FAKE NEWS DETECTION

TEAM MEMBERS: NANDINI ARIMANDA-22011101076

VENUPRIYA A-22011101123

**AIM:**

The aim of a Fake News Detection System is to automatically identify fake news articles from genuine sources using natural language

processing and machine learning techniques, thereby mitigating the spread of false information and promoting media literacy.

**DATASET DESCRIPTION:**

The dataset for a Fake News Detection System typically consists of a collection of news articles labeled as either genuine or fake. It is

taken from Kaggle and are given by Marwan ElMahalawy. The

training dataset consists of article ID, news, author and the label if it is true or false. Whereas the test data consists of article ID,news and the author. There are two datasets one used for training and the

other for testing.

**DIAGRAM-**

**PROJECT OUTLINE:**

## ACCURACY TABLE:

Input data from the training

dataset

Data Exploration:

Data statistics.

Data preprocessing:

Text cleaning,lemmatisation, tokenization,stopword removal.

Model evaluation:

Cross validation,performance matrix,confusion matrix.

Model selection and model

training:Logistic regression,SVM, descion tree,Naïve bayes

Feature Extraction:

(TF-IDF,word vectors).

Prediction:

Classify news and predict

Web Page Creation:

Using FLASK .

Takes user input and

predicts based on that

Predicts based on data in

dataset

Output:

Print the predicted labels: fake/real

|  |  |  |  |
| --- | --- | --- | --- |
| **Classifier** | **SVM** | **Maximum entropy** | **Naïve bayes** |
| **Accuracy** | **0.943** | **0.942** | **0.816** |

**CODE:**

1. PYTHON.PY

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv) import nltk

from sklearn.model\_selection import GridSearchCV from sklearn.linear\_model import LogisticRegression

from sklearn.feature\_extraction.text import CountVectorizer from sklearn.naive\_bayes import MultinomialNB

from sklearn.feature\_extraction.text import TfidfTransformer from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import ShuffleSplit import matplotlib.pyplot as plt

from nltk.corpus import stopwords import os

import warnings import seaborn as sns import re

import string

from termcolor import colored from nltk import word\_tokenize import string

from nltk import pos\_tag

from nltk.corpus import stopwords

from nltk.tokenize import WhitespaceTokenizer from nltk.stem import WordNetLemmatizer import nltk

nltk.download('averaged\_perceptron\_tagger')

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import cross\_val\_score

warnings.filterwarnings('ignore') from matplotlib.pyplot import \*

from sklearn.pipeline import Pipeline

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.naive\_bayes import MultinomialNB

from sklearn.svm import LinearSVC

from sklearn.tree import DecisionTreeRegressor

from sklearn.linear\_model import LogisticRegression from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn import preprocessing

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

from nltk.corpus import wordnet

from sklearn.feature\_extraction.text import TfidfTransformer seed = 12345

cv = ShuffleSplit(n\_splits=5, test\_size=0.2, random\_state=seed) encoder = preprocessing.LabelEncoder()

def get\_wordnet\_pos(pos\_tag): if pos\_tag.startswith('J'):

return wordnet.ADJ

elif pos\_tag.startswith('V'): return wordnet.VERB

elif pos\_tag.startswith('N'): return wordnet.NOUN

elif pos\_tag.startswith('R'):

return wordnet.ADV else:

return wordnet.NOUN

def preprocess(text):

# lowercase the text text = text.lower()

# remove the words counting just one letter text = [t for t in text.split(" ") if len(t) > 1]

# remove the words that contain numbers

text = [word for word in text if not any(c.isdigit() for c in word)] # tokenize the text and remove puncutation

text = [word.strip(string.punctuation) for word in text] # remove all stop words

stop = stopwords.words('english') text = [x for x in text if x not in stop] # remove tokens that are empty text = [t for t in text if len(t) > 0]

# pos tag the text

pos\_tags = pos\_tag(text) # lemmatize the text

text = [WordNetLemmatizer().lemmatize(t[0], get\_wordnet\_pos(t[1])) for t in pos\_tags]

# join all

text = " ".join(text) return (text)

def split\_train\_holdout\_test(encoder, df, verbose=True):

# Resplit original train and test train = df[df["label"] != "None"] test = df[df["label"] == "None"]

# Encode Target

train["encoded\_label"] = encoder.fit\_transform(train.label.values)

# Take holdout from train

train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = train\_test\_split(train, train.encoded\_label, test\_size=0.33, random\_state=seed)

if(verbose):

print("\nTrain dataset (Full)") print(train.shape)

print("Train dataset cols") print(list(train.columns))

print("\nTrain CV dataset (subset)") print(train\_cv.shape)

print("Train Holdout dataset (subset)") print(train\_holdout.shape)

print("\nTest dataset") print(test.shape)

print("Test dataset cols") print(list(test.columns))

return encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label

def runModel(encoder, train\_vector, train\_label, holdout\_vector, holdout\_label, type, name): global cv

global seed

## Classifier types if (type == "svc"):

classifier = SVC() grid = [

{'C': [1, 10, 50, 100], 'kernel': ['linear']},

{'C': [10, 100, 500, 1000], 'gamma': [0.0001], 'kernel': ['rbf']},

]

if (type == "nb"):

classifier = MultinomialNB() grid = {}

if (type == "maxEnt"):

classifier = LogisticRegression()

grid = {'penalty': ['l1','l2'], 'C': [0.001,0.01,0.1,1,10,100,1000]}

# Model

print(colored(name, 'red'))

model = GridSearchCV(estimator=classifier, cv=cv, param\_grid=grid) print(colored(model.fit(train\_vector, train\_label), "yellow"))

# Score

print(colored("\nCV-scores", 'blue'))

means = model.cv\_results\_['mean\_test\_score'] stds = model.cv\_results\_['std\_test\_score']

for mean, std, params in sorted(zip(means, stds, model.cv\_results\_['params']), key=lambda x: -x[0]): print("Accuracy: %0.3f (+/-%0.03f) for params: %r" % (mean, std \* 2, params))

print()

print(colored("\nBest Estimator Params", 'blue')) print(colored(model.best\_estimator\_, "yellow"))

# Predictions

print(colored("\nPredictions:", 'blue'))

model\_train\_pred = encoder.inverse\_transform( model.predict(holdout\_vector) ) print(model\_train\_pred)

# Confusion Matrix

cm = confusion\_matrix(holdout\_label, model\_train\_pred)

# Transform to df for easier plotting cm\_df = pd.DataFrame(cm,

index = ['FAKE','REAL'],

columns = ['FAKE','REAL'])

plt.figure(figsize=(5.5,4))

sns.heatmap(cm\_df, annot=True, fmt='g') plt.ylabel('True label')

plt.xlabel('Predicted label') plt.show()

# Accuracy

acc = accuracy\_score(holdout\_label, model\_train\_pred) print(colored("\nAccuracy:", 'blue'))

print(colored(acc, 'green')) return [name, model, acc]

def pos\_tag\_words(text):

pos\_text = nltk.pos\_tag(nltk.word\_tokenize(text))

return " ".join([pos + "-" + word for word, pos in pos\_text]) input\_str = open("fake\_or\_real\_news\_training.csv", encoding= 'utf-8')

# Remove all new lines

noNewLines = re.sub("\n", "", input\_str.read()) # re-add new line at end of each row

noNewLines = re.sub("X1,X2", "X1,X2\n", noNewLines) noNewLines = re.sub(",FAKE[,]+", ",FAKE,,\n", noNewLines)

# noNewLines = re.sub(",FAKE,(?!,)",",FAKE,,\n",noNewLines)

# noNewLines = re.sub(",FAKE,,(?!,)",",FAKE,,\n",noNewLines)

noNewLines = re.sub(",REAL[,]+", ",REAL,,\n", noNewLines)

# noNewLines = re.sub(",REAL,(?!,)",",REAL,,\n",noNewLines)

# noNewLines = re.sub(",REAL,,(?!,)",",REAL,,\n",noNewLines) # Replace any commas between two quotes with |

lines = noNewLines.split('\n') def removeComma(g):

t = g.groups()

t = [t[0], t[1].replace(',', ' |'), t[2], t[3]] return "".join(t)

betweenQuotes = lambda line: re.sub(r'(.\*,")(.\*)(",)(.\*)', lambda x: removeComma(x), line)

secondCol = lambda line: re.sub(r'^([0-9]+,)(.\*,.\*)(,\")(.\*)$', lambda x: removeComma(x), line, 1) lines = [betweenQuotes(l) for l in lines]

lines = [secondCol(l) for l in lines] finalString = '\n'.join(lines)

file = open('fake\_or\_real\_news\_training\_CLEANED.csv', 'w',encoding= 'utf-8') file.write(finalString)

file.close()

train = pd.read\_csv("fake\_or\_real\_news\_training\_CLEANED.csv") test = pd.read\_csv("fake\_or\_real\_news\_test.csv")

train = train.drop(['X1', 'X2'], axis=1)

from collections import Counter

ax = sns.countplot(train.label, order=[x for x, count in sorted(Counter(train.label).items(), key=lambda x: -x[1])])

for p in ax.patches:

height = p.get\_height()

ax.text(p.get\_x()+p.get\_width()/2., height + 3,

'{:1.2f}%'.format(height/len(train)\*100), ha="center")

ax.set\_title("Test dataset target") show()

test['label'] = None # empty label for test df = pd.concat([train, test])

df['title\_and\_text'] = df['title'] +' '+ df['text'] df.tail()

df['preprocessed\_text'] = df['title\_and\_text'].apply(lambda x: preprocess(x)) ## Save preprocessed df

df.to\_csv("fake\_or\_real\_news\_train\_PREPROCESSED.csv", index=False) df = pd.read\_csv("fake\_or\_real\_news\_train\_PREPROCESSED.csv")

df = df.astype(object).replace(np.nan, 'None') df.tail()

encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = split\_train\_holdout\_test(encoder, df)

models = pd.DataFrame(columns=['model\_name', 'model\_object', 'score']) count\_vect = CountVectorizer(analyzer = "word")

count\_vectorizer = count\_vect.fit(df.preprocessed\_text)

train\_cv\_vector = count\_vectorizer.transform(train\_cv.preprocessed\_text)

train\_holdout\_vector = count\_vectorizer.transform(train\_holdout.preprocessed\_text) test\_vector = count\_vectorizer.transform(test.preprocessed\_text)

count\_vect.get\_feature\_names()[:10] SVC\_classifier = runModel(encoder,

train\_cv\_vector, train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "svc",

"Baseline Model 1: SVC")

models.loc[len(models)] = SVCNB = runModel(encoder, train\_cv\_vector,

train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "nb",

"Baseline Model 2: Naiive Bayes")

models.loc[len(models)] = NBmaxEnt = runModel(encoder, train\_cv\_vector,

train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "maxEnt",

"Baseline Model 3: MaxEnt Classifier") models.loc[len(models)] = maxent

df['pos\_tagged\_text'] = df['preprocessed\_text'].apply(lambda x: pos\_tag\_words(x)) df.head()

encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = split\_train\_holdout\_test(encoder, df, False)

count\_vect = CountVectorizer(analyzer = "word") count\_vectorizer = count\_vect.fit(df.preprocessed\_text)

train\_cv\_vector = count\_vectorizer.transform(train\_cv.pos\_tagged\_text)

train\_holdout\_vector = count\_vectorizer.transform(train\_holdout.pos\_tagged\_text) test\_vector = count\_vectorizer.transform(test.pos\_tagged\_text)

SVC\_pos\_tag = runModel(encoder,

train\_cv\_vector, train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "svc",

"SVC on pos-tagged text") models.loc[len(models)] = SVC\_pos\_tag NB\_pos\_tag = runModel(encoder,

train\_cv\_vector, train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "nb",

"Naiive Bayes on pos-tagged text") models.loc[len(models)] = NB\_pos\_tag maxEnt\_pos\_tag = runModel(encoder,

train\_cv\_vector, train\_cv\_label,

train\_holdout\_vector, train\_holdout.label, "maxEnt",

"MaxEnt Classifier on pos-tagged text") models.loc[len(models)] = maxEnt\_pos\_tag

df["clean\_and\_pos\_tagged\_text"] = df['preprocessed\_text'] + ' ' + df['pos\_tagged\_text'] df.head(1)

encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = split\_train\_holdout\_test(encoder, df, False)

count\_vect = CountVectorizer(analyzer = "word")

count\_vectorizer = count\_vect.fit(df.clean\_and\_pos\_tagged\_text)

train\_cv\_vector = count\_vectorizer.transform(train\_cv.clean\_and\_pos\_tagged\_text)

train\_holdout\_vector = count\_vectorizer.transform(train\_holdout.clean\_and\_pos\_tagged\_text)

test\_vector = count\_vectorizer.transform(test.clean\_and\_pos\_tagged\_text) tf\_idf = TfidfTransformer(norm="l2")

train\_cv\_tf\_idf = tf\_idf.fit\_transform(train\_cv\_vector)

train\_holdout\_tf\_idf = tf\_idf.fit\_transform(train\_holdout\_vector) test\_tf\_idf = tf\_idf.fit\_transform(test\_vector)

SVC\_tf\_idf = runModel(encoder, train\_cv\_tf\_idf,

train\_cv\_label,

train\_holdout\_tf\_idf, train\_holdout.label, "svc",

"SVC on preprocessed+pos-tagged TF-IDF weighted text") models.loc[len(models)] = SVC\_tf\_idf

NB\_tf\_idf = runModel(encoder, train\_cv\_tf\_idf,

train\_cv\_label,

train\_holdout\_tf\_idf, train\_holdout.label, "nb",

"Naiive Bayes on preprocessed+pos-tagged TF-IDF weighted text") models.loc[len(models)] = NB\_tf\_idf

maxEnt\_tf\_idf = runModel(encoder, train\_cv\_tf\_idf,

train\_cv\_label,

train\_holdout\_tf\_idf, train\_holdout.label, "maxEnt",

"MaxEnt on preprocessed+pos-tagged TF-IDF weighted text") models.loc[len(models)] = maxEnt\_tf\_idf

encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = split\_train\_holdout\_test(encoder, df, False)

trigram\_vect = CountVectorizer(analyzer = "word", ngram\_range=(1,2)) trigram\_vect = count\_vect.fit(df.clean\_and\_pos\_tagged\_text)

train\_cv\_vector = trigram\_vect.transform(train\_cv.clean\_and\_pos\_tagged\_text)

train\_holdout\_vector = trigram\_vect.transform(train\_holdout.clean\_and\_pos\_tagged\_text) test\_vector = trigram\_vect.transform(test.clean\_and\_pos\_tagged\_text)

tf\_idf = TfidfTransformer(norm="l2")

train\_cv\_bigram\_tf\_idf = tf\_idf.fit\_transform(train\_cv\_vector)

train\_holdout\_bigram\_tf\_idf = tf\_idf.fit\_transform(train\_holdout\_vector) test\_bigram\_tf\_idf = tf\_idf.fit\_transform(test\_vector)

SVC\_trigram\_tf\_idf = runModel(encoder, train\_cv\_bigram\_tf\_idf,

train\_cv\_label,

train\_holdout\_bigram\_tf\_idf, train\_holdout.label,

"svc",

"SVC on bigram vect.+ TF-IDF")

models.loc[len(models)] = SVC\_trigram\_tf\_idf

encoder, train, test, train\_cv, train\_holdout, train\_cv\_label, train\_holdout\_label = split\_train\_holdout\_test(encoder, df, False)

trigram\_vect = CountVectorizer(analyzer = "word", ngram\_range=(1,3)) trigram\_vect = count\_vect.fit(df.clean\_and\_pos\_tagged\_text)

train\_cv\_vector = trigram\_vect.transform(train\_cv.clean\_and\_pos\_tagged\_text)

train\_holdout\_vector = trigram\_vect.transform(train\_holdout.clean\_and\_pos\_tagged\_text)

tf\_idf = TfidfTransformer(norm="l2")

train\_cv\_trigram\_tf\_idf = tf\_idf.fit\_transform(train\_cv\_vector)

train\_holdout\_trigram\_tf\_idf = tf\_idf.fit\_transform(train\_holdout\_vector) maxEnt\_tf\_idf = runModel(encoder,

train\_cv\_trigram\_tf\_idf, train\_cv\_label,

train\_holdout\_trigram\_tf\_idf, train\_holdout.label, "maxEnt",

"MaxEnt on trigram vect.+ TF-IDF") models.loc[len(models)] = maxEnt\_tf\_idf

1. preprocess.py

test = pd.read\_csv("fake\_or\_real\_news\_test.csv")

train = pd.read\_csv("fake\_or\_real\_news\_training\_CLEANED.csv")

train['title\_and\_text'] = train['title'] +' '+ train['text']

train['preprocessed\_text'] = train['title\_and\_text'].apply(lambda x: preprocess(x))

test['title\_and\_text'] = test['title'] +' '+ test['text']

test['preprocessed\_text'] = test['title\_and\_text'].apply(lambda x: preprocess(x))

## Save preprocessed df

train.to\_csv("fake\_or\_real\_news\_train\_PREPROCESSED.csv", index=False)

# Save preprocessed df

test.to\_csv("fake\_or\_real\_news\_test\_PREPROCESSED.csv", index=False) train = pd.read\_csv("fake\_or\_real\_news\_train\_PREPROCESSED.csv")

train = train.astype(object).replace(np.nan, 'None')

test = pd.read\_csv("fake\_or\_real\_news\_test\_PREPROCESSED.csv") test = test.astype(object).replace(np.nan, 'None')

train['pos\_tagged\_text'] = train['preprocessed\_text'].apply(lambda x: pos\_tag\_words(x)) test['pos\_tagged\_text'] = test['preprocessed\_text'].apply(lambda x: pos\_tag\_words(x))

train["clean\_and\_pos\_tagged\_text"] = train['preprocessed\_text'] + ' ' + train['pos\_tagged\_text'] test["clean\_and\_pos\_tagged\_text"] = test['preprocessed\_text'] + ' ' + train['pos\_tagged\_text']

from sklearn.pipeline import Pipeline

trigram\_vectorizer = CountVectorizer(analyzer = "word", ngram\_range=(1,3)) tf\_idf = TfidfTransformer(norm="l2")

classifier = LogisticRegression(C=1000, class\_weight=None, dual=False, fit\_intercept=True, intercept\_scaling=1, max\_iter=100, multi\_class='multinomial',

n\_jobs=None, penalty='l2', random\_state=None, solver='saga', tol=0.0001, verbose=0, warm\_start=False)

pipeline = Pipeline([

('trigram\_vectorizer', trigram\_vectorizer), ('tfidf', tf\_idf),

('clf', classifier),

])

pipeline.fit(train.clean\_and\_pos\_tagged\_text, encoder.fit\_transform(train.label.values)) import pickle

pickle.dump( pipeline, open( "pipeline.pkl", "wb" ) ) print(colored("Predicting on test", 'blue'))

test\_predictions = test\_predictions = pipeline.predict(test.clean\_and\_pos\_tagged\_text) test\_predictions\_decoded = encoder.inverse\_transform( test\_predictions )

predictions = test

predictions["label"] = test\_predictions\_decoded import collections

ax = sns.countplot(predictions.label,

order=[x for x, count in sorted(collections.Counter(predictions.label).items(), key=lambda x: -x[1])])

for p in ax.patches:

height = p.get\_height()

ax.text(p.get\_x()+p.get\_width()/2., height + 3,

'{:1.2f}%'.format(height/len(predictions)\*100), ha="center")

ax.set\_title("Test dataset target") show()

predictions.drop(columns=["title","text","title\_and\_text","preprocessed\_text","pos\_tagged\_text","cl ean\_and\_pos\_tagged\_text"]).head()

predictions.to\_csv("TEST\_PREDICTIONS.csv", index=False) 3)prediction.py

#This is predictionModel.py File # preprocessing

import timeit

from nltk.stem import WordNetLemmatizer from nltk import pos\_tag

from nltk.corpus import stopwords from nltk.corpus import wordnet import \_pickle as pickle

import pickle import string import nltk

nltk.data.path.append('./nltk\_data')

start = timeit.default\_timer()

with open("pickle/pipeline.pkl", 'rb') as f: pipeline = pickle.load(f)

stop = timeit.default\_timer()

print('=> Pickle Loaded in: ', stop - start)

class PredictionModel: output = {}

# constructor

def init (self, text):

self.output['original'] = text

def predict(self):

self.preprocess()

self.pos\_tag\_words()

# Merge text

clean\_and\_pos\_tagged\_text = self.output['preprocessed'] + \ ' ' + self.output['pos\_tagged']

self.output['prediction'] = 'FAKE' if pipeline.predict( [clean\_and\_pos\_tagged\_text])[0] == 0 else 'REAL'

return self.output

# Helper methods def preprocess(self):

# lowercase the text

text = self.output['original'].lower()

# remove the words counting just one letter text = [t for t in text.split(" ") if len(t) > 1]

# remove the words that contain numbers

text = [word for word in text if not any(c.isdigit() for c in word)]

# tokenize the text and remove puncutation

text = [word.strip(string.punctuation) for word in text]

# remove all stop words

stop = stopwords.words('english') text = [x for x in text if x not in stop]

# remove tokens that are empty text = [t for t in text if len(t) > 0]

# pos tag the text pos\_tags = pos\_tag(text)

# lemmatize the text

text = [WordNetLemmatizer().lemmatize(t[0], self.get\_wordnet\_pos(t[1])) for t in pos\_tags]

# join all

self.output['preprocessed'] = " ".join(text)

def get\_wordnet\_pos(self, pos\_tag): if pos\_tag.startswith('J'):

return wordnet.ADJ

elif pos\_tag.startswith('V'): return wordnet.VERB

elif pos\_tag.startswith('N'): return wordnet.NOUN

elif pos\_tag.startswith('R'): return wordnet.ADV

else:

return wordnet.NOUN

def pos\_tag\_words(self):

pos\_text = nltk.pos\_tag(

nltk.word\_tokenize(self.output['preprocessed'])) self.output['pos\_tagged'] = " ".join(

[pos + "-" + word for word, pos in pos\_text]) 4)app.py

from flask import Flask, jsonify, request, render\_template from predictionModel import PredictionModel

import pandas as pd

from random import randrange

app = Flask( name , static\_folder="./public/static", template\_folder="./public")

@app.route("/") def home():

return render\_template('index.html')

@app.route('/predict', methods=['POST']) def predict():

model = PredictionModel(request.json) return jsonify(model.predict())

@app.route('/random', methods=['GET']) def random():

data = pd.read\_csv("data/fake\_or\_real\_news\_test.csv") index = randrange(0, len(data)-1, 1)

return jsonify({'title': data.loc[index].title, 'text': data.loc[index].text})

# Only for local running

if name == ' main ': app.run()

## CLASSIFIERS/PREDICTORS USED:

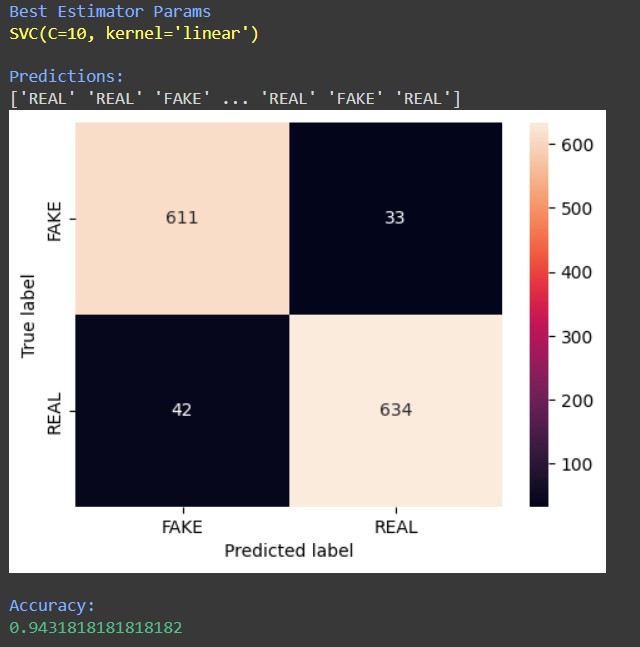
The Predictors used are:

1. Support Vector Classifier(SVC)
2. Multinomial Naïve Bayes Classifier
3. Logistic Regression Classifier

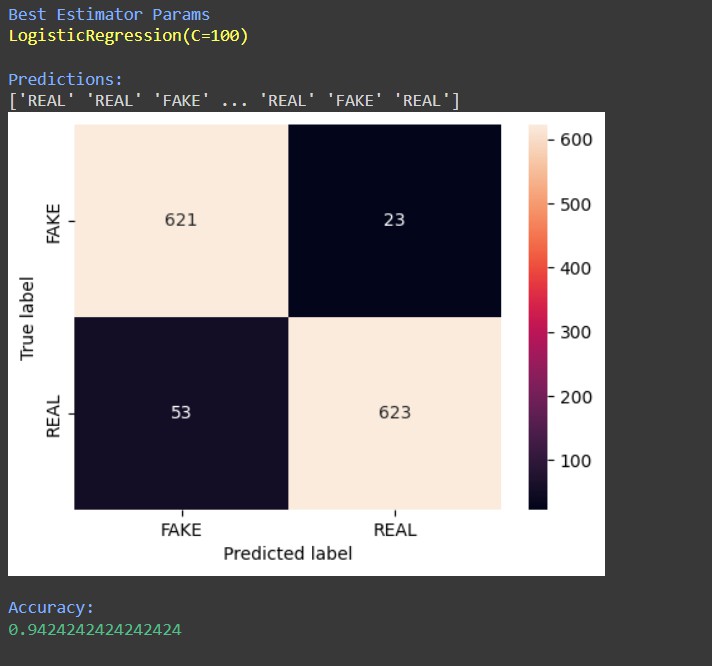
These classifiers are trained and evaluated using different feature representations, such as Bag-of-Words (Count Vectorizer), TF-IDF (Term Frequency-Inverse Document Frequency), and a combination of preprocessed and part-of-speech (POS) tagged text.

RESULT COMPARISION:

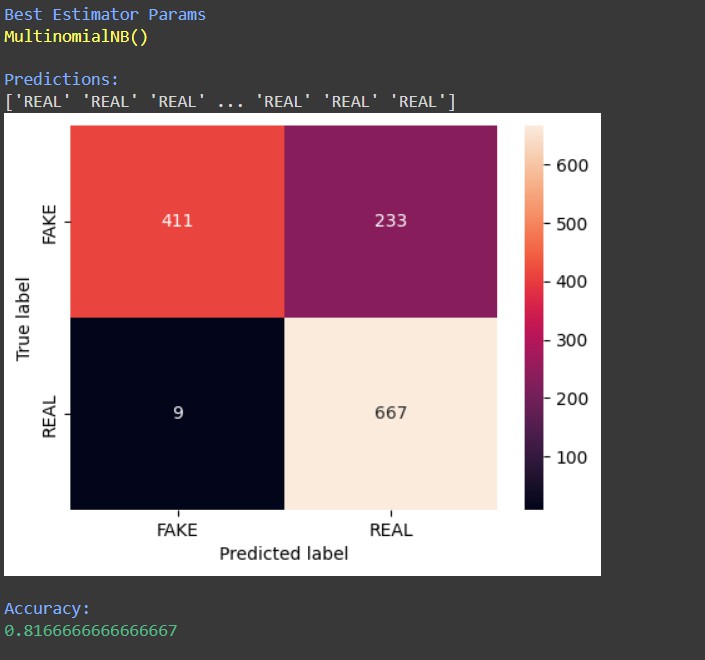
1. SVC:



1. Linear Regression:

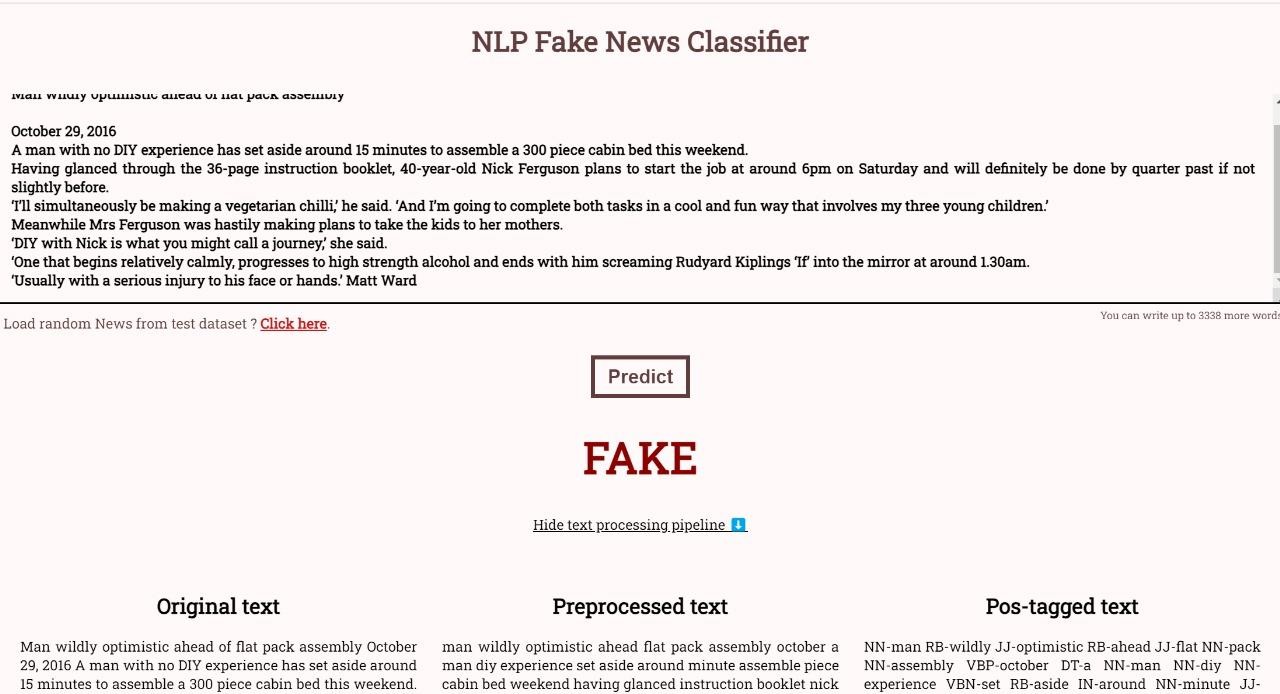


1. Naïve Bayes:



**OUTPUT: 1)**





**2) BASED ON USER INPUT:**



**OVERALL ANALYSIS:**

Objective:

* Text classification using ML algorithms. Preprocessing:
* Lowercasing, noise removal, lemmatization, and POS tagging. Model Training:
* SVC, MNB, and Logistic Regression with hyperparameter tuning. TF-IDF vectorization.

Evaluation:

* Accuracy and confusion matrix comparison.
* Performance across classifiers assessed. Strengths:
* Robust preprocessing.
* Hyperparameter optimization.
* Comparative analysis aids algorithm selection. Limitations:
* Limited feature engineering.
* Lack of detailed explanation.
* Single metric evaluation. Future Improvements:
* Ensemble methods exploration.
* Feature selection techniques.
* Diverse cross-validation strategies.
* Deep learning investigation.

**CONCLUSION:**

* Project is aimed at detecting fake news through preprocessing, feature engineering, training, and evaluating models.
* Classifiers like SVC, MultinomialNB, and Logistic Regression were employed.
* Preprocessing included lowercase conversion, punctuation removal, stop word elimination, and lemmatization.
* Feature engineering involved TF-IDF vectors and POS-tagged text.
* Models were trained and evaluated using cross-validation and grid search.
* Performance metrics included accuracy, precision, and confusion matrices.
* Despite varying performance, the project emphasized preprocessing and feature engineering's significance.
* Further refinement is needed for more robust fake news detection systems.

# REFERENCES:

* + [**https://ieeexplore.ieee.org/document/9378748**](https://ieeexplore.ieee.org/document/9378748)
  + [**https://irjmets.com/uploadedfiles/paper/issue\_6\_june\_20**](https://irjmets.com/uploadedfiles/paper/issue_6_june_2022/26445/final/fin_irjmets1655633757.pdf)[**22/26445/final/fin\_irjmets1655633757.pdf**](https://irjmets.com/uploadedfiles/paper/issue_6_june_2022/26445/final/fin_irjmets1655633757.pdf)
  + [**https://www.researchgate.net/publication/352021652\_Fak**](https://www.researchgate.net/publication/352021652_Fake_News_Detection_Using_Machine_Learning_and_Deep_Learning_Algorithms)[**e\_News\_Detection\_Using\_Machine\_Learning\_and\_Deep\_L**](https://www.researchgate.net/publication/352021652_Fake_News_Detection_Using_Machine_Learning_and_Deep_Learning_Algorithms)[**earning\_Algorithms**](https://www.researchgate.net/publication/352021652_Fake_News_Detection_Using_Machine_Learning_and_Deep_Learning_Algorithms)
  + **Machine learning notes and texts from Google.**