Fake News Classification Using Word Frequency Features

Jong Wook Choe
Department of Mathematics, Statistical Data Science
San Francisco State University
jchoe3@sfsu.edu

1 Introduction

Have you ever wondered if the news is real or not? Because the internet became widespread, fake news has become an ever more prevalent means of manipulating public opinion.

The Nayirah testimony is a more well-known instance. On October 10, 1990, a 15-year-old Kuwaiti female gave false testimony before the US Congressional Human Rights Caucus. She reported on volunteering as a nurse at a Kuwaiti hospital during the invasion of Iraq. She testified that the Iraqi soldiers took premature infants out of incubators leaving them to die. This emotional account developed public support for U.S. military action.

However, as reported by the *New York Times*, it "was shown to be almost certainly false" in 1991.In 1992 it was revealed that she never became a nurse, but was the daughter of the Kuwaiti ambassador.

This raises a crucial question: **What should we believe, and what should we not?**

This project aims to build a machine learning model to classify news as real or fake based on word-frequency-driven binary features from titles and body text.

2 Data Description

The dataset consists of two BuzzFeed CSV files:

- `BuzzFeed_real_news_content.csv`
- `BuzzFeed_fake_news_content.csv`

We merged the datasets and encoded with a binary variable `news_type` (0 = fake, 1 = real), encoded with `LabelEncoder`.

2.1 Text Preprocessing

A custom preprocessing process was developed to prepare the text for analysis. This application runs through the pipeline. a process of transformations that normalizes text with less noise prior to extracting word-frequency features.

Preprocessing steps include:

- converting all text to lowercase,
- removing numbers and punctuation,
- removing special characters,
- removing English stopwords (using `sklearn`'s stopword list),
- applying Porter stemming to reduce words to their base form,

- stripping extra whitespace and tokenizing the cleaned text.

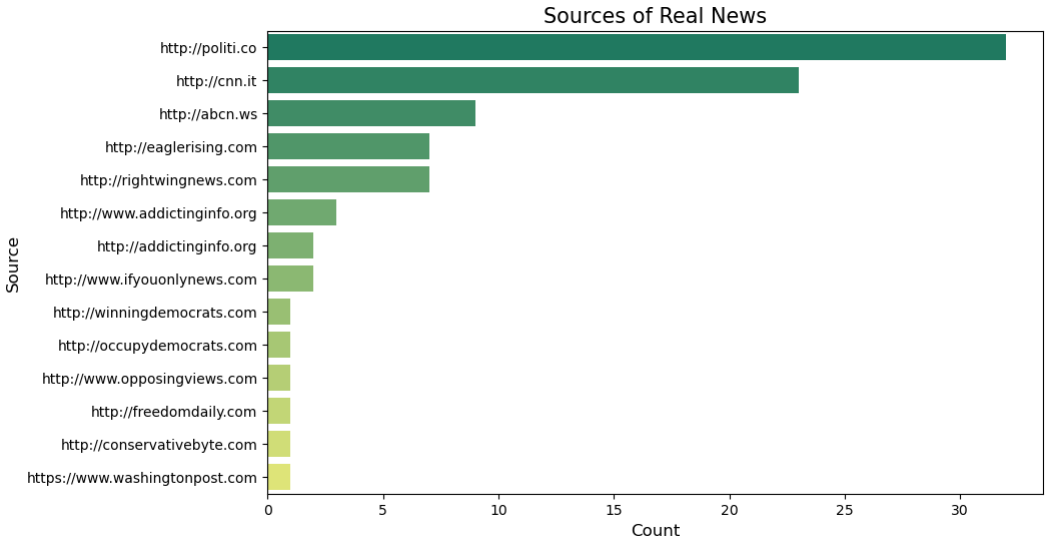
Using a functional composition approach, a single preprocessing was achieved in this step. function as the analyzer in the `CountVectorizer`. This guaranteed similar cleaning logic, which was deployed consistently through all EDA and modeling.

3 Exploratory Data Analysis (EDA)

The Exploratory Data Analysis focuses on understanding structural patterns, source characteristics, and linguistic differences between real and fake news articles in the BuzzFeed dataset. Since the final model employs engineered binary word indicators and examines the distributions of title and body text. was essential.

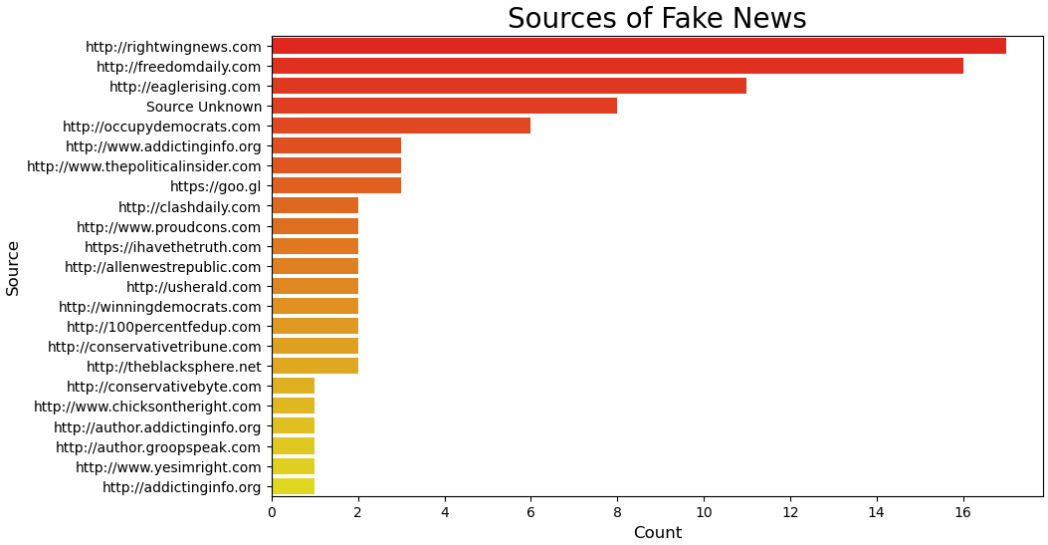
3.1 Source Distribution

First we did an academic look at domains Real and Fake news articles.



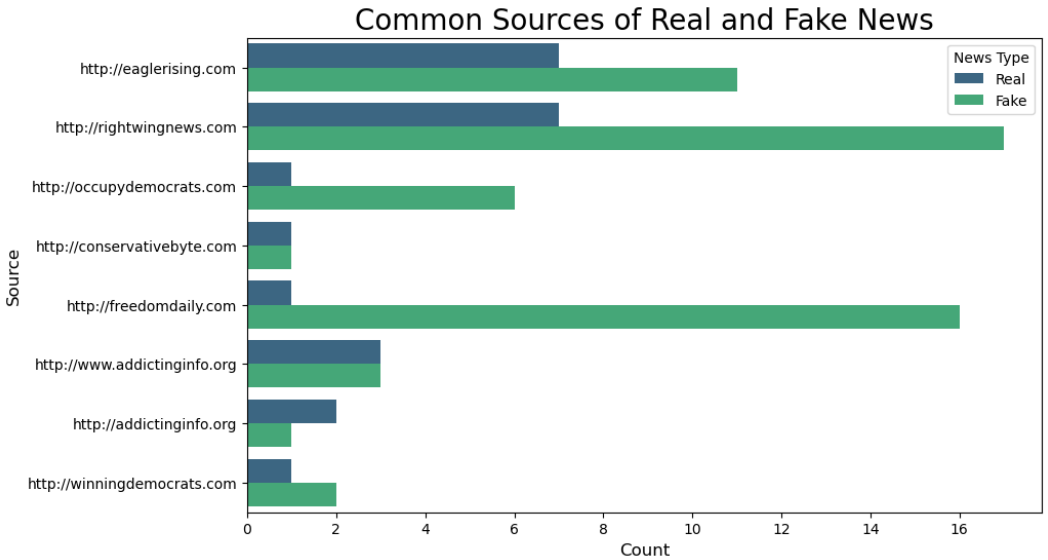
Top Sources of Real News Articles

Real news articles were concentrated among established outlets such as `cnn.it` and `politi.co`, with other domains appearing only once.



Top Sources of Fake News Articles

Fake news domains were more fragmented, with frequent appearances from `rightwingnews.com` and `freedomdaily.com`, but many sources appeared only once. There are also several articles with unknown sources, which reflect weaker metadata quality.



Shared Sources Between Real and Fake News

Some domains appeared in both real and fake subsets with the same frequency, confirming that the domain alone cannot reliably classify article validity.

3.2 Image and Video Link Distribution

We investigated embedded counts of multimedia as well.

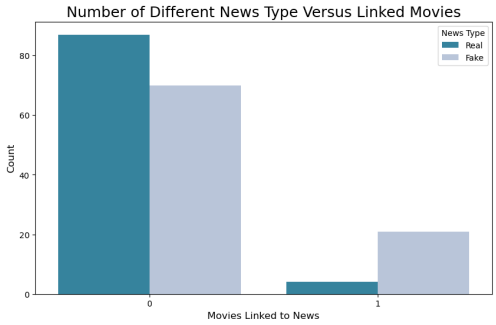
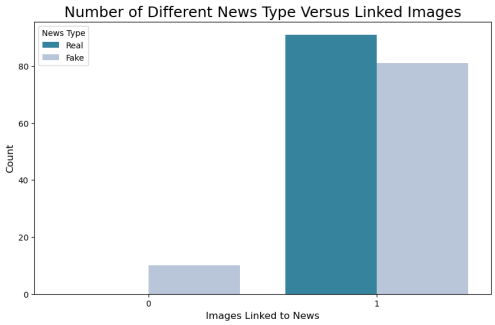


Image Link Distribution

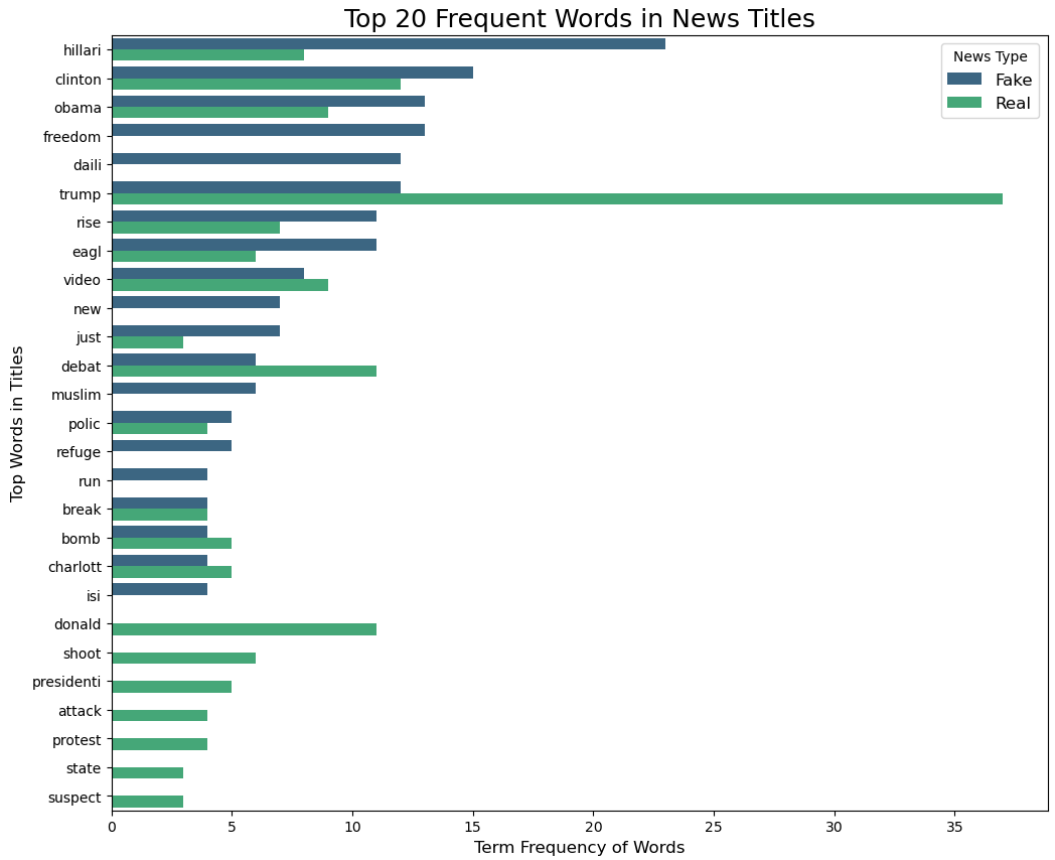


Video Link Distribution

Both distributions were extremely heavily skewed as we often saw zero image or video links in the articles. Just because these were non-informative raw counts, they became binary indicators `contain_images` and `contain_movies`.

3.3 Frequent Words in Titles

The most common words in article titles revealed strong polarization.



Top 20 Frequent Words in Titles (Real vs. Fake)

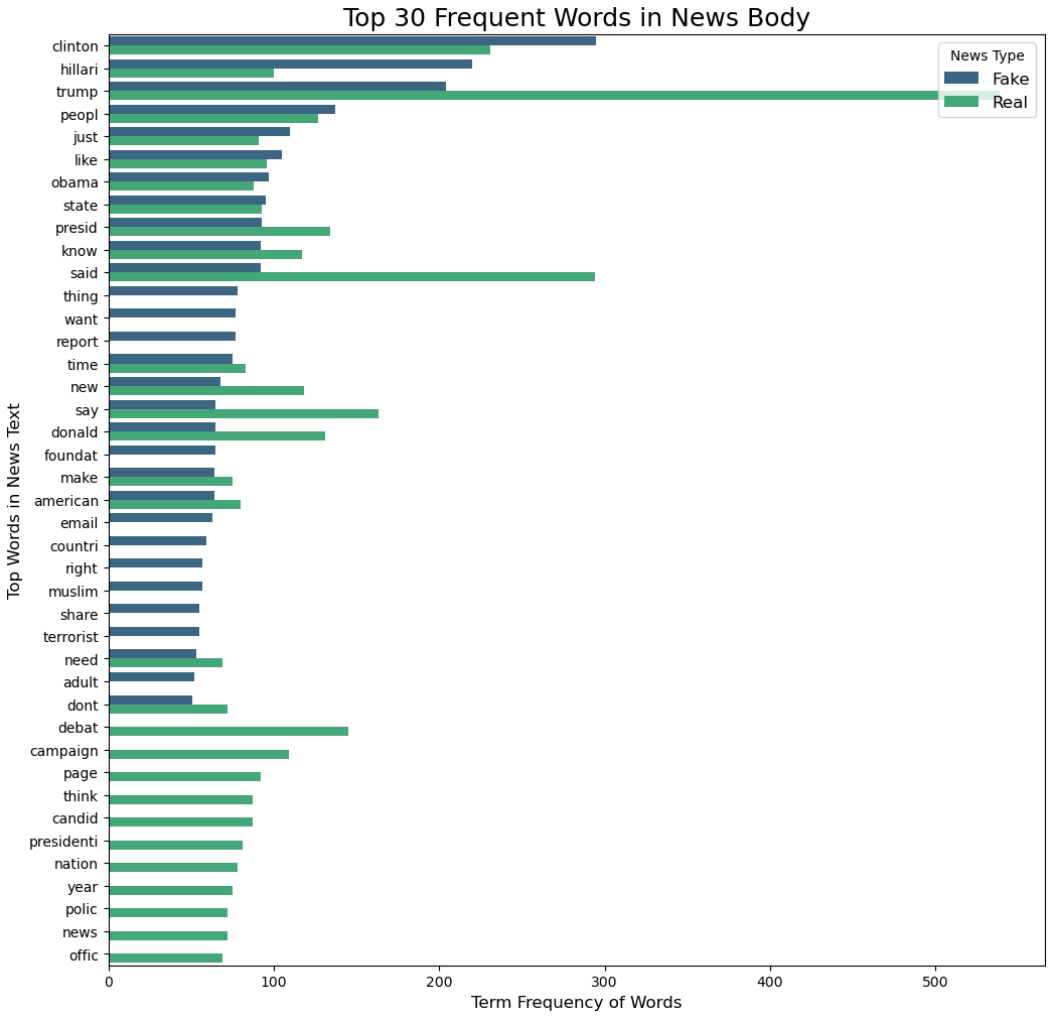
- **Fake News Titles:** High frequency of emotionally charged political names such as `hillari`, `clinton`, and `obama`.

- **Real News Titles:** Dominated by terms like `trump`, `donald`, and more neutral reporting words such as `presidenti`.

This motivated the creation of binary title-word indicators such as `fake_title_clinton` and `real_title_trump`.

3.4 Frequent Words in Body Text

We performed the same analysis on the full article text.



Top 30 Frequent Words in Body Text (Real vs. Fake)

- **Real News:** Extremely heavy usage of `trump`, appearing over 500 times.
- **Fake News:** Higher frequencies of `clinton`, `hillari`, and `obama` in body text.

These substantial lexical variations in both the titles and the body text led to a feature-engineering approach, which assesses accuracy depending on the various text inputs applied during modeling.

The EDA certainly demonstrates that real and fake articles contain well-defined and homogeneous vocabulary patterns, to be able to create binary indicator features for the most informative words and lays the basis for the subsequent classification models.

4 Data Analysis

4.1 Data Preprocessing

Before modeling, the BuzzFeed dataset was cleaned by removing non-informative columns such as `id`, `source`, URLs, and all metadata fields.

Also, the author information was removed because it did not contribute meaningfully to distinguishing real from fake articles.

After preprocessing, the dataset retained:

- `news_type` (0 = fake, 1 = real)
- Binary media indicators:
  - `contain_images`
  - `contain_movies`
- Text fields used for extracting word-frequency features (`title` and `text`)

4.2 Feature Engineering

Feature engineering focused on capturing the strongest lexical differences found in the EDA.

For both the titles and the body text, the following steps were taken

1. Find the top 5 most frequent words in real news.
2. Find the top 5 most frequent words in fake news.
3. Convert each of these words into a binary indicator feature showing whether the word appears in the article.

This produced two parallel feature sets

- **10 title-based features** (top 5 real-title words + top 5 fake-title words)
- **10 body-based features** (top 5 real-body words + top 5 fake-body words)

In total, the model used

- **10 binary lexical features**
- **2 media-indicator features** (`contain_images`, `contain_movies`)

Two different modeling datasets were constructed using these features. One with only title indicators as indicators, and one with indices for body only. The performance of multiple classifiers was compared over the two feature sets.

4.3 Model Training and Selection

We split the dataset into an 80% training set and 20% test set using stratified sampling. The following models were evaluated:

- Logistic Regression
- Random Forest
- Gradient Boosting
- Bagging (Decision Tree)
- KNN (k = 5 baseline)

4.3.1 Forward Feature Selection (SFS)

Sequential Forward Selection was applied using Logistic Regression as the base estimator. SFS selected the 10 most predictive engineered features. The model achieved

- **Accuracy with selected features:** 0.7297
- **Accuracy with all 20 engineered features:** 0.7297

Because performance was identical, the engineered feature set was already compact and effective.

4.3.2 Classifier Performance Comparison

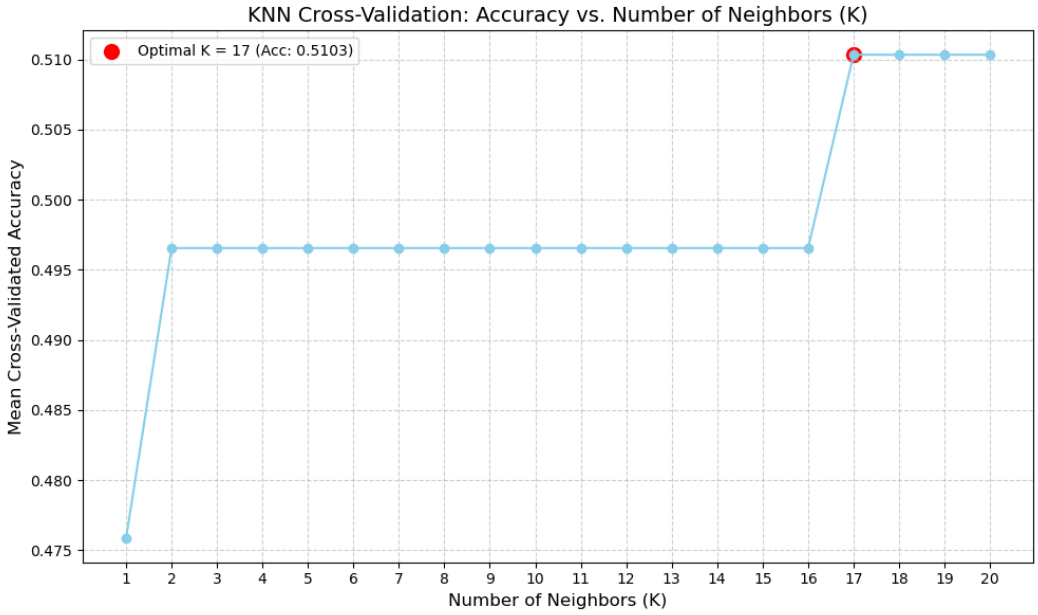
Table 1: Accuracy of All Models on Test Set

Model	Accuracy
Logistic Regression	0.7027
Random Forest	0.7027
Gradient Boosting	0.7027
Bagging (DT)	0.7027
KNN (k = 5)	0.5135

- Four models tied at the highest accuracy of 0.7027.
- KNN performed notably

4.4 K-Nearest Neighbors (KNN) Optimization

To evaluate whether KNN could improve upon the baseline performance, we ran a grid search over $k = 1$ to 20 using both uniform and distance-based weighting. The results are shown below.



Cross-validated accuracy across different values of k

Best configuration:

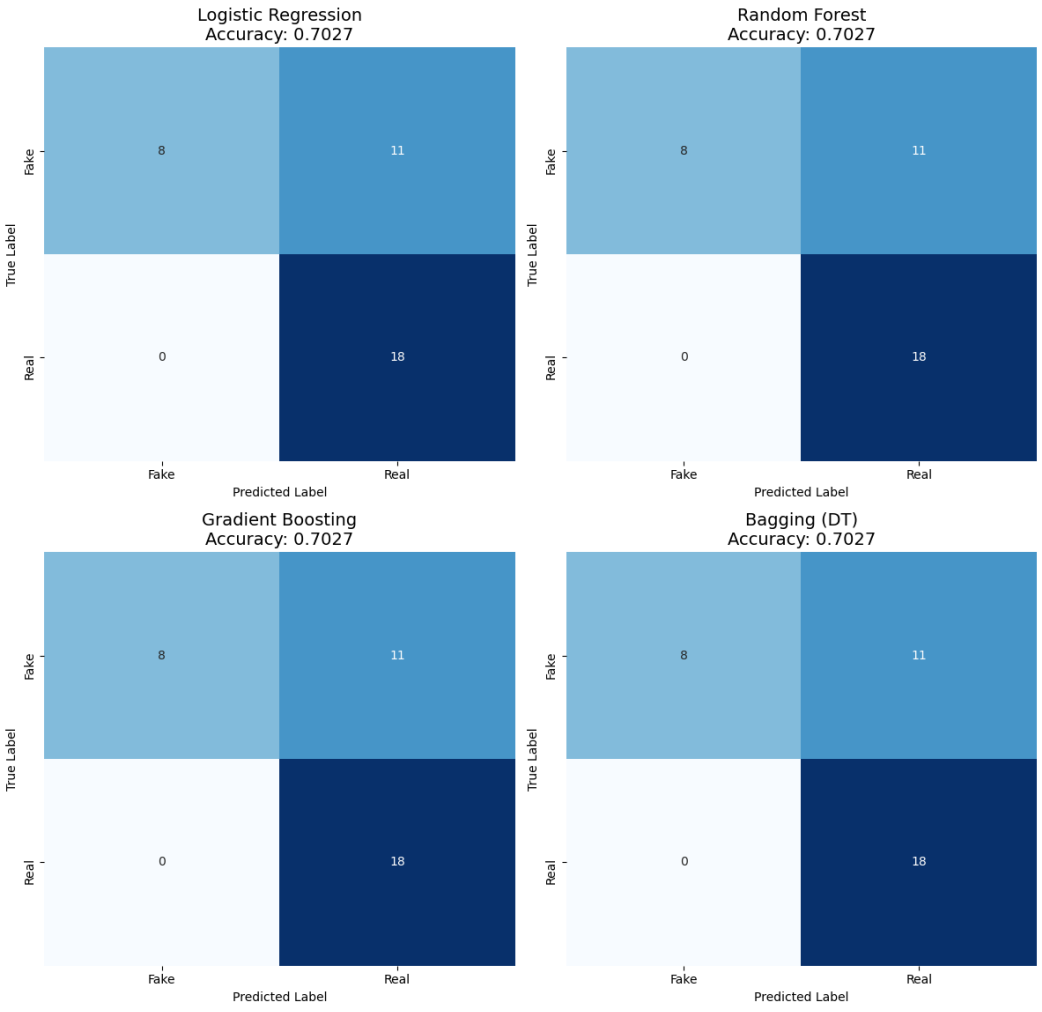
- Optimal k: **17**

- Weighting: **uniform**
- Cross-validated accuracy: **0.5103**

Despite tuning, KNN consistently underperformed due to the high-dimensional, sparse binary feature space. Its locality-based decision rule is not well-suited for this type of representation, making it the weakest model in this analysis.

4.5 Confusion Matrix Analysis

To better understand model behavior, we compared the confusion matrices of the four optimally successful models.



Confusion matrix comparison across top four models

- Found that all four models produced the same performance (accuracy = 0.7027).
- Every real article (18/18) was correctly classified.
- 11 fake articles were misclassified as real.

The confusion matrices demonstrate that these models are great at detecting real news, they are bad at spotting fake news that decreases the overall reliability.

5 Conclusion

This project explored fake news detection using engineered binary indicators based on the most frequently occurring real and fake words in titles and body text. Five classical machine-learning models were evaluated.

- Logistic Regression, Random Forest, Gradient Boosting, and Bagging all achieved the same accuracy: **0.7027**.
- KNN performed considerably worse, even after hyperparameter tuning.

The results show that the models captured real news more easily than fake news patterns, leading to perfect recall for real articles but frequent misclassification of fake articles as real. This shows the overlap between the two groups and the constraints of using simple binary word indicators. Overall, performance is also constrained by the small dataset of 182 samples, which limits model complexity and generalization. Future improvements could come from richer text representations, such as TF-IDF or embeddings, a larger and more diverse dataset, and methods that address class imbalance to reduce the model's bias toward predicting real news.

6 References

- "Remember Nayirah, Witness for Kuwait?" *New York Times*, 1992.
- Shu, Kai et al., "FakeNewsNet Dataset," arXiv, 2019.
- Mahudeswaran, Deepak. "FakeNewsNet." Kaggle, 2018.

7 Appendix

Listing 1: Notebook Code Excerpt

```
# %% [markdown]
# ### Exploratory Data Analysis on Fake News Dataset

# %%
# Import Required Library
import pandas as pd

# %%
# Load Datasets
Buzzfeed_f = pd.read_csv("data/BuzzFeed_fake_news_content.csv")
Buzzfeed_r = pd.read_csv("data/BuzzFeed_real_news_content.csv")

gossipcop_f =  pd.read_csv("data/gossipcop_fake.csv")
gossipcop_r =  pd.read_csv("data/gossipcop_real.csv")

politifact_f =  pd.read_csv("data/politifact_fake.csv")
politifact_r =  pd.read_csv("data/politifact_real.csv")


# %%
# Check Structure of Buzzfeed Fake Dataset
Buzzfeed_f.info()

# %%
# Check Structure of GossipCop Fake Dataset
gossipcop_f.info()


# %%
# Check Structure of PolitiFact Fake Dataset
```

```
politifact_f.info()

# %%
# Merge Real and Fake BuzzFeed Data
Buzzfeed_merge=pd.concat([Buzzfeed_r,Buzzfeed_f],axis=0)

# Extract News Type (Real or Fake) from 'id' Column
Buzzfeed_merge['news_type']=Buzzfeed_merge['id'].apply(lambda x: x.split('_')[0])

# Preview the Combined Dataset
Buzzfeed_merge.head()

# %%
# Check Shape (Rows, Columns)
Buzzfeed_merge.shape

# %%
# Check DataFrame Info After Merge
Buzzfeed_merge.info()

# %%
# Add Binary Columns Indicating Media Presence
Buzzfeed_merge['contain_movies']=Buzzfeed_merge['movies'].apply(lambda x: 0 if str(x)
Buzzfeed_merge['contain_images']=Buzzfeed_merge['images'].apply(lambda x: 0 if str(x)

# Drop Unnecessary Columns for Simplified Dataset
Buzzfeed_drop = Buzzfeed_merge.drop(['id','url',
                    'top_img',
                    'authors',
                    'publish_date',
                    'canonical_link',
                    'meta_data',
                    'movies',
                    'images'],axis=1)

# Preview the Dropped Dataset
Buzzfeed_drop.head()


# %%
# Confirm Data Types and Missing Values
Buzzfeed_drop.info()

# %%
# Duplicate the data for version control
Buzzfeed_clean = Buzzfeed_drop.copy()

#  Fill Missing 'source' Values
Buzzfeed_clean["source"] = Buzzfeed_drop["source"].fillna("Source Unknown")

# %%
Buzzfeed_clean

# %%
# Save Cleaned Data to CSV
output_directory = 'data/'  # Replace with your actual path
filename = 'Buzzfeed_data.csv'
full_path = output_directory + filename
```

```
# Export cleaned DataFrame to CSV without row index
Buzzfeed_clean.to_csv(full_path, index=False)


# %%
# Import Required Library
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import string
import nltk
import re

from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize, WhitespaceTokenizer
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS, CountVectorizer


# %%
Buzzfeed = pd.read_csv("data/BuzzFeed_data.csv")

# %%
real_order=Buzzfeed[Buzzfeed['news_type']=='Real']['source'].value_counts().sort_valu

# %%
real_order

# %%
plt.figure(figsize=(10,6))
sns.countplot(y='source', data=Buzzfeed[Buzzfeed['news_type']=='Real'],order=real_ord
plt.xlabel('Count',fontsize=12)
plt.ylabel('Source',fontsize=12)
plt.title('Sources of Real News',fontsize=15)
plt.show()

# %%
fake_order=Buzzfeed[Buzzfeed['news_type']=='Fake']['source'].value_counts().sort_valu

# %%
plt.figure(figsize=(10,6))
sns.countplot(y='source',data=Buzzfeed[Buzzfeed['news_type']=='Fake'],order=fake_orde
plt.xlabel('Count',fontsize=12)
plt.ylabel('Source',fontsize=12)
plt.title('Sources of Fake News',fontsize=20)
plt.show()

# %%
new=[]
for x in Buzzfeed[Buzzfeed['news_type']=='Fake']['source'].unique():
    if x in Buzzfeed[Buzzfeed['news_type']=='Real']['source'].unique():
        new.append(x)
print(new)

Buzzfeed_copy = Buzzfeed.copy()

Buzzfeed_copy['common']=Buzzfeed_copy['source'].apply(lambda x: x if x in new else 0
```

```
Buzzfeed_plot = Buzzfeed_copy[Buzzfeed_copy['common']!=0]

# %%
plt.figure(figsize=(10,6))
sns.countplot(y='common',data=Buzzfeed_plot,hue='news_type',palette='viridis')
plt.xlabel('Count',fontsize=12)
plt.ylabel('Source',fontsize=12)
plt.legend(loc='best', title='News Type',fontsize=10)
plt.title('Common Sources of Real and Fake News',fontsize=20)
plt.show()

# %%
plt.figure(figsize=(10,6))
sns.countplot(x='contain_movies', data=Buzzfeed_copy, hue='news_type', palette='PuBu
plt.xlabel('Movies Linked to News',fontsize=12)
plt.ylabel('Count',fontsize=12)
plt.legend(loc='best', title='News Type',fontsize=10)
plt.title('Number of Different News Type Versus Linked Movies',fontsize=18)
plt.show()

# %%
plt.figure(figsize=(10,6))
sns.countplot(x='contain_images', data=Buzzfeed_copy, hue='news_type', palette='PuBu
plt.xlabel('Images Linked to News',fontsize=12)
plt.ylabel('Count',fontsize=12)
plt.legend(loc='upper left', title='News Type',fontsize=10)
plt.title('Number of Different News Type Versus Linked Images',fontsize=18)
plt.show()

# %%
# Text Preprocessing Functions
ps = PorterStemmer()
wst = WhitespaceTokenizer()

# Lowercase
def lower_func(x):
    return x.lower()

# Remove numbers
def remove_number_func(x):
    return ''.join([a for a in x if not a.isdigit()])

# Remove punctuation
def remove_punc_func(x):
    return ''.join([a for a in x if a not in string.punctuation])

# Remove special characters
def remove_spec_char_func(x):
    return ''.join([a for a in x if a.isalnum() or a == ' '])

# Remove English stopwords (using sklearn)
def remove_stopwords(x):
    new = []
    for a in x.split():
        if a not in ENGLISH_STOP_WORDS:
            new.append(a)
    return " ".join(new)
```

```
# Stemming
def stem_func(x):
    wordlist = word_tokenize(x)
    psstem = [ps.stem(a) for a in wordlist]
    return '■'.join(psstem)

# Remove extra whitespaces
def remove_whitespace_func(x):
    return(wst.tokenize(x))

# Function composition helper
def compose(f, g):
    return lambda x: f(g(x))

# Final preprocessing pipeline
final = compose(
    compose(
        compose(
            compose(
                compose(
                    compose(remove_whitespace_func, stem_func),
                    remove_stopwords
                ),
                remove_spec_char_func
            ),
            remove_punc_func
        ),
        remove_number_func
    ),
    lower_func
)

# %%
nltk.download('punkt')
nltk.download('punkt_tab')


# %%
# Separate fake and real subsets
df_fake = Buzzfeed[Buzzfeed['news_type'] == 'Fake']
df_real = Buzzfeed[Buzzfeed['news_type'] == 'Real']

# Fake News Titles
cv1 = CountVectorizer(analyzer=final)
bow1 = cv1.fit_transform(df_fake['title'])
matrix1 = pd.DataFrame(bow1.toarray(), columns=cv1.get_feature_names_out())

# Sum word frequencies
matrix1_sum = matrix1.sum().sort_values(ascending=False).head(20)
top1 = matrix1_sum.reset_index()
top1.columns = ['word', 'sum']
top1['type'] = 'Fake'

# Real News Titles
cv2 = CountVectorizer(analyzer=final)
bow2 = cv2.fit_transform(df_real['title'])
matrix2 = pd.DataFrame(bow2.toarray(), columns=cv2.get_feature_names_out())

matrix2_sum = matrix2.sum().sort_values(ascending=False).head(20)
```

```
top2 = matrix2_sum.reset_index()
top2.columns = ['word', 'sum']
top2['type'] = 'Real'

# Combine for Visualization
conc1 = pd.concat([top1, top2])

plt.figure(figsize=(12, 10))
sns.barplot(y='word', x='sum', hue='type', data=conc1, palette='viridis')
plt.xlabel('Term Frequency of Words', fontsize=12)
plt.ylabel('Top Words in Titles', fontsize=12)
plt.title('Top 20 Frequent Words in News Titles', fontsize=18)
plt.legend(title='News Type', fontsize=12)
plt.show()


# %%
conc1.sort_values(by='word')

# %%
# Fake News Body
cv3 = CountVectorizer(analyzer=final)
bow3 = cv3.fit_transform(df_fake['text'])
matrix3 = pd.DataFrame(bow3.toarray(), columns=cv3.get_feature_names_out())

matrix3_sum = matrix3.sum().sort_values(ascending=False).head(30)
top3 = matrix3_sum.reset_index()
top3.columns = ['word', 'sum']
top3['type'] = 'Fake'

# Real News Body
cv4 = CountVectorizer(analyzer=final)
bow4 = cv4.fit_transform(df_real['text'])
matrix4 = pd.DataFrame(bow4.toarray(), columns=cv4.get_feature_names_out())

matrix4_sum = matrix4.sum().sort_values(ascending=False).head(30)
top4 = matrix4_sum.reset_index()
top4.columns = ['word', 'sum']
top4['type'] = 'Real'

# Combine and Visualize
conc2 = pd.concat([top3, top4])

plt.figure(figsize=(12, 12))
sns.barplot(y='word', x='sum', hue='type', data=conc2, palette='viridis')
plt.xlabel('Term Frequency of Words', fontsize=12)
plt.ylabel('Top Words in News Text', fontsize=12)
plt.title('Top 30 Frequent Words in News Body', fontsize=18)
plt.legend(title='News Type', fontsize=12)
plt.show()


# %%
# Define output directory
output_directory = 'data/'

# Save each top DataFrame to CSV
top1.to_csv(output_directory + 'top1_fake_title.csv', index=False)
top2.to_csv(output_directory + 'top2_fake_title.csv', index=False)
```

```
top3.to_csv(output_directory + 'top3_fake_title.csv', index=False)
top4.to_csv(output_directory + 'top4_fake_title.csv', index=False)


# %%
import pandas as pd
import numpy as np
import re
import seaborn as sns
import matplotlib.pyplot as plt


import string
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize, WhitespaceTokenizer
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

from sklearn.preprocessing import LabelEncoder

from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score,
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC

from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import GradientBoostingClassifier


import joblib
import os



# %%
# Load data
Buzzfeed = pd.read_csv('data/Buzzfeed_data.csv')
Buzzfeed_title = Buzzfeed.copy()
Buzzfeed_body = Buzzfeed.copy()

top1 = pd.read_csv('data/top1_fake_title.csv').head(5)
# top 5 fake title words
top2 = pd.read_csv('data/top2_real_title.csv').head(5)
# top 5 real title words
top3 = pd.read_csv('data/top3_fake_body.csv').head(5)
# top 5 fake body words
top4 = pd.read_csv('data/top4_real_body.csv').head(5)
# top 5 real body words

# %%
# Convert top words to list
```

```
fake_title_words = top1['word'].tolist()
real_title_words = top2['word'].tolist()
fake_body_words = top3['word'].tolist()
real_body_words = top4['word'].tolist()


# Helper to count occurrences of a word
# def count_word(text, word):
#     if isinstance(text, str):
#         return len(re.findall(rf'\b{word}\b', text))
#     else:
#         return 0

def has_word(text, word):
    if isinstance(text, str):
        return 1 if re.search(rf'\b{word}\b', text) else 0
    else:
        return 0

# %%
Buzzfeed

# %%
# Create fake title word columns
for word in fake_title_words:
    col_name = f"fake_title_{word}"
    Buzzfeed_title[col_name] = Buzzfeed_title['title'].apply(lambda x: has_word(x, w

# Create real title word columns
for word in real_title_words:
    col_name = f"real_title_{word}"
    Buzzfeed_title[col_name] = Buzzfeed_title['title'].apply(lambda x: has_word(x, w


# Create fake body word columns
for word in fake_body_words:
    col_name = f"fake_body_{word}"
    Buzzfeed_body[col_name] = Buzzfeed_body['text'].apply(lambda x: has_word(x, word

# Create real body word columns
for word in real_body_words:
    col_name = f"real_body_{word}"
    Buzzfeed_body[col_name] = Buzzfeed_body['text'].apply(lambda x: has_word(x, word

# %%
# Show new columns
# Start from your Buzzfeed_title DataFrame
title = Buzzfeed_title.copy()

# Drop columns that shouldn't be used as predictors
title = title.drop(columns=['title', 'text', 'source'])

# # Convert 'source' to categorical (one-hot encoding)
# title = pd.get_dummies(title, columns=['source'], drop_first=True)

# Encode the target variable (news_type)
le = LabelEncoder()
title['news_type'] = le.fit_transform(title['news_type'])
```

```
# %%
# Show new columns
# Start from your Buzzfeed_body DataFrame
body = Buzzfeed_body.copy()

# Drop columns that shouldn't be used as predictors
body = body.drop(columns=['title', 'text','source'])

# # Convert 'source' to categorical (one-hot encoding)
# body = pd.get_dummies(body, columns=['source'], drop_first=True)

# Encode the target variable (news_type)
le = LabelEncoder()
body['news_type'] = le.fit_transform(body['news_type'])


# %%
# Define features and target
X = title.drop(columns=['news_type'])
y = title['news_type']

# Now perform the train/test split:
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# %%
# Logistic Regression model (Forward Selection setup remains the same)
lr = LogisticRegression(max_iter=500)

# Forward Selection (choose 10 best features)
# k_features is set to 10 here, which means 10 non-'news_type' features will be selec
sfs_forward = SFS(
    lr,
    k_features=10,
    forward=True,
    floating=False,
    scoring='accuracy',
    cv=5
)

sfs_forward = sfs_forward.fit(X_train, y_train)

print("Selected Features (Forward):")
print(list(sfs_forward.k_feature_names_))

# %%
# Full Model (all features, excluding news_type)
lr_full = LogisticRegression(max_iter=1000, solver='liblinear')
lr_full.fit(X_train, y_train)

y_pred_full = lr_full.predict(X_test)
acc_full = accuracy_score(y_test, y_pred_full)

print(f"Full Model Accuracy (Corrected): {acc_full:.4f}")

# Selected Feature Model (using SFS)
selected_features = list(sfs_forward.k_feature_names_)
```

```
# X_train and X_test are now subsetted correctly using only the selected features
X_train_sfs = X_train[selected_features]
X_test_sfs = X_test[selected_features]

lr_sfs = LogisticRegression(max_iter=1000, solver='liblinear')
lr_sfs.fit(X_train_sfs, y_train)

y_pred_sfs = lr_sfs.predict(X_test_sfs)
acc_sfs = accuracy_score(y_test, y_pred_sfs)

print(f"Selected Feature Model Accuracy (Corrected): {acc_sfs:.4f}")

improvement = acc_sfs - acc_full
print(f"Accuracy Difference (SFS - Full): {improvement:.4f}")

# %%
# Define features and target
X = body.drop(columns=['news_type'])
y = body['news_type']

# Now perform the train/test split:
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# %%
# Logistic Regression model (Forward Selection setup remains the same)
lr = LogisticRegression(max_iter=500)

# Forward Selection (choose 10 best features)
# k_features is set to 10 here, which means 10 non-'news_type' features will be selec
sfs_forward = SFS(
    lr,
    k_features=10,
    forward=True,
    floating=False,
    scoring='accuracy',
    cv=5
)

sfs_forward = sfs_forward.fit(X_train, y_train)

# Full Model (all features, excluding news_type)
lr_full = LogisticRegression(max_iter=1000, solver='liblinear')
lr_full.fit(X_train, y_train)

y_pred_full = lr_full.predict(X_test)
acc_full = accuracy_score(y_test, y_pred_full)

print(f"Full Model Accuracy (Corrected): {acc_full:.4f}")

# Selected Feature Model (using SFS)
selected_features = list(sfs_forward.k_feature_names_)

# X_train and X_test are now subsetted correctly using only the selected features
X_train_sfs = X_train[selected_features]
X_test_sfs = X_test[selected_features]
```

```
lr_sfs = LogisticRegression(max_iter=1000, solver='liblinear')
lr_sfs.fit(X_train_sfs, y_train)

y_pred_sfs = lr_sfs.predict(X_test_sfs)
acc_sfs = accuracy_score(y_test, y_pred_sfs)

print(f"Selected Feature Model Accuracy (Corrected): {acc_sfs:.4f}")

improvement = acc_sfs - acc_full
print(f"Accuracy Difference (SFS - Full): {improvement:.4f}")

# %%
X = title.drop(columns=['news_type'])
y = title['news_type']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Define the new set of models for comparison
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000, solver='liblinear', ran

    # Ensemble Methods
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, learning_rate=
    "Bagging (DT)": BaggingClassifier(
        estimator=DecisionTreeClassifier(random_state=42),
        n_estimators=100,
        random_state=42
    ),

    # Baseline
    "K-Nearest Neighbors (k=5)": KNeighborsClassifier(n_neighbors=5)
}

results = []

# Loop through models and evaluate (Your existing loop structure)
print("Running model training and evaluation...")
for name, model in models.items():
    # Train the full model on X_train
    model.fit(X_train, y_train)

    # Predict on the hold-out test set
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)

    # Store results
    results.append({
        "Model": name,
        "Accuracy": accuracy
    })

# Create Comparison DataFrame
comparison_df = pd.DataFrame(results).sort_values(by='Accuracy', ascending=False).re
```

```
print("\n--- New Model Accuracy Comparison ---")
print(comparison_df)


# %%
# Define the 5 Models (as used in last comparison)
models_to_plot = {
    "Logistic Regression": LogisticRegression(max_iter=1000, solver='liblinear', ran
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, learning_rate=
    "Bagging (DT)": BaggingClassifier(
        estimator=DecisionTreeClassifier(random_state=42),
        n_estimators=100,
        random_state=42
    )
}

# Train Models and Get Predictions (Re-running the fit on X_train/y_train)
all_preds = {}
print("\nGenerating predictions for confusion matrix plots...")

# Assuming X_train, X_test, y_train, y_test are available from the previous block
for name, model in models_to_plot.items():
    model.fit(X_train, y_train)
    all_preds[name] = model.predict(X_test)

# Generate and Plot Confusion Matrices (3x2 Layout)
fig, axes = plt.subplots(2, 2, figsize=(12, 12))
plt.suptitle('Confusion Matrices Comparison (Top 4 Models on Engineered Features)',
axes = axes.flatten() # Flattens the 2x2 grid of axes into a 1D array

# Assuming 0=Fake, 1=Real
class_names = ['Fake', 'Real']

for i, (name, y_pred) in enumerate(all_preds.items()):
    cm = confusion_matrix(y_test, y_pred)
    ax = axes[i]

    # Calculate the accuracy for the title
    acc = accuracy_score(y_test, y_pred)

    sns.heatmap(
        cm,
        annot=True,
        fmt='d',
        cmap='Blues',
        cbar=False,
        xticklabels=class_names,
        yticklabels=class_names,
        ax=ax
    )

    ax.set_title(f'{name}\nAccuracy: {acc:.4f}', fontsize=14)
    ax.set_ylabel('True Label')
    ax.set_xlabel('Predicted Label')

# IMPORTANT: The deletion line (fig.delaxes(axes[5])) is removed as the 2x2 grid is

plt.tight_layout()
```

```
plt.savefig('confusion_matrices_2x2_comparison.png')
plt.close()

print("\nConfusion matrices visualization saved as 'confusion_matrices_2x2_comparison

# %%
# We will test k values from 1 up to 20. Odd numbers are usually preferred for binary
param_grid = {
    'n_neighbors': np.arange(1, 21),  # Test k from 1 to 20
    'weights': ['uniform', 'distance'] # Also check uniform vs. distance weighting
}

# Use Stratified K-Fold to ensure the proportion of Real/Fake news is maintained in
cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)


knn = KNeighborsClassifier()

grid_search = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    scoring='accuracy',  # Use accuracy as the metric to optimize
    cv=cv_strategy,
    verbose=1,
    n_jobs=-1  # Use all available cores for faster processing
)

print("Starting Grid Search for optimal K...")
grid_search.fit(X_train, y_train)

best_k = grid_search.best_params_['n_neighbors']
best_weights = grid_search.best_params_['weights']
best_score = grid_search.best_score_

print("\n--- Cross-Validation Results ---")
print(f"Optimal K (n_neighbors): {best_k}")
print(f"Optimal Weights: {best_weights}")
print(f"Cross-Validated Accuracy: {best_score:.4f}")

# %%
# 1. Define the Optimized KNN Model
# Use the best parameters found by the Grid Search cross-validation
knn_optimized = KNeighborsClassifier(
    n_neighbors=best_k,
    weights=best_weights
)

# 2. Train the Model (on the entire training set)
print(f"Training optimized KNN model with K={best_k} and weights='{best_weights}'...
knn_optimized.fit(X_train, y_train)

# 3. Predict on the Test Set
y_pred_optimized = knn_optimized.predict(X_test)

# 4. Calculate Final Test Accuracy
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)

print("\n--- Optimized K-Nearest Neighbors Results ---")
print(f"Test Accuracy (Optimal K={best_k}): {accuracy_optimized:.4f}")
```

```
# %%
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Assuming the 'grid_search' object is available from the previous step
results = pd.DataFrame(grid_search.cv_results_)

# Filter results for the best weight ('uniform') and extract k values and scores
best_weights = 'uniform'
filtered_results = results[results['param_weights'] == best_weights]

# Extract mean scores and the k values
mean_scores = filtered_results['mean_test_score']
k_values = filtered_results['param_n_neighbors'].apply(lambda x: x)

# Find the best score index
best_k_index = np.argmax(mean_scores)

# --- Plotting Code ---
plt.figure(figsize=(10, 6))
plt.plot(k_values, mean_scores, marker='o', linestyle='-', color='skyblue')

# Highlight the best K=17
plt.scatter(
    k_values.iloc[best_k_index],
    mean_scores.iloc[best_k_index],
    color='red',
    s=100,
    label=f'Optimal K = {k_values.iloc[best_k_index]} (Acc: {mean_scores.iloc[best_k
)

plt.title('KNN Cross-Validation: Accuracy vs. Number of Neighbors (K)', fontsize=14)
plt.xlabel('Number of Neighbors (K)', fontsize=12)
plt.ylabel('Mean Cross-Validated Accuracy', fontsize=12)
plt.xticks(k_values, rotation=0)
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()
plt.close()

# %%
try:
    import tensorflow as tf
    print(f"TensorFlow is installed. Version: {tf.__version__}")
except ImportError:
    print("TensorFlow is NOT installed. You can install it using: pip install tensorf

# %%
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# 1. Split the data
X_train_raw, X_test_raw, y_train, y_test = train_test_split(
    Buzzfeed['title'], Buzzfeed['news_type'], test_size=0.2, random_state=42, stratif
)
```

```
# 2. Tokenize the text
max_words = 10000  # Vocabulary size
tokenizer = Tokenizer(num_words=max_words, oov_token="<unk>")
tokenizer.fit_on_texts(X_train_raw)

# 3. Convert text to sequences and pad
maxlen = 20 # Maximum length for a title
X_train_seq = tokenizer.texts_to_sequences(X_train_raw)
X_test_seq = tokenizer.texts_to_sequences(X_test_raw)

X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen, padding='post', truncating='
X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen, padding='post', truncating='po

# %%
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

# Model parameters
embedding_dim = 100

# Build the LSTM model
model = Sequential([
    # Input layer: Turns index sequence into dense vectors (embeddings)
    Embedding(max_words, embedding_dim, input_length=maxlen),

    # LSTM layer: The core recurrent layer to capture sequential information
    LSTM(64),

    # Dropout for regularization (to prevent overfitting)
    Dropout(0.5),

    # Output layer: Sigmoid activation for binary classification (Real/Fake)
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model (Using 10 epochs as a starting point)
history = model.fit(
    X_train_pad,
    y_train,
    epochs=10,
    batch_size=32,
    validation_data=(X_test_pad, y_test)
)

# Evaluate on the test set
loss, accuracy = model.evaluate(X_test_pad, y_test, verbose=0)
print(f"\nLSTM Model Test Accuracy: {accuracy:.4f}")

# %%
# Text Preprocessing Functions

ps = PorterStemmer()
wst = WhitespaceTokenizer()

# Lowercase
def lower_func(x):
```

```
    return x.lower()

# Remove numbers
def remove_number_func(x):
    return ''.join([a for a in x if not a.isdigit()])

# Remove punctuation
def remove_punc_func(x):
    return ''.join([a for a in x if a not in string.punctuation])

# Remove special characters
def remove_spec_char_func(x):
    return ''.join([a for a in x if a.isalnum() or a == '■'])

# Remove English stopwords (using sklearn)
def remove_stopwords(x):
    new = []
    for a in x.split():
        if a not in ENGLISH_STOP_WORDS:
            new.append(a)
    return "■".join(new)

# Stemming
def stem_func(x):
    wordlist = word_tokenize(x)
    psstem = [ps.stem(a) for a in wordlist]
    return '■'.join(psstem)

# Remove extra whitespaces
def remove_whitespace_func(x):
    return(wst.tokenize(x))

# Function composition helper
def compose(f, g):
    return lambda x: f(g(x))

# Final preprocessing pipeline
final = compose(
    compose(
        compose(
            compose(
                compose(
                    compose(remove_whitespace_func, stem_func),
                    remove_stopwords
                ),
                remove_spec_char_func
            ),
            remove_punc_func
        ),
        remove_number_func
    ),
    lower_func
)

# %%


# %%
# Split features and target
```

```
X = Buzzfeed['title']
y = Buzzfeed['news_type']

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state

# Preprocessing + RandomForest pipeline
pp = Pipeline([
    ('bow', CountVectorizer(analyzer=final)),
# final is your preprocessing function
    ('tfidf', TfidfTransformer()),
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Fit model
pp.fit(X_train, y_train)

# Predictions
predictions = pp.predict(X_test)

# Evaluate
print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))

# %%
# Split features and target
X = Buzzfeed['text']
y = Buzzfeed['news_type']

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state

# Preprocessing + RandomForest pipeline
pp = Pipeline([
    ('bow', CountVectorizer(analyzer=final)),
# final is your preprocessing function
    ('tfidf', TfidfTransformer()),
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Fit model
pp.fit(X_train, y_train)

# Predictions
predictions = pp.predict(X_test)

# Evaluate
print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))

# %%
# Define models to compare
models = {
    "RandomForest": RandomForestClassifier(n_estimators=100, random_state=42),
    "LogisticRegression": LogisticRegression(max_iter=500, random_state=42),
    "NaiveBayes": MultinomialNB(),
    "SVM": SVC(kernel='linear', random_state=42)
}
```

```
# Feature types
feature_types = {
    "Title": Buzzfeed['title'],
    "Body": Buzzfeed['text']
}

results = []

for feat_name, X in feature_types.items():
    y = Buzzfeed['news_type']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

    for model_name, model in models.items():
        pipeline = Pipeline([
            ('bow', CountVectorizer(analyzer=final)),
            ('tfidf', TfidfTransformer()),
            ('clf', model)
        ])

        pipeline.fit(X_train, y_train)
        preds = pipeline.predict(X_test)

        results.append({
            "Feature": feat_name,
            "Model": model_name,
            "Accuracy": accuracy_score(y_test, preds),
            "Precision": precision_score(y_test, preds, pos_label='Real'),
            "Recall": recall_score(y_test, preds, pos_label='Real'),
            "F1": f1_score(y_test, preds, pos_label='Real')
        })

# Convert results to DataFrame
results_df = pd.DataFrame(results)

# Pivot table to show only accuracy
accuracy_df = results_df.pivot(index='Model', columns='Feature', values='Accuracy')
accuracy_df = accuracy_df.reset_index()
accuracy_df

# %%
# Pivot table to show only accuracy
accuracy_df = results_df.pivot(index='Model', columns='Feature', values='Accuracy')
accuracy_df = accuracy_df.reset_index()
accuracy_df

# %%
# lambda cause error to download
def finals(text):
    text = lower_func(text)
    text = remove_number_func(text)
    text = remove_punc_func(text)
    text = remove_spec_char_func(text)
    text = remove_stopwords(text)
    text = stem_func(text)
    text = '■'.join(remove_whitespace_func(text))
    return text

# Split features and target
X = Buzzfeed['text']
```

```
y = Buzzfeed['news_type']

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Preprocessing + SVM pipeline
svm_pipeline = Pipeline([
    ('bow', CountVectorizer(analyzer=finals)),
# your preprocessing function
    ('tfidf', TfidfTransformer()),
    ('classifier', SVC(kernel='linear', random_state=42))
])

# Fit model
svm_pipeline.fit(X_train, y_train)


# Path to folder
os.makedirs('data', exist_ok=True)

# Save your trained pipeline inside the data folder
joblib.dump(svm_pipeline, 'data/svm_body_model.pkl')
```