Appendix

Code in python 3.12.7 tested on Manjaro Linux

Schizophrenia: 96% accuracy

```
import os,mne,warnings
import numpy as np
from sklearn.preprocessing import StandardScaler,RobustScaler,MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
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
from sklearn.cluster import KMeans
from concurrent.futures import ProcessPoolExecutor
# Paths ---------------------------------------------------------
base_path="/home/a/Downloads/schizophrenia/eeg/"
output_path="/home/a/Downloads/schizophrenia/eeg/results.txt"
# ---------------------------------------------------------------
# Ignore specific warnings to clean the output
warnings.filterwarnings("ignore",category=UndefinedMetricWarning)
# 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(['F7','Fz'])
    return raw
# 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)
# 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)
# Add KMeans clustering as feature
def add_clustering_feature(features):
    kmeans=KMeans(n_clusters=2,random_state=1) # 2 1
    clusters=kmeans.fit_predict(features)
    return np.hstack((features,clusters.reshape(-1,1)))
# Parallel processing for EEG loading and feature extraction
def parallel_load_and_process_data(subjects,base_path):
    with ProcessPoolExecutor() as executor:
        futures=[]
        for subject_id in subjects:
            file_path=os.path.join(base_path,f"{subject_id}.edf")
            if os.path.exists(file_path):
                futures.append(executor.submit(process_subject_data,file_path,subject_id))
        return [future.result() for future in futures]
def process_subject_data(file_path,subject_id):
    raw_data=load_eeg_data(file_path)
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
        print(f"No adjacency matrices generated for {subject_id}")
        return None
    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}")
        return None
    features=np.hstack((dft_data.mean(axis=0),spectral_data.mean(axis=0)))
    label=1 if 's' in subject_id else 0  # Schizophrenia if subject_id contains 's',Control otherwise
    return features,label
# Model optimization and classification with custom validation scheme
def fine_tuned_classification(subjects,features,labels,output_path):
    # Define pipeline
    pipeline=Pipeline([('imputer',SimpleImputer(strategy='mean')),('scaler',RobustScaler()),
                       ('pca',PCA(n_components=2)),('clf',RandomForestClassifier(random_state=1))]) # 2 1
    param_grid={'clf__n_estimators':[100],'clf__max_depth':[None],'clf__min_samples_split':[14]}
    all_results=[]
    all_subjects=[]
    all_predictions=np.full(len(subjects),-1)  # Initialize array to hold predictions
    all_true_labels=np.full(len(subjects),-1)  # Initialize array for true labels
    # Split subjects into controls and schizophrenia subjects
    control_subjects=[f"h{i:02d}" for i in range(1,15)]
    schizophrenia_subjects=[f"s{i:02d}" for i in range(1,15)]
    # 7. Custom validation loop for balance:13 controls and 13 schizophrenia subjects for training
    for i in range(14): # 14 folds (one for each control and schizophrenia pair)
        # Select a control and schizophrenia subject for validation
        ctrl_val_subject=control_subjects[i]
        sz_val_subject=schizophrenia_subjects[i]
        # Select 13 controls and 13 schizophrenia subjects for training
        ctrl_train_subjects=[control_subjects[j] for j in range(14) if j!=i]
        sz_train_subjects=[schizophrenia_subjects[j] for j in range(14) if j!=i]
        # Combine train subjects (13 controls+13 schizophrenia)
        train_subjects=ctrl_train_subjects+sz_train_subjects
        # Load training data in parallel
        train_data=parallel_load_and_process_data(train_subjects,base_path)
```

```
        train_features=[data[0] for data in train_data if data is not None]
        train_labels=[data[1] for data in train_data if data is not None]
        # Add clustering features
        train_features=add_clustering_feature(np.array(train_features))
        # Train the model on the training data
        train_features=np.array(train_features)
        train_labels=np.array(train_labels)
        grid_search=GridSearchCV(pipeline,param_grid,cv=13,n_jobs=-1,scoring='accuracy')
        grid_search.fit(train_features,train_labels)
        # Load testing data for validation
        test_data=parallel_load_and_process_data([ctrl_val_subject,sz_val_subject],base_path)
        test_features=[data[0] for data in test_data if data is not None]
        test_labels=[data[1] for data in test_data if data is not None]
        # Add clustering features for the test set
        test_features=add_clustering_feature(np.array(test_features))
        # Get predictions for the test set
        y_pred=grid_search.predict(test_features)
        # Store predictions and true labels
        all_predictions[subjects.index(ctrl_val_subject)]=y_pred[0]
        all_predictions[subjects.index(sz_val_subject)]=y_pred[1]
        all_true_labels[subjects.index(ctrl_val_subject)]=test_labels[0]
        all_true_labels[subjects.index(sz_val_subject)]=test_labels[1]
    # Classification report and confusion matrix

report=classification_report(all_true_labels,all_predictions,target_names=['Control','Schizophrenia'],ze
ro_division=0)
    cm=confusion_matrix(all_true_labels,all_predictions)
    print("\nClassification Report:"); print(report); print("Confusion Matrix:"); print(cm)
    # Save results for all subjects
    results=[]
    for subj,pred in zip(subjects,all_predictions):
        results.append(f"{subj}-{['Control','Schizophrenia'][pred]}")
    # Calculate and display accuracy metrics
    correct_count=np.sum(all_predictions==all_true_labels)
    total_accuracy=(correct_count/len(all_true_labels))*100
    h_correct=np.sum((all_true_labels==0)&(all_predictions==0))
    s_correct=np.sum((all_true_labels==1)&(all_predictions==1))
    h_total=np.sum(all_true_labels==0)
    s_total=np.sum(all_true_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}")
# Execute complete pipeline with validation and optimization
if __name__=="__main__":
    all_subjects=[f"h{i:02d}" for i in range(1,15)]+[f"s{i:02d}" for i in range(1,15)]
    fine_tuned_classification(all_subjects,None,None,output_path)
```

Alzheimer's Disease: 97% accuracy

```
import os,mne,warnings
import numpy as np
from sklearn.preprocessing import StandardScaler,RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
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
from sklearn.cluster import KMeans
from concurrent.futures import ProcessPoolExecutor
# Paths ---------------------------------------------------------------
base_path='/home/a/ds004504/'
output_path='/home/a/Downloads/alzheimer/results.txt'
# ---------------------------------------------------------------------
# Ignore specific warnings to clean the output
warnings.filterwarnings('ignore',category=UndefinedMetricWarning)
# Load EEG data and filter by selected channels
def load_eeg_data(file_path):
    raw=mne.io.read_raw_eeglab(file_path,preload=True)
    raw.pick(['O2','Cz'])
    return raw
# Generate adjacency matrices for the selected channels
def get_adjacency_matrix(raw,window_size_sec=300,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)
# 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)
# Add KMeans clustering as feature
def add_clustering_feature(features):
    kmeans=KMeans(n_clusters=2,random_state=1) #2 1
    clusters=kmeans.fit_predict(features)
    return np.hstack((features,clusters.reshape(-1,1)))
# Parallel processing for EEG loading and feature extraction
def parallel_load_and_process_data(subjects,base_path):
    with ProcessPoolExecutor() as executor:
        futures=[]
        for subject_id in subjects:
            file_path=os.path.join(base_path,subject_id,'eeg',f"{subject_id}_task-eyesclosed_eeg.set")
            if os.path.exists(file_path):
                futures.append(executor.submit(process_subject_data,file_path,subject_id))
        return [future.result() for future in futures]
def process_subject_data(file_path,subject_id):
    raw_data=load_eeg_data(file_path)
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
        print(f"No adjacency matrices generated for {subject_id}")
        return None
    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}")
        return None
    features=np.hstack((dft_data.mean(axis=0),spectral_data.mean(axis=0)))
    label=1 if int(subject_id[4:7])<=29 else 0  # Alzheimer if subject_id is 001-029,Control if 037-065
    return features,label
# Model optimization and classification with custom validation scheme
def fine_tuned_classification(subjects,features,labels,output_path):
    # Define pipeline
    pipeline=Pipeline([('imputer',SimpleImputer(strategy='mean')),('scaler',RobustScaler()),
                ('pca',PCA(n_components=2)),('clf',RandomForestClassifier(random_state=33))]) #2 33
    param_grid={'clf__n_estimators':[100],'clf__max_depth':[None],'clf__min_samples_split':[37]}#37
    all_results=[]
    all_subjects=[]
    all_predictions=np.full(58,-1)  # Initialize an array to hold predictions for all 58 subjects (29 AD+29
CN)
    all_true_labels=np.full(58,-1)  # Initialize array for true labels of all 58 subjects
    # Custom validation loop for 29 folds (one for each Alzheimer's subject)
    for i in range(29):  # 29 folds for both Alzheimer and Control
        alz_val_subject=f"sub-{i+1:03d}"  # Alzheimer (AD)
        ctrl_val_subject=f"sub-{i+37:03d}"  # Control (CN)
        # Exclude validation subjects for training
        alz_train_subjects=[f"sub-{j+1:03d}" for j in range(29) if j!=i]  # All except current AD validation
        ctrl_train_subjects=[f"sub-{j+37:03d}" for j in range(29) if j!=i]  # All except current CN validation
        # Create training and testing dataset
        train_subjects=alz_train_subjects+ctrl_train_subjects
        test_subjects=[alz_val_subject,ctrl_val_subject]
        # Load training data in parallel
        train_data=parallel_load_and_process_data(train_subjects,base_path)
        train_features=[data[0] for data in train_data if data is not None]
        train_labels=[data[1] for data in train_data if data is not None]
        # Load testing data in parallel
        test_data=parallel_load_and_process_data(test_subjects,base_path)
```

```
        test_features=[data[0] for data in test_data if data is not None]
        test_labels=[data[1] for data in test_data if data is not None]
        # Add clustering features
        train_features=add_clustering_feature(np.array(train_features))
        test_features=add_clustering_feature(np.array(test_features))
        # Train the model on the training data
        train_features=np.array(train_features)
        train_labels=np.array(train_labels)
        grid_search=GridSearchCV(pipeline,param_grid,cv=28,n_jobs=-1,scoring='accuracy')
        grid_search.fit(train_features,train_labels)
        # Predict for the test set
        test_features=np.array(test_features)
        test_labels=np.array(test_labels)
        y_pred=grid_search.predict(test_features)
        # Store predictions and true labels for all 58 subjects
        # Store predictions and true labels for all 58 subjects
        for subject,pred,true in zip(test_subjects,y_pred,test_labels):
            subject_num=int(subject[4:7])  # Extract the subject number
            if subject_num<=29:  # Alzheimer subjects (001-029)
                idx=subject_num-1  # 0-28 for Alzheimer
            else:  # Control subjects (037-065)
                idx=subject_num-37+29  # 29-57 for Control
            all_predictions[idx]=pred
            all_true_labels[idx]=true
    # Classification report and confusion matrix

report=classification_report(all_true_labels,all_predictions,target_names=['Control','Alzheimer'],zero_
division=0)
    cm=confusion_matrix(all_true_labels,all_predictions)
    print("\nClassification Report:"); print(report); print("Confusion Matrix:"); print(cm)
    # Save results for all subjects (001-029 for Alzheimer,037-065 for Control)
    results=[]
    for subj,pred in zip(range(58),all_predictions):
        results.append(f"sub-{subj+1:03d}-{['Control','Alzheimer'][pred]}")
    # Calculate and display accuracy metrics
    correct_count=np.sum(all_predictions==all_true_labels)
    total_accuracy=(correct_count/len(all_true_labels))*100
    h_correct=np.sum((all_true_labels==0) & (all_predictions==0))
    a_correct=np.sum((all_true_labels==1) & (all_predictions==1))
    h_total=np.sum(all_true_labels==0)
    a_total=np.sum(all_true_labels==1)
    h_accuracy=(h_correct/h_total)*100 if h_total>0 else 0
    a_accuracy=(a_correct/a_total)*100 if a_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 Alzheimer's (a):{a_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}")
# 7. Execute complete pipeline with validation and optimization
if __name__=="__main__":
    fine_tuned_classification(None,None,None,output_path)
```

Frontotemporal dementia: 96% accuracy

```
import os,mne,warnings
import numpy as np
from sklearn.preprocessing import StandardScaler,RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
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
from sklearn.cluster import KMeans
from concurrent.futures import ProcessPoolExecutor
# Paths ---------------------------------------------------------
base_path='/home/a/ds004504/'
output_path='/home/a/Downloads/frontotemporaldementia/results.txt'
# ---------------------------------------------------------------
# Ignore specific warnings to clean the output
warnings.filterwarnings('ignore',category=UndefinedMetricWarning)
# Load EEG data and filter by selected channels
def load_eeg_data(file_path):
    raw=mne.io.read_raw_eeglab(file_path,preload=True)
    raw.pick(['F7','F3'])
    return raw
# Generate adjacency matrices for the selected channels
def get_adjacency_matrix(raw,window_size_sec=300,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)
# 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)
```

```
# Add KMeans clustering as feature
def add_clustering_feature(features):
    kmeans=KMeans(n_clusters=2,random_state=1) #2 1
    clusters=kmeans.fit_predict(features)
    return np.hstack((features,clusters.reshape(-1,1)))
# Parallel processing for EEG loading and feature extraction
def parallel_load_and_process_data(subjects,base_path):
    with ProcessPoolExecutor() as executor:
        futures=[]
        for subject_id in subjects:
            file_path=os.path.join(base_path,subject_id,'eeg',f"{subject_id}_task-eyesclosed_eeg.set")
            if os.path.exists(file_path):
                futures.append(executor.submit(process_subject_data,file_path,subject_id))
        return [future.result() for future in futures]
def process_subject_data(file_path,subject_id):
    raw_data=load_eeg_data(file_path)
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
        print(f"No adjacency matrices generated for {subject_id}")
        return None
    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}")
        return None
    features=np.hstack((dft_data.mean(axis=0),spectral_data.mean(axis=0)))
    # Label for FTD (0 for control,1 for FTD)
    label=1 if 66<=int(subject_id[4:7])<=88 else 0  # FTD subjects are from 066 to 088
    return features,label
# Model optimization and classification with custom validation scheme
def fine_tuned_classification(subjects,features,labels,output_path):
    # Define pipeline
    pipeline=Pipeline([('imputer',SimpleImputer(strategy='mean')),('scaler',RobustScaler()),
        ('pca',PCA(n_components=2)),('clf',RandomForestClassifier(random_state=2))]) #2 2
    param_grid={'clf__n_estimators':[100],'clf__max_depth':[None],'clf__min_samples_split':[21]}#21
    all_results=[]
    all_subjects=[]
    all_predictions=[]
    all_true_labels=[]
    # Custom validation loop for 23 folds (1 per subject)
    for i in range(23): # 23 folds for FTD (066-088)
        ftd_val_subject=f"sub-{i+66:03d}"  # FTD subjects 066-088
        ctrl_val_subject=f"sub-{i+37:03d}"  # Control subjects 037-059
        # Exclude validation subjects for training
        ftd_train_subjects=[f"sub-{j+66:03d}" for j in range(23) if j!=i]  # All except current FTD validation
        ctrl_train_subjects=[f"sub-{j+37:03d}" for j in range(23) if j!=i]  # All except current Control
validation
        # Create training and testing dataset
        train_subjects=ftd_train_subjects + ctrl_train_subjects
        test_subjects=[ftd_val_subject,ctrl_val_subject]
        # Load training data in parallel
        train_data=parallel_load_and_process_data(train_subjects,base_path)
        train_features=[data[0] for data in train_data if data is not None]
        train_labels=[data[1] for data in train_data if data is not None]
        # Load testing data in parallel
        test_data=parallel_load_and_process_data(test_subjects,base_path)
        test_features=[data[0] for data in test_data if data is not None]
```

```
        test_labels=[data[1] for data in test_data if data is not None]
        # Add clustering features
        train_features=add_clustering_feature(np.array(train_features))
        test_features=add_clustering_feature(np.array(test_features))
        # Train the model on the training data
        grid_search=GridSearchCV(pipeline,param_grid,cv=22,n_jobs=-1,scoring='accuracy')
        grid_search.fit(train_features,train_labels)
        # Predict for the test set
        y_pred=grid_search.predict(test_features)
        # Store results
        all_subjects.append(test_subjects)
        all_predictions.append(y_pred)
        all_true_labels.append(test_labels)
    # Classification report and confusion matrix
    all_true_labels=np.concatenate(all_true_labels)
    all_predictions=np.concatenate(all_predictions)

report=classification_report(all_true_labels,all_predictions,target_names=['Control','FTD'],zero_divisio
n=0)
    cm=confusion_matrix(all_true_labels, all_predictions)
    print("\nClassification Report:");print(report);print("Confusion Matrix:");print(cm)
    all_subjects_flat=[s for fold_subjects in all_subjects for s in fold_subjects]
    results=[]
    for subj, pred in zip(all_subjects_flat,all_predictions):
        results.append(f"{subj}-{['Control','FTD'][pred]}")
    # Calculate and display accuracy metrics
    correct_count=np.sum(all_predictions==all_true_labels)
    total_accuracy=(correct_count/len(all_true_labels))*100
    h_correct=np.sum((all_true_labels==0)&(all_predictions==0))
    a_correct=np.sum((all_true_labels==1) & (all_predictions==1))
    h_total=np.sum(all_true_labels==0)
    a_total=np.sum(all_true_labels==1)
    h_accuracy=(h_correct/h_total)*100 if h_total>0 else 0
    a_accuracy=(a_correct/a_total)*100 if a_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 FTD (a): {a_accuracy:.2f}%")
    with open(output_path, "w") as f:
        f.write("\n".join(results))
    print(f"Results saved in {output_path}")
# Execute complete pipeline with validation and optimization
if __name__=="__main__":
    fine_tuned_classification(None,None,None,output_path)
```

Parkinson's Disease: 96% accuracy

```
import os,mne,warnings
import numpy as np
from sklearn.preprocessing import StandardScaler,RobustScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
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
from sklearn.cluster import KMeans
from concurrent.futures import ProcessPoolExecutor
# Paths ---------------------------------------------------------
base_path='/home/a/Downloads/parkinson/'
output_path='/home/a/Downloads/parkinson/results.txt'
# ---------------------------------------------------------------
# Ignore specific warnings to clean the output
warnings.filterwarnings('ignore',category=UndefinedMetricWarning)
# Define subject lists for control and Parkinson
control_subjects=['1021','1041','1061','1081','1101','1111','1191','1201','1211','1231','1291','1351','1381'
,'1411']  # Label 0
parkinson_subjects=['1001','1021','1031','1061','1091','1101','1151','1201','1251','1261','1311','1571','1
661','1681']  # Label 1
def load_eeg_data(file_path):
    raw=mne.io.read_raw_brainvision(file_path,preload=True)  # This reads the .vhdr file
    raw.pick(['Fz','T8'])
    return raw
# Generate adjacency matrices for the selected channels
def get_adjacency_matrix(raw,window_size_sec=120,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)
# 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)
# Add KMeans clustering as feature
def add_clustering_feature(features):
    kmeans=KMeans(n_clusters=2,random_state=1) # 2 1
    clusters=kmeans.fit_predict(features)
    return np.hstack((features,clusters.reshape(-1,1)))
# Parallel processing for EEG loading and feature extraction
def process_subject_data(subject_id,label):
    file_name=f"{'Control' if label==0 else 'PD'}{subject_id}.vhdr" # Use .vhdr files
    file_path=os.path.join(base_path,file_name)
    if not os.path.exists(file_path):
        print(f"The file {file_name} does not exist.")
        return None
    try:
        raw_data=load_eeg_data(file_path)
    except Exception as e:
        print(f"Error loading {file_name}:{e}")
        return None
    matrices=get_adjacency_matrix(raw_data)
    if matrices.size==0:
        print(f"No adjacency matrices generated for {file_name}")
        return None
    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 {file_name}")
        return None
    # Extract mean features from DFT and spectral eigenvalues
    features=np.hstack((dft_data.mean(axis=0),spectral_data.mean(axis=0)))
    return (features,label)
def parallel_load_and_process(subject_list,label):
    results=[]
    with ProcessPoolExecutor() as executor:
        futures={executor.submit(process_subject_data,subj,label):subj for subj in subject_list}
        for future in futures:
            res=future.result()
            if res is not None:
                results.append(res)
    return results
# 6. Training and validation with 14 folds,1 subject per fold in each group
def fine_tuned_classification():
    """
    Divide the 14 control subjects and 14 Parkinson's subjects into 14 folds.
    In each fold,reserve one subject from each group for testing and train on the other 13.
    Then report the overall performance.
    """
    pipeline=Pipeline([('imputer',SimpleImputer(strategy='mean')),('scaler',RobustScaler()),
                       ('pca',PCA(n_components=2)),('clf',RandomForestClassifier(random_state=2))]) #2 2
    param_grid={'clf__n_estimators':[100],'clf__max_depth':[None],'clf__min_samples_split':[14]} #14
    all_predictions=np.full(28,-1)
    all_true_labels=np.full(28,-1)
    results=[]  # Define the list to store results
    # Perform 14 folds (1 subject from each list per fold)
    for i in range(14):
```

```
        # Test subject in control group
        test_control_subj=control_subjects[i]
        # Test subject in Parkinson group
        test_parkinson_subj=parkinson_subjects[i]
        # Training=rest of the subjects
        train_control=[s for j,s in enumerate(control_subjects) if j!=i]
        train_parkinson=[s for j,s in enumerate(parkinson_subjects) if j!=i]
        # Load and process training data
        train_data_control=parallel_load_and_process(train_control,label=0)
        train_data_parkinson=parallel_load_and_process(train_parkinson,label=1)
        train_data=train_data_control+train_data_parkinson
        train_features=[d[0] for d in train_data]
        train_labels=[d[1] for d in train_data]
        # Load and process test data
        test_data_control=parallel_load_and_process([test_control_subj],label=0)
        test_data_parkinson=parallel_load_and_process([test_parkinson_subj],label=1)
        test_data=test_data_control+test_data_parkinson
        test_features=[d[0] for d in test_data]
        test_labels=[d[1] for d in test_data]
        # Add clustering feature
        train_features=add_clustering_feature(np.array(train_features))
        test_features=add_clustering_feature(np.array(test_features))
        # GridSearchCV with internal validation (cv=?)
        grid_search=GridSearchCV(pipeline,param_grid,cv=13,n_jobs=-1,scoring='accuracy')
        grid_search.fit(train_features,train_labels)
        # Prediction
        y_pred=grid_search.predict(test_features)
        control_index=i
        parkinson_index=i+14
        all_predictions[control_index]=y_pred[0]
        all_true_labels[control_index]=test_labels[0]
        all_predictions[parkinson_index]=y_pred[1]
        all_true_labels[parkinson_index]=test_labels[1]
        # Store individual subject results
        results.append(f"Subject {test_control_subj} (Control)-
Predicted:{y_pred[0]},True:{test_labels[0]}")
        results.append(f"Subject {test_parkinson_subj} (Parkinson)-
Predicted:{y_pred[1]},True:{test_labels[1]}")
    # Global Report

report=classification_report(all_true_labels,all_predictions,target_names=['control','parkinson'],zero_di
vision=0)
    cm=confusion_matrix(all_true_labels,all_predictions)
    print("\nClassification Report:")
    print(report)
    print("Confusion Matrix:")
    print(cm)
    # Save to file
    results.append("Classification Report:\n"+report)
    results.append("\nConfusion Matrix:\n"+str(cm))
    # Save results to file
    with open(output_path,"w") as f:
        f.write("\n\n".join(results))  # Ensure the results are written correctly
    print(f"Results saved in {output_path}")
# 7. Execute the complete pipeline
if __name__=="__main__":
```

fine_tuned_classification()