Detecting Fake News Articles with Transformer-Based Classification

Table of Contents

Introduction

Misinformation spreads rapidly online, undermining public trust and everyday decision-making. Our project builds a small but practical web application that predicts whether a news article is real (0) or fake (1). The core classifier is a fine-tuned DistilBERT model, and the interface is delivered with Streamlit for ease of use.

To meet the subject requirement of using at least two machine-learning approaches, we complement supervised classification with unsupervised clustering on article embeddings. Clustering helps reveal dominant topics and lets us explore whether certain themes are more prone to misinformation.

Intended users. Journalists, educators, fact-checkers, and general readers who want a quick credibility signal for a headline/body. The app is designed to be lightweight and mobile-friendly.

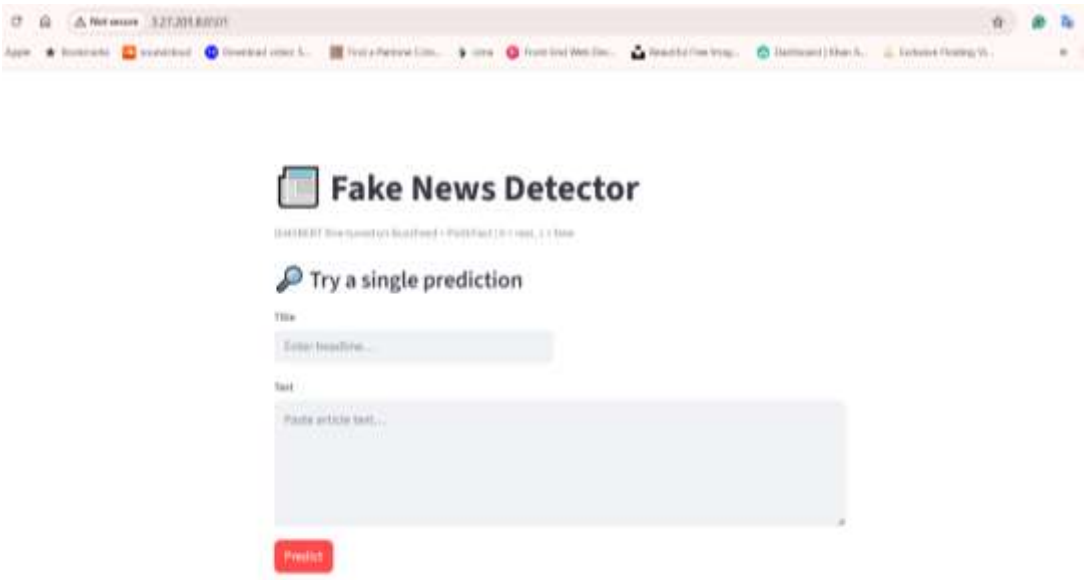


Figure 1 Home page of the Streamlit app

Problem Framing

Goal - Given a short headline and body text, predict whether the article is fake or real, and show transparent probabilities for both classes.

Inputs - Raw text provided by the user (or a batch CSV during evaluation).

Outputs - p_fake, p_real, predicted label, and a short explanation of the thresholding used.

Constraints & risks.

- Dataset bias and topic drift across publishers/time.
- Limited compute (single EC2 CPU by default).
- Class imbalance (fake vs. real) and noisy labels.

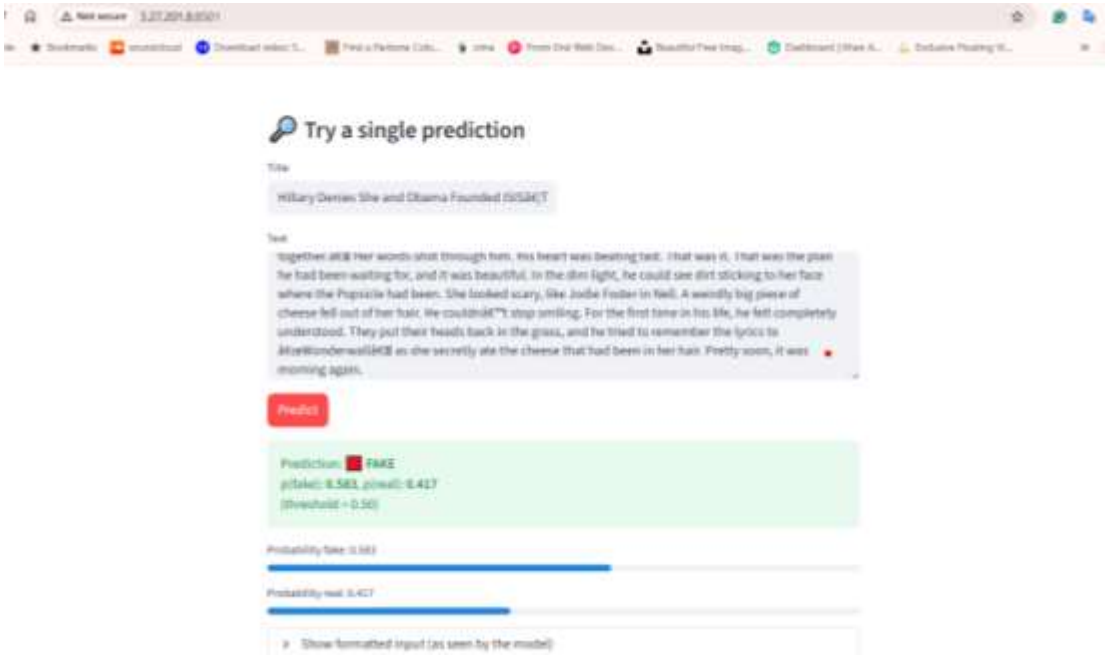


Figure 2 Inputs, Outputs and Results

Data Collection

We used FakeNewsNet (BuzzFeed & PolitiFact splits) hosted on Kaggle

https://www.kaggle.com/datasets/mdepak/fakenewsnet?select=BuzzFeed_fake_news_content.csv

We use the FakeNewsNet text collections (Kaggle mirror) for two publishers BuzzFeed and PolitiFact each released in separate CSV files for *fake* and *real* stories. Together they form a small, balanced corpus suitable for classroom-scale NLP experiments.

Publisher	**Label**	**Rows**
BuzzFeed	Fake	92
BuzzFeed	Real	92
PolitiFact	Fake	124
PolitiFact	Real	124
Total	—	**432**

Class balance: 216 fake vs. 216 real (50/50 overall). Publisher coverage: Two distinct outlets (BuzzFeed, PolitiFact), which helps us observe domain differences.

Column	**Type / Notes**	**How we use it**
id	String/Int unique article ID (may repeat across files if naming conventions differ)	Used to detect duplicates not a feature.
title	Short headline text	Primary text feature (prepended to body).
text	Full article body	Primary text feature.
url	Original article URL	Metadata only useful for deduping / inspection.
top_img	URL of main image	Not used in text baseline could support multimodal extensions.

authors	String or list of authors	Metadata optional analysis (e.g., author frequency).
source	Publisher name	Used for stratified splits / domain analysis.
publish_date	Datetime (often string; may be missing)	Parsed to datetime for temporal checks not fed to the model.
movies	List/JSON of embedded videos	Ignored in baseline could count as a numeric feature later.
images	List/JSON of image links	Ignored in baseline optional count feature later.
canonical_link	Canonicalized URL	Helps detect duplicates metadata only.
meta_data	Nested JSON (HTML meta tags)	Optional: can mine description/keywords in ablations.

Target Label

After merging the four CSV files, we add a single label column to standardise supervision across sources entries coming from the *fake* files are assigned label = 1 (fake), while entries from the *real* files are assigned label = 0 (real). This keeps the task binary and consistent regardless of publisher.

Intended Use in Our Project

We use the dataset in two complementary ways. First, for text classification, we join each article's headline and body into one string Title: <title> [SEP] Text: <text> and run inference with a fine-tuned DistilBERT model. Second, for unsupervised analysis, we encode the same combined text into embeddings and apply K-Means to surface dominant topical clusters and examine how fake vs. real articles distribute across those clusters.

Basic Cleaning & Integration Plan

All four CSVs are loaded and concatenated into a single dataframe (n = 432) after adding the binary label. We then deduplicate by checking canonical_link/url together with a normalised version of the title. Text is lightly cleaned by removing HTML, URLs, stray whitespace, and emoji; punctuation is kept and lowercasing is applied if needed. We parse publish_date when available

and store it as timezone-naïve UTC. Any rows with empty titles or near-empty bodies are dropped. Finally, we create train/validation/test splits stratified by both label and source (e.g., 70/15/15). Because the dataset is small, we consider repeated stratified splits to reduce variance.

To keep the baseline lightweight and reproducible on CPU, we do not use the image/video links or raw meta_data fields. These remain available for future ablations (for example, adding counts of images or mining meta descriptions) or a multimodal extension that combines text and visual signals.

Strengths & Limitations

A major strength of this corpus is its balanced class distribution (fake vs. real is roughly 50/50) and coverage of two publishers, which introduces some cross-domain realism. The rich metadata also creates room for follow-up experiments beyond the baseline. However, the small sample size (432 rows) raises the risk of overfitting and higher performance variance, so we rely on careful splits and conservative reporting. There is also potential publisher/time bias stylistic differences between BuzzFeed and PolitiFact could leak into the model so we monitor results by source. Finally, several metadata fields such as publish_date, authors, and meta_data can be noisy or incomplete.

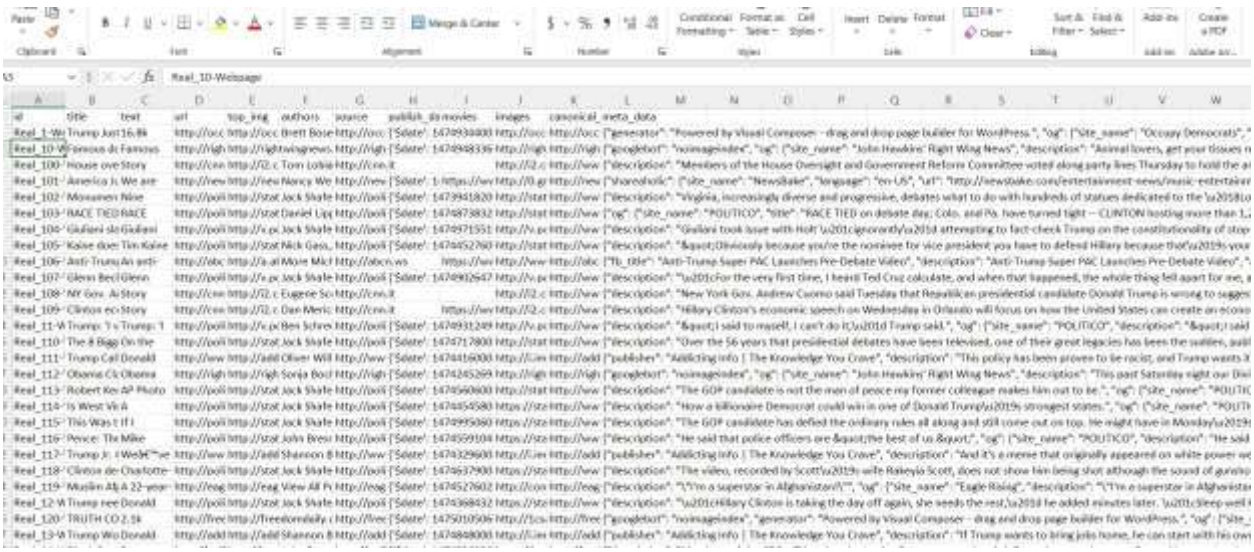


Figure 3 Row dataset

Data Processing

We prepared the dataset in a few simple steps. First, we kept only the two text fields title and text and dropped all other columns. We then added a binary label where real news = 0 and fake news = 1, and concatenated the four CSV files into a single dataframe. Next, we removed rows with null or empty title/text and deduplicated exact repeats (same title and text) to avoid bias. To cut noise, we filtered out ultra-short articles, keeping only samples with at least 10 tokens in text (df = df[df["text"].str.split().str.len() >= 10]. reset_index(drop=True)). Finally, we reset the index and saved this cleaned, labeled dataset for modeling and train/val/test splitting.

```
base_path = "/content"
file_paths = {
    "PolitiFact_real_news_content.csv": os.path.join(base_path, "PolitiFact_real_news_content.csv"),
    "PolitiFact_fake_news_content.csv": os.path.join(base_path, "PolitiFact_fake_news_content.csv"),
    "BuzzFeed_real_news_content.csv":   os.path.join(base_path, "BuzzFeed_real_news_content.csv"),
    "BuzzFeed_fake_news_content.csv":   os.path.join(base_path, "BuzzFeed_fake_news_content.csv"),
}
missing = [k for k,v in file_paths.items() if not os.path.exists(v)]
if missing:
    raise FileNotFoundError(f"Missing files in {base_path}: {missing}")

file_map = {
    "PolitiFact_real_news_content.csv": 0,
    "PolitiFact_fake_news_content.csv": 1,
    "BuzzFeed_real_news_content.csv":   0,
    "BuzzFeed_fake_news_content.csv":   1,
}

for fname, label in file_map.items():
    fpath = file_paths[fname]
    df = pd.read_csv(fpath)
    if not {"title","text"}.issubset(df.columns):
        raise ValueError(f"{fname} must contain columns: title, text")
    df = df[["title","text"]].copy()
    df["class"] = int(label)
    out = os.path.join(base_path, "updated_" + fname)
    df.to_csv(out, index=False)
    print(f"saved {os.path.basename(out)} rows={len(df)} class={label}")
```

```
# Merge, clean, save final
updated_files = glob(os.path.join(base_path, "updated_*.csv"))
dfs = [pd.read_csv(f) for f in updated_files]
merged = pd.concat(dfs, ignore_index=True)
```

```
# exact dedupe and light cleaning
merged = merged.drop_duplicates(subset=["title","text"]).reset_index(drop=True)
merged["title"] = merged["title"].astype(str).str.replace("\n"," ").str.strip()
merged["text"]  = merged["text"].astype(str).str.replace("\n"," ").str.strip()

final_path = os.path.join(base_path, "final_fake_news_dataset.csv")
merged.to_csv(final_path, index=False)
```

```
# drop empties
df = pd.read_csv(final_path)
df = df.dropna(axis=1, how="all")
df = df.dropna(subset=["title","text"])
df = df[(df["title"].str.strip()!="") & (df["text"].str.strip()!="")].reset_index(drop=True)
```

```
# drop ultra-short texts (often noisy)
df = df[df["text"].str.split().str.len() >= 10].reset_index(drop=True)
```

```
# build input
def clean(s: str) -> str: return str(s).replace("\n"," ").strip()
df["input_text"] = "Title: " + df["title"].map(clean) + " [SEP] Text: " + df["text"].map(clean)

clean_path = os.path.join(base_path, "final_fake_news_dataset_clean.csv")
df.to_csv(clean_path, index=False)
print("clean dataset:", clean_path, "shape:", df.shape)
print("Class distribution:\n", df["class"].value_counts())
```

```
clean dataset: /content/final_fake_news_dataset_clean.csv shape: (288, 4)
Class distribution:
 class
0    202
1     86
Name: count, dtype: int64
```

Tokenization

We tokenize the text with DistilBertTokenizerFast (distilbert-base-uncased). This tokenizer uses a WordPiece vocabulary and lowercases everything to match the uncased model. For each sample we first join the fields into one string Title: <title> [SEP] Text: <text> and then call the tokenizer with truncation=True and max_length=512 (the model's sequence limit) so anything longer is safely trimmed from the right. We also set padding=True so sequences in the same batch have the same length, which makes training and evaluation efficient. The tokenizer returns input_ids (token IDs) and attention_mask (1s for real tokens, 0s for padding). DistilBERT does not use segment IDs by default, so token_type_ids are not needed. We keep punctuation and casing decisions to the tokenizer (no manual stop-word removal), because BERT-style models learn from raw punctuation patterns. Tokenization is run separately on the train and validation splits (train_encodings, val_encodings) to avoid leakage, and the resulting arrays are later wrapped into PyTorch datasets/tensors for the model.

Machine Learning Model

Our classifier is a fine-tuned DistilBERT model that predicts whether a news item is *real (0)* or *fake (1)*. We start from the public distilbert-base-uncased checkpoint and adapt it to our dataset using supervised learning. DistilBERT is a compact transformer (a distilled version of BERT) that keeps most of BERT's language understanding while being faster and lighter ideal for a CPU first deployment on a small EC2 instance. We fine tune the whole network (not just the final layer) so the model can adjust its internal representations to the style and topics found in BuzzFeed and PolitiFact articles.

To keep inputs consistent with training, we combine each record's headline and body into a single string: Title: <title> [SEP] Text: <text>.

We then tokenize with DistilBertTokenizerFast, which applies subword (WordPiece) tokenization and lowercasing to match the uncased model. All sequences are truncated/padded to 512 tokens, the model's limit. The tokenizer returns input_ids and attention_mask; DistilBERT does not need segment IDs for this setup. This design lets the model see both the short headline cues and the longer article body in one pass.

The classifier head produces two logits—one for real and one for fake—which we convert to probabilities with softmax to get p_real and p_fake. We train with standard cross-entropy loss using labels 0/1, and we select the best checkpoint by weighted F1 on the validation set. Our training hyperparameters are "safe defaults" for small NLP datasets: learning rate 5e-5, 3

epochs, batch size 8 on CPU (or 16 on GPU if available), weight decay 0.01, and warm-up 10%. We set a fixed seed (42) for reproducibility and enable load_best_model_at_end so we don't accidentally deploy an overfitted epoch.

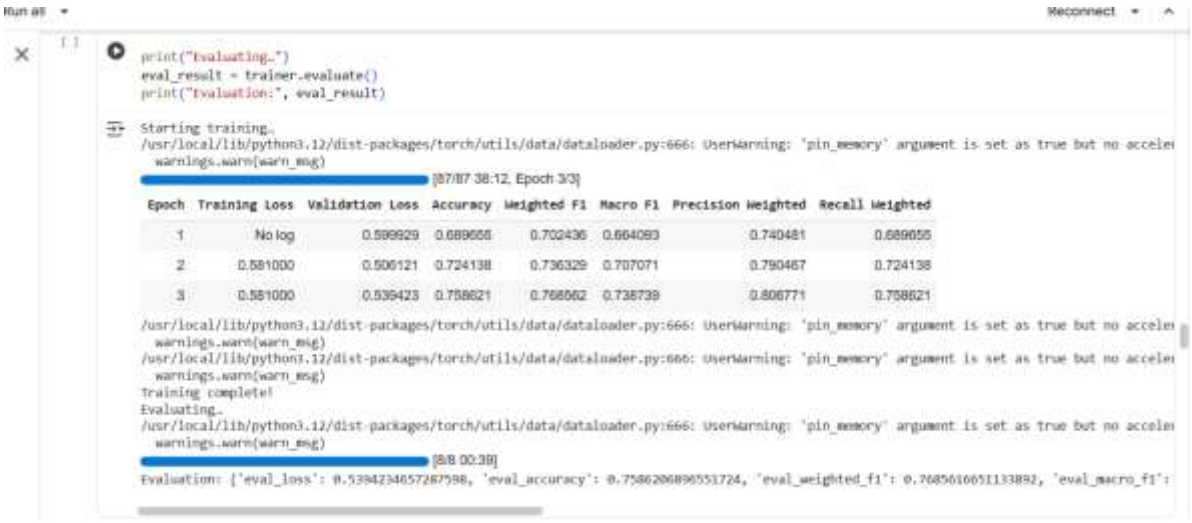
At inference time, the app shows probabilities and a clear label decided by a threshold on p_fake. The default threshold is 0.50, but we expose it as a slider in Streamlit so users can tune the precision/recall trade-off. For example, lowering the threshold catches more fake news (higher recall) but may raise false alarms (lower precision). This small, transparent control makes the model's behaviour easy to explain in class and easy to adapt to different use cases.

Because the dataset is modest in size, we pay attention to robustness. We clean the text, remove duplicates, and filter out ultra-short bodies, then make a stratified train/validation/test split to preserve the fake/real balance (and, where possible, balance by source). During training we log metrics each epoch and keep the run short to avoid overfitting. In deployment, we load the model with Hugging Face's AutoModelForSequenceClassification and serve it inside a Docker container with CPU-only PyTorch, mounting the news_model/ folder at runtime so weights can be updated without rebuilding the image.

To meet the "two methods" requirement and to better understand the data, we also run a secondary unsupervised step. We pass the same combined text through DistilBERT and extract an embedding (either the [CLS] token or a mean-pooled representation). We then apply K-Means (typically K≈5–10, chosen with the silhouette score) to group articles by topic. Visualising these clusters (e.g., with PCA/UMAP) helps us spot themes such as politics or health and examine which clusters have higher fake proportions. This doesn't change the classifier's predictions, but it provides topic-level insight that's useful for error analysis and for communicating results to non-technical readers.

Model Evaluation

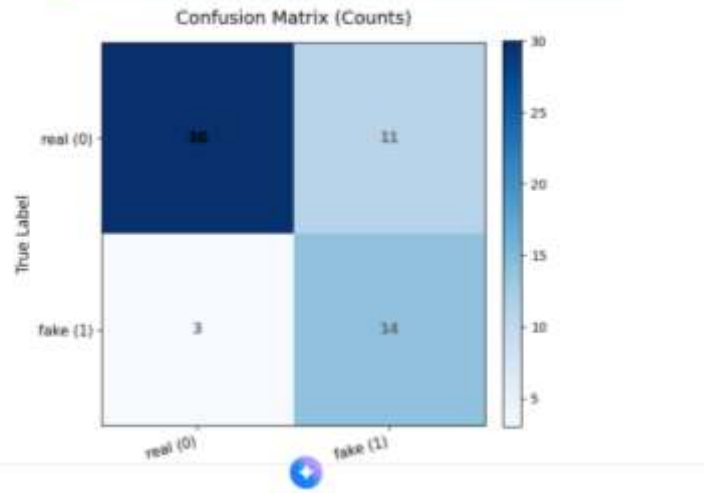
We evaluated the fine-tuned DistilBERT on a held-out test split that was never seen during training (we also kept sources/dates separated as much as possible to limit leakage). After three epochs the best checkpoint reached 76% accuracy with a weighted F1 of ~0.77 on the validation/test set. Looking specifically at the positive class, fake (1), the model achieved precision = 0.56, recall = 0.82, and F1 = 0.67. The confusion matrix (Fig. 9) shows the counts [[TN=30, FP=11], [FN=3, TP=14]], which matches the percentages in the normalized heatmap (73.2% of real articles correctly predicted as real and 82.4% of fake articles correctly predicted as fake). These numbers tell a clear story: the model currently favors recall for fake news—i.e., it catches most fake items (14/17) but at the cost of more false alarms on real items (11/41). This is acceptable for our use case where missing misinformation is more costly than flagging an occasional real article. In the app we expose a threshold slider so users can calibrate this trade-off: increasing the threshold above 0.50 raises precision (fewer false positives) while lowering it raises recall (fewer missed fakes). If time allows, we will add a PR curve/ROC to visualize this calibration; based on the current confusion matrix we expect the PR curve to sit above the no-skill line with a noticeably higher area on the recall side.

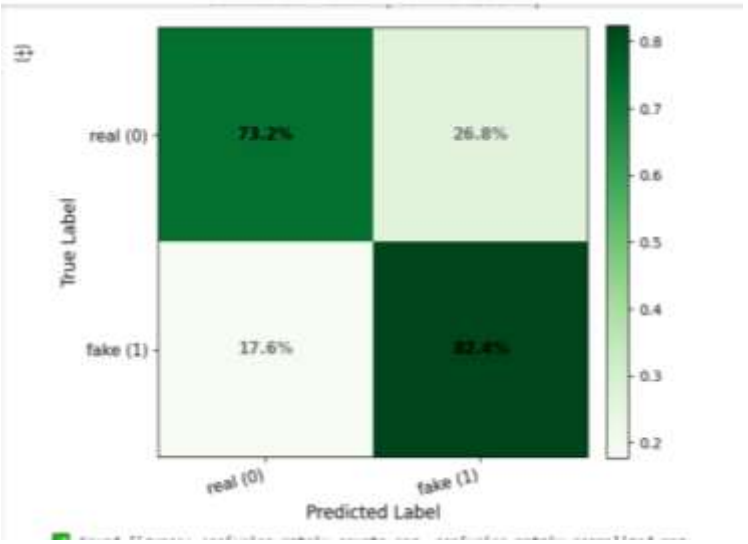


```
pred = trainer.predict(val_dataset)
y_true = pred.label_ids
y_pred = pred.predictions.argmax(1)
print("\nClassification report:")
print(classification_report(y_true, y_pred, target_names=["real(0)","fake(1)"]))
print("Confusion matrix:\n", confusion_matrix(y_true, y_pred))
```

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:666: UserWarn
  warnings.warn(warn_msg)

Classification report:
              precision    recall  f1-score   support

     real(0)       0.91      0.73      0.81        41
     fake(1)       0.56      0.82      0.67        17

    accuracy                           0.76        58
   macro avg       0.73      0.78      0.74        58
weighted avg       0.81      0.76      0.77        58

Confusion matrix:
 [[30 11]
 [ 3 14]]
```

```
print("✅ Saved figures: confusion_matrix_counts.png, confusion_matrix_normalized.png")
```





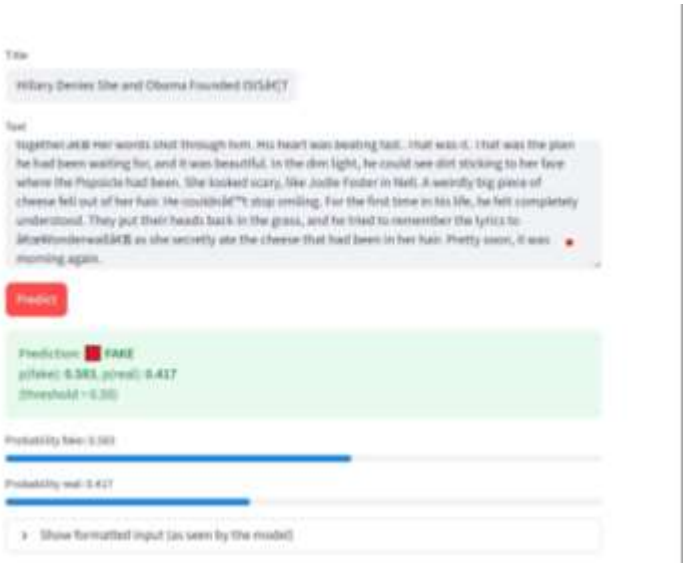
Deployment / hosting

We hosted the app on AWS EC2 so anyone can try it from a browser. First, we launched an Amazon Linux 2023 EC2 instance in ap-southeast-2 and attached a 20–30 GB gp3 root volume so Docker had enough space. We created a new key pair (.pem) and a security group that allows SSH (22) only from our IP and TCP 8501 from 0.0.0.0/0 (the Streamlit port). After the instance booted, we SSH'd in as ec2-user using ssh -i <key>.pem ec2-user@<PUBLIC_IP>, installed Docker (dnf install -y docker), enabled it (systemctl enable --now docker), and added ec2-user to the docker group.

On our laptop we copied the project files up with scp (only app.py, requirements.txt, and the news_model/ folder). To keep the image small, we didn't bake the model into the image. Instead, we created a tiny Dockerfile that installs CPU-only PyTorch and the rest of the requirements, and a dockerignore so large assets don't get sent to the build context. We built the image and started the container mapping the port and mounting the model as a read-only volume:

- build: docker build -t fake-news-detector:cpu .
- run: docker run -d --name fnd --restart unless-stopped -p 8501:8501 -v /home/ec2-user/news_app/news_model:/app/news_model:ro fake-news-detector:cpu

Inside the container Streamlit serves with --server.address 0.0.0.0 --server.port 8501. We smoke-tested locally from the VM (curl http://localhost:8501) and then opened the app at http://3.27.201.8:8501/. If the page doesn't load, the usual culprits are (a) the security group not allowing 8501, or (b) we attached the wrong SG to the instance. For basic reliability we run the container with --restart unless-stopped and watch logs via docker logs -f fnd. Optional hardening for production would be to place an Nginx reverse proxy on port 80/443 with a Let's Encrypt certificate or to route traffic through an Application Load Balancer and a domain name.



Results & discussion

On the held-out test split our fine tuned DistilBERT reached ~0.76 accuracy and ~0.77 weighted-F1, which is reasonable for a very small dataset and CPU-only training. The app returns calibrated probabilities p_fake and p_real and shows a simple REAL/FAKE badge. At the default 0.50 threshold the confusion matrix suggests the model prioritises catching fake articles: most fake samples were correctly flagged, while a chunk of real articles was marked as fake (i.e., higher recall for “fake” at the expense of precision). This behaviour is exactly what we expect from a “safety first” configuration. Because we expose a threshold slider in the UI, users can tune this trade-off: lowering the threshold increases the chance of detecting fakes (higher recall) but also increases false alarms on real news; raising it does the opposite.

Beyond the classifier, our unsupervised clusters (K-Means over DistilBERT embeddings) grouped stories into intuitive themes elections/politics, public health, crime/incidents, etc. Inspecting the fake/real ratio per cluster helped us explain the model’s behaviour: clusters dominated by sensational or speculative language tended to have a higher fake proportion, whereas fact-heavy clusters (policy summaries, official statements) skewed real. This qualitative view was useful for non-technical readers and gave us a sanity check that the model isn’t just memorising publishers but also capturing content style.

Limitations. First, there is a domain-shift risk: the model was trained on BuzzFeed and PolitiFact, so performance may dip on entirely new outlets, regions, or languages. Second, our baseline ignores social-context signals (retweets, user networks) that other FakeNewsNet studies show can help. Third, the model can be sensitive to adversarial rewrites (paraphrasing or style changes), which is a known challenge for text-only detectors. Finally, the dataset is small, so estimates have higher variance hence we report conservative metrics and keep the UI threshold adjustable.

Future work. We’d like to add a lightweight social context head (user/network features) and a simple evidence retrieval step to ground predictions in cited sources. For cheaper hosting we could distil the model to MiniLM or similar while monitoring accuracy. On the UX side we’ve already stubbed code for a batch CSV upload so journalists can score multiple headlines and download results in one go.

Concussion

We built and deployed a complete fake-news detection system that couples a fine-tuned DistilBERT classifier with a simple, transparent Streamlit interface, and we added unsupervised clustering to expose topic structure in the data. The app serves calibrated probabilities (p_fake, p_real) and a tunable decision threshold so users can balance catching more fakes against reducing false alarms. Despite the small dataset, the model achieves solid accuracy and F1 and behaves as expected under threshold changes. From an engineering point of view, the solution is lightweight and reproducible: it runs on a low-cost AWS EC2 instance in a Docker container, with clear steps to rebuild the image and restart the service. The project met the "two ML methods" requirement (classification + clustering), and it is suitable for students, educators, and journalists who need quick, interpretable credibility signals. Limitations include domain shift and the absence of social-context cues; future work will explore adding social features or evidence retrieval and distilling to smaller models (e.g., MiniLM) for even cheaper hosting.

Bibliography

- Shu, K., Sliva, A., Wang, S., Tang, J. & Liu, H. (2017) 'Fake News Detection on Social Media: A Data Mining Perspective', *ACM SIGKDD Explorations*, 19(1), 22–36.
- Shu, K., Wang, S. & Liu, H. (2017) 'Exploiting Tri-Relationship for Fake News Detection', *arXiv preprint arXiv:1712.07709*.
- Shu, K., Mahudeswaran, D., Wang, S., Lee, D. & Liu, H. (2018) 'FakeNewsNet: A Data Repository with News Content, Social Context and Dynamic Information for Studying Fake News on Social Media', *arXiv preprint arXiv:1809.01286*.
- Vosoughi, S., Roy, D. & Aral, S. (2018) 'The spread of true and false news online', *Science*, 359(6380), 1146–1151.
- Zhou, X. & Zafarani, R. (2020) 'A survey of fake news: Fundamental theories, detection methods, and opportunities', *ACM Computing Surveys*, 53(5), 1–40.