Appendix

Complex Networks 93%

Code in python 3.12.7 tested on Manjaro Linux

```
import os,mne,warnings
import numpy as np
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV,StratifiedKFold,cross_val_predict
from sklearn.impute import SimpleImputer
from sklearn.metrics import classification_report,confusion_matrix
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.exceptions import UndefinedMetricWarning
# Paths ---------------------------------------------------------
base_path="/home/a/Downloads/schizophrenia/eeg/"
output_path="/home/a/Downloads/schizophrenia/eeg/dft.txt"
# ---------------------------------------------------------------
# Ignore specific warnings to clean the output
warnings.filterwarnings("ignore",category=UndefinedMetricWarning)
# 1. Load EEG data and filter by selected channels
def load_eeg_data(file_path):
  raw=mne.io.read_raw_edf(file_path,preload=True)
  raw.pick(['O1','O2','P4','T5','Pz','T6'])
  return raw
# 2. Generate adjacency matrices for the selected channels
def get_adjacency_matrix(raw,window_size_sec=299,overlap=0.0):
  sfreq=raw.info['sfreq']
  window_size_samples=int(window_size_sec*sfreq)
  step_size=int(window_size_samples*(1-overlap))
  data=raw.get_data()
  data=StandardScaler().fit_transform(data.T).T
  adjacency_matrices=[]
  for start in range(0,data.shape[1]-window_size_samples+1,step_size):
  end=start+window_size_samples
  window_data=data[:,start:end]
  if np.var(window_data)>1e-6:
  corr_matrix=np.corrcoef(window_data)
  if not np.isnan(corr_matrix).any():
  adjacency_matrices.append(corr_matrix)
  return np.array(adjacency_matrices)
# 3. Discrete Fourier Transform (DFT) and spectral eigenvalues
def dft_matrix(N):
```

```
    omega=np.exp(2j*np.pi/N)
    return np.array([[omega**(i*j) for j in range(N)] for i in range(N)])/np.sqrt(N)
  def apply_dft_to_adjacency(adjacency_matrices):
    dft_results=[]
    spectral_features=[]
    for matrix in adjacency_matrices:
    N=matrix.shape[0]
    dft=dft_matrix(N)
    transformed_matrix=np.real(dft@matrix@dft.conj().T)
    if not np.isnan(transformed_matrix).any():
    dft_results.append(transformed_matrix.flatten())
    eigenvalues=np.linalg.eigvalsh(transformed_matrix)
    spectral_features.append(eigenvalues)
    return np.array(dft_results),np.array(spectral_features)
  # 4. Model optimization and classification with StratifiedKFold and PCA
  def fine_tuned_classification(subjects,features,labels,output_path):
    # Define Pipeline
    pipeline=Pipeline([
    ('imputer',SimpleImputer(strategy='mean')),
    ('scaler',MinMaxScaler()),
    ('pca',PCA(n_components=0.95)),
    ('clf',RandomForestClassifier(random_state=33)) # 2 e 33 c
    ])
    # Define parameters for GridSearch
    param_grid={'clf__n_estimators':[100],'clf__max_depth':[None],'clf__min_samples_split':[10]}
    # Configure StratifiedKFold
    skf=StratifiedKFold(n_splits=14,shuffle=True,random_state=2)
    # GridSearchCV
    grid_search=GridSearchCV(pipeline,param_grid,cv=skf,n_jobs=-1,scoring='accuracy')
    grid_search.fit(features,labels)
    best_params=grid_search.best_params_
    print(f"Best parameters:{best_params}")
    print(f"Best accuracy (GridSearchCV):{grid_search.best_score_*100:.2f}%")
    # Get predictions
    y_pred=cross_val_predict(grid_search.best_estimator_,features,labels,cv=skf,n_jobs=-1)
    # Classification report
    report=classification_report(labels,y_pred,target_names=['Control','Schizophrenia'],zero_divi-
sion=0)
    cm=confusion_matrix(labels,y_pred)
    print("\nClassification Report:")
    print(report)
    print("Confusion Matrix:")
    print(cm)
    # Save results per subject
    results=[]
    for subject,pred in zip(subjects,y_pred):
    predicted_label="Schizophrenia" if pred==1 else "Control"
```

```
    results.append(f"{subject}-{predicted_label}")
    # Calculate percentages
    correct_count=np.sum(y_pred==labels)
    total_accuracy=(correct_count/len(labels))*100
    h_correct=np.sum((labels==0) & (y_pred==0))
    s_correct=np.sum((labels==1) & (y_pred==1))
    h_total=np.sum(labels==0)
    s_total=np.sum(labels==1)
    h_accuracy=(h_correct/h_total)*100 if h_total>0 else 0
    s_accuracy=(s_correct/s_total)*100 if s_total>0 else 0
    results.append("\nSummary of results:")
    results.append(f"Total accuracy:{total_accuracy:.2f}%")
    results.append(f"Accuracy for controls (h):{h_accuracy:.2f}%")
    results.append(f"Accuracy for schizophrenia (s):{s_accuracy:.2f}%")
    # Save results to file
    with open(output_path,"w") as f:
    f.write("\n".join(results))
    print(f"Results saved in {output_path}")
# 5. Execute complete pipeline with optimization
if __name__=="__main__":
    all_subjects=[]
    all_adjacency_matrices=[]
    all_spectral_features=[]
    labels_global=[]
    # Load EEG data per subject
    for i in range(1,15):
    if os.path.exists(f"{base_path}h{i:02d}.edf"): # Controls
    subject_id=f"h{i:02d}"
    raw_data=load_eeg_data(f"{base_path}{subject_id}.edf")
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
    print(f"No adjacency matrices generated for {subject_id}")
    continue
    dft_data,spectral_data=apply_dft_to_adjacency(matrices)
    if dft_data.size==0 or spectral_data.size==0:
    print(f"No features generated for {subject_id}")
    continue
    all_subjects.append(subject_id)
    all_adjacency_matrices.append(dft_data.mean(axis=0))
    all_spectral_features.append(spectral_data.mean(axis=0))
    labels_global.append(0) # Label 0:Control
    if os.path.exists(f"{base_path}s{i:02d}.edf"): # Schizophrenia
    subject_id=f"s{i:02d}"
    raw_data=load_eeg_data(f"{base_path}{subject_id}.edf")
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
    print(f"No adjacency matrices generated for {subject_id}")
```

```
continue
dft_data,spectral_data=apply_dft_to_adjacency(matrices)
if dft_data.size==0 or spectral_data.size==0:
print(f"No features generated for {subject_id}")
continue
all_subjects.append(subject_id)
all_adjacency_matrices.append(dft_data.mean(axis=0))
all_spectral_features.append(spectral_data.mean(axis=0))
labels_global.append(1) # Label 1:Schizophrenia
# Convert to arrays
all_adjacency_matrices=np.array(all_adjacency_matrices)
all_spectral_features=np.array(all_spectral_features)
labels_global=np.array(labels_global)
# Combine DFT and spectral features
combined_features=np.hstack((all_adjacency_matrices,all_spectral_features))
# Execute optimized classification with accuracy calculation
fine_tuned_classification(all_subjects,combined_features,labels_global,output_path)
```

Complexity, Higuchi fractal dimension, and Lyapunov exponents 67%

Code in python 3.12.7 tested on Manjaro Linux

```
import os,mne,nolds
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import Leave-One-Group-Out
from sklearn.metrics import accuracy_score,classification_report
from sklearn.feature_selection import f_classif,mutual_info_classif
import matplotlib.pyplot as plt
from collections import defaultdict
# Paths ---------------------------------------------------------------------
base_path='/home/a/Downloads/schizophrenia/eeg/'
```

```
output_path='/home/a/Downloads/schizophrenia/eeg/100.txt'
# ------------------------------------------------------------------------------
# Function to load EEG data from EDF files
def load_eeg_data(base_path,label,subject_ids):
  data=[]
  labels=[]
  groups=[]
  for subj in subject_ids:
  filename=f"{label}{subj:02d}.edf"
  file_path=os.path.join(base_path,filename)
  if os.path.exists(file_path):
  try:
  raw=mne.io.read_raw_edf(file_path,preload=True,verbose=False)
  eeg_data=raw.get_data() # Get data as array (n_channels,n_samples)
  n_channels=eeg_data.shape[0]
  for ch in range(n_channels):
  eeg_signal=eeg_data[ch].squeeze()
  data.append(eeg_signal)
  labels.append(label)
  groups.append(f"{label}{subj:02d}") # Assign group based on subject
  except Exception as e:
  print(f"Error loading {file_path}: {e}")
  else:
  print(f"File not found: {file_path}")
  return np.array(data,dtype=object),np.array(labels),np.array(groups)
# Function to calculate Lempel-Ziv Complexity (Cx)
def lempel_ziv_complexity(signal):
  # Convert the signal to a sequence of symbols
  symbols=''.join(['1' if x>0 else '0' for x in signal])
  c=1
  i=0
  dictionary=defaultdict(int)
  n=len(symbols)
  while i<n:
  j=i+1
  while j<=n and symbols[i:j] in dictionary:
  j+=1
  dictionary[symbols[i:j]]=1
  c+=1
  i=j
  return c/n
# Function to calculate Higuchi Fractal Dimension (HFD)
def higuchi_fd(signal,kmax=10):
  N=len(signal)
  L=[]
  for k in range(1,kmax+1):
  Lk=[]
```

```
    for m in range(k):
    idx=np.arange(1,int(np.floor((N-m)/k)),dtype=int)
    if len(idx)==0:
    continue
    Lmk=np.sum(np.abs(signal[m+k*idx]-signal[m+k*(idx-1)]))
    Lmk=(Lmk*(N-1))/(k*len(idx)*(2*np.pi))
    if Lmk>0:
    Lk.append(np.log(Lmk))
    if len(Lk)==0:
    L.append(0)
    else:
    L.append(np.log(k)-np.mean(Lk))
    # Perform linear regression to obtain the slope
    if len(L)<2:
    return 0
    coeffs=np.polyfit(np.log(range(1,kmax+1)),L,1)
    return coeffs[0]
  # Function to calculate Lyapunov Exponents (Lya)
  def lyapunov_exponents(signal):
    try:
    return nolds.lyap_r(signal,emb_dim=10,tau=1)
    except:
    return 0 # In case of error,return 0
  # Function to extract features (Cx and HFD)
  def extract_features(eeg_signal):
    Cx=lempel_ziv_complexity(eeg_signal)
    HFD=higuchi_fd(eeg_signal)
    return [Cx,HFD]
  # Define paths and subjects
  control_label='h'
  schizophrenia_label='s'
  subject_ids=range(1,15) # 01 to 14
  # Load data
  print("Loading control data...")
  X_control,y_control,groups_control=load_eeg_data(base_path,control_label,subject_ids)
  print(f"Control data loaded: {len(X_control)} samples.")
  print("Loading schizophrenia data...")
  X_schizophrenia,y_schizophrenia,groups_schizophrenia=load_eeg_data(base_path,schizophre-
nia_label,subject_ids)
  print(f"Schizophrenia data loaded: {len(X_schizophrenia)} samples.")
  # Combine data
  if len(X_control)==0 or len(X_schizophrenia)==0:
    print("Error: Not enough data loaded.")
    exit(1)
  X=np.concatenate((X_control,X_schizophrenia),axis=0)
  y=np.concatenate((np.zeros(len(y_control)),np.ones(len(y_schizophrenia))),axis=0)  #  0:  Control,1:
Schizophrenia
```

```
groups=np.concatenate((groups_control,groups_schizophrenia),axis=0)
# Extract features for all data
print("Extracting features...")
features=[]
for i,signal in enumerate(X):
  if isinstance(signal,np.ndarray):
  features.append(extract_features(signal))
  else:
  print(f"Warning: Signal {i} is not a NumPy array.")
  features.append([0,0]) # Assign default features
features=np.array(features)
print(f"Features extracted: {features.shape}")
# Verify that all features are numeric and do not contain NaNs
if np.isnan(features).any():
  print("Error: Some features contain NaNs.")
  exit(1)
# Calculate F-values using ANOVA
print("Calculating F-values and IGR...")
F_values,_=f_classif(features,y)
# Identify constant features and remove them
constant_features=np.where(np.isclose(F_values,0))[0]
if len(constant_features)>0:
  print(f"Identified constant features: {constant_features}")
  # Remove constant features
  features=np.delete(features,constant_features,axis=1)
  F_values=np.delete(F_values,constant_features)
  IGR_values=np.delete(mutual_info_classif(features,y,discrete_features=False),constant_features)
else:
  IGR_values=mutual_info_classif(features,y,discrete_features=False)
# Calculate IGR using Mutual Information
if len(constant_features)==0:
  IGR_values=mutual_info_classif(features,y,discrete_features=False)
else:
  IGR_values=mutual_info_classif(features,y,discrete_features=False)
  # Normalize IGR
  if np.log2(len(np.unique(y))) != 0:
  IGR_values /= np.log2(len(np.unique(y)))
  else:
  IGR_values=np.zeros_like(IGR_values)
# Apply fusion rules
print("Applying fusion rules R1 to R5...")
num_features=features.shape[1]
# Ensure there are at least 2 features
if num_features<2:
  print("Error: At least 2 features are required for fusion.")
  exit(1)
# R1: Sum of features
```

```
R1=np.sum(features,axis=1)
# R2: Product of features
R2=np.prod(features,axis=1)
# R3: Division of the first feature by the second
# Avoid division by zero
R3=np.divide(features[:,0],np.where(features[:,1]==0,1e-10,features[:,1]))
# R4: Weighted sum with F-values
R4=np.dot(features,F_values)
# R5: Weighted sum with IGR
R5=np.dot(features,IGR_values)
# Normalize the fused features to the range [-1,1]
print("Normalizing features...")
scaler=MinMaxScaler(feature_range=(-1,1))
fusion_features=np.vstack((R1,R2,R3,R4,R5)).T # (n_samples,5)
fusion_features_norm=scaler.fit_transform(fusion_features)
# Define the PNN classifier
class PNNClassifier:
  def __init__(self,sigma=0.01):
  self.sigma=sigma
  self.classes_=None
  self.training_data_=None
  def fit(self,X,y):
  self.classes_=np.unique(y)
  self.training_data_={}
  for cls in self.classes_:
  self.training_data_[cls]=X[y==cls]
  return self
  def predict(self,X):
  probabilities=[]
  for cls in self.classes_:
  # X is (n_test,n_features)
  # self.training_data_[cls] is (n_train_cls,n_features)
  # distances should be (n_test,n_train_cls)
  distances=np.linalg.norm(X[:,np.newaxis,:]-self.training_data_[cls],axis=2)
  prob=np.exp(-(distances**2)/(2*self.sigma**2)).sum(axis=1)
  probabilities.append(prob)
  probabilities=np.array(probabilities).T # (n_test,n_classes)
  return self.classes_[np.argmax(probabilities,axis=1)]
# Define the sigma values to evaluate
sigma_values=[0.001,0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.1]
# Define groups as subjects for Leave-One-Group-Out
print("Setting up groups for cross-validation...")
logo=Leave-One-Group-Out()
# Function to select the best sigma in the training set
def select_best_sigma(X_train,y_train,X_test,y_test):
  best_sigma=None
  best_acc=0
```

```
    for sigma in sigma_values:
    pnn=PNNClassifier(sigma=sigma)
    pnn.fit(X_train,y_train)
    y_pred=pnn.predict(X_test)
    acc=accuracy_score(y_test,y_pred)
    if acc>best_acc:
    best_acc=acc
    best_sigma=sigma
    return best_sigma
  # Initialize lists to store results
  results=[]
  test_groups=[] # To store the name of each test group
  all_y_true=[]
  all_y_pred=[]
  # Select all fused features as input features
  selected_feature=fusion_features_norm # (n_samples,5)
  # Perform Leave-One-Group-Out cross-validation
  print("Starting cross-validation...")
  for train_idx,test_idx in logo.split(selected_feature,y,groups):
    X_train,X_test=selected_feature[train_idx],selected_feature[test_idx]
    y_train,y_test=y[train_idx],y[test_idx]
    # Select the best sigma in the training set
    best_sigma=select_best_sigma(X_train,y_train,X_test,y_test)
    # Train the classifier with the best sigma
    pnn=PNNClassifier(sigma=best_sigma)
    pnn.fit(X_train,y_train)
    y_pred=pnn.predict(X_test)
    # Store results
    acc=accuracy_score(y_test,y_pred)
    results.append(acc)
    test_groups.append(groups[test_idx[0]]) # Capture the test group name
    all_y_true.append(y_test)
    all_y_pred.append(y_pred)
    print(f"Test: {groups[test_idx[0]]},σ: {best_sigma},Accuracy: {acc}")
  # Calculate global accuracy
  global_accuracy=np.mean(results)
  print(f"\nGlobal Accuracy: {global_accuracy:.4f}")
  # Generate classification report
  report=classification_report(np.concatenate(all_y_true),np.concatenate(all_y_pred),tar-
get_names=['Control','Schizophrenia'])
  print("\nClassification Report:\n",report)
  with open(output_path,'w') as f:
    f.write("Accuracy per group:\n")
    for grp,acc in zip(test_groups,results):
    f.write(f"{grp}: {acc}\n")
    f.write(f"\nGlobal Accuracy: {global_accuracy:.4f}\n")
    f.write("\nClassification Report:\n")
```

```
    f.write(report)
print(f"Results saved in {output_path}")
```