

SOCIAL NETWORK THEORY AND PRACTICE

PROJECT



BSE-7A2

UMAIR, NAYYAR , UMER

21F-9612,21F-9173,21F-9291

**PROJECT**

**CODE:**

import tensorflow as tf

import numpy as np

import random

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

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

from sklearn.metrics import precision\_recall\_curve, confusion\_matrix, precision\_score, recall\_score, f1\_score, roc\_auc\_score

from sklearn.utils.class\_weight import compute\_class\_weight

from imblearn.over\_sampling import SMOTE

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv1D, BatchNormalization, MaxPooling1D, GlobalAveragePooling1D, Dense, Dropout

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

np.random.seed(42)

random.seed(42)

tf.random.set\_seed(42)

file\_path = '/content/bio-WormNet-v3.edges'

data = pd.read\_csv(file\_path, sep=' ', header=None, names=['source', 'target', 'weight'])

data['weight'] = data['weight'].fillna(data['weight'].mean())

def frequency\_processing(q):

    data = np.zeros(256)

    for value in q:

        if value < 256:

            data[value] += 1

    data /= len(q)

    return data

def frequency\_weighted\_processing(q):

    data = np.zeros(256)

    for value in q:

        if value < 256:

            weight = 1 / (1 + abs(value - 128))

            data[value] += weight

    data /= len(q)

    return data

def difference\_processing(q):

    data = np.zeros(256)

    for i in range(1, len(q)):

        diff = abs(q[i] - q[i-1])

        if diff < 256:

            data[diff] += 1

    data /= len(q)

    return data

source\_data = data['source'].values

source\_data = np.mod(source\_data, 256)

X\_data\_freq = np.array([frequency\_processing(np.random.randint(0, 256, 256)) for \_ in range(1000)])

X\_data\_freq\_weighted = np.array([frequency\_weighted\_processing(np.random.randint(0, 256, 256)) for \_ in range(1000)])

X\_data\_diff = np.array([difference\_processing(np.random.randint(0, 256, 256)) for \_ in range(1000)])

y\_data = np.random.randint(0, 2, len(X\_data\_freq))

scaler = MinMaxScaler()

X\_data\_freq = scaler.fit\_transform(X\_data\_freq)

X\_data\_freq\_weighted = scaler.fit\_transform(X\_data\_freq\_weighted)

X\_data\_diff = scaler.fit\_transform(X\_data\_diff)

X\_train\_freq, X\_test\_freq, y\_train, y\_test = train\_test\_split(X\_data\_freq, y\_data, test\_size=0.2, random\_state=42)

X\_train\_freq\_weighted, X\_test\_freq\_weighted, \_, \_ = train\_test\_split(X\_data\_freq\_weighted, y\_data, test\_size=0.2, random\_state=42)

X\_train\_diff, X\_test\_diff, \_, \_ = train\_test\_split(X\_data\_diff, y\_data, test\_size=0.2, random\_state=42)

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_train\_freq\_res, y\_train\_res = smote.fit\_resample(X\_train\_freq, y\_train)

X\_train\_freq\_weighted\_res, y\_train\_weighted\_res = smote.fit\_resample(X\_train\_freq\_weighted, y\_train)

X\_train\_diff\_res, y\_train\_diff\_res = smote.fit\_resample(X\_train\_diff, y\_train)

class\_weights = compute\_class\_weight('balanced', classes=np.unique(y\_train\_res), y=y\_train\_res)

class\_weights\_dict = dict(zip(np.unique(y\_train\_res), class\_weights))

X\_train\_freq\_cnn = X\_train\_freq\_res.reshape(X\_train\_freq\_res.shape[0], X\_train\_freq\_res.shape[1], 1)

X\_train\_freq\_weighted\_cnn = X\_train\_freq\_weighted\_res.reshape(X\_train\_freq\_weighted\_res.shape[0], X\_train\_freq\_weighted\_res.shape[1], 1)

X\_train\_diff\_cnn = X\_train\_diff\_res.reshape(X\_train\_diff\_res.shape[0], X\_train\_diff\_res.shape[1], 1)

X\_test\_freq\_cnn = X\_test\_freq.reshape(X\_test\_freq.shape[0], X\_test\_freq.shape[1], 1)

X\_test\_freq\_weighted\_cnn = X\_test\_freq\_weighted.reshape(X\_test\_freq\_weighted.shape[0], X\_test\_freq\_weighted.shape[1], 1)

X\_test\_diff\_cnn = X\_test\_diff.reshape(X\_test\_diff.shape[0], X\_test\_diff.shape[1], 1)

def create\_cnn\_model(input\_shape):

    model = Sequential([

        Conv1D(128, kernel\_size=3, activation='relu', input\_shape=input\_shape, padding='same'),

        BatchNormalization(),

        MaxPooling1D(pool\_size=2),

        Conv1D(256, kernel\_size=3, activation='relu', padding='same'),

        BatchNormalization(),

        MaxPooling1D(pool\_size=2),

        Conv1D(512, kernel\_size=3, activation='relu', padding='same'),

        BatchNormalization(),

        MaxPooling1D(pool\_size=2),

        GlobalAveragePooling1D(),

        Dense(1024, activation='relu'),

        Dropout(0.5),

        Dense(1, activation='sigmoid')  # Binary classification

    ])

    model.compile(optimizer=Adam(learning\_rate=0.0001), loss='binary\_crossentropy', metrics=['accuracy'])

    return model

def evaluate\_model(X\_train, y\_train, X\_test, y\_test):

    cnn\_model = create\_cnn\_model((X\_train.shape[1], 1))

    callbacks = [

        EarlyStopping(monitor='val\_loss', patience=10, restore\_best\_weights=True),

        ModelCheckpoint('best\_cnn\_model.keras', monitor='val\_loss', save\_best\_only=True)

    ]

    history\_cnn = cnn\_model.fit(X\_train, y\_train, epochs=12, batch\_size=32, validation\_split=0.2,

                                callbacks=callbacks, class\_weight=class\_weights\_dict, shuffle=True)

    accuracy\_cnn = history\_cnn.history['accuracy'][-1]

    # Predictions

    preds\_cnn = cnn\_model.predict(X\_test)

    # Precision-Recall and AUC

    precision, recall, thresholds = precision\_recall\_curve(y\_test, preds\_cnn)

    optimal\_idx = np.argmax(precision + recall)

    optimal\_threshold = thresholds[optimal\_idx]

    final\_predictions\_cnn = (preds\_cnn > optimal\_threshold).astype(int)

    tn, fp, fn, tp = confusion\_matrix(y\_test, final\_predictions\_cnn).ravel()

    accuracy = (tp + tn) / (tp + tn + fp + fn)

    precision\_value = precision\_score(y\_test, final\_predictions\_cnn)

    recall\_value = recall\_score(y\_test, final\_predictions\_cnn)

    f1\_value = f1\_score(y\_test, final\_predictions\_cnn)

    auc\_value = roc\_auc\_score(y\_test, preds\_cnn)

    loss\_value = history\_cnn.history['loss'][-1]

    return accuracy, accuracy\_cnn, precision\_value, recall\_value, f1\_value, auc\_value, loss\_value

accuracy\_freq, history\_accuracy\_freq, precision\_freq, recall\_freq, f1\_freq, auc\_freq, loss\_freq = evaluate\_model(X\_train\_freq\_cnn, y\_train\_res, X\_test\_freq\_cnn, y\_test)

accuracy\_freq\_weighted, history\_accuracy\_weighted, precision\_weighted, recall\_weighted, f1\_weighted, auc\_weighted, loss\_weighted = evaluate\_model(X\_train\_freq\_weighted\_cnn, y\_train\_weighted\_res, X\_test\_freq\_weighted\_cnn, y\_test)

accuracy\_diff, history\_accuracy\_diff, precision\_diff, recall\_diff, f1\_diff, auc\_diff, loss\_diff = evaluate\_model(X\_train\_diff\_cnn, y\_train\_diff\_res, X\_test\_diff\_cnn, y\_test)

# Print accuracies and additional metrics

print(f"Final Training Accuracy with Frequency Processing: {history\_accuracy\_freq \* 100:.2f}%")

print(f"Final Training Accuracy with Frequency-Weighted Processing: {history\_accuracy\_weighted \* 100:.2f}%")

print(f"Final Training Accuracy with Difference Processing: {history\_accuracy\_diff \* 100:.2f}%")

**OUTPUT:**

