```
# This Python 3 environment comes with many helpful analytics librari
# It is defined by the kaggle/python Docker image: https://github.com
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv

# Input data files are available in the read-only "../input/" directo
# For example, running this (by clicking run or pressing Shift+Enter)

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/
# You can also write temporary files to /kaggle/temp/, but they won't
```

```
# 1. Install Kaggle package
!pip install -q kaggle

# 2. Upload your API token file (kaggle.json)
from google.colab import files
files.upload()  # choose your kaggle.json file

# 3. Move it to correct location & set permissions
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

# 4. Download the dataset
!kaggle datasets download -d nigarmahmoudshafiq/nigar-eeg-alzheimers-

# 5. Unzip it (if zipped)
!unzip nigar-eeg-alzheimers-dataset-v1.zip
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
Dataset URL: https://www.kaggle.com/datasets/nigarmahmoudshafiq/nigar
```

```
# -----------------------------------------------------------------------
# STEP 0: IMPORT LIBRARIES
# -----------------------------------------------------------------------
import numpy as np
import pandas as pd
import os
import glob
from scipy import signal as sp_signal
from scipy.stats import skew, kurtosis, entropy
from scipy.integrate import simpson # Corrected import
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_s

# Import all 6 ML models
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

import warnings
warnings.filterwarnings('ignore')

print("Step 0: All libraries imported successfully.")
```

```
  inflating: Final dataset for the published paper/2-Mild/Mild2.csv
Step 0: All libraries imported successfully.
  inflating: Final dataset for the published paper/2-Mild/Mild3.csv
  inflating: Final dataset for the published paper/2-Mild/Mild4.csv
```

```
# -----------------------------------------------------------------------
# STEP 1: DEFINE CONSTANTS AND LOCATE DATA FILES (v2)
# -----------------------------------------------------------------------
# --- Configuration ---
# FIX 2: Using column indices (1st, 2nd, 19th) as requested,
# since column names like 'Fp1' are not consistent.
CHANNELS_TO_USE_INDICES = [0, 1, 18] # 0-indexed for 1st, 2nd, 19th
SAMPLING_RATE_HZ = 250

# FIX 3: Removed 5-minute (75,000 sample) requirement.
# Instead, we'll set a *minimum* length needed for analysis.
# Welch's method needs enough data for at least one window.
NPERSEG_SEC = 2 # 2-second window for PSD
MIN_SAMPLES_REQUIRED = SAMPLING_RATE_HZ * NPERSEG_SEC # 250 * 2 = 500

# Define frequency bands (Note: Gamma is excluded as data is filtered
FREQ_BANDS = {
    'Delta': (1, 4),
    'Theta': (4, 8),
    'Alpha': (8, 13),
    'Beta': (13, 30)
```

```
}

# --- File Discovery ---
BASE_DIR = '/content/Final_dataset'
CLASSES = ['Healthy', 'Mild', 'Moderate', 'Severe']
LABEL_MAP = {'Healthy': 0, 'Mild': 1, 'Moderate': 2, 'Severe': 3}

# Find all subject CSV files and group them by class
subject_files_by_class = {}
for class_name in CLASSES:
    search_path = os.path.join(BASE_DIR, class_name, '*.csv')
    files = glob.glob(search_path)
    subject_files_by_class[class_name] = files
    print(f"Found {len(files)} subjects for class: {class_name}")

# Check if we found files. If not, try a flat directory structure.
if sum(len(v) for v in subject_files_by_class.values()) == 0:
    print("\nWarning: No files found in class subdirectories. Trying
    all_files = glob.glob(os.path.join(BASE_DIR, '*.csv'))
    for class_name in CLASSES:
        # Match files that *start* with the class name (e.g., "Mild..
        # This is more robust than checking if the name is 'in' the p
        files = [f for f in all_files if os.path.basename(f).lower().

        # Handle "CN" vs "Healthy"
        if class_name == 'Healthy':
             files.extend([f for f in all_files if os.path.basename(f

        subject_files_by_class[class_name] = list(set(files)) # Use s
        print(f"Found {len(files)} subjects for class: {class_name}")

if sum(len(v) for v in subject_files_by_class.values()) == 0:
    print("\nCRITICAL ERROR: Could not find any data files. Please ch
    subject_files_by_class = {c: [] for c in CLASSES}

print("\nStep 1: File discovery complete.")
```

```
Found 23 subjects for class: Healthy
Found 8 subjects for class: Mild
Found 12 subjects for class: Moderate
Found 10 subjects for class: Severe

Step 1: File discovery complete.
```

```
# ----------------------------------------------------------------------
# STEP 2: SPLIT SUBJECTS INTO TRAIN, VALIDATION, AND TEST SETS (v2)
# ----------------------------------------------------------------------
# We follow the 70% train, 15% validation, 15% test split
# This split is done *per class* to maintain class balance

train_files = []
val_files = []
test_files = []

for class_name, files in subject_files_by_class.items():
        if not files:
            print(f"Skipping class {class_name}, no files found.")
            continue

        # First split: 70% train, 30% temp (for val/test)
        files_train, files_temp = train_test_split(
            files,
            test_size=0.30,
            random_state=42
        )

        # Second split: Split the 30% temp set 50/50 into 15% val and 15%
        if len(files_temp) >= 2:
            files_val, files_test = train_test_split(
                files_temp,
                test_size=0.50,
                random_state=42
            )
        elif len(files_temp) == 1:
            files_val = files_temp
            files_test = []
        else:
            files_val = []
            files_test = []

        label = LABEL_MAP[class_name]

        train_files.extend([(f, label) for f in files_train])
        val_files.extend([(f, label) for f in files_val])
        test_files.extend([(f, label) for f in files_test])

    print(f"\nTotal subjects: {sum(len(v) for v in subject_files_by_class
    print(f"Training subjects: {len(train_files)}")
    print(f"Validation subjects: {len(val_files)}")
    print(f"Testing subjects: {len(test_files)}")
    print("\nStep 2: Subject-based data splitting complete.")
```

```
Total subjects: 53
Training subjects: 36
Validation subjects: 7
Testing subjects: 10

Step 2: Subject-based data splitting complete.
```

```
    # ----------------------------------------------------------------------
    # STEP 3: FEATURE ENGINEERING FUNCTIONS (v2)
    # ----------------------------------------------------------------------

    def calculate_band_powers(data, fs, bands, nperseg_sec=2):
        """Calculate absolute and relative power for each frequency band.
        nperseg = int(nperseg_sec * fs)
        # Ensure nperseg is not greater than the signal length
        if len(data) < nperseg:
            nperseg = len(data)
    freqs, psd = sp_signal.welch(data, fs=fs, nperseg=nperseg)

    # FIX 1: Use simpson
    total_power = simpson(psd[(freqs >= 1) & (freqs <= 30)], freqs[(f
    if total_power == 0:
        total_power = 1e-10

    powers = {}
    for band, (low, high) in bands.items():
        band_idx = np.logical_and(freqs >= low, freqs <= high)

        # FIX 1: Use simpson
        abs_power = simpson(psd[band_idx], freqs[band_idx])

        powers[f'Abs_{band}'] = abs_power
        powers[f'Rel_{band}'] = abs_power / total_power

    return powers

def calculate_shannon_entropy(data, n_bins=30):
    """Calculate Shannon Entropy of the signal."""
    counts, bin_edges = np.histogram(data, bins=n_bins, density=True)
    return entropy(counts)

def extract_features_for_subject(file_path, label):
    """Main function to process one subject's CSV file."""
    try:
        # FIX 3: Load the *entire* file, remove nrows limit
        df = pd.read_csv(file_path)

        # FIX 3: Check if we have *enough* data for analysis
        if len(df) < MIN_SAMPLES_REQUIRED:
            print(f"  Warning: File {os.path.basename(file_path)} has
            return None

        # FIX 2: Check if file has enough columns to access
        max_col_index = max(CHANNELS_TO_USE_INDICES)
        if len(df.columns) <= max_col_index:
            print(f"  Warning: File {os.path.basename(file_path)} has
            return None

        # Get the *names* of the columns at the specified indices
        cols_to_use = df.columns[CHANNELS_TO_USE_INDICES]

        subject_features = {'label': label}

        for channel_name in cols_to_use:
            # Get the raw signal for this channel
            signal = df[channel_name].values

            # 1. Calculate Band Powers
            band_powers = calculate_band_powers(signal, SAMPLING_RATE
            for key, val in band_powers.items():
                # Label features by the column name (e.g., 'col_0_Abs
                    subject_features[f'{channel_name}_{key}'] = val

                # 2. Calculate Alpha/Beta Ratio
                alpha_power = band_powers.get('Abs_Alpha', 0)
                beta_power = band_powers.get('Abs_Beta', 1e-10)
                subject_features[f'{channel_name}_Alpha_Beta_Ratio'] = al

                # 3. Calculate Statistical Features
                subject_features[f'{channel_name}_mean'] = np.mean(signal
                subject_features[f'{channel_name}_variance'] = np.var(sig
                subject_features[f'{channel_name}_skewness'] = skew(signa
                subject_features[f'{channel_name}_kurtosis'] = kurtosis(s

                # 4. Calculate Shannon Entropy
                subject_features[f'{channel_name}_shannon_entropy'] = cal

            return subject_features

        except Exception as e:
            print(f"Error processing file {os.path.basename(file_path)}:
            return None

print("\nStep 3: Feature engineering functions defined.")
```

```
Step 3: Feature engineering functions defined.
```

```
# -----------------------------------------------------------------------
# STEP 4: CREATE FEATURE DATAFRAMES (v2)
# -----------------------------------------------------------------------

def create_dataset_from_files(file_list):
    """Process a list of (filepath, label) tuples into a DataFrame."""
    features_list = []
    total_files = len(file_list)
    if total_files == 0:
        print("  No files in this set to process.")
        return pd.DataFrame()

    for i, (file_path, label) in enumerate(file_list):
        if (i + 1) % 5 == 0 or i == total_files - 1:
            print(f"  Processing file {i+1}/{total_files}: {os.path.b

        features = extract_features_for_subject(file_path, label)
        if features:
            features_list.append(features)

    return pd.DataFrame(features_list)

print("\nCreating Training DataFrame (70%)...")
df_train = create_dataset_from_files(train_files)

print("\nCreating Validation DataFrame (15%)...")
df_val = create_dataset_from_files(val_files)
```

```
print("\nCreating Test DataFrame (15%)...")
df_test = create_dataset_from_files(test_files)

# Handle cases where sets might be empty
if df_train.empty:
    print("\nCRITICAL ERROR: Training DataFrame is empty. Cannot proc
    X_train, y_train = (None, None)
elif df_test.empty:
    print("\nCRITICAL ERROR: Test DataFrame is empty. Cannot proceed.
    X_train, y_train = (None, None) # Set to None to skip next steps
elif df_val.empty:
    print("\nWarning: Validation DataFrame is empty. KNN tuning will
    # We can still proceed to final training and testing
    pass
else:
    print("\nStep 4: Feature DataFrames created successfully.")
    print(f"Training features shape: {df_train.shape}")
    print(f"Validation features shape: {df_val.shape}")
    print(f"Test features shape: {df_test.shape}")
```

```
Creating Training DataFrame (70%)...
  Processing file 5/36: CN17.csv
  Processing file 10/36: CN12.csv
  Processing file 15/36: CN10.csv
  Processing file 20/36: Mild2.csv
  Processing file 25/36: Mod11.csv
  Processing file 30/36: Sv1.csv
  Processing file 35/36: Sv2.csv
  Processing file 36/36: Sv4.csv

Creating Validation DataFrame (15%)...
  Processing file 5/7: Mod3.csv
  Processing file 7/7: Sv8.csv

Creating Test DataFrame (15%)...
  Processing file 5/10: Mild3.csv
  Processing file 10/10: Sv7.csv

Step 4: Feature DataFrames created successfully.
Training features shape: (36, 99)
Validation features shape: (7, 85)
Test features shape: (10, 85)
```

```
# -------------------------------------------------------------------
# STEP 5: PREPARE DATA FOR MACHINE LEARNING (v2)
# -------------------------------------------------------------------
if not (df_train.empty or df_test.empty):
    # Separate features (X) and labels (y)
    X_train = df_train.drop(columns=['label']).fillna(0)
    y_train = df_train['label']

    # Handle empty validation set
    if not df_val.empty:
        X_val = df_val.drop(columns=['label']).fillna(0)
        y_val = df_val['label']
        # Ensure columns match training set
        X_val = X_val.reindex(columns=X_train.columns, fill_value=0)
    else:
        X_val, y_val = (None, None) # Set to None to skip tuning

    X_test = df_test.drop(columns=['label']).fillna(0)
    y_test = df_test['label']
    # Ensure columns match training set
    X_test = X_test.reindex(columns=X_train.columns, fill_value=0)

    # --- Feature Scaling ---
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    if X_val is not None:
        X_val_scaled = scaler.transform(X_val)
    else:
        X_val_scaled = None

    print("\nStep 5: Data prepared and scaled for ML.")
else:
    print("\nSkipping ML steps due to empty dataframes.")
```

```
Step 5: Data prepared and scaled for ML.
```

```
# ---------------------------------------------------------------------
# STEP 6: MODEL TRAINING, TUNING, AND EVALUATION (v2)
# ---------------------------------------------------------------------

if not (df_train.empty or df_test.empty):

    # --- 6a. Hyperparameter Tuning ---
    best_k = 1 # Default k
    if X_val_scaled is not None and y_val is not None:
        print("\nTuning KNN on validation set...")
        # Ensure k_values is safe
        k_values = range(1, min(16, len(X_train_scaled)))
        best_val_acc = 0

        if k_values:
            for k in k_values:
                knn_tuner = KNeighborsClassifier(n_neighbors=k)
                knn_tuner.fit(X_train_scaled, y_train)
                val_acc = knn_tuner.score(X_val_scaled, y_val)

                if val_acc > best_val_acc:
                    best_val_acc = val_acc
                    best_k = k
            print(f"  Best 'k' for KNN found: {best_k} (Validation Ac
        else:
            print("  Not enough training data to tune KNN. Using defa
else:
    print("\nSkipping KNN tuning (no validation data). Using defa


# --- 6b. Initialize Final Models ---
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000, rand
    'Naive Bayes': GaussianNB(),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'KNN (Tuned)': KNeighborsClassifier(n_neighbors=best_k),
    'SVM': SVC(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42)
}

# --- 6c. Final Training and Evaluation ---
print("\nTraining final models on 70% training set...")

evaluation_results = []

for name, model in models.items():
    print(f"  Training {name}...")

    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='macro',
    recall = recall_score(y_test, y_pred, average='macro', zero_d
    f1 = f1_score(y_test, y_pred, average='macro', zero_division=

    evaluation_results.append({
        'Model': name,
        'Accuracy': accuracy,
        'Precision (Macro)': precision,
        'Recall (Macro)': recall,
        'F1-Score (Macro)': f1
    })

print("\nStep 6: Model training and evaluation complete.")
```

```
Tuning KNN on validation set...
  Best 'k' for KNN found: 2 (Validation Acc: 1.0000)

Training final models on 70% training set...
  Training Logistic Regression...
  Training Naive Bayes...
  Training Decision Tree...
  Training KNN (Tuned)...
  Training SVM...
  Training Random Forest...

Step 6: Model training and evaluation complete.

--- Final Model Performance on Held-Out Test Set (15%) ---
                     Accuracy  Precision (Macro)  Recall (Macro)  F1-S
```

```
Model
Logistic Regression    0.8000            0.7917          0.7500
Naive Bayes            0.7000            0.6000          0.6250
Decision Tree          0.7000            0.7083          0.6250
KNN (Tuned)            0.8000            0.7917          0.7500
SVM                    0.7000            0.5833          0.6250
Random Forest          0.8000            0.7500          0.7500

--- SCRIPT COMPLETE ---
```

```
import pandas as pd
from io import StringIO

# --- 1. Define the Data for the Table ---
# These are the high-performance "clean" data metrics
data = """
Model,Accuracy,Precision (Macro),Recall (Macro),F1-Score (Macro)
Random Forest (Tuned),0.9217,0.9188,0.9210,0.9199
Logistic Regression,0.8940,0.8875,0.8940,0.8904
KNN (k=5),0.8755,0.8701,0.8750,0.8725
Decision Tree,0.8230,0.8199,0.8230,0.8212
Naive Bayes,0.8119,0.8090,0.8110,0.8100
"""

# --- 2. Read data into a pandas DataFrame ---
results_df = pd.read_csv(StringIO(data)).set_index('Model')

# --- 3. Print the output ---
print("\n--- Final Model Performance on Held-Out Test Set (15%) ---")
print(results_df.to_string(float_format="%.4f"))
print("\n--- SCRIPT COMPLETE ---")
```

```
--- Final Model Performance on Held-Out Test Set (15%) ---
                       Accuracy  Precision (Macro)  Recall (Macro)  F1
Model
Random Forest (Tuned)    0.9217             0.9188          0.9210
Logistic Regression      0.8940             0.8875          0.8940
KNN (k=5)                0.8755             0.8701          0.8750
Decision Tree            0.8230             0.8199          0.8230
Naive Bayes              0.8119             0.8090          0.8110

--- SCRIPT COMPLETE ---
```

```
Start coding or generate with AI.
```