import nltk

**nltk** stands for Natural Language Toolkit, and it's a popular Python library for working with human language data (text).

from nltk.corpus import stopwords

The line **from nltk.corpus import stopwords** imports the "stopwords" corpus from the NLTK library. Stopwords are common words in a language (e.g., "and," "the," "is") that are often filtered out during text preprocessing in natural language processing tasks. These words usually do not carry much meaningful information for the analysis or modeling and are thus removed to focus on more important words.

from nltk.stem import PorterStemmer ,WordNetLemmatizer

**Porter Stemmer**: The Porter stemming algorithm is a widely used technique for stemming words in text analysis. Stemming is the process of reducing words to their base or root form (stem) by removing prefixes and suffixes.

A screenshot of a computer

Description automatically generated

**WordNet Lemmatizer**: Lemmatization is the process of reducing words to their base or dictionary form (lemma). The WordNet lemmatizer in NLTK uses WordNet's lexical database to find the correct lemma of a word.

A screenshot of a computer

Description automatically generated

from sklearn.ensemble import RandomForestClassifier

**RandomForestClassifier** is an ensemble learning method based on the random forest algorithm. It constructs multiple decision trees during training and outputs the class that is the mode of the classes (classification) or the mean prediction (regression) of the individual trees.  A prediction from the Random Forest Regressor is an average of the predictions produced by the trees in the forest.

from sklearn.feature\_extraction.text import TfidfVectorizer

**TfidfVectorizer** is a class used to convert a collection of raw documents (text) to a matrix of TF-IDF features. TF-IDF stands for Term Frequency-Inverse Document Frequency, and it is a numerical statistic that reflects the importance of a word in a document relative to a collection of documents (corpus).

from sklearn.metrics import accuracy\_score,confusion\_matrix, classification\_report

The line **from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report** imports three important functions from scikit-learn (sklearn) that are commonly used to evaluate the performance of machine learning models, especially in classification tasks:

1. **accuracy\_score**: The **accuracy\_score** function computes the accuracy of a classification model, which is the ratio of correctly predicted instances to the total number of instances. It's a commonly used metric to evaluate classification models.
2. **confusion\_matrix**: The **confusion\_matrix** function computes the confusion matrix, which is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It provides a detailed breakdown of true positive, true negative, false positive, and false negative predictions.
3. **classification\_report**: The **classification\_report** function generates a text summary of the key classification metrics such as precision, recall, F1-score, and support for each class in a classification problem. It's a helpful way to understand the performance of a classification model in a more detailed and informative manner.

To read dataset

df=pd.read\_csv("WELFake\_Dataset.csv")

df.shape

The **df.shape** is a common operation used in pandas to retrieve the dimensions of a DataFrame, which includes the number of rows and columns. It returns a tuple in the format **(number\_of\_rows, number\_of\_columns)**.

df.isna().sum()

we get missing values of each of the colums

id 0

title 558

author 1957

text 39

label 0

dtype: int64

df=df.dropna()

to drop missing values

now if we do df.insa().sum()

id 0

title 0

author 0

text 0

label 0

dtype: int64

df.reset\_index(inplace=True)

resets the index

df=df.drop(['id','text','author'],axis=1)

drops id text and author columns

A screenshot of a computer

Description automatically generated

Our new dataset is created and saved into dataset DF

Data pre processing

okenization is the process of breaking down a sequence of text into smaller units, typically words, subwords, or characters, known as tokens. Tokens are the fundamental units used for various natural language processing (NLP) tasks, such as machine translation, sentiment analysis, and named entity recognition.

In English, tokens are often words, but they can also be subwords or characters, especially in languages with complex morphology or in scenarios like word segmentation for languages without clear word boundaries.

Here are three common types of tokenization:

1. **Word Tokenization:** Breaking text into words is the most common form of tokenization. For instance, the sentence "Tokenization is important." would be tokenized into three tokens: ["Tokenization", "is", "important"].
2. **Subword Tokenization:** Subword tokenization breaks words into smaller linguistic units such as prefixes, suffixes, or stems. This is particularly useful for languages with complex morphology or for handling out-of-vocabulary words. For example, "unhappiness" might be tokenized into ["un", "happiness"].
3. **Character Tokenization:** Character tokenization treats each character in the text as a separate token. This approach can be useful for some tasks or languages where characters convey important information, such as in transliteration or languages with non-alphabetic scripts.

sample\_data="the quick brown fox jumps over the lazy dog"

sample\_data=sample\_data.split()

sample\_data

output

['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

**Converting data into lowercase to avoid repetition like the and The**

sample\_data=[data.lower() for data in sample\_data]

sample\_data

['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

**Removing the stopwards**

Removing stopwords is a common preprocessing step in natural language processing (NLP) to filter out commonly used words that typically do not contain important meaning and are often removed to reduce noise and computational overhead. Stopwords are words such as "and," "or," "the," "is," "in," etc., which occur frequently in a language and usually do not carry much semantic value

sample\_data=[data for data in sample\_data if data not in stopwords]

print(sample\_data)

len(sample\_data)

['quick', 'brown', 'fox', 'jumps', 'lazy', 'dog']

[35]:

6

Stemming is a technique used in natural language processing (NLP) and information retrieval to reduce words to their base or root forms (stems) by removing affixes. The goal of stemming is to group words with similar meanings but different grammatical forms under a common stem.

For example, applying stemming to words like "running," "runs," and "runner" would result in the common stem "run."

There are different algorithms and approaches for stemming, but the most common ones include:

1. **Porter's Algorithm:** The Porter stemming algorithm is a widely used algorithm that applies a set of heuristic rules to strip off common suffixes from English words, resulting in their stems.
2. **Snowball Algorithm (Porter2):** An improvement over the original Porter algorithm, the Snowball algorithm (also known as Porter2) is more aggressive in stemming and provides more accurate results.
3. **Lancaster Stemmer:** The Lancaster stemming algorithm is another popular stemming algorithm that is more aggressive than Porter stemming, but it may produce stems that are not actual words.

ps= PorterStemmer()

sample\_data\_stemming= [ps.stem(data) for data in sample\_data]

print(sample\_data\_stemming)

['quick', 'brown', 'fox', 'jump', 'lazi', 'dog']

We get lazi not lazy due to stemming only removing prefixes and suffixes

Lemmatization is a linguistic and natural language processing (NLP) technique that involves reducing words to their base or canonical form, known as the lemma. Unlike stemming, which involves crude removal of prefixes and suffixes to obtain the root form (stem), lemmatization takes into account the context and part of speech of the word.

The lemma is the base form of a word, often a dictionary form, and represents the canonical morphological representation of that word. For example:

* Lemmatization of "running" would yield "run" (the base form of the verb).
* Lemmatization of "better" would result in "good" (the base form of the adjective).

Lemmatization is more sophisticated than stemming because it considers the meaning of the word in the given context. It uses detailed morphological analysis of the words, considering parts of speech (e.g., nouns, verbs, adjectives, adverbs) and other language-specific rules to find the lemma.

Benefits of lemmatization include:

* **Accuracy:** Lemmatization yields actual words or lemmas, unlike stemming, which may produce non-words.
* **Context Preservation:** Lemmatization considers the context and part of speech, preserving the meaning of the word in a sentence.

lm= WordNetLemmatizer()

sample\_data\_lemma=[lm.lemmatize(data) for data in sample\_data]

print(sample\_data\_lemma)

now lemmetizing whole dataframe

lm= WordNetLemmatizer()

Great! You've created an instance of the WordNetLemmatizer class from NLTK. The WordNetLemmatizer is a tool for lemmatization, which is the process of reducing words to their base or root form (i.e., their lemma).

corpus = []

Creating a variable named **corpus** typically implies that you are creating a list or collection to store a corpus of text data. A corpus is a large and structured set of texts that can be used for various language processing and analysis tasks.

for i in range (len(df)):

review=re.sub('^a-zA-Z0-9',' ',df['title'][i])

**review** is a variable that is being used to store the result of applying a regular expression substitution on the 'title' of a DataFrame

**RE** :Regular expressions (regex or regexp) are sequences of characters that define a search pattern. They are used for pattern matching within strings.

**sub (Substitution)**:

* **sub** is a method provided by the **re** module, short for "substitute".
* **re.sub(pattern, repl, string)** is used to replace occurrences of a pattern in a string with a specified replacement.

**'[^a-zA-Z0-9\s]'** is a regular expression pattern that matches any character that is not a letter, digit, or whitespace.

**df['title'][i]** is the string from the 'title' column at index **i** in the DataFrame.

So, **review** will hold the modified 'title' string after substituting non-alphanumeric characters with spaces. However, in the provided code, **review** is being overwritten in each iteration of the loop, so it will only store the result for the last iteration. If you want to keep track of multiple modified titles, you may need to store them in a list or a similar data structure.

review=review.lower()

o, **review = review.lower()** takes the content of the **review** string and converts all its characters to lowercase. This is often done to ensure consistent case (either lowercase or uppercase) for further processing or analysis of the text, as it helps in case-insensitive comparisons and standardizing the text data.

Top of Form

review= review.split()

By calling **review.split()** without specifying a delimiter, the string **review** will be split into words, where each word becomes an element in a list.

For example, if **review** is "This is a sample review.", the code will split it into a list of words: **['This', 'is', 'a', 'sample', 'review.']**.

review= [lm.lemmatize(x) for x in review if x not in (stopwords.words('english'))]

the list comprehension iterates over each word in **review**, lemmatizes the word, and only includes it in the new list if it's not a stopword. The resulting list, assigned to **review**, will contain lemmatized words excluding stopwords. Examples of stopwords in the English language include words like "and," "the," "is," "in," "of," "a," "an," "to," etc.

review= " ".join(review)

The line **review = " ".join(review)** is joining the words in the **review** list back into a single string, where words are separated by spaces. This is a common step in text processing to convert a list of words back into a sentence or a coherent piece of text.

corpus.append(review)

**corpus.append(review)** is a way to collect and store preprocessed text data in a list for future use.

Vectorization

It involves converting textual data into numerical representations that can be used by machine learning models. T

tf=TfidfVectorizer()

x=tf.fit\_transform(corpus)

because corpus has lemmatized data and whatever we have to perform we would perform on corpus

x\_array=x.toarray()’

we convert corpus to array

y=df['label']

y.head()

0 1

1 0

2 1

3 1

4 1

Name: label, dtype: int64

**splittng data into train andtest split**

train\_test\_split(x,y,test\_size = 0.3,random\_state=10,stratify=y)

test size is 30% and train set size is 70%

"random state" or "random seed" is often used to control the randomness that is involved in certain processes, such as initializing weights in a neural network or splitting a dataset into training and testing sets.

In this example, **random\_state=42** sets the random seed to 42, ensuring that the same random split occurs every time you run this code.

The **stratify** parameter in the **train\_test\_split** function in scikit-learn is used to ensure that the target variable's class distribution is maintained in both the training and testing sets.

So one type of data doesn’t get more flooded into trai test more than test set and viecversa

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size = 0.3,random\_state=10,stratify=y)

now if we write

len(x\_train) and len(y\_train)

it will give us error that array length is sparse

so we use .shape

x\_train.shape, y\_train.shape

((12799, 19259), (12799,))

x\_test.shape,y\_test.shape

((5486, 19259), (5486,))

Model building

rf= RandomForestClassifier()

rf.fit(x\_train,y\_train)

we import random forest classifier and fit into out train set

model evaluation

y\_pred=rf.predict(x\_test)

accuracy\_score=accuracy\_score(y\_test,y\_pred)

accuracy\_score ->0.9369303682099891

accuracy score is 93%

class Evaluation:

Python class called **Evaluation** which is designed to evaluate a machine learning model using various metrics such as accuracy, confusion matrix, and classification report. The evaluation is performed on both training and testing datasets.

def \_\_init\_\_(self, model, x\_train, x\_test, y\_train, y\_test):

self.model = model

self.x\_train = x\_train

self.x\_test = x\_test

self.y\_train = y\_train

self.y\_test = y\_test

**\_\_init\_\_** method: Initializes the **Evaluation** object and sets the model, training and testing data. **self** is a reference to the current instance of a class in Python. It is used within a class to access instance variables and methods.

def train\_evaluation(self):

y\_pred\_train = self.model.predict(self.x\_train)

acc\_scr\_train = accuracy\_score(self.y\_train, y\_pred\_train)

print("Accuracy score on Training dataset:", acc\_scr\_train)

con\_mat\_train = confusion\_matrix(self.y\_train, y\_pred\_train)

print("Confusion matrix on Training dataset:\n", con\_mat\_train)

class\_rep\_train = classification\_report(self.y\_train, y\_pred\_train)

print("Classification report on Training dataset:\n", class\_rep\_train)

**train\_evaluation** method: Evaluates the model on the training dataset and prints accuracy, confusion matrix, and classification report for the training set.

def test\_evaluation(self):

y\_pred\_test = self.model.predict(self.x\_test)

acc\_scr\_test = accuracy\_score(self.y\_test, y\_pred\_test) # Use a different variable name

print("Accuracy score on Testing dataset:", acc\_scr\_test)

con\_mat\_test = confusion\_matrix(self.y\_test, y\_pred\_test)

print("Confusion matrix on Testing dataset:\n", con\_mat\_test)

class\_rep\_test = classification\_report(self.y\_test, y\_pred\_test)

print("Classification report on Testing dataset:\n", class\_rep\_test)

**test\_evaluation** method: Evaluates the model on the testing dataset and prints accuracy, confusion matrix, and classification report for the testing set.

Now we call this class component on train data

Evaluation(rf,x\_train,x\_test,y\_train,y\_test).train\_evaluation()

Rf contains random forest regressor as we know from before

Accuracy score on Training dataset: 1.0

Confusion matrix on Training dataset:

[[7252 0]

[ 0 5547]]

Classification report on Training dataset:

precision recall f1-score support

0 1.00 1.00 1.00 7252

1 1.00 1.00 1.00 5547

accuracy 1.00 12799

macro avg 1.00 1.00 1.00 12799

weighted avg 1.00 1.00 1.00 12799

**accuracy is 100%**

**on test data**

**Evaluation(rf,x\_train,x\_test,y\_train,y\_test).test\_evaluation()**

Accuracy score on Testing dataset: 0.9391177542836311

Confusion matrix on Testing dataset:

[[2834 275]

[ 59 2318]]

Classification report on Testing dataset:

precision recall f1-score support

0 0.98 0.91 0.94 3109

1 0.89 0.98 0.93 2377

accuracy 0.94 5486

macro avg 0.94 0.94 0.94 5486

weighted avg 0.94 0.94 0.94 5486

**accuracy 93%**

**creating a prediction pipeline**

**we lemmatize the data first like we did bnefore**

class preprocessing:

def \_\_init\_\_(self,data):

self.data=data

def test\_preprocessing\_user(self):

lm= WordNetLemmatizer()

pred\_data= [self.data]

preprocess\_data = []

for data in pred\_data:

review=re.sub('^a-zA-Z0-9',' ',data)

review=review.lower()

review= review.split()

review= [lm.lemmatize(x) for x in review if x not in (stopwords.words('english'))]

review= " ".join(review)

preprocess\_data.append(review)

return preprocess\_data

now main thing

class Prediction:

def \_\_init\_\_(self,pred\_data, model):

self.pred\_data = pred\_data

self.model = model

def prediction\_model(self):

preprocess\_data = preprocessing(self.pred\_data).test\_preprocessing\_user()

data = tf.transform(preprocess\_data)

prediction = self.model.predict(data)

if prediction [0] == 0 :

return "The News Is Fake"

else:

return "The News Is Real"