#=== Packages used in this research ===#

import pandas as FNPDS

import seaborn as FNSBN

import matplotlib.pyplot as FNMTP

import warnings as FNWRN

FNWRN.filterwarnings("ignore")

import time as FNTM

from collections import defaultdict as FNDDict

import numpy as FNmpy

#=== Import the Fake News Data using Pandas ===#

FN\_Classification = FNPDS.read\_csv('FNews\_Classification.csv')

FN\_Classification = FN\_Classification.sample(1000, random\_state=7) #=== Runtime crashed if i take large or full amount of data ===#

FN\_Classification

FN\_input = FN\_Classification['text']

FN\_output = FN\_Classification['label']

#====Vectorization TF IDF Process ====#

FN\_input = FN\_input.fillna('').astype(str)

#==== Generate vocabulary and calculate TF-IDF ====#

def find\_tfidf(fncorpus):

    fnvocab = {}

    fn\_idf = FNDDict(int)

    num\_docs = len(fncorpus)

    #==== Calculate term frequencies and update vocab ====#

    FNews\_tf = []

    for fndoc in fncorpus:

        fndoc\_tf = FNDDict(int)

        for fnword in fndoc.split():

            fndoc\_tf[fnword] += 1

            if fnword not in fnvocab:

                fnvocab[fnword] = len(fnvocab)

        FNews\_tf.append(fndoc\_tf)

        for fnword in set(fndoc.split()):

            fn\_idf[fnword] += 1

    #=== Calculate IDF ===#

    for fnword in fn\_idf:

        fn\_idf[fnword] = FNmpy.log(num\_docs / (1 + fn\_idf[fnword]))

    #=== Calculate TF-IDF matrix ====#

    fntf\_idf = FNmpy.zeros((num\_docs, len(fnvocab)))

    for i, fndoc\_tf in enumerate(FNews\_tf):

        for fnword, fncount in fndoc\_tf.items():

            if fnword in fnvocab:

                fntf\_idf[i, fnvocab[fnword]] = fncount \* fn\_idf[fnword]

    return fntf\_idf, fnvocab

#=== Get TF-IDF vectors and vocabulary ===#

FN\_tfidf\_ip, fnvocab = find\_tfidf(FN\_input)

#=== Convert output labels to array ===#

FN\_output = FNmpy.array(FN\_output)

print("Fake News data shape using TF-IDF Vectorization =", FN\_tfidf\_ip.shape)

#=== Split data manually for train and test (80-20) ===#

def FNTrnTstSplt(fnX, fnY, fntst\_sze=0.2, random\_state=7):

    FNmpy.random.seed(random\_state)

    fnindices = FNmpy.random.permutation(len(fnX))

    fntst\_sze = int(len(fnX) \* fntst\_sze)

    fntrn\_indices = fnindices[fntst\_sze:]

    fntst\_indices = fnindices[:fntst\_sze]

    return fnX[fntrn\_indices], fnX[fntst\_indices], fnY[fntrn\_indices], fnY[fntst\_indices]

fnXtr, fnXtst, fnYtr, fnYtst = FNTrnTstSplt(FN\_tfidf\_ip, FN\_output, fntst\_sze=0.2)

fnXtr.shape

fnXtst.shape

from collections import Counter as FNCntr

#=== Decision Tree Classifier from Scratch ====#

class FND\_Tree:

    def \_\_init\_\_(self, max\_depth=10, min\_samples\_split=2):

        self.max\_depth = max\_depth

        self.min\_samples\_split = min\_samples\_split

        self.tree = None

    def fit(self, fnX, fnY):

        self.tree = self.\_build\_tree(fnX, fnY)

    def \_build\_tree(self, fnX, fnY, depth=0):

        num\_sampFN, num\_featFN = fnX.shape

        if (depth >= self.max\_depth or num\_sampFN < self.min\_samples\_split or len(set(fnY)) == 1):

            return self.\_majority\_vote(fnY)

        #=== Finding best split ===#

        FNbest\_feat, FNbest\_thres = self.\_best\_split(fnX, fnY, num\_featFN)

        if FNbest\_feat is None:

            return self.\_majority\_vote(fnY)

        #=== Split data  ===#

        FNleft\_indices = fnX[:, FNbest\_feat] < FNbest\_thres

        FNright\_indices = fnX[:, FNbest\_feat] >= FNbest\_thres

        FNleft\_tree = self.\_build\_tree(fnX[FNleft\_indices], fnY[FNleft\_indices], depth + 1)

        FNright\_tree = self.\_build\_tree(fnX[FNright\_indices], fnY[FNright\_indices], depth + 1)

        return {

            'feature': FNbest\_feat,

            'threshold': FNbest\_thres,

            'left': FNleft\_tree,

            'right': FNright\_tree

        }

    def \_best\_split(self, fnX, fny, num\_featFN):

        best\_gain = 0

        FNbest\_feat, FNbest\_thres = None, None

        for feature in range(num\_featFN):

            thresholds = FNmpy.unique(fnX[:, feature])

            for threshold in thresholds:

                left = fny[fnX[:, feature] < threshold]

                right = fny[fnX[:, feature] >= threshold]

                if len(left) == 0 or len(right) == 0:

                    continue

                gain = self.\_information\_gain(fny, left, right)

                if gain > best\_gain:

                    best\_gain, FNbest\_feat, FNbest\_thres = gain, feature, threshold

        return FNbest\_feat, FNbest\_thres

    def \_information\_gain(self, parent, left, right):

        #=== Gini impurity calculation ===#

        def gini\_impurity(fnY):

            counts = FNmpy.bincount(fnY)

            probabilities = counts / len(fnY)

            return 1 - FNmpy.sum(probabilities \*\* 2)

        p = float(len(left)) / len(parent)

        return gini\_impurity(parent) - p \* gini\_impurity(left) - (1 - p) \* gini\_impurity(right)

    def \_majority\_vote(self, fnY):

        counter = FNCntr(fnY)

        return counter.most\_common(1)[0][0]

    def \_predict\_row(self, row, node):

        if not isinstance(node, dict):

            return node

        if row[node['feature']] < node['threshold']:

            return self.\_predict\_row(row, node['left'])

        else:

            return self.\_predict\_row(row, node['right'])

    def predict(self, fnX):

        return FNmpy.array([self.\_predict\_row(row, self.tree) for row in fnX])

#=== Decision Tree Training ===#

FNstart = FNTM.time()

FNtree = FND\_Tree(max\_depth=10, min\_samples\_split=5)

FNtree.fit(fnXtr, fnYtr)

FNend = FNTM.time()

print("Training Time of Decision Tree = ",FNend-FNstart)

#=== Decision Tree Testing ===#

FNstart = FNTM.time()

fnYpd = FNtree.predict(fnXtst)

FNacc = FNmpy.mean(fnYpd == fnYtst)

FNend = FNTM.time()

print("Testing Time of Decision Tree = ",FNend-FNstart)

print(f"Decision Tree Accuracy in Fake News = {FNacc \* 100:.2f}%")

#=== Decision Tree Classification Report (Precision, Recall, F1-Score) ===#

def classReport(fnY\_tru, fnYpd):

    fntru\_pos = sum((fnY\_tru == 1) & (fnYpd == 1))

    fntru\_neg = sum((fnY\_tru == 0) & (fnYpd == 0))

    fnfals\_pos = sum((fnY\_tru == 0) & (fnYpd == 1))

    fnfals\_neg = sum((fnY\_tru == 1) & (fnYpd == 0))

    fnprecis = fntru\_pos / (fntru\_pos + fnfals\_pos) if (fntru\_pos + fnfals\_pos) > 0 else 0

    fnrecal = fntru\_pos / (fntru\_pos + fnfals\_neg) if (fntru\_pos + fnfals\_neg) > 0 else 0

    fnf1\_scr = 2 \* fnprecis \* fnrecal / (fnprecis + fnrecal) if (fnprecis + fnrecal) > 0 else 0

    fnacc = (fntru\_pos + fntru\_neg) / len(fnY\_tru)

    print(f"Decision Tree Classification Report:")

    print(f"Fake News - DT Accuracy = {fnacc:.2f}")

    print(f"Fake News - DT Precision= {fnprecis:.2f}")

    print(f"Fake News - DT Recall   = {fnrecal:.2f}")

    print(f"Fake News - DT F1 Score = {fnf1\_scr:.2f}")

classReport(fnYtst, fnYpd)

#=== AUC and ROC Curve for Decision Tree ===#

from sklearn.metrics import roc\_curve, auc

fnfpr, fntpr, \_ = roc\_curve(fnYtst, fnYpd)

fnRocAuc = auc(fnfpr, fntpr)

FNMTP.figure()

FNMTP.plot(fnfpr, fntpr, color='springgreen', label=f'ROC curve (AUC = {fnRocAuc:.2f})')

FNMTP.plot([0, 1], [0, 1], color='mediumvioletred', linestyle='--')

FNMTP.xlabel('False Pos-Rate')

FNMTP.ylabel('True Pos-Rate')

FNMTP.title('Receiver Operating Characteristic')

FNMTP.legend(loc="lower right")

FNMTP.show()