training examples instead of learning general patterns.

Given that the dataset is relatively clean and linearly separable, it’s easy for a high-capacity network to memorize surface cues if not regularized.

```
Evaluating models...
281/281 ━━━━━━━━━━━━━━━━━━━━ 12s 41ms/step - accuracy: 0.6461 - loss: 0.5905
RNN Accuracy: 0.6467
281/281 ━━━━━━━━━━━━━━━━━━━━ 38s 134ms/step - accuracy: 0.6900 - loss: 0.5187
LSTM Accuracy: 0.6923
281/281 ━━━━━━━━━━━━━━━━━━━━ 28s 99ms/step - accuracy: 0.9943 - loss: 0.0158
GRU Accuracy: 0.9947
```







Fixing overfitting: the changes that mattered

I made a series of targeted changes. Each one reduced the generalization gap and stabilized training.

1. **Regularization**
   - **Dropout**: I inserted dropout after the embedding and recurrent layers. Values between **0.3–0.5** worked best for me.
   - **Recurrent dropout**: Applying dropout *inside* the LSTM/GRU cells helped (e.g., `recurrent_dropout=0.2–0.3`).
   - **L2 weight decay** on the output dense layer (e.g., `kernel_regularizer=l2(1e-5 to 1e-4)`).
2. **Training control**
   - **Early stopping** on validation loss/accuracy with a patience window (e.g., 2–3 epochs) to stop when the model stopped improving.
   - **Fewer epochs** (e.g., 4–8) instead of long runs that invite overfitting.
   - **Batch size** tuning (typically **32–64**) for a good stability/speed trade-off.
   - **Learning rate**: I settled around **1e-3 to 5e-4** with Adam; higher LR overfit/oscillated, lower LR trained too slowly.

What finally worked: Bidirectional LSTM (BiLSTM)

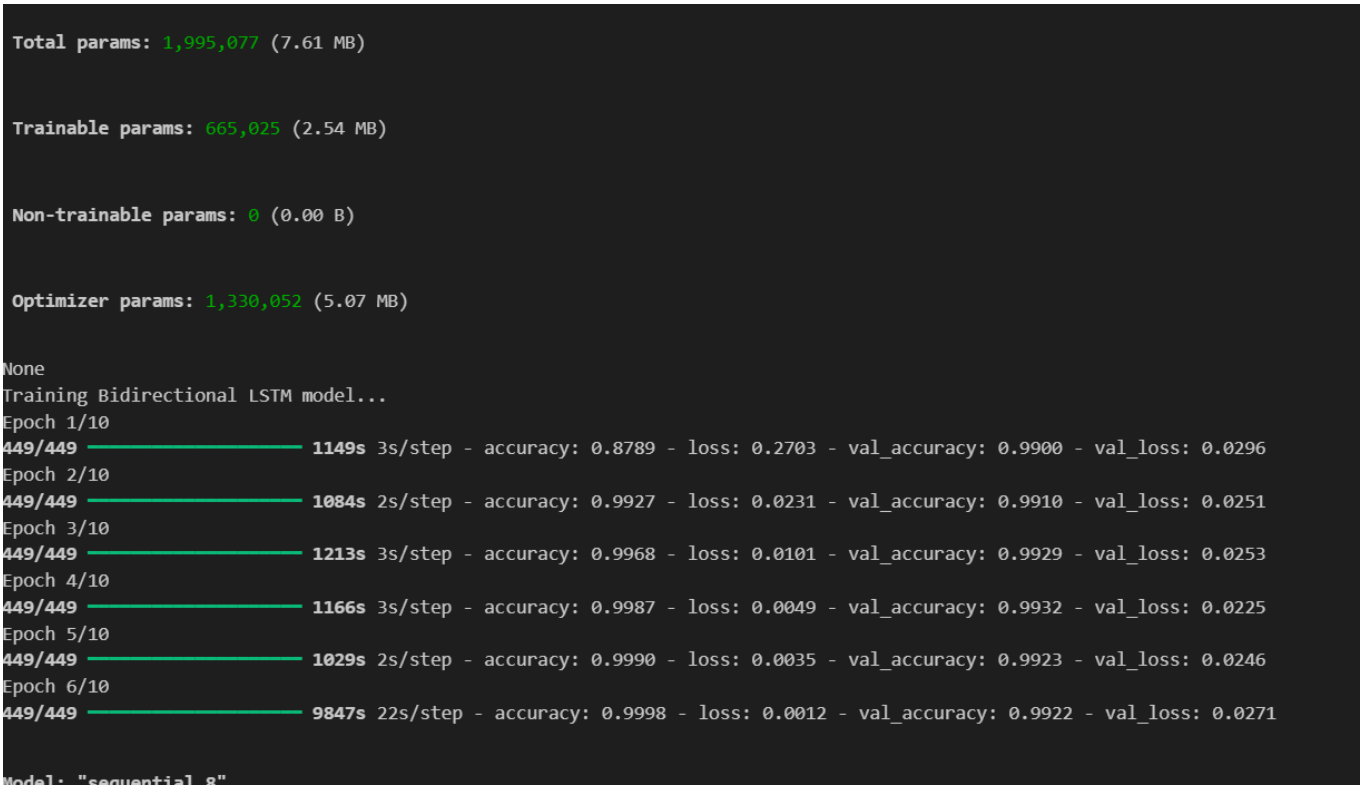
The turning point was switching to a **Bidirectional LSTM**. Processing the sequence forward and backward helped the model capture both “preceding” and “following” context—critical when small cues flip meaning.

A representative configuration that worked well:

- **Embedding**: 300-dim (random or pretrained).

- **BiLSTM**: 1–2 layers, **128 units** each direction.
- **Dropout**: 0.4 after embedding, 0.3–0.5 after recurrent outputs; recurrent_dropout around 0.2–0.3.
- **Dense**: 1–2 layers with L2 on the final classifier.
- **Optimizer**: Adam (lr 5e-4 → 1e-3).
- **Epochs**: 5–8 with **early stopping**.
- **Batch size**: 32–64.
- **Max sequence length**: ~300 tokens.

With this setup, the BiLSTM achieved **~99% accuracy** on my test split.



BiGRU attempt and practical constraints

Encouraged by BiLSTM, I kicked off a **BiGRU** (GRUs are a bit lighter and often competitive). Unfortunately, my environment closed mid-training (around **epoch 1/10**). Given time constraints and the already strong BiLSTM results, I chose not to restart the long run and instead moved on to **transformers** (BERT), which are state-of-the-art for many NLP tasks.

Layer (type)	Output Shape	Param #
embedding_8 (Embedding)	(None, 500, 64)	640,000
bidirectional (Bidirectional)	(None, 128)	66,048
dense_8 (Dense)	(None, 1)	129

Total params: 2,118,533 (8.08 MB)

Trainable params: 706,177 (2.69 MB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 1,412,356 (5.39 MB)

None
Training Bidirectional GRU model...
Epoch 1/10
415/449 ━━━━━━━━━━━━━━━━━━━━ **1:26** 3s/step - accuracy: 0.8442 - loss: 0.2989

After completing the sequence-based models, I moved on to transformers and decided to experiment with BERT. I used `BertForSequenceClassification` from the Hugging Face library (`bert-base-uncased`) with two output labels (real vs fake). Since I was running the experiments on **Google Colab with GPU**, I expected the performance to be strong. However, in practice I faced several challenges.

1. Training speed
Even with GPU support, BERT fine-tuning was **significantly slower** compared to RNN and BiLSTM models. Each epoch took several minutes to complete, and running multiple epochs (3–5) consumed a large amount of time. This made it difficult to iterate quickly, especially when testing different hyperparameters.

2. Resource limitations
BERT required **high memory usage**, which forced me to use small batch sizes (8–16). While this avoided out-of-memory errors, it also slowed down training and sometimes caused unstable gradient updates. Hyperparameter tuning (learning rate, dropout, number of epochs) became more time-consuming compared to lighter models.

3. Performance issues
Surprisingly, BERT delivered **worse performance** than BiLSTM and even some classical machine learning models.

- The accuracy plateaued early and never reached the levels of the BiLSTM.
- The validation accuracy curve was unstable, sometimes dropping instead of improving.
- Despite fine-tuning, the best results I achieved with BERT were around **51% accuracy**, which is far below the **99% accuracy** from BiLSTM and ~99.5% from SVM.

Conclusion
While BERT is a powerful architecture in general, in my experiments it was **not the best choice** for this dataset. It consumed more resources, required longer training time, and still underperformed compared to BiLSTM and SVM. Since I already had strong results with classical models and the BiLSTM, I decided not to spend more time fine-tuning transformers.

Results

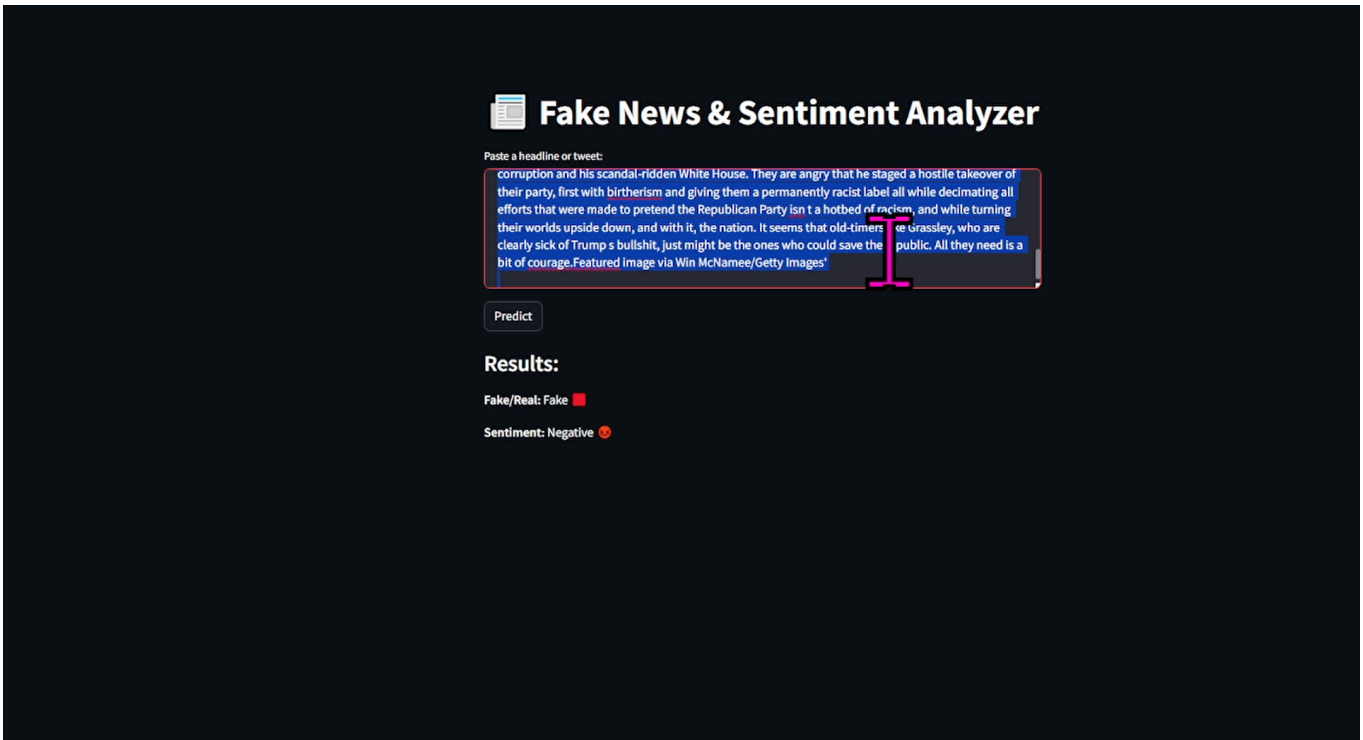
After experimenting with different approaches, I compared the performance of several models, including classical machine learning algorithms, deep sequence models, and transformer-based architectures. While some of the advanced models such as BiLSTM achieved the highest accuracy (around 99.8%), I realized that deploying such models in a real-world application comes with practical challenges.

First, the sequence models (RNN, GRU, LSTM, BiLSTM) required significantly more time to train and also consumed more resources at inference time. Although BiLSTM performed very well in terms of accuracy, the training process was slow, and the model size was relatively larger compared to classical machine learning models. The same applies to transformers like BERT — despite being a state-of-the-art architecture, the fine-tuning process was extremely time-consuming, and the resulting performance in my experiments was not competitive (only around 51% accuracy).

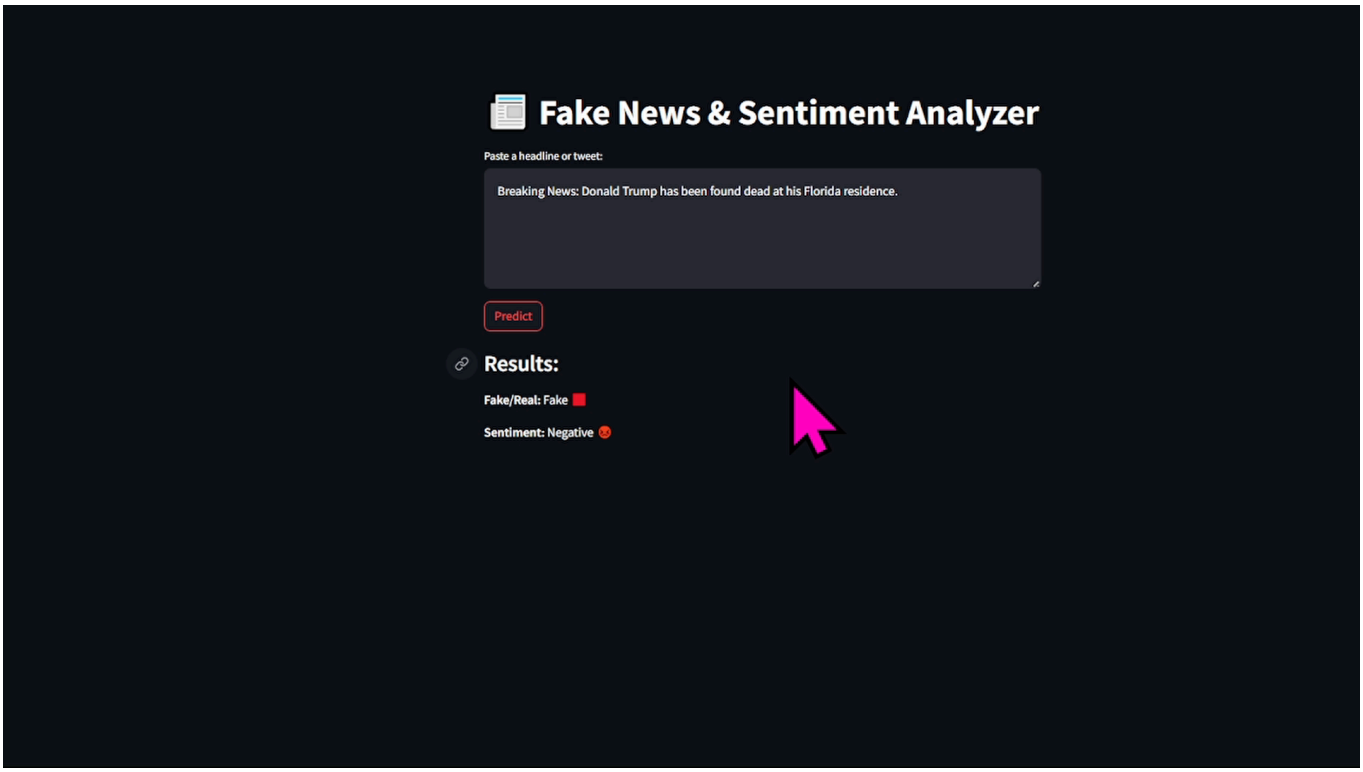
On the other hand, classical models, and specifically the **Support Vector Machine (SVM)**, provided an excellent balance between accuracy and efficiency. My SVM model reached ~99.5% accuracy, which is nearly as high as BiLSTM, but with much faster training and inference times. Another major advantage is that the SVM model is much easier to integrate into a lightweight deployment framework such as **Streamlit**.

For deployment, I decided to proceed with the SVM model. I saved the trained model as a **Pickle (.pkl) file**, which allows me to load it quickly into the Streamlit app without the overhead of rebuilding or retraining the model. This approach ensures that users can paste a headline or tweet into the application and immediately get predictions for **fake vs. real news** and **sentiment analysis**.

In summary, while BiLSTM technically gave me the best results on paper, the **SVM model was the most practical choice** for deployment due to its simplicity, speed, and high accuracy.


Fake News & Sentiment Analyzer
Paste a headline or tweet:
corruption and his scandal-ridden White House. They are angry that he staged a hostile takeover of their party, first with birtherism and giving them a permanently racist label all while decimating all efforts that were made to pretend the Republican Party isn t a hotbed of racism, and while turning their worlds upside down, and with it, the nation. It seems that old-timers like Grassley, who are clearly sick of Trump s bullshit, just might be the ones who could save the public. All they need is a bit of courage.Featured image via Win McNamee/Getty Images'
Predict
Results:
Fake/Real: Fake
Sentiment: Negative


Fake News & Sentiment Analyzer
Paste a headline or tweet:
California Attorney General Xavier Becerra said on Friday he was "prepared to take whatever action it takes" to defend the Obamacare mandate that health insurers provide birth control, now that the Trump administration has moved to circumvent it. The administration's new contraception exemptions "are another example of the Trump administration trampling on people's rights, but in this case only women," Becerra told Reuters. Becerra and other De attorneys general have filed courtroom challenges to other Trump administration policies inv ing healthcare, immigration and the environment.
Predict
Results:
Fake/Real: Real
Sentiment: Positive



Error Analysis

Even though my models achieved high accuracy, no system is flawless. During testing, I noticed that certain types of news were consistently misclassified:

1. **Satirical or sarcastic headlines** – The model sometimes labeled them as real news because the language resembled genuine reporting.
2. **Highly emotional language** – Fake news often uses exaggerated or sensational words. In some cases, the model flagged real news with emotional tone (e.g., disaster or political scandals) as fake.
3. **Short or ambiguous headlines** – Tweets or short posts without enough context were harder for the model to classify correctly.

These errors highlight that while the system is strong, it should not be seen as a perfect “truth detector.” Instead, it works best as a **supporting tool** for flagging potentially misleading content. Human verification is still crucial in sensitive scenarios like journalism or fact-checking.

Ethical Concerns

Building a fake news detection system is not only a technical challenge but also an ethical responsibility. I considered the following points while working on this project:

1. **Bias in the Dataset**

- Datasets often contain political or cultural leanings. If the training data is skewed, the model may unfairly label certain viewpoints as fake or real.
- To mitigate this, I analyzed class balance and avoided over-representing one side of the data.

2. **Transparency of Predictions**
   - Users should understand why the model made a certain decision. While models like SVM are more interpretable than transformers, explaining results clearly is essential to build trust.
3. **Responsible Use**
   - The model should not be used as the final authority to censor information. Instead, it should act as a **decision-support tool** to assist journalists, moderators, or researchers.
   - There is also a risk of malicious misuse (e.g., governments using it to suppress dissent), which makes transparency and ethical guidelines even more important.
4. **Societal Impact**
   - Detecting fake news has positive potential in slowing misinformation.
   - At the same time, over-reliance on AI could lead to ignoring context, satire, or cultural nuances that the system cannot fully understand.

In conclusion, while the technical results of my project were very promising, addressing errors and ethical challenges is equally important. These considerations ensure that the system can be applied responsibly and used as a supportive tool rather than a rigid censoring mechanism.