The Fake News Detection model by applying NLP techniques and training a classification model.

Text Preprocessing and Feature Extraction

Model training and evaluation

# AI-PHASE 4 PROJECT

SELVAM COLLEGE OF TECHNOLOGY

**Key steps, including Text Preprocessing, Feature Extraction, Model training, and Evaluation.**

# Imports:

* The code begins by importing necessary Python libraries, including Pandas (**pandas**), regular expressions (**re**), NumPy (**numpy**), Natural Language Toolkit (**nltk**), TensorFlow (**tensorflow**), and specific modules and functions from these libraries.

# NLTK Resource Downloads:

* The code downloads NLTK resources for tokenization (**punkt**), stopwords (**stopwords**), and WordNet lemmatizer (**wordnet**) using **nltk.download**.

# Loading and Combining Datasets:

* The code loads two datasets: "True.csv" and "Fake.csv" using Pandas. Each dataset contains news articles labeled as either "true" or "fake."
* Labels are added to the datasets to identify them as "true" or "fake."
* The datasets are combined into one (**combined\_df**) using **pd.concat**.

# Data Cleaning:

* The **combined\_df['text']** column is cleaned in several ways:
* **<[^>]+>** is removed from the text using regular expressions, which is a common way to remove HTML tags from text data.
* Special characters and punctuation are removed using **re.sub**.
* All text is converted to lowercase.
* Numbers are removed using regular expressions.

# Tokenization:

* The text in **combined\_df['text']** is tokenized into words using NLTK's **word\_tokenize** function. This step converts the text into a list of words.

# Stop-word Removal:

* Stop words (common words like "the," "and," "in") are removed from the tokenized text using NLTK's list of English stop words. This step reduces the number of low-information words.

# Splitting the Dataset:

* The dataset is split into training and testing sets using **train\_test\_split** from Scikit-Learn. This is a common practice in machine learning to evaluate the model's performance.

# Deep Learning Model (LSTM) Preparation:

* The text data in the training and testing sets is joined into strings and tokenized using Keras' **Tokenizer** and padded to ensure uniform length using **pad\_sequences**.

# Encoding Labels:

* Labels (i.e., "true" and "fake") are encoded into numerical values using Scikit-Learn's**LabelEncoder**. "true" might be encoded as 0, and "fake" as 1.

# Deep Learning Model (LSTM) Creation:

* A sequential deep learning model is built using TensorFlow and Keras. This model consists of an embedding layer, an LSTM layer, and a dense layer with sigmoid activation. This architecture is common for text classification.

# Model Compilation and Training:

* The model is compiled with an optimizer, loss function, and metrics. Then, it is trained on the training data using the **fit** method.

# Deep Learning Model Evaluation:

* The trained deep learning model is evaluated on the testing data, and various metrics, including accuracy, precision, recall, and F1-score, are calculated and printed.

# Text Preprocessing for Naive Bayes:

* This section defines a preprocessing function for text data that tokenizes, lemmatizes, and removes stop words from the text. This will be used for the Naive Bayes model.

# TF-IDF Vectorization and Naive Bayes Training:

* The text data is preprocessed and then converted into TF-IDF features using **TfidfVectorizer** from Scikit-Learn. It is followed by training a Naive Bayes classifier.

# Naive Bayes Model Evaluation:

* The Naive Bayes model is evaluated on the testing data, and similar metrics to the deep learning model are calculated and printed.

*The code showcases two different approaches to text classification: deep learning (LSTM) and traditional machine learning (Naive Bayes) for classifying news articles as "true" or "fake". It includes data preprocessing, model building, training, and evaluation for both approaches.*

let's break down the code into different parts and explaining each part separately:

# Part 1: Imports and NLTK Resource Downloads

## Python Code

|  |
| --- |
| import pandas aspd import re import numpyas np import nltk import tensorflowastf from nltk.tokenizeimportword\_tokenize from nltk.corpusimportstopwords from sklearn.model\_selectionimporttrain\_test\_split from sklearn.preprocessingimportLabelEncoder from tensorflow.keras.preprocessing.textimportTokenizer from tensorflow.keras.preprocessing.sequenceimportpad\_sequences from tensorflow.keras.modelsimport Sequential from tensorflow.keras.layersimport Embedding, LSTM, Dense from sklearn.metricsimportaccuracy\_score, precision\_score, recall\_score, f1\_score from sklearn.feature\_extraction.textimportTfidfVectorizer from sklearn.naive\_bayesimportMultinomialNB from sklearn.metricsimport accuracy\_score,confusion\_matrix, classification\_report from nltk.stemimportPorterStemmer, WordNetLemmatizer nltk.download('punkt') nltk.download('stopwords') nltk.download('wordnet') |

* This part of the code begins by importing necessary libraries and resources, including Pandas for data handling, regular expressions, NumPy for numerical operations, NLTK for natural language processing, TensorFlow for deep learning, and various specific functions and classes from these libraries.
* NLTK resources are downloaded using **nltk.download()** to access tokenization, stopwords, and WordNet lemmatizer.

# Part 2: Loading and Combining Datasets

## Python code

|  |
| --- |
| true\_df = pd.read\_csv("D:\\naan mudhalvan\\True.csv", encoding='utf-8') true\_df['title'] = 'true'  fake\_df = pd.read\_csv("D:\\naan mudhalvan\\Fake.csv", encoding='utf-8') fake\_df['title'] = 'fake'  combined\_df = pd.concat([true\_df, fake\_df], ignore\_index=True) |

* In this part, two datasets are loaded into Pandas DataFrames: "True.csv" and "Fake.csv." The **'title'** column is added to each DataFrame to label articles as "true" or "fake."
* The **pd.concat** function combines these DataFrames into a single DataFrame named **combined\_df**.

# Part 3: Data Cleaning

## Python Code

|  |
| --- |
| combined\_df['text'] = combined\_df['text'].apply(lambda x: re.sub('<[^>]+>', '', x)) combined\_df['text'] = combined\_df['text'].replace('[^\w\s]', '') combined\_df['text'] = combined\_df['text'].apply(lambda x: x.lower()) combined\_df['text'] = combined\_df['text'].apply(lambda x: re.sub('\d+', '', x)) |

* In this section, data cleaning is performed on the text content of the DataFrame:
* HTML tags are removed using regular expressions.
* Special characters and punctuation are removed using **re.sub**.
* All text is converted to lowercase.
* Numbers are removed using regular expressions.

# Part 4: Tokenization

## Python Code

|  |
| --- |
| combined\_df['text'] = combined\_df['text'].apply(lambda x: word\_tokenize(x)) |

* The text in the **combined\_df**DataFrame is tokenized into words using NLTK's **word\_tokenize** function.

# Part 5: Stop-word Removal

## Python Code

|  |
| --- |
| stop\_words = set(stopwords.words('english')) combined\_df['text'] = combined\_df['text'].apply(lambda x: [word for word in x if word notinstop\_words]) |

* A set of English stopwords is defined, and then stop words are removed from the tokenized text in the DataFrame using a list comprehension.

This is the first part of the code explained in detail. If you'd like to continue with the explanation of the subsequent parts, please let me know.

# Part 6: Splitting the Dataset

## Python Code

|  |
| --- |
| combined\_df['text'].apply(lambda x: ' '.join(x)) y = combined\_df['title']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) |

* X = In this section, the dataset is split into training and testing sets using **train\_test\_split** from scikit-learn.
* The **X** variable contains the combined and tokenized text data, where individual words are joined into a single string.
* The **y** variable contains the labels, which are either 'true' or 'fake'.
* The dataset is split into training (**X\_train**, **y\_train**) and testing (**X\_test**, **y\_test**) sets. The testing set constitutes 20% of the data, and a random seed of 42 is set for reproducibility.

# Part 7: Tokenization and Padding

## Python Code

|  |
| --- |
| max\_words = 10000 tokenizer = Tokenizer(num\_words=max\_words) tokenizer.fit\_on\_texts(X\_train) X\_train\_seq = tokenizer.texts\_to\_sequences(X\_train) X\_test\_seq = tokenizer.texts\_to\_sequences(X\_test)  max\_sequence\_length = 250 X\_train\_pad = pad\_sequences(X\_train\_seq, maxlen=max\_sequence\_length) X\_test\_pad = pad\_sequences(X\_test\_seq, maxlen=max\_sequence\_length) |

* Tokenization and padding are performed to prepare the text data for deep learning.
* **max\_words** defines the maximum number of unique words in the tokenized dictionary.
* A **Tokenizer** is created with a specified maximum number of words, and it is fitted on the training data.
* The text data is converted to sequences of numbers using **texts\_to\_sequences**.
* **max\_sequence\_length** specifies the maximum length of sequences, and padding is added to ensure all sequences have the same length.
* **pad\_sequences** is used to pad or truncate sequences as needed.

# Part 8: Encoding Labels

## Python Code

|  |
| --- |
| label\_encoder = LabelEncoder() y\_train\_encoded = label\_encoder.fit\_transform(y\_train) y\_test\_encoded = label\_encoder.transform(y\_test) |

* Labels ('true' and 'fake') are encoded into numerical values. Label encoding assigns 0 to 'true' and 1 to 'fake.'

# Part 9: Building a Simple LSTM Model

## Python Code

|  |
| --- |
| model = Sequential() model.add(Embedding(input\_dim=max\_words, output\_dim=128, input\_length=max\_sequence\_length)) model.add(LSTM(128)) model.add(Dense(1, activation='sigmoid')) |

* A simple LSTM (Long Short-Term Memory) model is built using the Keras Sequential API.
* The model consists of an embedding layer for word embeddings, an LSTM layer with 128 units, and a dense output layer with a sigmoid activation function.

# Part 10: Compiling and Training the Model

## Python Code

|  |
| --- |
| model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy']) model.fit(X\_train\_pad, y\_train\_encoded, epochs=5, batch\_size=32, validation\_split=0.2) |

* The model is compiled with the Adam optimizer, binary cross-entropy loss, and accuracy as a metric.
* The model is trained on the training data with 5 epochs, a batch size of 32, and a validation split of 20%.

# Part 11: Evaluating the Deep Learning Model

## Python Code

|  |
| --- |
| y\_pred\_dl = model.predict(X\_test\_pad) y\_pred\_dl\_binary = (y\_pred\_dl>0.5).flatten()  accuracy\_dl = accuracy\_score(y\_test\_encoded, y\_pred\_dl\_binary) precision\_dl = precision\_score(y\_test\_encoded, y\_pred\_dl\_binary) recall\_dl = recall\_score(y\_test\_encoded, y\_pred\_dl\_binary) f1\_dl = f1\_score(y\_test\_encoded, y\_pred\_dl\_binary) |

* The deep learning model is evaluated by making predictions on the test set and calculating accuracy, precision, recall, and F1-score.

# Part 12: Preprocessing for Naive Bayes Classifier

Python Code

|  |
| --- |
| lemmatizer = WordNetLemmatizer() stemmer = PorterStemmer()  defpreprocess\_text(text):  words = text.split()  words = [lemmatizer.lemmatize(stemmer.stem(word)) for word in words if word notinstop\_words] return' '.join(words)  X\_train\_nb = X\_train.apply(preprocess\_text) X\_test\_nb = X\_test.apply(preprocess\_text) |

e

* Text data preprocessing is defined for a Naive Bayes classifier. It involves lemmatization, stemming, and stop-word removal.
* Preprocessing functions are applied to the training and testing data.

# Part 13: Vectorization with TF-IDF

## Python Code

|  |
| --- |
| tfidf\_vectorizer = TfidfVectorizer(max\_features=5000) X\_train\_tfidf = tfidf\_vectorizer.fit\_transform(X\_train\_nb) X\_test\_tfidf = tfidf\_vectorizer.transform(X\_test\_nb) |

* TF-IDF (Term Frequency-Inverse Document Frequency) vectorization is applied to the preprocessed text data.
* A **TfidfVectorizer** is created with a maximum of 5000 features (words).
* **fit\_transform** is used on the training data to learn the vocabulary and transform it into a TF-IDF matrix.
* **transform** is used on the test data to transform it into a TF-IDF matrix using the vocabulary learned from the training data.

# Part 14: Training a Naive Bayes Classifier

## Python Code

|  |
| --- |
| naive\_bayes\_classifier = MultinomialNB() naive\_bayes\_classifier.fit(X\_train\_tfidf, y\_train) |

* A Naive Bayes classifier, specifically the **MultinomialNB** variant, is initialized.
* It is trained on the TF-IDF vectorized training data and corresponding labels.

# Part 15: Making Predictions with Naive Bayes

## Python Code

|  |
| --- |
| y\_pred\_nb = naive\_bayes\_classifier.predict(X\_test\_tfidf) |

* The trained Naive Bayes classifier is used to make predictions on the TF-IDF vectorized test data.

# Part 16: Evaluating the Naive Bayes Model

## Python Code

|  |
| --- |
| accuracy\_nb = accuracy\_score(y\_test, y\_pred\_nb) confusion\_nb = confusion\_matrix(y\_test, y\_pred\_nb) classification\_rep\_nb = classification\_report(y\_test, y\_pred\_nb) |

* The performance of the Naive Bayes classifier is evaluated with accuracy, confusion matrix, and a classification report.

# Part 17: Printing Results

The final part of the code prints the results of both the deep learning model and the Naive Bayes classifier.

I Encountered some errors in our code. Let's address those issues.

# Lemmatizer Error:

* Csharp Code

|  |
| --- |
| * NameError: name 'lemmatizer'isnot defined |

* This error occurs because the **lemmatizer** object is defined inside the function **preprocess\_text**, and it's not accessible outside that function. To fix this, you should define **lemmatizer** outside the function so it's accessible globally. Here's how to fix it:
* Python Code

|  |
| --- |
| * lemmatizer = WordNetLemmatizer()  # Preprocess the text data for the Naive Bayes classifier defpreprocess\_text(text):  words = text.split()  words = [lemmatizer.lemmatize(stemmer.stem(word)) for word in words if word notinstop\_words] return' '.join(words) |

* Make sure to add the **lemmatizer = WordNetLemmatizer()** line before defining the **preprocess\_text** function.

# Undefined Variable Error:

* Scss Code

|  |
| --- |
| * print(F1-score) |

* This error is caused by trying to print a variable that is not defined. The correct variable name should be **f1**, not **F1-score**. Update the print statement as follows:
* Python Code

|  |
| --- |
| * print(f'F1-score: {f1:.2f}') |

# Backslash in File Paths:

* You are using backslashes in your file paths (e.g., **"D:\\naan mudhalvan\\True.csv"**). Make sure you escape backslashes in your file paths or use raw string literals (prefixed with **r**) to avoid issues with backslashes. **For example:**
* Python Code

|  |
| --- |
| * true\_df = pd.read\_csv(r"D:\naan mudhalvan\True.csv", encoding='utf-8') fake\_df = pd.read\_csv(r"D:\naan mudhalvan\Fake.csv", encoding='utf-8') |