Fake News Detection

Project Report-(Group13)

MBA750M: SOCIAL MEDIA ANALYTICS II

Submitted By:
Harsh (231140016)
Rohit (231140019)

Submitted To:
Dr. Shankar Prawes
DoMS Department
IIT Kanpur

INTRODUCTION

In the contemporary digital landscape, the rapid proliferation of information through online platforms has transformed the way individuals consume news. While this accessibility fosters informed communities, it also opens avenues for the spread of misinformation and deceptive content, collectively known as "fake news." The dissemination of fake news poses significant threats, including eroding public trust, influencing political outcomes, and inciting social discord. Consequently, developing effective mechanisms to identify and mitigate fake news is imperative for maintaining the integrity of information ecosystems.

This project focuses on implementing a machine learning-based approach to detect fake news articles. The dataset utilised comprises three distinct CSV files: training, testing, and submission sets. The training dataset serves as the foundation for building and refining the predictive model. Each entry in this dataset includes a unique identifier, the title of the news article, the author's name, the main textual content, and a binary label indicating the veracity of the news—where a label of 1 denotes fake news, and 0 signifies reliable information.

The dataset includes a wide range of topics, with a significant focus on political and world news. The data is provided in two CSV files. The first file, "train.csv," has over 20,800 articles for training purposes, while the second file, "test.csv," contains over 5,200 articles from various news sources, some of which are known to publish fake news. Each article entry includes an ID, title, author, text content, and a label indicating its classification (fake or real). The structure of the training data is (20800, 5), and the test data is (5200, 5).

Label	Count
1	10413
0	10387

Table 1: Label Distribution

Based on the above table, the training dataset is balanced with 10,413 instances of fake news (label 1) and 10,387 instances of reliable news (label 0). Since the data contains an equal number of fake and reliable news articles, there's no need for techniques like oversampling, undersampling, or class weighting to address class imbalance.

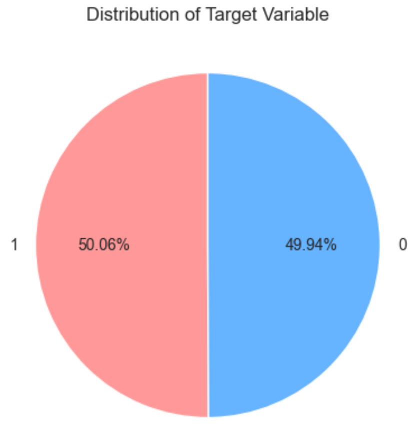


Figure 1: Label Distribution

Exploratory Data Analysis (EDA)

Token Descriptive Statistics:

Variables\range	Min	1st Quartile	Median	Mean	3rd Quartile	Max
Title	0	7	9	8.49	10	43
Author	0	2	2	2.10	2	16
Text	0	152	316	426.88	590	12445

Table 2: Token Statistics

The mean text length suggests articles have substantial content, with length variations helping distinguish genuine, in-depth reporting from sensationalised fake news.

The high standard deviation in text length indicates significant variability, with zero token counts showing missing values. Including token counts as features allows the model to handle this variability and data imbalance, helping it recognise length-related patterns that often separate real from fake news.

Token counts also support imputation by guiding how missing values are filled. For example, titles with low token counts might be imputed differently from longer ones to maintain data distribution.

MISSING VALUES

Based on Table 2, we can say there are some missing values in the training data as there token count is zero. The train dataset has some missing information in a few columns. In the *author* column, there are 1,957 missing values, which is about 9.4% of the data. The *title* column has 558 missing entries, making up around 2.7% of the data. The *text* column has 39 values listed as NaN (missing), which is 0.2% of the data. In total, there are 116 missing values in the *text* column; 39 of these are marked as NaN, while the rest are empty, meaning they have no content. It will be important to handle these missing values carefully to make sure our model is accurate and reliable.

	Train Data		**Test Data**	
	MISSING COUNT	**MISSING COUNT**	**MISSING COUNT**	**MISSING COUNT**
AUTHOR	1957	9.41	503	9.67
TITLE	558	2.68	122	2.35
TEXT	39	0.19	7	0.13

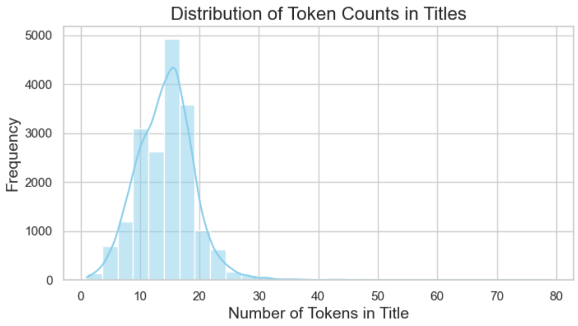
Table 3: Missing Value Distribution

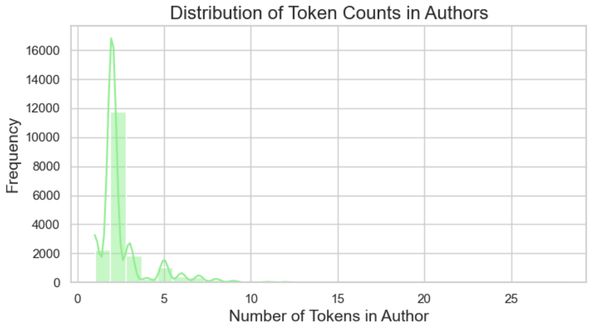
The test dataset has some missing information in a few columns. The *author* column is missing 503 values, which is about 9.7% of the data. The *title* column has 122 missing entries, making up around 2.4% of the dataset. In the *text* column, there are 7 values marked as NaN, which is 0.13% of the data. In total, there are 38 missing values in the *text* column; 7 are marked as NaN, while the remaining are empty with no actual content. Handling these missing values properly is important for accurate analysis and reliable model performance.

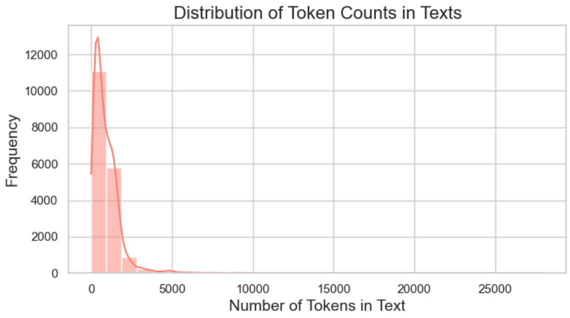
Missing Value Imputation:

To impute missing entries for features like author, title, and text, we use a systematic approach involving feature engineering, similarity matching, and weighted imputation. First, we identify missing values and combine relevant text fields (like title and main text) into a single representation, vectorized with TF-IDF to capture essential terms and context using unigrams and bigrams. We also scale numerical features, like token counts, for consistency. Using **a K-Nearest Neighbours (KNN)** model, we then identify similar entries based on cosine similarity across both text and numerical features. For imputation, a weighted approach assigns greater influence to closer neighbours, and for categorical fields, the highest-weighted value is chosen as the imputed entry. This process is applied throughout the dataset, improving data completeness by leveraging patterns from similar entries.

Token Distribution







Title Token Counts: The histogram shows a narrow, symmetric distribution, with most titles containing 15-20 tokens. Few exceed 30 tokens, reflecting concise news titles. Given the limited length, titles alone might not provide a large vocabulary for classification. However, they could serve as valuable keywords to help differentiate between fake and real news if certain words are more likely to appear in one category over the other.

Author Token Counts: Author names mostly have 1-2 tokens, typically just first and last names, with a few exceptions having more tokens (e.g., middle names or suffixes). This results in a limited vocabulary.
Text Token Counts: The text token distribution is highly skewed, with most articles under 2000 tokens but some outliers reaching 25,000. Token count serves as a feature to manage variations in article length, potentially correlating with news authenticity.

To address these variations, we use token count as a feature to address these changes in token length in correlation with real or fake news.

Data Preprocessing:

Convert to Lowercase: Converts all text to lowercase to ensure uniformity and prevent case-based inconsistencies using text.lower().

Remove URLs: Removes URLs to clean data and focus on content, using re.sub(r'http\S+', '', text).

Remove Non-Alphanumeric Characters: Removes punctuation and symbols to reduce noise, using re.sub(r'[^a-zA-Z0-9\s]', '', text).

Remove Stopwords and Tokenize: Splits text into tokens and removes common stopwords to focus on meaningful words with if word not in stop_words.

Lemmatize Words: Reduces words to their base forms (e.g., "running" to "run") using lemmatizer.lemmatize(word) for consistency**.**

Data Preparation:

The final dataset incorporated a combination of raw textual data and numerical features representing the complexity and length of the text. Specifically, the following components were utilised to build the predictive model:

- **Title Text:** The actual content of the article's title, which is obtained by combining Title, Author and Text, providing contextual and semantic information.
- **Title Token Count:** A numerical representation of the number of words in the title, offering insights into the title's length and potential verbosity.
- **Text Token Count:** The total number of words in the main body of the article, indicating the article's depth and coverage.
- **Author Token Count:** The number of words in the author's name, which can capture variations in name lengths and possibly reflect on author-related patterns.

Model

Frequency Based Model:

This model classifies text data by extracting features from titles and content. Key steps include:

1. Data Preparation: Text features are created from titles and content, with the label as the target for classification.
2. Train-Test Split: Training data is divided into an 80/20 split to allow for training and separate evaluation using cross-validation.
3. Pipeline Construction:
   - *Text Vectorization*: Transforms text into token counts using unigrams and bigrams, limits vocabulary, and removes stopwords to capture context while managing memory.
   - *Feature Selection*: Reduces the feature space to 20,000 using chi-squared testing to focus on relevant terms.
   - *Classification*: A classifier with smoothing handles discrete text data.
4. Hyperparameter Tuning: Grid search with cross-validation optimises model parameters for accuracy.
5. Evaluation: The final model is tested on the held-out data, with accuracy and other metrics assessing classification performance.

Result

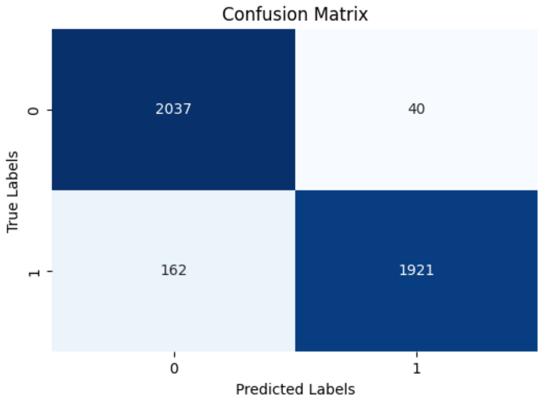


Figure 2 : Confusion Matrix for Cross Validation Data

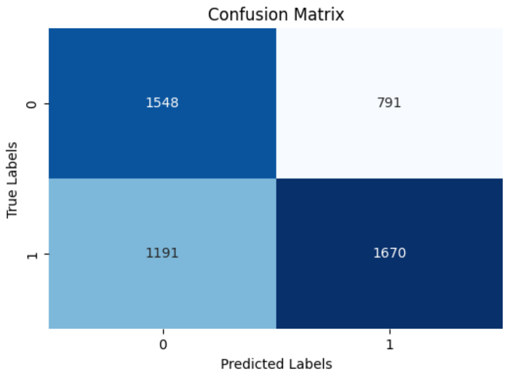


Figure 3 : Confusion Matrix for Test Data

Classification Report (Cross Validation Data): <u>**Cross Validation Accuracy: 95.14%**</u>

	Precision	**Recall**	**F1-score**
1	0.98	0.92	0.95
0	0.93	0.98	0.95

Table 4: Cross Validation Data Evaluation

Classification Report (Test Data): <u>**Test Data Accuracy: 61.69%**</u>

	Precision	**Recall**	**F1-score**
1	0.68	0.58	0.63
0	0.56	0.66	0.61

Table 5: Test Data Evaluation

- ## Tf-Idf Based Model

This model aims to build a well-tuned Naive Bayes classifier that combines TF-IDF vectorization, feature scaling, and chi-squared feature selection within an organised pipeline.

1. Data Preparation: Both text data and numerical token counts are combined to enrich the feature set. The data is split into training and cross-validation sets in a ratio of 80:20.
2. Pipeline Creation:
   - *Text Processing and Scaling*: A ColumnTransformer applies TF-IDF vectorization to text. L2 regulariser and MinMax scaling to numerical features, enabling the model to handle diverse data types.
   - *Feature Selection*: Chi-squared feature selection retains only the most relevant 20,000 features, reducing model complexity and enhancing efficiency.
   - *Classification*: A Naive Bayes classifier is applied, ideal for text data due to its feature independence assumption.
3. Hyperparameter Tuning: A grid search with 5-fold cross-validation tests different parameters for TF-IDF, n-grams, and Naive Bayes smoothing to optimize accuracy.
4. Evaluation: The final model is tested on the held-out data, with accuracy and other metrics assessing classification performance.

Result:

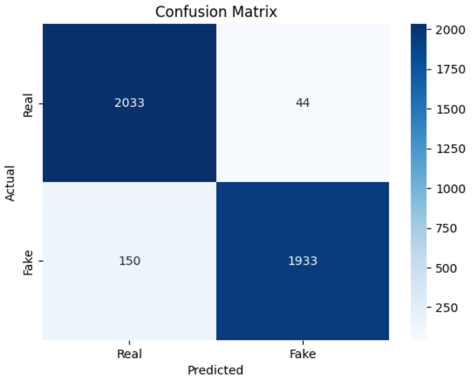


Figure 3 : Confusion Matrix for Cross Validation Data

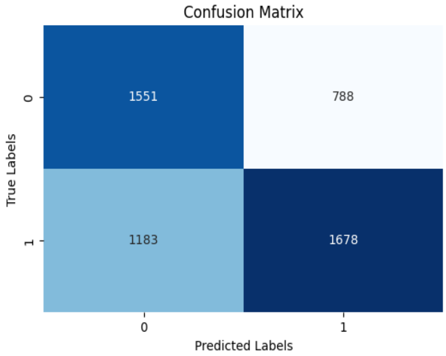


Figure 3: Confusion Matrix for Test Data

Classification Report (Cross Validation Data): **Cross Validation Accuracy: 95.34%**

	Precision	Recall	F1-score
1	0.98	0.93	0.95
0	0.93	0.98	0.95

Table 6: Train Data Evaluation

Classification Report (Test Data): **Test Set Accuracy: 62.09%**

	Precision	Recall	F1-score
1	0.68	0.59	0.63
0	0.57	0.66	0.61

Table 7: Test Data Evaluation

Conclusion

	Cross-Validation Accuracy (%)	Test Accuracy (%)
Frequency Based Model	95.14	61.69
Tf-Idf Based Model	95.4	62.1

Table 8: Model Performance

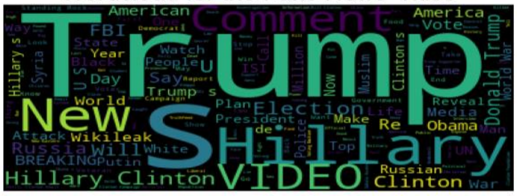
Both the frequency-based and TF-IDF models show excellent performance on the cross-validation data, with accuracies around 95% and high F1-scores for both classes. This indicates that the models are capturing key distinguishing features between real and fake news within the training data, demonstrating a solid fit on known data. Specifically, each class shows high precision and recall, implying that the models effectively classify both real and fake news in the cross-validation phase.

However, on the test data, both models perform significantly worse, with accuracies dropping to significantly lower test accuracies (61.69% and 62.1%). This drop suggests that the models are overfitting the training data, capturing specific patterns that do not generalise well to unseen data. It may also mean the separate testing dataset might have a different distribution, or characteristics compared to the training/test sets effective .The test classification reports further highlight this issue: for both models, the F1-scores for each class decrease sharply, particularly with a noticeable dip in recall for fake news (class "1"). This lower recall implies that the models fail to flag a substantial portion of fake news instances in new data, which is a critical shortfall for fake news detection.

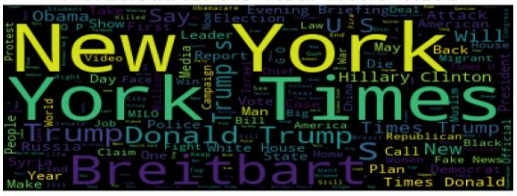
Appendix:

Word cloud Representation

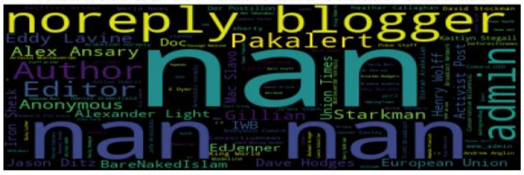
Wordcloud for the Fake News' Titles



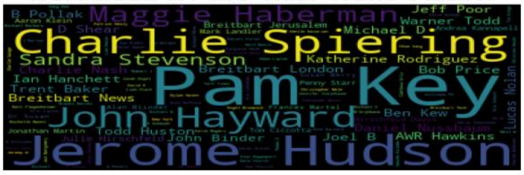
Wordcloud for the 'Real' News' Titles



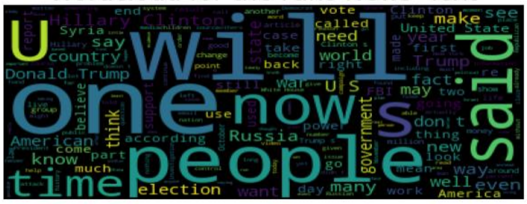
Wordcloud for the Fake News' Author



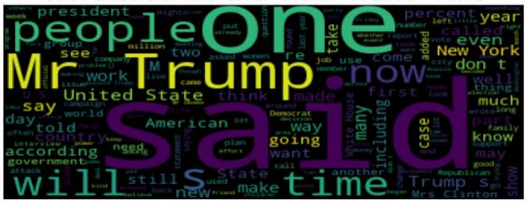
Wordcloud for the 'Real' News' Author



Wordcloud for the Fake News' Text



Wordcloud for the 'Real' News' Text



Top 30 Token in each category

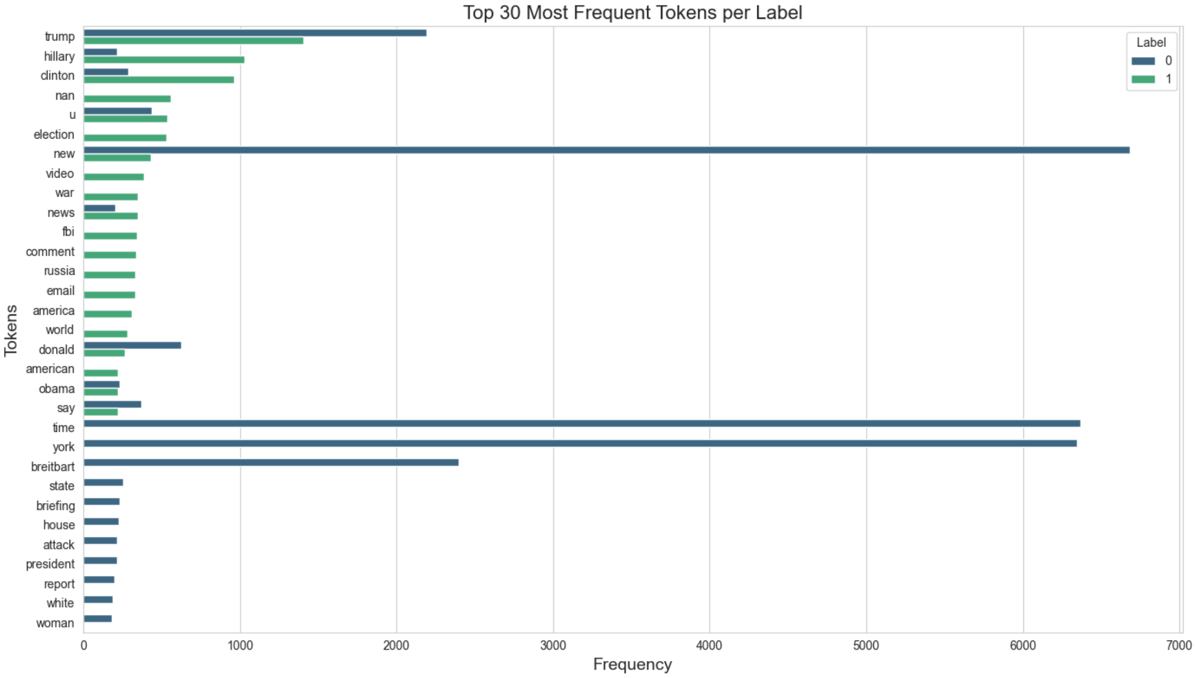


Figure 4: Title

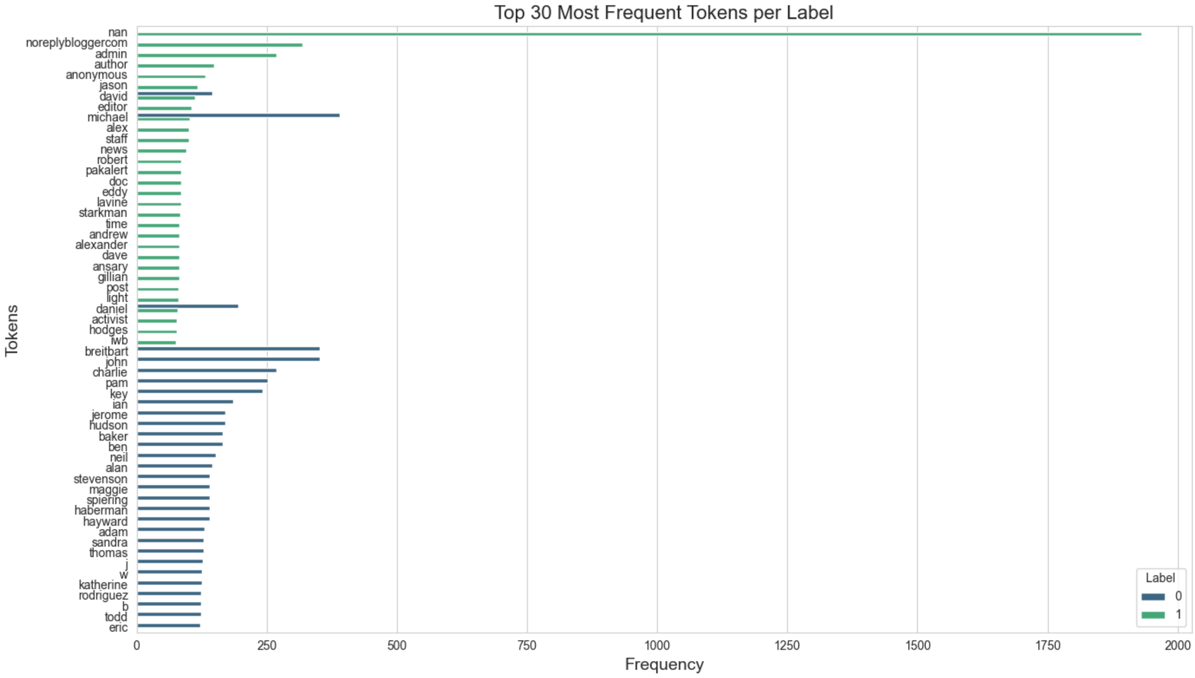


Figure 5: Author

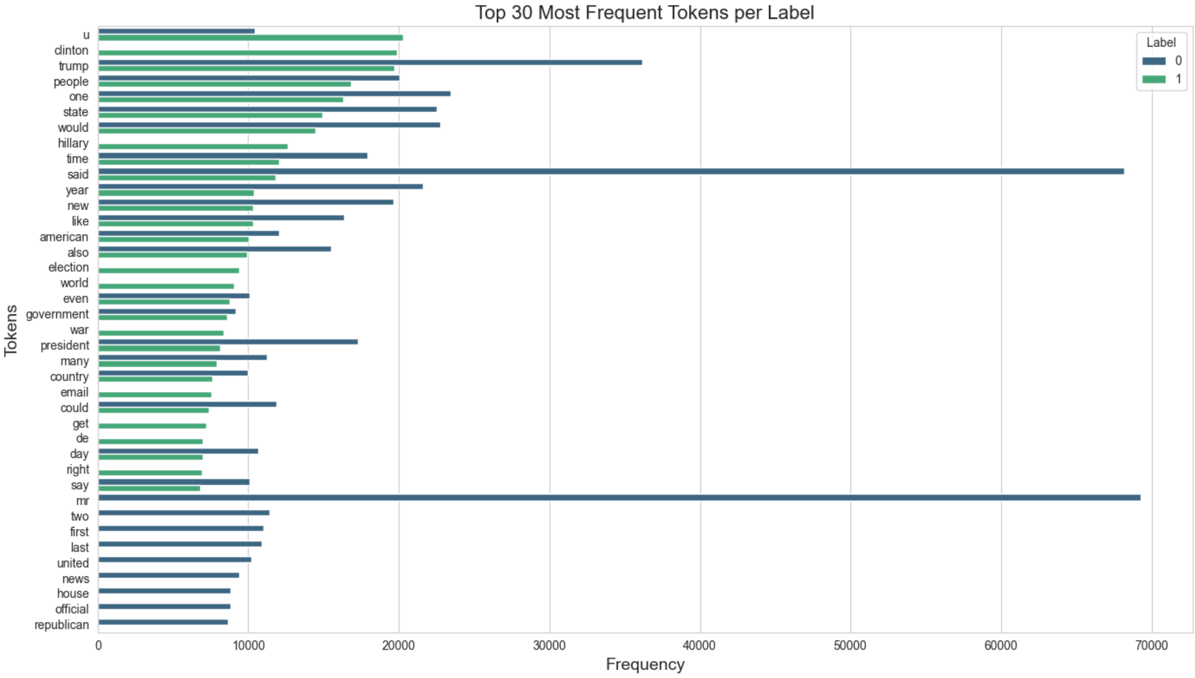


Figure 6: Text

CODE

Code for Missing Value Imputation:

```
# Identify missing 'author' entries
missing_author = df['author'].isnull() | (df['author'].str.strip() == '')
print(f"Number of missing 'author' entries: {missing_author.sum()}")

# Feature Engineering for 'author' imputation
# Combine 'title' and 'text' for similarity features
df['combined_features_author'] = df['title'].fillna('') + ' ' + df['text'].fillna('')

# Vectorize combined features using TF-IDF
tfidf_author_vectorizer = TfidfVectorizer(
    max_features=5000,
    ngram_range=(1, 2),
    stop_words='english'
)
tfidf_author_matrix = tfidf_author_vectorizer.fit_transform(df['combined_features_author'])

# Scale numerical token counts
scaler_author = StandardScaler()
token_counts_author_scaled = scaler_author.fit_transform(df[['title_token_count',
'text_token_count']])

# Combine TF-IDF and token counts
X_author = hstack([tfidf_author_matrix, token_counts_author_scaled])

# Initialize and fit KNN on non-missing 'author' entries
knn_author = NearestNeighbors(n_neighbors=5, metric='cosine')
knn_author.fit(X_author)

# Define weighted imputation function for 'author'
def weighted_impute_author(row, knn, vectorizer, scaler):
    if pd.notnull(row['author']) and row['author'].strip() != '':
        return row['author']
    else:
        combined = row['combined_features_author']
        combined_vec = vectorizer.transform([combined])
        token_counts = scaler.transform([[row['title_token_count'], row['text_token_count']]])
        X_current = hstack([combined_vec, token_counts])

        # Find KNN neighbors
        distances, indices = knn.kneighbors(X_current, n_neighbors=5)
        neighbor_authors = df.iloc[indices[0]]['author'].dropna()
        neighbor_distances = distances[0]
        if neighbor_authors.empty:
            return "Unknown Author"
        else:
            # Compute weights based on distances (closer neighbors have higher weights)
            weights = 1 / (neighbor_distances + 1e-5)  # Add small value to avoid division by zero
            weights = weights / weights.sum()  # Normalize weights

            # Aggregate weights for each author
            author_weights = {}
            for author, weight in zip(neighbor_authors, weights):
                author_weights[author] = author_weights.get(author, 0) + weight

            # Select the author with the highest total weight
            imputed_author = max(author_weights, key=author_weights.get)
            return imputed_author

# Apply weighted imputation for 'author'
df['author_imputed'] = df.apply(
    lambda row: weighted_impute_author(row, knn_author, tfidf_author_vectorizer,
scaler_author),
    axis=1
)

# Verify imputation
print(f"Missing 'author' before imputation: {missing_author.sum()}")
missing_imputed_author = df['author_imputed'].isnull() | (df['author_imputed'].str.strip() == '')
print(f"Missing 'author' after imputation: {missing_imputed_author.sum()}")

# Optional: Replace original 'author' with imputed values
df['author'] = df['author_imputed']
df.drop('author_imputed', axis=1, inplace=True)
```

Text Processing:

```
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
# Initialize the lemmatizer and set of stopwords
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
# Define preprocessing function
def preprocess_text(text):
    text = re.sub(r'http\S+', '', text) # Remove URLs
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text)  # Remove non-alphanumeric characters
    tokens = text.split()
    processed_tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in
stop_words]  # Remove stopwords and lemmatize
    return ' '.join(processed_tokens)
# Apply preprocessing to 'title' and 'text' columns
df['title'] = df['title'].apply(preprocess_text)
df['text'] = df['text'].apply(preprocess_text)
df['author'] = df['author'].apply(preprocess_text)
```

Frequency Based Model:

```
# Split the data into training and test sets
X = df[['title_text', 'title_token_count', 'text_token_count', 'author_token_count']]
y = df['label']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)

# Define the pipeline
pipeline_count = Pipeline([
    ('countvec', CountVectorizer(
        ngram_range=(1, 2),     # Unigrams and Bigrams
        min_df=4,               # Minimum document frequency
        max_features=50000,     # Maximum number of features
        stop_words='english'    # Remove English stopwords
    )),
    ('feature_selection', SelectKBest(chi2, k=20000)),
    ('clf', MultinomialNB())
])

# Define the parameter grid
param_grid = {
    'clf__alpha': [0.1, 1.0],                   # MultinomialNB smoothing parameter
    'countvec__ngram_range': [(1, 1), (1, 2)],     # Unigrams or Unigrams + Bigrams
    'feature_selection__k': [20000, 30000]      # Number of top features to select
}

# Initialize the MultinomialNB classifier
nb_classifier = MultinomialNB()
```

```
# Set up GridSearchCV
grid_search = GridSearchCV(
    pipeline_count,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='accuracy',
    n_jobs=-1  # Use all available cores
)

# Perform Grid Search
grid_search.fit(X_train, y_train)

# Retrieve best parameters and score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print(f'Best Parameters: {best_params}')
print(f'Best Cross-Validation Accuracy: {best_score}')

# Evaluate the best model on the test set
best_model_count = grid_search.best_estimator_
y_pred = best_model_count.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f'Test Set Accuracy: {accuracy}')

print('Classification Report:')
print(classification_report(y_test, y_pred))
```

Tf-Idf Based Model:

```
# Define feature columns and target variable
X = df[['title_text', 'title_token_count', 'text_token_count', 'author_token_count']]
y = df['label']  # Replace 'label' with your actual target variable if different
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,         # 20% for testing
    random_state=42,       # Ensures reproducibility
    stratify=y             # Maintains the distribution of classes
)
print(f"Training Set Shape: {X_train.shape}")
print(f"Testing Set Shape: {X_test.shape}")

# Define the preprocessor using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('tfidf', TfidfVectorizer(
            ngram_range=(1, 2),
            min_df=4,
            max_features=50000,
            stop_words='english',
            norm='l2'
        ), 'title_text'),
        ('scaler', MinMaxScaler(), ['title_token_count', 'text_token_count',
'author_token_count'])
    ],
    remainder='passthrough'  # Keep other columns if any
)
# Define the complete pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('feature_selection', SelectKBest(chi2, k=20000)),
    ('clf', MultinomialNB())
])
# Define the parameter grid for GridSearchCV
param_grid = {
    'preprocessor__tfidf__max_features': [20000, 30000],
    'preprocessor__tfidf__ngram_range': [(1, 1), (1, 2)],
    'clf__alpha': [0.01, 1.0]
}
# Initialize GridSearchCV with the pipeline
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,                # 5-fold cross-validation
    scoring='accuracy',
    n_jobs=-1,           # Use all available cores
    verbose=1
)
# Perform Grid Search on the training data
grid_search.fit(X_train, y_train)
# Retrieve best parameters and score
best_params = grid_search.best_params_
best_score = grid_search.best_score_
print(f'Best Parameters: {best_params}')
print(f'Best Cross-Validation Accuracy: {best_score:.4f}')
# Evaluate the best model on the test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'\nTest Set Accuracy: {accuracy:.4f}')
print('\nClassification Report:')
print(classification_report(y_test, y_pred))
# Plot Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Real', 'Fake'],
          yticklabels=['Real', 'Fake'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```