CSBB 311: Machine Learning

Fake News Detection: Analyzing the Impact of Text Preprocessing on Decision Trees and Naive Bayes Classifiers

Submitted By:

Abhiraj Banerjee (221210005)

Aditya Shaurya Singh Negi (221210012)

Akshat Singh (221210015)

Submitted To: Dr. Preeti Mehta

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

2024

Abstract

The proliferation of misinformation has become a critical challenge in the digital age, undermining public trust and informed decision-making. This project addresses the urgent need for an automated system to detect fake news articles. Utilizing a machine learning approach, we developed a classifier that leverages Natural Language Processing (NLP) techniques in **Python** programming language to analyze textual features and assess the veracity of news content. The methodology involved collecting a diverse dataset of news articles labeled as true or false, followed by preprocessing steps such as **tokenization** and **TF-IDF** vectorization.

We implemented classification algorithms namely the **Naive Bayes** and **Decision Trees** algorithms, evaluating their performance using accuracy, precision, recall, and F1-score metrics. Both the Naive Bayes and Decision Tree algorithms emerged very effective, achieving an accuracy of up to 97%. Our results indicate that NLP techniques combined with ensemble learning can significantly enhance the detection of fake news. This project highlights the potential of machine learning in combating misinformation and suggests avenues for further research, including real-time detection and integration into news consumption platforms. The findings underscore the importance of developing robust tools to empower users in discerning credible information sources in an increasingly complex media landscape.

Introduction

In today's digital age, the rapid dissemination of information through social media and online platforms has transformed the way we consume news. However, this shift has also led to the proliferation of fake news—misleading or fabricated information presented as factual reporting. The impact of fake news is profound, contributing to the spread of misinformation, polarization of public opinion, and erosion of trust in legitimate news sources. Consequently, detecting and combating fake news has become an essential endeavor for journalists, policymakers, and technology developers alike.

The primary objective of this project is to develop a machine learning model capable of accurately identifying fake news articles. By leveraging natural language processing (NLP) techniques, the model aims to analyze textual content for linguistic cues and patterns indicative of misinformation.

To achieve these objectives, the project employs a comprehensive methodology that includes data collection, preprocessing, feature extraction, model training, and evaluation. We gather a diverse dataset consisting of both genuine and fake news articles, followed by the implementation of various machine learning algorithms. Through rigorous training, testing and validation, we aim to enhance the model's accuracy and reliability, ensuring its practical applicability in real-world scenarios.

Based on Bayes' Theorem, the *Naive Bayes* classifier is a probabilistic method. It makes the assumption that given the class label (false or true news), features (in this case, words in the news articles) are conditionally independent. This premise of independence may appear straightforward, but Naive Bayes has proven to be useful for text categorisation, particularly in cases with high-dimensional data like textual content. The algorithm calculates the likelihood that a given item of news belongs to a specific class based on the presence of certain phrases or traits. It is a well-liked option for early-stage models due to its effectiveness and simplicity, particularly in NLP applications.

A *Decision Tree* is a structure that resembles a flowchart, with nodes standing in for decisions made based on features and leaves for the result (false or actual news). It creates a recursive tree that depicts the decision-making process by dividing the data into subgroups according to feature values. Because they make it possible to comprehend the decision rules that the model forms, decision trees are very interpretable. Decision Trees are non-parametric and can capture complicated interactions between features, unlike Naive Bayes, which relies on a strong independence assumption. This could result in improved accuracy when the relationship between features is more nuanced.

Methodology

- ## Dataset Description:

The dataset we used contains labeled news articles classified as real or fake. The dataset includes text data for training machine learning models aimed at distinguishing between genuine and false news. The data in the dataset is in the following format:

title (string)	text (string)	subject (string)	date (string)	label (bool)

Note: *The 'subject' label is not very useful as there are very few distinct subjects in the dataset, so we won't be using that column.*

You can explore the dataset here.

- ## Text Preprocessing Techniques used:

1. **Text Cleaning:**

This process includes:

a) **Stopword removal**: Removing common words like "the," "is," "and," that do not contribute to the meaning. These words are unnecessary for model training and increase computational time.
b) **Punctuation marks removal**: Stripping out punctuation marks.
c) **Character filtering**: Removing irrelevant characters (e.g., special symbols, numbers).
d) **Link filtering:** Removing the unnecessary links in the dataset which contribute nothing to the training.

These tasks are performed by some great python libraries which are discussed later.

2. **Tokenization:**

This process refers to the splitting of text into words or *'tokens'* which then are used to represent numerical values in the machine learning models.

Tokenization reduces the size of raw text so that it can be handled more easily for processing and analysis.

3. **Term-Frequency/ Inverse Document Frequency (TF-IDF):**

In information retrieval, TFIDF ,short for term frequency–inverse document frequency, is a measure of importance of a word to a document in a collection or corpus, adjusted for the fact that some words appear more frequently in general.

a) Term-Frequency:

Term frequency, ***tf(t,d)***, is the relative frequency of term ***t*** within document ***d***, is defined as:

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

where $f_{t,d}$ is the raw count of a term in a document, i.e., the number of times that term t occurs in document d. *Note:* the denominator is simply the total number of terms in document d (counting each occurrence of the same term separately).

b) Inverse-Document Frequency:

The inverse document frequency is a measure of how much information the word provides, i.e., how common or rare it is across all documents.

It is the logarithmically scaled inverse fraction of the documents that contain the word (obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient).

It's defined as:

$$\log\left(\frac{N}{1+n_t}\right)+1$$

Where N is ***total no. of documents*** and n_t is ***no. of documents in which the term "t"*** *appears.*

Note: this is the smoothed version of IDF to handle zeroes.

c) TFIDF:

The tf–idf is calculated as:

$$\mathrm{tfidf}(t,d,D)=\mathrm{tf}(t,d)\cdot\mathrm{idf}(t,D)$$

A high weight in TFIDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms.

- Model Descriptions:

1. Decision Trees

a) **Overview**: Decision tree is a supervised learning algorithm used for both classification and regression tasks. They work by recursively splitting the dataset based on feature values to form a tree structure.

b) **How It Works**: At each node, the algorithm selects the feature that best separates the data according to a criterion like Gini impurity or information gain (for classification) and mean squared error (for regression).

c) **Pros**:
- Easy to understand and visualize.
- Handles both numerical and categorical data.
- Requires minimal data preprocessing (no need for scaling).

d) **Cons**:
- Prone to overfitting, especially on noisy data.
- Can create biased trees if some classes dominate.

e) **Applications**:
- **Classification**: Used in spam detection, medical diagnoses, and sentiment analysis to classify data into categories.
- **Regression**: Predicting continuous values like house prices or stock trends.
- **Customer Segmentation**: Marketing analysis to segment customers based on behavior or demographics.
- **Fraud Detection**: Identifying fraudulent activities in transactions.

2. Naive Bayes:

a) **Overview**: Naive Bayes is a probabilistic classifier based on Bayes' theorem, assuming strong (naive) independence between features.

b) **How It Works**: It calculates the posterior probability for each class based on the input features, using the formula:

$$P(Class \mid Features) = P(Features) P(Features \mid Class) \cdot P(Class)$$

The class with the highest probability is chosen as the prediction.

c) **Pros**:
- Works well with small datasets.
- Fast and efficient, especially for text classification tasks.
- Performs well even with independent feature assumptions.

d) **Cons**:
- Assumes feature independence, which may not hold in real-world data.
- Not ideal for continuous variables without proper handling (e.g., Gaussian Naive Bayes).

e) **Applications**:
- **Spam Filtering**: Classifying emails as spam or not spam.
- **Text Classification**: Sentiment analysis, fake news detection, and categorizing documents.
- **Medical Diagnosis**: Predicting diseases based on patient symptoms.
- **Recommendation Systems**: Suggesting products or content to users based on their preferences.
- **Sentiment Analysis**: Analyzing customer reviews to determine positive or negative sentiment.

- Code Workflow:

Tech Stack:

a) Programming Language: *Python*
b) Standard Libraries: *re (regex), collections, math, string*
c) Additional Libraries: *nltk, sklearn, matplotlib, seaborn*

❖ Decision Trees

1) Data Loading

- **Purpose**:

Loading the data into a format that can be easily manipulated, using Pandas DataFrames. Separating the dataset into training and test sets. Training data will be used to teach the model, and test data will evaluate the model's performance.

- **Details**:

Load the datasets from here.

The dataset consists of various columns and thousands of rows which are full of articles. The main columns are "text" which includes the news, and "label" which contains binary values, 1 for real news and 0 for fake news.

2) Preprocessing (Text Cleaning with `wordopt` function)

- **Purpose**:

The article data in its raw form is noisy and contains irrelevant information, like punctuation, links, special characters, etc. Preprocessing removes this noise, leading to a more accurate model in the end.

- **Details**:

The `wordopt` function converts text to lowercase, removes extra spaces, special characters, URLs, and numbers. For this we will be importing the regular expression library: “import re”. This will help in screening the text and giving us the processed text. Removing punctuation, decapitalizing, removing links, and all sorts of noises from the article, to enhance the result we will obtain in the end.

3) Feature Extraction (TF-IDF)

- **Purpose**:

Convert the raw text into a numerical format that machine learning models can work with. TF-IDF (Term Frequency-Inverse Document Frequency) is one way to represent text data as features.

- **Details**:

The process involved in using `TfidfVectorizer()` includes aspects of tokenization. Tokenization is the process of splitting text into smaller pieces, called tokens. These tokens can be words, phrases, or symbols, depending on the specific application.

TF-IDF computes the importance of each word in the document by considering how often it appears in a document relative to how often it appears in other documents. The `TfidfVectorizer()` transforms each text document into a vector (a row of numbers) where each number represents the importance of a word. Here is the formula which is used to calculate it.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Machine learning models can’t work directly on raw text, so this transformation allows text data to be used in models like Decision Trees or Naive Bayes. TF-IDF is especially useful for filtering out common words and focusing on more informative words.

4) Model Training

- **Purpose**:

Train the Decision Tree on the TF-IDF vectors derived from the text. The model "learns" to associate patterns in the text with the labels (0 for fake news, 1 for real news). The TF-IDF vectors are the input features to the model. These vectors represent the importance of words in the articles while eliminating common words that don't carry significant meaning. By feeding these vectors into the Decision Tree, the model "learns" to associate specific patterns in the text like the presence or absence of certain words or word combinations with the corresponding labels.

- **Details**:

You can use `DecisionTreeClassifier()` to fit the model on the training data (`xv_train` and `y_train`). The model will learn the rules that determine whether a piece of news is fake or real based on the patterns in the training data.

5) Model Testing

- **Purpose**:

Evaluate how well the trained model performs on **unseen data** (the test dataset). This helps you understand whether the model can generalize beyond the training data.

- **Details**:

You use the `.predict()` method on `xv_test` (the test data's vectorized text) to see how well the model performs. The `classification_report()` provides performance metrics like:

  - **Accuracy**: Percentage of correctly predicted labels.
  - **Precision**: Out of all instances predicted as fake news, how many were actually fake news.
  - **Recall**: Out of all actual fake news articles, how many were correctly identified by the model.
  - **F1-score**: A balance between precision and recall.

Additionally, the `confusion_matrix()` method provides a confusion matrix from which the above metrics can easily be calculated. More details on performance metrics are provided here.

6) Manual Testing

- **Purpose**:

Allow for manual input to test the model on specific news articles, outside the regular training/test set. This can provide real-world validation of the model's effectiveness.

- **Details**:

The `manual_testing()` function takes a news article as input, processes it using the wordopt function, and then transforms it using the trained vectorizer. The model makes a prediction (whether it's fake or real), and the result is printed.

❖ Naive Bayes:

The Naive Bayes Classifier for this project is completely implemented from scratch and only uses libraries for evaluation purposes. The data loading with the aid of pandas dataframe is the same as discussed previously.

The code consists of 5 functions and a class described below:

- `def clean_text(text: str) -> str:`

This function is completely analogous to the **`wordopt`** function that was used in the Decision Tree model. It removes unnecessary characters from the text and converts it to lower case.

- `def tokenize(data: pd.DataFrame, label: bool) -> Counter:`

This function takes the pandas dataframe and a label (0/1) as parameters and tokenizes the text according to the label i.e. counts the frequency of each word appearing in the news that are true/false and returns a *counter*(dictionary like) object. The main difference between a *Counter* and a *dict* object is that the latter raises a `KeyError` when a key is not present, whereas the former does not.

- `def IDF(df: pd.DataFrame) -> Counter:`

This function takes a dataframe (dataset) and returns a Counter containing all the words appearing in the corpus with its IDF (inverse document frequency) calculated as:

$$\log\left(\frac{N}{1+n_t}\right)+1$$

(See IDF here)

- `def predict_raw(parameters):`

This function takes a string (news to be classified) and other necessary parameters and returns the predicted class according to probability of the news to be fake/real. The formula that the code uses is the standard bayes theorem:

$$P(class|text) \propto P(class) * P(w1 \mid class) * P(w2 \mid class) * \cdots * P(wn \mid class)$$

Where w1, w2 ...wn are the words appearing in the text (news to be classified).

It seems that we can apply this formula directly *but*, as our dataset is very large (30,000 rows and each news column has 3000 characters approx) so probabilities can get very small and hence, their product can get *extremely small.* The smallest number that a float data type can represent in Python language is ***2.2250738585072014e-308***.

So, it's inconvenient to use the formula as it is.

To solve this problem we calculate the *log(probabilities)* so we remain free from underflow issues. The above formula modifies to:

$$log(P(class|text)) \propto log(P(class)) + \sum log(P(Wi \mid class))$$

In order to avoid zero probabilities we use Laplace smoothing,

For a given word w and class c:

$$P(w|c) = \frac{\text{count}(w \text{ in class } c) + 1}{\text{total words in class } c + V}$$

- `def predict_tf_idf(parameters):`

This does nearly the same as the above function; it additionally multiplies the TF-IDF weight of the Wi to $P(Wi \mid class)$. *See TFIDF here.*

- The `NB` class:

This class contains 3 functions:

a) `def __init__(self, dataset_path):`

This function is a typical constructor used to load and preprocess the data using above functions and precompute the necessary values and *Counters* which are used again and again to compute probabilities.

b) `def NaiveBayes(self):`

This function applies the `predict_raw` function to the dataframe and creates a new column for the predicted values in the same dataframe. Additionally, using this new column it generates the classification report and the confusion matrix.

c) `def NaiveBayesTFIDF(self):`

This function applies the `predict_tf_idf` function to the dataframe and creates a new column for the predicted values in the same dataframe. Additionally, using this new column it generates the classification report and the confusion matrix.

Evaluation Metrics

Evaluating the performance of a Machine learning model is one of the important steps while building an effective ML model. *To evaluate the performance or quality of the model, different metrics are used, and these metrics are known as performance metrics or evaluation metrics.*

- Confusion Matrix:

A confusion matrix is a tabular representation of prediction outcomes of any binary classifier, which is used to describe the performance of the classification model on a set of test data when true values are known.

	Predicted Fake News (0)	**Predicted Real News (1)**
Actual Fake News (0)	**True Negatives (TN)**	**False Positives (FP)**
Actual Real News (1)	**False Negatives (FN)**	**True Positives (TP)**

A Confusion Matrix

With the help of the ***confusion matrix*** these different types of other evaluation metrics can be calculated:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} * 100 \quad (3)$$

$$\text{Precision or } = \frac{\text{TP}}{\text{TP} + \text{FP}} * 100 \quad (4)$$

$$\text{Recall or TPR } = \frac{\text{TP}}{\text{TP} + \text{FN}} * 100 \quad (5)$$

$$\text{Specificity or TNR } = \frac{\text{TN}}{\text{TN} + \text{FP}} * 100 \quad (6)$$

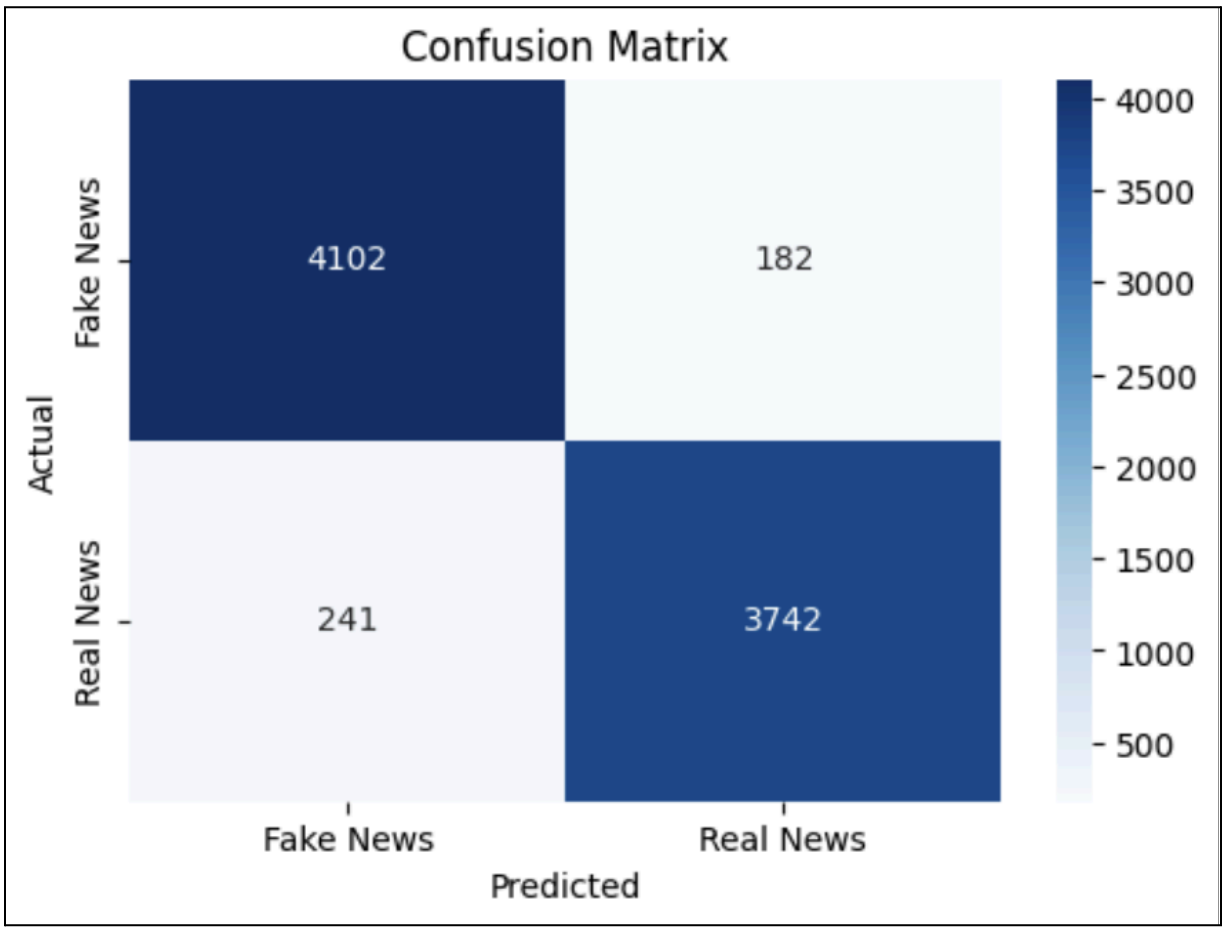
$$\text{F1 Score } = 2 * \frac{\text{Recall } \times \text{Precision}}{\text{Recall} + \text{Precision}} * 100 \quad (7)$$

$$\text{False Positive Ratio(FPR)} = \frac{\text{FP}}{\text{FP+FN}} * 100 \quad (8)$$

These performance metrics help us understand how well our model has performed for the given data. In this way, we can improve the model's performance by tuning the hyper-parameters.

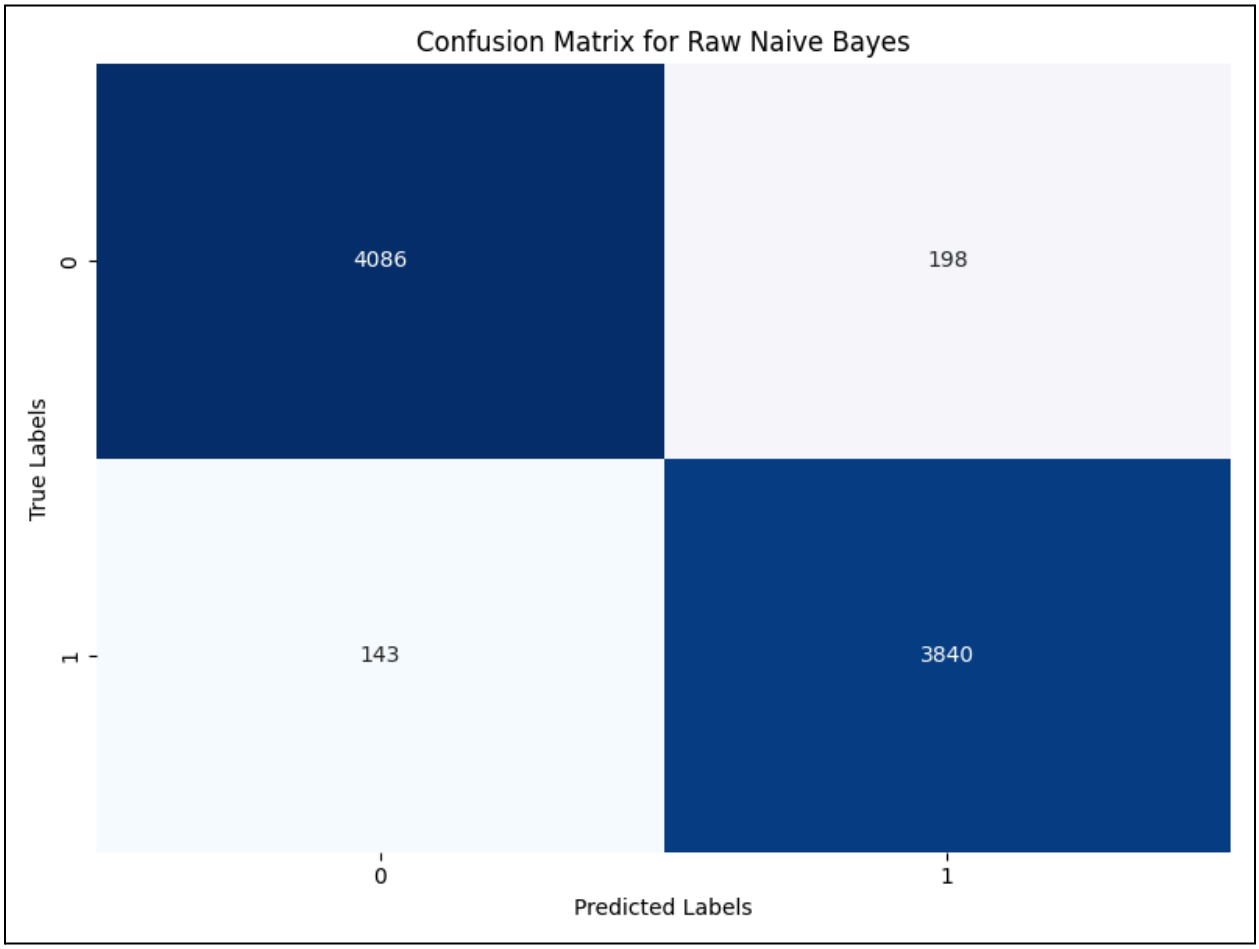
Results

- **Decision Tree**



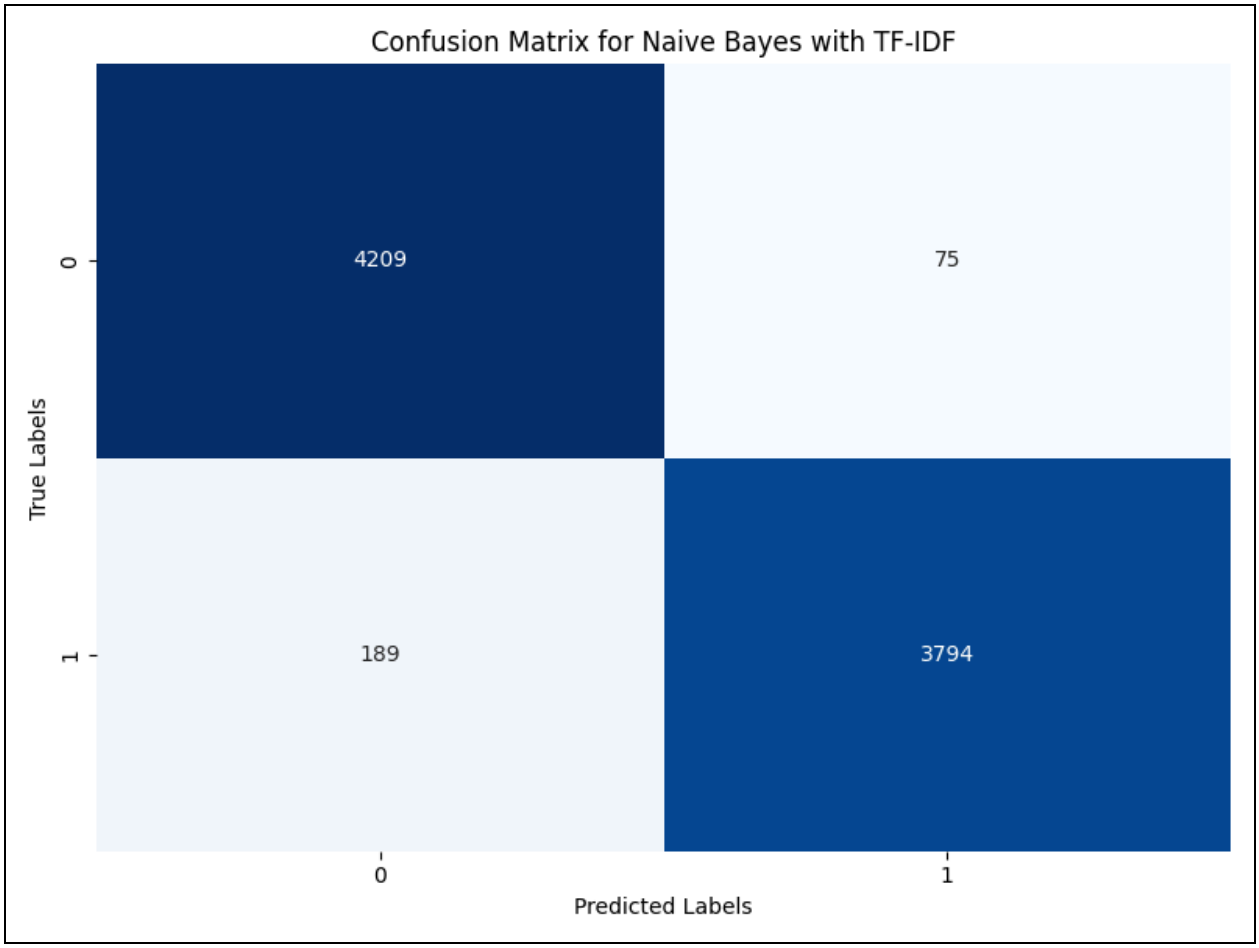

```
Accuracy: 0.9488327083585 34
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      4284
           1       0.95      0.94      0.95      3983

    accuracy                           0.95      8267
   macro avg       0.95      0.95      0.95      8267
weighted avg       0.95      0.95      0.95      8267
```

- **Naive Bayes**



Classification Report for Raw Naive Bayes:

	precision	recall	f1-score	support
0	0.97	0.95	0.96	4284
1	0.95	0.96	0.96	3983
accuracy			0.96	8267
macro avg	0.96	0.96	0.96	8267
weighted avg	0.96	0.96	0.96	8267



Classification Report of Naive Bayes with TF-IDF:

	precision	recall	f1-score	support
0	0.96	0.98	0.97	4284
1	0.98	0.95	0.97	3983
accuracy			0.97	8267
macro avg	0.97	0.97	0.97	8267
weighted avg	0.97	0.97	0.97	8267

CONCLUSION

*The Decision Tree (TF-IDF) model predicted the outcomes with **95 %** accuracy.*

*The Naive Bayes model predicted the outcomes with **96 %** accuracy.*

*The Naive Bayes (TF-IDF) model predicted the outcomes with **97 %** accuracy.*

The application of machine learning models for detecting fake news using natural language processing has produced encouraging results. In this project, we investigated two machine learning algorithms—Decision Trees and Naive Bayes—while utilizing the TF-IDF feature extraction technique to pinpoint patterns in the text that separate fake news from legitimate reports.

The Decision Tree (TF-IDF) model delivered reliable performance, achieving an accuracy rate of 95%. Decision Trees are particularly effective in identifying complex relationships between features and offer clear interpretability, helping us understand the model's decision-making process. However, these models can be prone to overfitting and are sensitive to small data changes, which can affect their consistency.

The Naive Bayes algorithm, known for its simplicity and effectiveness in text classification, performed slightly better with an accuracy of 96%. This method, grounded in Bayes' Theorem and the assumption of feature independence, worked well on this dataset. Naive Bayes is particularly suited to handling high-dimensional text data, making it efficient in managing large vocabularies and sparse matrices.

The Naive Bayes (TF-IDF) model delivered even stronger results, achieving an impressive accuracy of 97%, indicating the effectiveness of this approach. This suggests that while Naive Bayes is an inherently strong algorithm for text classification, combining it with TF-IDF vectorization further improves its ability to discern between fake and real news by focusing on the most important terms in the dataset.

In conclusion, while both models showed strong performance, Naive Bayes (TF-IDF) stood out as the best performer in this project.

Learning

Challenges:

During the development of this project, we encountered several challenges:

- **Preprocessing text:** There were many meaningless words in the dataset containing characters that seldom appear in normal texts.

- **Feature Importance & Feature Selection**:The model gave undue weight to certain words, leading to poor generalization. Identifying which features (e.g., text characteristics, sentiment analysis scores) were most effective for distinguishing between real and fake news proved difficult.

- **Sparse Data**: High-dimensional feature vectors from text (e.g., using TF-IDF) can lead to sparse data, making it harder for the model to capture patterns.

- **Handling Rare Words**: Rare or unseen words during training can cause incorrect probability calculations.

Throughout the project, we realized the importance of iterative testing. Initially, our focus was heavily on model complexity, but we shifted towards simpler models combined with effective feature selection. This change led to improvements in performance and reduced training time.
Additionally, ongoing discussions and feedback loops among team members fostered a culture of continuous improvement, where each iteration brought new insights that refined our approach.

Limitations

Despite the progress made, the project had its limitations:

- Limited Dataset Diversity: The dataset primarily consisted of articles from specific sources, which may not represent the broader landscape of news, potentially skewing results.
- Evolving Nature of Fake News: The characteristics of fake news can change rapidly, meaning that models may require frequent retraining to stay effective against new tactics used in misinformation.
- Subjectivity in Labeling: The process of labeling articles as real or fake is inherently subjective and can lead to inconsistencies, impacting the reliability of the training data.
- Computational Constraints: Resource limitations restricted the complexity of the models we could explore, preventing us from fully leveraging deep learning techniques that might have improved performance.
- Interpretability Issues: The more complex models, while accurate, lacked transparency, making it difficult to understand the basis of their predictions. This raised concerns about trust and accountability in real-world applications.

By acknowledging these limitations, the team is better equipped to address them in future iterations or related projects, ultimately enhancing the effectiveness and reliability of the fake news detection algorithm.

Appendix (Codes)

The code's repository can be accessed from here.

CODE:

```
import pandas as pd
from nltk.corpus import stopwords
import re
from collections import Counter
from math import log
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report

stopW = set(stopwords.words("english"))  # set of stopwords

def clean_text(text: str) -> str:
    # Remove special quotes
    text = re.sub(r"[“”‘’]", "", text)
    # Remove commas and periods, while ensuring spaces are handled
    text = re.sub(r"[,.]", " ", text)  # Replace commas and periods with spaces
    # Remove possessive forms
    text = re.sub(r"'s\b", "", text)
    # Remove non-word characters (except for apostrophes)
    text = re.sub(r"[^\w\s']", "", text)
    text = re.sub(r"_+", " ", text)
    # Convert to lowercase
    text = text.lower()
    # Split text into words and remove stopwords
    words = text.split()
    words = [word for word in words if word not in stopW]
    # Join the remaining words back into a string
    cleaned_text = " ".join(words)
    return cleaned_text
```

```
# word counts
def tokenize(data: pd.DataFrame, label: bool) -> Counter:
    tokens = Counter()
    for text in data[data["label"] == label]["text"]:
        tokens.update(str(text).split())
    return tokens

# apply bayes theorem for predicting
def predict_raw(
    text: str,
    real: Counter,
    fake: Counter,
    prob_real: float,
    tot_words_real: int,
    tot_words_fake: int,
    n_unique: int,
) -> bool:
    # prob(real|words)=prob(words|real)*prob(real)/prob(words)
    prob_r = log(prob_real)
    prob_f = log(1 - prob_real)
    list_of_words = text.split()
    for word in list_of_words:
        # laplace smoothing
        prob_r += log((1 + real[word]) / (tot_words_real + n_unique))
        prob_f += log((1 + fake[word]) / (tot_words_fake + n_unique))

    return prob_r > prob_f

def IDF(df: pd.DataFrame) -> Counter:
    idf = Counter()
    num_docs = len(df)  # Total number of documents

    for txt in df["text"]:
        # Split the text into words and use a set to count each word only once
per document
        words_in_doc = set(txt.split())
        # This will count each word only once per document
        for word in words_in_doc:
            idf[word] += 1
```

```
    for word in idf:
        idf[word] = log(num_docs / (1 + idf[word])) + 1  # idf smooth

    return idf

def predict_tf_idf(
    text: str,
    real: Counter,
    fake: Counter,
    tot_words_real: int,
    tot_words_fake: int,
    n_unique: int,
    idf_real: Counter,
    idf_fake: Counter,
    prob_real: float,
) -> bool:
    prob_r = log(prob_real)
    prob_f = log(1 - prob_real)
    epsilon = 1e-9
    tf = Counter(text.split())
    total = tf.total()

    for term in tf.keys():
        prob_r += log(
            (tf[term] / total)
            * (idf_real[term] + epsilon)
            * ((real[term] + 1) / (tot_words_real + n_unique))
        )
        prob_f += log(
            (tf[term] / total)
            * (idf_fake[term] + epsilon)
            * ((fake[term] + 1) / (tot_words_fake + n_unique))
        )

    return prob_r > prob_f

class NB:
    def __init__(self, dataset_path):
```

```
        # Load and preprocess training and testing data
        self.train_data = pd.read_csv(dataset_path[0], delimiter="\t")
        self.train_data = self.train_data.drop(columns=["title", "subject",
"date"])
        self.train_data["text"] = self.train_data["text"].apply(clean_text)

        self.test_data = pd.read_csv(dataset_path[1], delimiter="\t")
        self.test_data = self.test_data.drop(columns=["title", "subject",
"date"])
        self.test_data["text"] = self.test_data["text"].apply(clean_text)

        # these constants must be global
        self.train_true = self.train_data[self.train_data["label"] == 1]
        self.train_false = self.train_data[self.train_data["label"] == 0]

        self.prob_real = len(self.train_true) / len(self.train_data)

        self.real = tokenize(self.train_data, 1)
        self.fake = tokenize(self.train_data, 0)
        self.n_unique = len(self.real.keys() | self.fake.keys())
        self.tot_words_real = sum(self.real.values())
        self.tot_words_fake = sum(self.fake.values())

        # precomputing idf hash tables
        self.idf_real = IDF(self.train_true)
        self.idf_fake = IDF(self.train_false)

    def NaiveBayes(self):
        # Make predictions
        self.test_data["predicted_label_raw"] = self.test_data["text"].apply(
            lambda x: predict_raw(
                x,
                self.real,
                self.fake,
                self.prob_real,
                self.tot_words_real,
                self.tot_words_fake,
                self.n_unique,
            )
```

```
        )

        real_labels = self.test_data["label"]
        predicted_labels = self.test_data["predicted_label_raw"]

        print(classification_report(real_labels, predicted_labels))

        # Generate confusion matrix
        conf_matrix = confusion_matrix(real_labels, predicted_labels)

        # Create confusion matrix heatmap using seaborn
        plt.figure(figsize=(8, 6))
        sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)

        # Add plot labels and title
        plt.title("Confusion Matrix for Raw Naive Bayes")
        plt.xlabel("Predicted Labels")
        plt.ylabel("True Labels")

        # Show the plot
        plt.tight_layout()
        plt.show()

    def NaiveBayesTFIDF(self):
        self.test_data["predicted_label_tfidf"] = self.test_data["text"].apply(
            lambda x: predict_tf_idf(
                x,
                self.real,
                self.fake,
                self.tot_words_real,
                self.tot_words_fake,
                self.n_unique,
                self.idf_real,
                self.idf_fake,
                self.prob_real,
            )
        )

        real_labels = self.test_data["label"]
```

```
        predicted_labels = self.test_data["predicted_label_tfidf"]

        print(classification_report(real_labels, predicted_labels))

        # Generate confusion matrix
        conf_matrix = confusion_matrix(real_labels, predicted_labels)

        # Create confusion matrix heatmap using seaborn
        plt.figure(figsize=(8, 6))
        sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)

        # Add plot labels and title
        plt.title("Confusion Matrix for Naive Bayes with TF-IDF")
        plt.xlabel("Predicted Labels")
        plt.ylabel("True Labels")

        # Show the plot
        plt.tight_layout()
        plt.show()


if __name__ == "__main__":
    model = NB(["dataset/train.tsv", "dataset/test.tsv"])
    model.NaiveBayes()
    model.NaiveBayesTFIDF()
```

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import string
import re
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report


splits = {'train': 'train.tsv', 'validation': 'validation.tsv', 'test':
'test.tsv'}
```

```
# Load the training dataset
df_train =
pd.read_csv("hf://datasets/ErfanMoosaviMonazzah/fake-news-detection-dataset-Engli
sh/" + splits["train"], sep="\t")

# Load the test dataset
df_test =
pd.read_csv("hf://datasets/ErfanMoosaviMonazzah/fake-news-detection-dataset-Engli
sh/" + splits["test"], sep="\t")

#label'd as fake new 0 ; real news 1

# **Dropping unwanted columns**


df_train.columns

df_train = df_train.drop(['Unnamed: 0', 'title', 'subject', 'date'], axis=1)

df_test = df_test.drop(['Unnamed: 0', 'title', 'subject', 'date'], axis=1)

df_train.columns

df_test.columns

#count of missing values
df_train.isnull().sum()

"""# Data is already suffled so we do not require to do so"""

df_train.head()

"""# **Preprocessing Text**
# Creating a function to convert the text in lowercase, remove the extra space,
special chr., ulr and links.

"""
```

```
def wordopt(text):
    text = text.lower()
    text = re.sub('\[.*?\]','',text)
    text = re.sub("\\W"," ",text)
    text = re.sub('https?://\S+|www\.\S+','',text)
    text = re.sub('<.*?>+',b'',text)
    text = re.sub('[%s]' % re.escape(string.punctuation),'',text)
    text = re.sub('\w*\d\w*','',text)
    return text

df_train['text'] = df_train['text'].apply(wordopt)

df_test['text'] = df_test['text'].apply(wordopt)

df_train.head()

# Tokenization

x_train = df_train['text']  # Features for training
y_train = df_train['label']  # Labels for training
x_test = df_test['text']     # Features for testing
y_test = df_test['label']     # Labels for testing

from sklearn.feature_extraction.text import TfidfVectorizer

# Creating an instance of TfidfVectorizer
vectorization = TfidfVectorizer()

# Fitting the vectorizer on the training data and transforming it into TF-IDF
vectors
xv_train = vectorization.fit_transform(x_train)

# Transforming the test data into TF-IDF vectors
xv_test = vectorization.transform(x_test)

# Decision trees

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
```

```
# Initialize the Decision Tree Classifier
DT = DecisionTreeClassifier()

# Fit the model on the training data
DT.fit(xv_train, y_train)

# Make predictions on the test set
pred_dt = DT.predict(xv_test)

# Accuracy score
accuracy = DT.score(xv_test, y_test)
print(f'Accuracy: {accuracy}')

# Classification report for the predictions
print(classification_report(y_test, pred_dt))

# Manual Testing
# I entered news from the Validation.tsv file.

def output_label(n):
    if n == 0:
        return "Fake News"
    elif n == 1:
        return "Real News"

def manual_testing(news):
    # Creating a DataFrame for the input news
    testing_news = {"text": [news]}
    new_def_test = pd.DataFrame(testing_news)

    # Apply Preprocessing function
    new_def_test['text'] = new_def_test["text"].apply(wordopt)

    # Prepare the text for prediction
    new_x_test = new_def_test["text"]
    new_xv_test = vectorization.transform(new_x_test)

    # Predictions
```

```
    pred_DT = DT.predict(new_xv_test)

    # Print the results
    print("\nDT Prediction: {}".format(output_label(pred_DT[0])))

# this is a fake news
news_article = "Mexico has been the beneficiary of our open borders for decades.
It s really quite amazing how loudly they cry foul when we finally have a
presidential candidate willing to stand up to this insanity and say, ENOUGH! It s
Time to put Americans first! Illegal immigrants send home $50 billion annually
but cost taxpayers more than $113 billion. Approximately 126,000 illegal
immigrants emigrated from these three nations to the U.S. since last October and
federal officials estimate at least 95,500 more will enter next year.The Central
American governments have encouraged the high levels of emigration because it is
earning their economy billions of dollars! For every illegal alien that sneaks
into the U.S. and remits money back home, that grand total remittance number only
grows. But what if the millions of U.S. jobs now filled by illegal aliens were
done by American workers earning better wages, paying more in taxes and spending
their money in their communities rather than sending it abroad?Americans are the
ones forced to pick up the $113 billion tab for taking care of the country s 12
million illegal immigrants.r"
manual_testing(news_article)

# this is a real news
news_article2 = "German Foreign Minister Sigmar Gabriel said it was necessary to
do everything possible to make progress on the nuclear deal with Iran and that he
did not see any indications during a visit to the United States that Washington
would terminate it. U.S. President Donald Trump said earlier on Thursday that
“nothing is off the table” in dealing with Iran following its test launch of a
ballistic missile."
manual_testing(news_article2)


# Confusion matrix
from sklearn.metrics import confusion_matrix
# Create confusion matrix
cm = confusion_matrix(y_test, pred_dt)

# Plotting the confusion matrix
```

```
plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Fake News',
'Real News'], yticklabels=['Fake News', 'Real News'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```