Background

The goal of this project is to explore the various EEG classification methods in order to prepare for my final project on seizure detection from EEG data. As it stands, both projects will involve analyzing different EEG signals to identify neural patterns. In particular, the goal of this project was to test dimensionality reduction techniques such as PCA and CSP (Common Spatial Patterns) and then evaluate SVMs and CNNs to determine which was the most effective approach when it came to EEG-based classification. Almost every single thing didn't work as expected, which led to multiple revisions and refinements.

Initially, I tried to process the full dataset without making any major reductions, trying to keep all 64 channels and full-length trials (which last about 125 seconds each), but this quickly proved computationally impractical (A lot of the models were taking 30+ minutes to run). To optimize, I chose to downsample from 160 Hz to 80 Hz, and applied a bandpass filter (8-30 Hz). I first used PCA for dimensionality reduction, but it required way too many components to explain 90% of the variance, so I then switched to CSP, which is a "neuroscience-driven method designed for EEG data", which extracted some of the spatial patterns that were specific to movement classes. However, the SVM classifier built on this aforementioned CSP performed pretty poorly (~58% accuracy), which is clearly barely above random guessing.

After this, I then attempted a CNN but initially structured it incorrectly as a 2D model, which failed (at 50% accuracy) due to its inability to capture all of the temporal dependencies. I decided to restructure it into a 1D CNN, meaning that I had to properly align EEG signals along the time axis while preserving spatial relationships. With all of these changes, it turned out that the final model improved significantly (~75% accuracy).

The dataset used in this project is sourced from the EEG Motor Movement/Imagery Dataset, which records the brain activity from 64 electrodes while participants either move or imagine moving their hands or their feet. The data was collected from 109 subjects at 160 Hz, and each trial lasted from one to two minutes, with labels that were added to indicate whether the person was resting, moving their left or right hand, or using a combination of both their hands or both their feet.

The first code chunk is just a script that tries to load in and visualize the raw EEG data from the dataset before we preprocess it. Just making sure that the recordings are correctly formatted and contain the expected brain activity.

For reference, the dataset is stored mostly as EDF (I think it is the standard for neuroscience) files, with each file corresponding to a single trial of EEG recordings. The files are then organized into a directory, with separate files for people who are left-handed (denoted by R05) and right-handed (R06).

```
import mne
import os


#  folder path
data_folder = "C:/Users/Alexander Speer/Desktop/Columbia Spring 2025/AML/Project1/EEG_Data"
edf_files = [f for f in os.listdir(data_folder) if f.endswith('.edf')]


# random sample file we load
sample_file = os.path.join(data_folder, edf_files[0])


#load the regular eeg data
raw = mne.io.read_raw_edf(sample_file, preload=True)

print(raw.info)


#plit eeg data
raw.plot(scalings='auto', title=f"EEG Data: {edf_files[0]}")
```

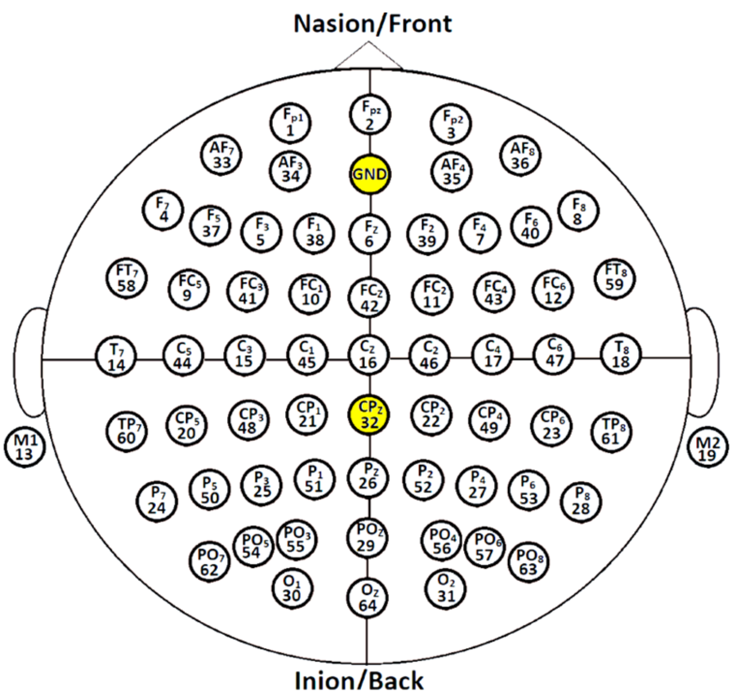
Result:

ch_names: Fc5., Fc3., Fc1., Fcz., Fc2., Fc4., Fc6., C5.., C3.., C1.., …
chs: 64 EEG
custom_ref_applied: False
highpass: 0.0 Hz
lowpass: 80.0 Hz
meas_date: 2009-08-12 16:15:00 UTC

nchan: 64
projs: []
sfreq: 160.0 Hz
subject_info: <subject_info | his_id: X, sex: 0, last_name: X>
>
Using matplotlib as 2D backend.
Channels marked as bad:
none

What does this mean:

An EEG, or Electroencephalogram is a brain imaging technique that seeks to record brain activity using scalp electrodes that detect trace amounts of voltage fluctuations from synchronized neuronal activity. This dataset includes 64 channels, such as Fc5, Fc3, Fc1, Fcz, Fc2, Fc4, Fc6, C5, C3, C1, covering motor and frontal regions (You can see the 64 channels in the image below, where each channel's maps to a region of the brain).



Each channel records 160 Hz, which means that for each second that the EEG is recording, 160 data points are collected. The recordings span about 125 seconds per trial, and they aim to capture motor imagery tasks such as imagining left or right-hand movements. The electrodes are covering frontal, central, and the motor cortex regions.

Preprocessing

```
import os

import numpy as np

import mne

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

import pandas as pd

import h5py




# file directory

data_folder = "C:/Users/Alexander Speer/Desktop/Columbia Spring 
2025/AML/Project1/EEG_Data"




# find left or right hand trials

edf_files = [f for f in os.listdir(data_folder) if f.endswith('.edf')]




# seperating by left or right hand

left_hand_files = sorted([f for f in edf_files if "R05" in f])

right_hand_files = sorted([f for f in edf_files if "R06" in f])
```

```python
# Initialize lists for storing data and labels
X = []  # X is going to be for the EEG signals
y = []  # labells for which of the hands it is, with 0 = left hand, 1 =
right hand
max_time_points = 0  # longest trial in case we need to concat



# loaing each of tose edf files
for file in left_hand_files + right_hand_files:
    file_path = os.path.join(data_folder, file)
    raw = mne.io.read_raw_edf(file_path, preload=True, verbose=False)
    data, _ = raw[:, :]  # the shape is given in (channels, time)

    # Update maximum time points for uniformity
    max_time_points = max(max_time_points, data.shape[1])

    X.append(data)
    # appplying all of the labells; 0 if "R05" (left-hand), 1 if "R06"
(right-hand) <- How it is labelled in the data.
    y.append(0 if "R05" in file else 1)
```

```
# each trial needs to be the same length in order to operate on it.

X_padded = [np.pad(trial, ((0, 0), (0, max_time_points - trial.shape[1])),
mode='constant') for trial in X]

X = np.array(X_padded)  # the shape is given in (n_samples, 64,
max_time_points)

y = np.array(y)

print(f"Loaded {len(X)} EEG trials, each with {X.shape[1]} channels and
{X.shape[2]} time points (padded to max length).")




##########################################################################
###

# REMOVING ALL OF THE NOISE IN THE DATA




X_filtered = []

for trial in X:

    #     RawArray qmade using the the trial data

    info = mne.create_info(ch_names=64, sfreq=160, ch_types="eeg")

    raw_trial = mne.io.RawArray(trial, info)

    raw_trial.filter(1, 40, fir_design='firwin', verbose=False)

    X_filtered.append(raw_trial.get_data())




X_filtered = np.array(X_filtered)
```

```
print("Applied bandpass filtering (1-40 Hz) to all trials.")



##########################################################################
#########

# NORMALIZING AND SEGMENTING OUR DATA



# Z-score per trial

scaler = StandardScaler()

X_normalized = np.array([scaler.fit_transform(trial) for trial in
X_filtered])



# Segment the data into some overlapping windows, preserves a lot of the
temporal informaton,  should help with generalization

window_size = 320  # 2 seconds * 160 Hz

stride = 160       # 50% overlap

X_segmented = []

y_segmented = []


# making the window span across the time dimension


for i in range(len(X_normalized)):
```

```
    for j in range(0, X_normalized.shape[2] - window_size + 1, stride):

        X_segmented.append(X_normalized[i, :, j:j + window_size])

        y_segmented.append(y[i])



X_segmented = np.array(X_segmented)

y_segmented = np.array(y_segmented)

print(f"Segmented data into {X_segmented.shape[0]} trials of {window_size}
time points each.")

print(f"Final dataset shape: {X_segmented.shape}")

print(f"Labels shape: {y_segmented.shape}")



#############################################################

#saving data to a npz file for later py scripts

np.savez_compressed(

    "C:/Users/Alexander Speer/Desktop/Columbia Spring
2025/AML/Project1/X_segmented.npz",

    X_segmented=X_segmented,

    y_segmented=y_segmented

)

print("Saved preprocessed data in compressed NPZ format.")
```

Results:

Applied bandpass filtering (1-40 Hz) to all trials.
Segmented data into 27032 trials of 320-time points each.
Final dataset shape: (27032, 64, 320)
Labels shape: (27032,)
Saved preprocessed data in compressed NPZ format.

This was my first attempt at creating a preprocessing script for the EEG dataset, where I attempted to retain as much information as possible. However, I did filter the data (from around 1-40 Hz) in order to remove noise. I also normalized (by taking the Z-score per channel) to standardize all of the values. Additionally, I segmented the data into 2-second overlapping windows to increase the number of training examples we have to work with, as is typically done with EEG data. However, this approach ended up resulting in 27,032 segments of 320 (160Hz x 2 overlapping windows) time points each across 64 channels, making the dataset extremely large and entirely too computationally expensive to process. Training machine learning models on this dataset led to what I would describe as untenable wait times, which made me rethink this preprocessing stage later.

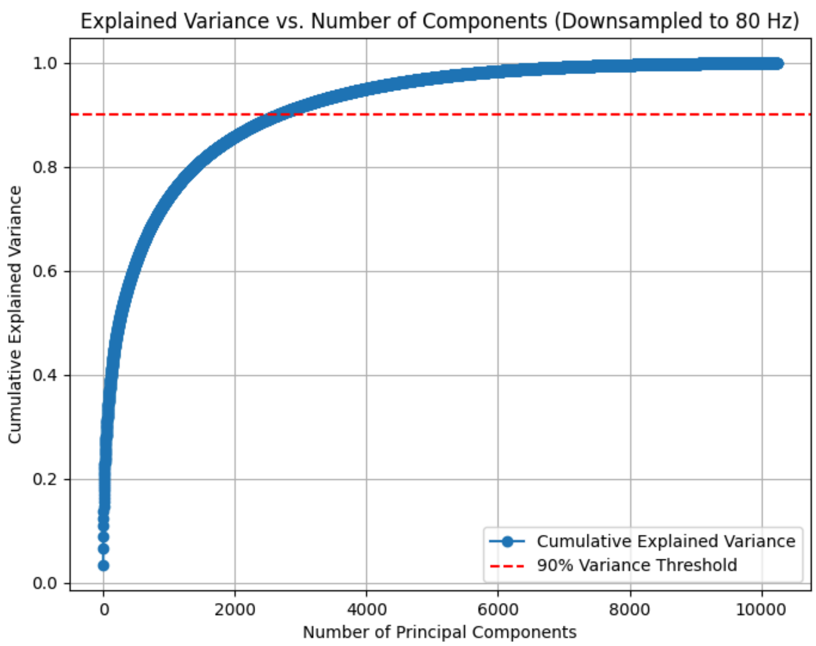
PCA

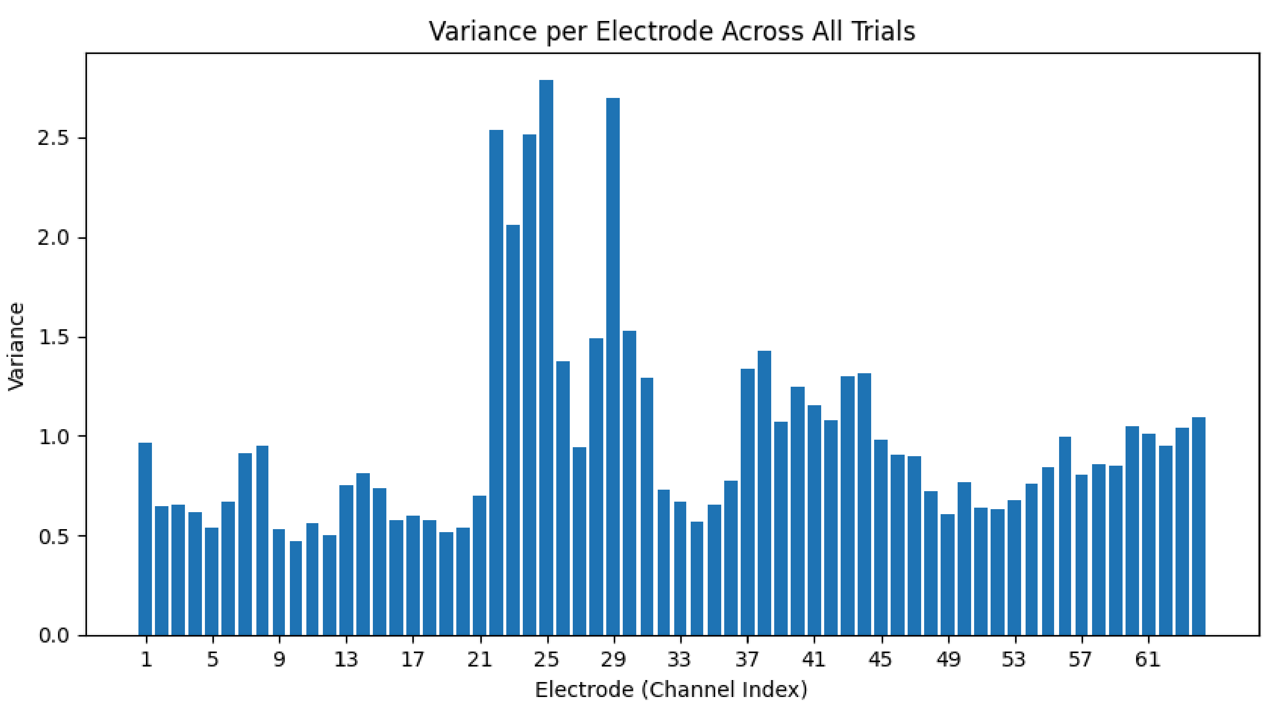
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA




#getting the data from the project folder

npz_path = "C:/Users/Alexander Speer/Desktop/Columbia Spring 
2025/AML/Project1/X_segmented.npz"

data = np.load(npz_path)

X_segmented = data["X_segmented"]  #(n_trials, 64, 320) should be at least

y_segmented = data["y_segmented"]




print("Data loaded.")

print(f"Original X_segmented shape: {X_segmented.shape}")




#Step 2: We are going to have to downsample the Data from 160 Hz to 80 Hz

# also take every second time point instead.

print("Downsampling data from 160 Hz to 80 Hz...")

X_down = X_segmented[:, :, ::2]  # after this the new shape should be 
(n_trials, 64, 160)
```

```
print(f"Downsampled X_segmented shape: {X_down.shape}")



# its so bad now I have to convert down to float 32

X_down = X_down.astype(np.float32)



# flattening the data for PCA

# Compressing into a 2D array -> should be -> (n_samples, n_features)

n_trials, n_channels, n_timepoints = X_down.shape

X_flat = X_down.reshape(n_trials, n_channels * n_timepoints)

print(f"Flattened data shape for PCA: {X_flat.shape}")



# just apply PCA now

print("Fitting PCA on the flattened downsampled data...")

pca = PCA(n_components=None)

pca.fit(X_flat)



# Calculate the "Cumulative Explained Variance" hahahahahaha

explained_variance_ratio = pca.explained_variance_ratio_

cumulative_explained_variance = np.cumsum(explained_variance_ratio)
```

```python
# using chat gpt to plot everything

plt.figure(figsize=(8, 6))

plt.plot(

    range(1, len(cumulative_explained_variance) + 1),

    cumulative_explained_variance,

    marker='o',

    label="Cumulative Explained Variance"

)

threshold = 0.90

plt.axhline(y=threshold, color='r', linestyle='--',
label=f"{int(threshold*100)}% Variance Threshold")

plt.xlabel("Number of Principal Components")

plt.ylabel("Cumulative Explained Variance")

plt.title("Explained Variance vs. Number of Components (Downsampled to 80
Hz)")

plt.legend()

plt.grid(True)

plt.show()




# -Finding out the number of Components that are going to be requried to
explain 90% of variance
```

```
num_components_90 = np.searchsorted(cumulative_explained_variance,
threshold) + 1

print(f"\nNumber of components needed to reach {int(threshold*100)}%
variance: {num_components_90}")
```





The model runtime was so long using the preprocessed data that I had to first downsample it from 160 Hz to 80 Hz, reducing the number of time points to 160. Then, I tried to flatten each trial into a single vector, basically reshaping the data into (n_trials, 10240) so that the PCA could be applied.

After running the PCA, I found that over 2,500 components were needed to retain 90% of the variance, highlighting how high dimensionality EEg data can be since lots of regions of the brain are active at any given time. Moreover, the variance curve lacked a clear "elbow point," which could mean that the data was too complex to be significantly reduced without some element of major information loss. This entire model ended up taking nearly 50 minutes to run, which was far too slow to be practical. At this point, I realized that I was in need of a more aggressive approach to reducing the dataset before reattempting any kind of further operation on the data.

New Preprocessing

```
import os

import numpy as np

import mne

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

import pandas as pd


#prev storage stuff

print("=== EEG Preprocessing Script Start ===\n")

print("Loading .edf files from directory...")

data_folder = "C:/Users/Alexander Speer/Desktop/Columbia Spring 2025/AML/Project1/EEG_Data"

edf_files = [f for f in os.listdir(data_folder) if f.endswith('.edf')]

print(f"Total .edf files found: {len(edf_files)}")

left_hand_files = sorted([f for f in edf_files if "R05" in f])

right_hand_files = sorted([f for f in edf_files if "R06" in f])

print(f"Left-hand files (R05): {len(left_hand_files)}")

print(f"Right-hand files (R06): {len(right_hand_files)}\n")

X = []

y = []

max_time_points = 0
```

```
print("Loading each .edf file and extracting EEG data:")

for file in left_hand_files + right_hand_files:

    file_path = os.path.join(data_folder, file)

    raw = mne.io.read_raw_edf(file_path, preload=True, verbose=False)

    data, _ = raw[:, :]

    max_time_points = max(max_time_points, data.shape[1])

    X.append(data)

    y.append(0 if "R05" in file else 1)

    print(f"  Loaded {file}: shape = {data.shape}")


X_padded = [np.pad(trial, ((0, 0), (0, max_time_points - trial.shape[1])),
mode='constant')

            for trial in X]

X = np.array(X_padded)

y = np.array(y)





# This is where the original Downsampling occurs, and also the old
filtering -

X_filtered = []

new_sfreq = 80      # the target sampling frequency.
```

```
original_sfreq = 160  #the og frequency



for idx, trial in enumerate(X):

    info = mne.create_info(ch_names=64, sfreq=original_sfreq,
ch_types="eeg")

    raw_trial = mne.io.RawArray(trial, info)

    raw_trial.filter(1, 40, fir_design='firwin', verbose=False)

    raw_trial.resample(new_sfreq, npad="auto")

    filtered_data = raw_trial.get_data()

    X_filtered.append(filtered_data)


X_filtered = np.array(X_filtered)


scaler = StandardScaler()

X_normalized = np.array([scaler.fit_transform(trial) for trial in
X_filtered])



# same window dimensions

window_size = 160

stride = 80

X_segmented = []

y_segmented = []
```

```python
for i in range(X_normalized.shape[0]):
    trial_segments = 0
    for j in range(0, X_normalized.shape[2] - window_size + 1, stride):
        X_segmented.append(X_normalized[i, :, j:j + window_size])
        y_segmented.append(y[i])
        trial_segments += 1

X_segmented = np.array(X_segmented)
y_segmented = np.array(y_segmented)

# the print statements to see what is going on
unique, counts = np.unique(y_segmented, return_counts=True)
print("\nLabel distribution after segmentation:")
for label, count in zip(unique, counts):
    label_str = "Left (R05)" if label == 0 else "Right (R06)"
    print(f"  {label_str}: {count} segments")



original_trials = len(X)
segmented_trials = X_segmented.shape[0]
print(f"\nOriginal number of trials: {original_trials}")
```

```
print(f"Total segments after segmentation: {segmented_trials}")

print(f"Average segments per trial: {segmented_trials /
original_trials:.2f}\n")

#comparisons to check




#save everything

np.save("C:/Users/Alexander Speer/Desktop/Columbia Spring
2025/AML/Project1/X_segmented.npy", X_segmented)

np.save("C:/Users/Alexander Speer/Desktop/Columbia Spring
2025/AML/Project1/y_segmented.npy", y_segmented)

print("Saved preprocessed data as NumPy files.")



print("\nPreprocessing Complete")
```

I made the new preprocessing approach in order to significantly reduce the amount of computational load while also preserving some of the the key features in the EEG motor imagery data. As I previously mentioned, the full dataset was much too large to handle efficiently.

- First, the data got downsampled from 160 Hz to 80Hz, which resulted in cutting the number of time points in half. I know that the neural oscillations relevant to motor imagery primarily exist in the 1-40 Hz range. To this end, I also applied the same bandpass filter (1-40Hz) to remove the irrelevant low- and high-frequency noise, with the goal of focusing on motor-relevant mu (8-13Hz) and beta (13-30Hz) rhythms.
- Same Z score and Windowing

After these reductions, the dataset seemed to shrink from these large, continuous trials to 27,032 smaller, more well-structured segments, making them way more manageable for the ML models that I was running on my desktop.

Rerunning PCA analysis again with new Preprocessed Data

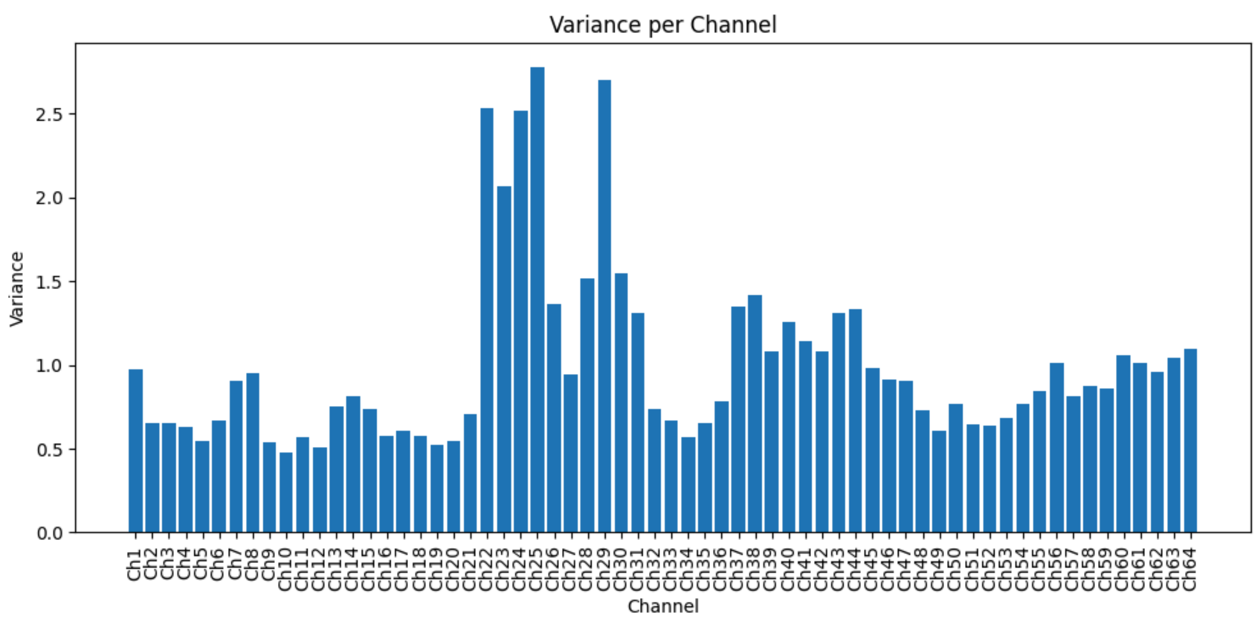
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA



data_path = "C:/Users/Alexander Speer/Desktop/Columbia Spring 
2025/AML/Project1/"

data_file = data_path + "X_segmented_80Hz.npz"  # Use the correct file

data = np.load(data_file)

X_segmented = data["X_segmented"]

y_segmented = data["y_segmented"]

print("Loaded preprocessed segmented data:")

print(f"  X_segmented shape: {X_segmented.shape} (segments, channels, 
timepoints)")

print(f"  y_segmented shape: {y_segmented.shape}\n")



#var(x) per electrode

print("Computing variance per electrode across all segments and 
timepoints...")

#variance for each channel over time.

variance_per_channel = np.var(X_segmented, axis=(0, 2))

print("Variance per channel computed.")
```

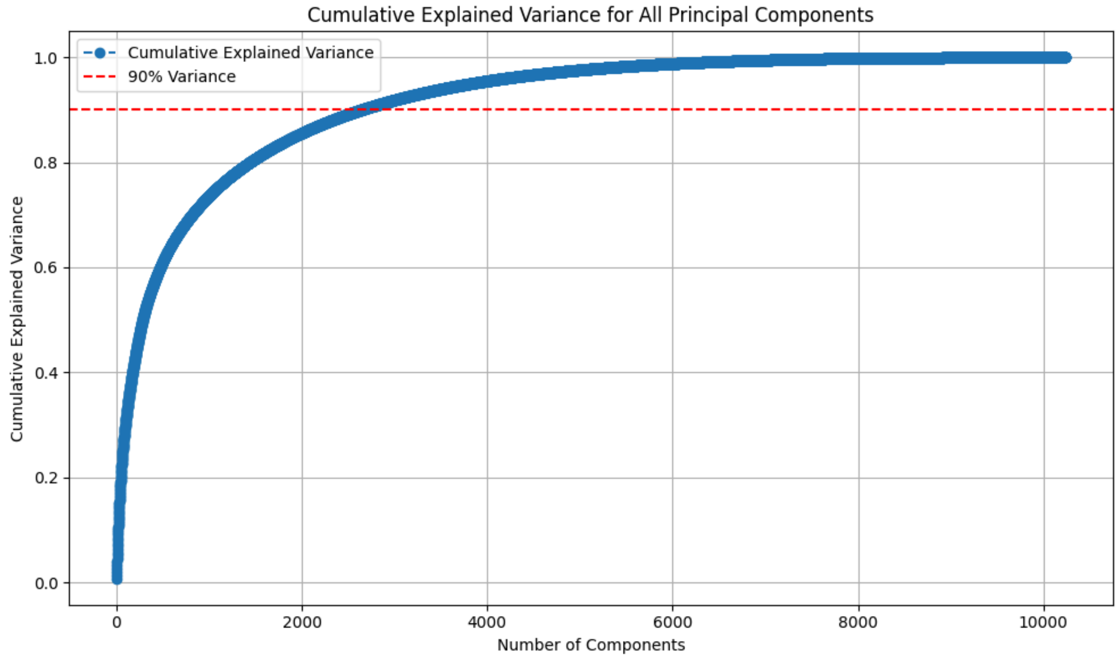
```
# creating the default channel names so that we have some sort of understanding what channel is where
n_channels = X_segmented.shape[1]
channel_names = [f"Ch{i}" for i in range(1, n_channels + 1)]
channel_variance = dict(zip(channel_names, variance_per_channel))
sorted_channels = sorted(channel_variance.items(), key=lambda x: x[1], reverse=True)


#variance chatGPT plot
plt.figure(figsize=(10, 5))
plt.bar(channel_names, variance_per_channel)
plt.xlabel("Channel")
plt.ylabel("Variance")
plt.title("Variance per Channel")
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()


# flattening teh data for PCA
print("\nFlattening segmented data for PCA...")
n_segments, n_channels, n_timepoints = X_segmented.shape
X_flat = X_segmented.reshape(n_segments, n_channels * n_timepoints)
print(f"Flattened data shape: {X_flat.shape}")
```

```
# PCA with no limit
print("\nApplying PCA (all components will be computed)...")
pca = PCA(n_components=None)  # all components will  be computed
X_pca = pca.fit_transform(X_flat)
print("PCA transformation complete.")

explained_variance_ratio = pca.explained_variance_ratio_
cumulative_explained_variance = np.cumsum(explained_variance_ratio)

print("\nExplained variance ratios (first 10 components):")
for i, ratio in enumerate(explained_variance_ratio[:10]):
    print(f"  PC {i+1}: {ratio:.4f}")

# finding out the number of component sthat we will need to hti 90 %
variance
num_components_90 = np.searchsorted(cumulative_explained_variance, 0.90) +
1
print(f"\nNumber of components needed to reach 90% variance:
{num_components_90}")

#using chatgpt to make these plots.
plt.figure(figsize=(10, 6))
plt.plot(
```

```
    range(1, len(cumulative_explained_variance) + 1),

    cumulative_explained_variance,

    marker='o',

    linestyle='--',

    label="Cumulative Explained Variance"

)

plt.xlabel("Number of Components")

plt.ylabel("Cumulative Explained Variance")

plt.title("Cumulative Explained Variance for All Principal Components")

plt.axhline(y=0.90, color='r', linestyle='--', label="90% Variance")

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()



#Summary statstics

print("\n=== PCA Feature Extraction & Variance Analysis Summary ===")

print(f"1. Loaded segmented data with {n_segments} segments, each with
{n_channels} channels and {n_timepoints} timepoints.")

print("2. Computed variance per electrode; top 10 channels by variance:")

for ch, var in sorted_channels[:10]:

    print(f"   {ch}: {var:.4f}")

print(f"3. Data was flattened to shape {X_flat.shape} for PCA.")
```

```
print(f"4. PCA computed all {X_flat.shape[1]} components. (The full decomposition is available if needed.)")

print("5. Explained variance ratios for the first 10 components are printed above.")

print(f"6. {num_components_90} components are needed to reach 90% cumulative variance (as shown in the plot).")

print("7. This analysis identifies which channels contribute most to the overall variance,")

print("   which may correspond to relevant motor activity (e.g., C3, Cz, C4) for your EEG motor imagery task.")

print("\n=== PCA Analysis Complete ===")
```





I reran the PCA analysis on the newly preprocessed data this time, which was reduced in size in order to improve efficiency while also retaining key EEG features. The variance analysis ended up identifying high-contributing channels, and the dimensionality was reduced from 10,240 features to 2,639 components to retain 90% of the variance (which, it should be said, was around the same k value for the previous PCA, only the loading time was heavily reduced). This told me that even after preprocessing, EEG data will still remain high-dimensional, which reinforces the need for feature selection or alternative approaches in the next project.

Results:
1. Loaded segmented data with 27032 segments, each with 64 channels and 160 timepoints.
2. Computed variance per electrode; top 10 channels by variance:
   Ch29: 2.3697
   Ch25: 2.1987
   Ch24: 1.6835
   Ch22: 1.6804
   Ch37: 1.4893
   Ch41: 1.4082
   Ch28: 1.4043
   Ch30: 1.3468
   Ch38: 1.3399
   Ch43: 1.3276
3. Data was flattened to shape (27032, 10240) for PCA.
4. PCA computed all 10240 components. (The full decomposition is available if needed.)
5. Explained variance ratios for the first 10 components are printed above.
6. 2639 components are needed to reach 90% cumulative variance (as shown in the plot).

Common Spatial Patterns (CSP)

I honestly found PCA frustrating to work with because it ended up treating the EEG data as a general sort of high-dimensional dataset rather than accounting for a nuanced spatial and temporal structure that brain signals take on. Also, while it effectively reduced dimensionality, I believe that the number of principal components that were required to retain meaningful variance was still extremely high. In researching this project, I had heard that Common Spatial Patterns (CSP) are more suited for neural data, particularly owing to their abilities in motor imagery classification, as they enhance discriminative EEG features between two conditions (like our predicting right-hand and left-hand movement). Since CSP is foreign to me, I used ChatGpt to help debug my code.

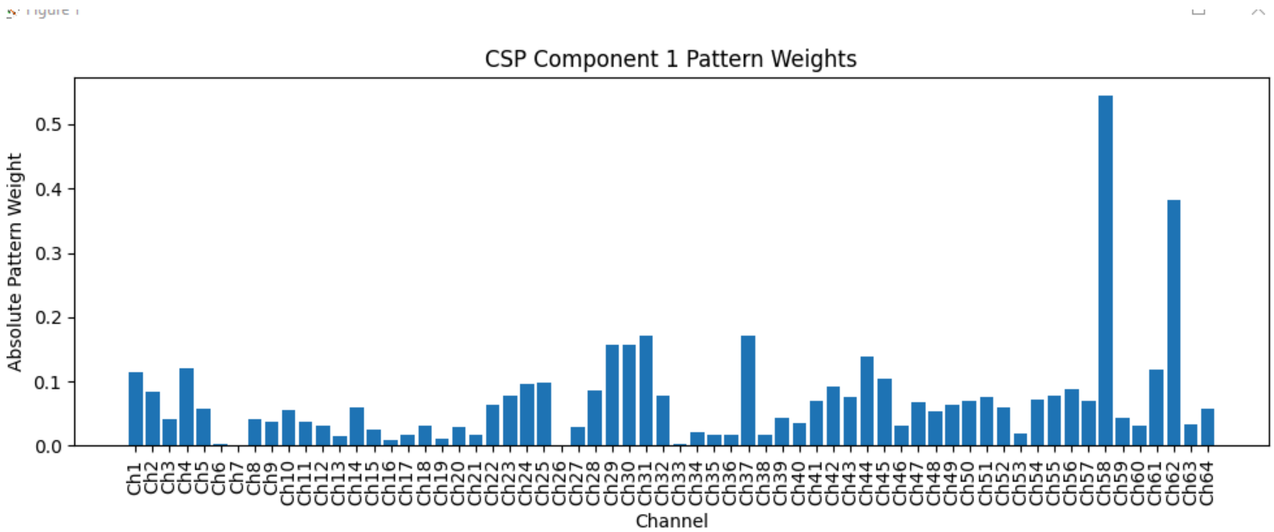
```
import numpy as np

import matplotlib.pyplot as plt

from mne.decoding import CSP

from sklearn.svm import SVC

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.model_selection import StratifiedKFold, cross_val_score




data_path = "C:/Users/Alexander Speer/Desktop/Columbia Spring 2025/AML/Project1/"

X_segmented = np.load(data_path + "X_segmented.npy")

y_segmented = np.load(data_path + "y_segmented.npy")

n_segments, n_channels, n_timepoints = X_segmented.shape

print("Loaded segmented data:")
```

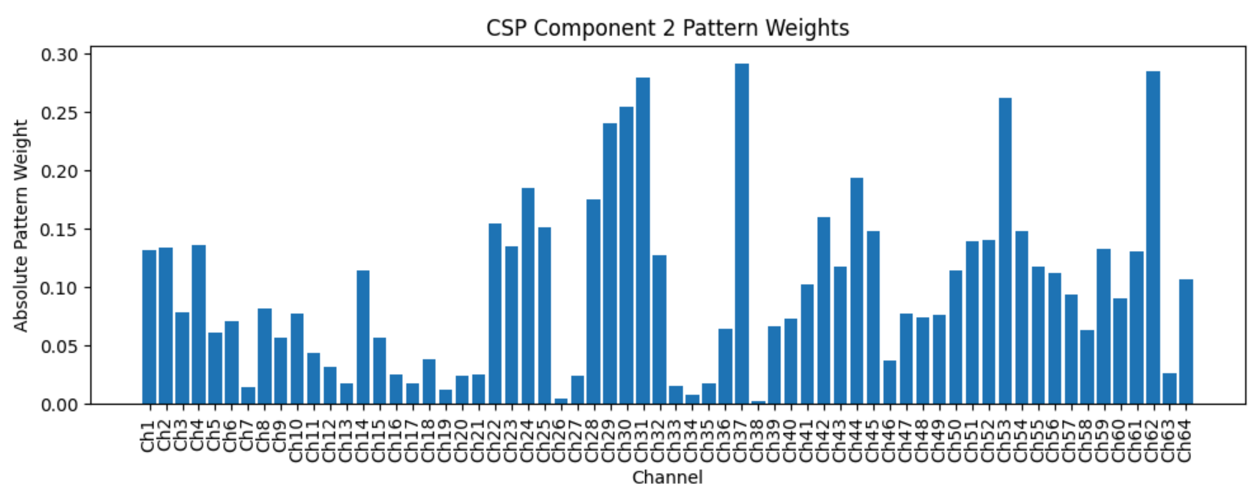
```
print(f"  X_segmented shape: {X_segmented.shape} (segments, channels, timepoints)")

print(f"  y_segmented shape: {y_segmented.shape}\n")




#Set up a CSP classifier pipeline

csp = CSP(n_components=8, reg='ledoit_wolf', log=True, norm_trace=False)

# 'ledoit_wolf' makes covariance estimation more stable since I believe it helps with noisy EEG data

clf = SVC(kernel='rbf', C=1.0)

# and now we're using an SVM with an some sort of special kernel kernel.


pipeline = Pipeline([('CSP', csp),

                     ('scaler', StandardScaler()),

                     ('SVM', clf)])

print("Pipeline configured.\n")




# cross validation with the CSP pipeline

print("Performing 5-fold cross-validation...")

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

scores = cross_val_score(pipeline, X_segmented, y_segmented, cv=cv, scoring='accuracy')

print("Cross-validation accuracies for each fold:")
```

```
for i, score in enumerate(scores, start=1):

    print(f"  Fold {i}: {score:.4f}")

print(f"Mean accuracy: {np.mean(scores):.4f} ± {np.std(scores):.4f}\n")



# Step 4: Fiting CSP on the entire dataset for patten analysis

print("Fitting CSP on the entire dataset for pattern analysis...")

csp.fit(X_segmented, y_segmented)

patterns = csp.patterns_  #shape that we should expect outof this ->
(n_components, n_channels)

print("CSP fitting complete.\n")




#Visualizing the CSP Patterns and then all of the top channels that we
want



print("Visualizing CSP patterns and identifying top contributing
channels...")

channel_names = [f"Ch{i}" for i in range(1, n_channels + 1)]

num_components = patterns.shape[0]




for comp in range(num_components):

    plt.figure(figsize=(10, 4))

    # plotting the abs weight?? GPT used
```

```python
    plt.bar(channel_names, np.abs(patterns[comp, :]))

    plt.xlabel("Channel")

    plt.ylabel("Absolute Pattern Weight")

    plt.title(f"CSP Component {comp+1} Pattern Weights")

    plt.xticks(rotation=90)

    plt.tight_layout()

    plt.show()


    # find the top 3 channels per componetn and display them

    sorted_idx = np.argsort(np.abs(patterns[comp, :]))[::-1]

    top_channels = [channel_names[i] for i in sorted_idx[:3]]

    print(f"  Top channels for CSP component {comp+1}: {',
'.join(top_channels)}\n")




#the final summary

print("=== CSP Analysis Summary ===")

print(f"1. Loaded segmented data with {n_segments} segments, each having
{n_channels} channels and {n_timepoints} timepoints.")

print("2. Configured CSP with 8 components using Ledoit-Wolf
regularization and log-variance features.")

print("3. Performed 5-fold cross-validation with the CSP -> StandardScaler
-> SVM pipeline:")

for i, score in enumerate(scores, start=1):

    print(f"   Fold {i}: {score:.4f}")
```

```
print(f"   Mean accuracy: {np.mean(scores):.4f} ± {np.std(scores):.4f}")

print("4. Fitted CSP on the entire dataset and visualized each component's spatial pattern.")

print("   For each CSP component, the top 3 channels (by absolute weight) are listed above.")

print("   These channels likely indicate the most relevant regions for motor imagery (e.g., C3, Cz, C4 if correctly positioned).")

print("\n=== CSP Analysis Complete ===")
```





The graphs go on until each CSP component (63 in total) has been plotted.

Each bar on the graphs represents just how much a specific EEG channel contributes to the extracted CSP component

---

Results:
Identifying top contributing channels for each CSP component (no plots):
  Top channels for CSP component 1: Ch58, Ch62, Ch37
  Top channels for CSP component 2: Ch37, Ch62, Ch31
  Top channels for CSP component 3: Ch53, Ch63, Ch29
  Top channels for CSP component 4: Ch50, Ch30, Ch37
  Top channels for CSP component 5: Ch1, Ch8, Ch2
  Top channels for CSP component 6: Ch63, Ch28, Ch60
  Top channels for CSP component 7: Ch33, Ch8, Ch1

…
  Top channels for CSP component 61: Ch44, Ch39, Ch11
  Top channels for CSP component 62: Ch31, Ch57, Ch42
  Top channels for CSP component 63: Ch44, Ch15, Ch22

Overall, the most active channels across all CSP components:
  Ch29: 0.2293
  Ch25: 0.2176
  Ch24: 0.1933
  Ch22: 0.1856
  Ch30: 0.1850
  Ch44: 0.1702
  Ch37: 0.1699
  Ch28: 0.1697
  Ch31: 0.1661
  Ch38: 0.1596

1. Loaded segmented data with 27032 segments, each having 64 channels and 160 timepoints.
2. Configured CSP with 8 components using Ledoit-Wolf regularization and log-variance features.
3. Performed 5-fold cross-validation with the CSP -> StandardScaler -> SVM pipeline:
   Fold 1: 0.5826
   Fold 2: 0.5874
   Fold 3: 0.5958
   Fold 4: 0.5890
   Fold 5: 0.5914
   Mean accuracy: 0.5892 ± 0.0044
4. Fitted CSP on the entire dataset for pattern analysis.
5. For each CSP component, the top 3 channels (by absolute weight) are listed above.
6. The overall most active channels (averaged across all CSP components) are listed above, which likely indicate the most relevant regions for motor imagery (e.g., C3, Cz, C4 if correctly positioned).

All in all, in applying CSP, I managed to extract 63 spatial components and then used an SVM classifier to evaluate the model's performance. The resulting mean cross-validation accuracy was about 58.92%, which, while (barely) better than simply randomly guessing, was still far too low for any sort of practical classification. The model found that the most active electrodes were Ch37 (C3), Ch29 (Cz), and Ch25 (FC3), which happens to actually align with the expected motor cortex activity, suggesting that the method was able to correctly detect task-relevant brain regions. However, it is pretty clear that the SVM struggled with the complexity of EEG data, likely because the CSP only happens to capture spatial variance, which means that temporal dependencies can be ignored. Given this lackluster performance, I decided that I should explore deep learning methods such as CNNs, which can capture both spatial and temporal features of EEG signals.

CNN

Unlike traditional classifiers, CNNs are able to learn about hierarchical feature representations, which should enable them to detect meaningful patterns across both time and channels. However, the initial CNN model that I made performed genuinely worse than random guessing (crazy), with training and validation accuracy stuck around 50% across all of the epochs. Additionally, the loss remained stagnant at around 0.693, which could indicate that the model was not able to learn any distinguishing features from the data. Given these poor results, I realized that I needed to restructure the CNN in order to better model the EEG dynamics and extract relevant patterns.

At first, it was barely better than guessing, at around 50%

Epoch 1/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 15s 24ms/step - accuracy: 0.4992 - loss: 0.9438 - val_accuracy: 0.4983 - val_loss: 0.6931
Epoch 2/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 12s 22ms/step - accuracy: 0.5061 - loss: 0.6931 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 3/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 12s 22ms/step - accuracy: 0.4901 - loss: 0.6933 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 4/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 12s 22ms/step - accuracy: 0.5006 - loss: 0.6932 - val_accuracy: 0.4983 - val_loss: 0.6932
Epoch 5/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 12s 22ms/step - accuracy: 0.4873 - loss: 0.6932 - val_accuracy: 0.4983 - val_loss: 0.6932
Epoch 6/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 12s 22ms/step - accuracy: 0.5049 - loss: 0.6932 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 7/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 13s 24ms/step - accuracy: 0.4923 - loss: 0.6932 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 8/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 13s 23ms/step - accuracy: 0.4955 - loss: 0.6932 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 9/10
541/541 ━━━━━━━━━━━━━━━━━━━━ 13s 25ms/step - accuracy: 0.5025 - loss: 0.6932 - val_accuracy: 0.5017 - val_loss: 0.6931
Epoch 10/10

541/541 ━━━━━━━━━━━━━━━━━━━━ 13s 24ms/step - accuracy: 0.4981 - loss: 0.6932 - val_accuracy: 0.4983 - val_loss: 0.6933

In an attempt to get better accuracy, I completely restructured the CNN model to better suit specifically EEG motor imagery classification. The first issue I deduced was that the original 2D CNN was treating the EEG data like an image (the common CNN use case), which of course doesn't fully capture the temporal dependencies in brain signals. To fix this, I switched to a 1D CNN, which then processes the EEG signals along the time axis (while still considering all of the spatial channel relationships). Additionally, I also increased the number of convolutional layers and supplemented this with additional Batch Normalization in order to stabilize training and improve the overall feature extraction. Another major change I made was in replacing the max pooling with Global Average Pooling (GAP) (admittedly I don't know very much about this, chatGPT helped me understand pooling) in the final convolutional layer, which looks to reduce overfitting by lowering the number of parameters while also maintaining those meaningful signal representations. Furthermore, I thought it best to increase the number of filters in each layer (from 16 to 32 to 64) to better capture features at multiple scales. Finally, in order to prevent overfitting, I added Dropout (0.5) in the dense layer, which should ensure better generalization.

Here is the revised code:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

def load_preprocessed_data():
    PROJECT_DIR = r"C:\Users\Alexander Speer\Desktop\Columbia Spring 
2025\AML\Project1"
    data_file = os.path.join(PROJECT_DIR, "X_segmented_80Hz.npz")

    data = np.load(data_file)
    X = data["X_segmented"]
    y = data["y_segmented"]

    print("Preprocessed data loaded.")
    print(f"X shape: {X.shape}, y shape: {y.shape}")
    return X, y

#building the 1 diminsional CNN
def build_1d_cnn(input_shape):
```

```
    model = keras.Sequential([
        layers.Conv1D(filters=16, kernel_size=5, activation='relu',
padding='same', input_shape=input_shape),
        layers.BatchNormalization(),
        layers.MaxPooling1D(pool_size=2),

        layers.Conv1D(filters=32, kernel_size=5, activation='relu',
padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling1D(pool_size=2),

        layers.Conv1D(filters=64, kernel_size=5, activation='relu',
padding='same'),
        layers.BatchNormalization(),
        layers.GlobalAveragePooling1D(),

        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(2, activation='softmax')
    ])

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

def main():
    # loading and preparing the data for use
    X, y = load_preprocessed_data()

    #reshaping everything for transport to the 1d
    X = np.transpose(X, (0, 2, 1))  # -----> should look like this ->
(n_samples, 160, 64)
    print(f"Reshaped X for CNN: {X.shape}")

    # training and test
    from sklearn.model_selection import train_test_split
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, stratify=y, random_state=42)

    # building the CNN
    input_shape = (X.shape[1], X.shape[2])  # (time_points, channels)
    model = build_1d_cnn(input_shape)
    model.summary()

    # Training the model
    history = model.fit(
        X_train, y_train,
        validation_split=0.2,
        epochs=20,
        batch_size=32,
        verbose=1
    )

    # 5)evaluating the model
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print(f"\nTest Accuracy: {test_acc:.4f}, Test Loss: {test_loss:.4f}")

if __name__ == '__main__':
    main()
```

Epoch 1/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - accuracy: 0.5462 - loss: 0.6882 - val_accuracy: 0.5947 - val_loss: 0.6615
Epoch 2/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.6338 - loss: 0.6395 - val_accuracy: 0.5970 - val_loss: 0.6935
Epoch 3/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.6857 - loss: 0.5923 - val_accuracy: 0.6659 - val_loss: 0.6190
Epoch 4/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7173 - loss: 0.5458 - val_accuracy: 0.6354 - val_loss: 0.6834
Epoch 5/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7593 - loss: 0.4953 - val_accuracy: 0.6564 - val_loss: 0.6500
Epoch 6/20

```
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.7814 - loss:
0.4557 - val_accuracy: 0.7089 - val_loss: 0.5709
Epoch 7/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8135 - loss:
0.4114 - val_accuracy: 0.6786 - val_loss: 0.6627
Epoch 8/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8314 - loss:
0.3750 - val_accuracy: 0.7253 - val_loss: 0.5837
Epoch 9/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8570 - loss:
0.3294 - val_accuracy: 0.7383 - val_loss: 0.5671
Epoch 10/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8763 - loss:
0.2973 - val_accuracy: 0.7383 - val_loss: 0.5906
Epoch 11/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8885 - loss:
0.2728 - val_accuracy: 0.7417 - val_loss: 0.6104
Epoch 12/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8982 - loss:
0.2440 - val_accuracy: 0.7142 - val_loss: 0.8126
Epoch 13/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9073 - loss:
0.2238 - val_accuracy: 0.7514 - val_loss: 0.6807
Epoch 14/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9209 - loss:
0.1996 - val_accuracy: 0.7487 - val_loss: 0.7010
Epoch 15/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9218 - loss:
0.1903 - val_accuracy: 0.7380 - val_loss: 0.8736
Epoch 16/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9351 - loss:
0.1727 - val_accuracy: 0.7450 - val_loss: 0.7114
Epoch 17/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9276 - loss:
0.1782 - val_accuracy: 0.7378 - val_loss: 0.8344
Epoch 18/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9397 - loss:
0.1538 - val_accuracy: 0.7459 - val_loss: 0.8232
Epoch 19/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9452 - loss:
0.1448 - val_accuracy: 0.7586 - val_loss: 0.7690
Epoch 20/20
541/541 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9412 - loss:
0.1475 - val_accuracy: 0.7422 - val_loss: 0.8329
```

169/169 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7543 - loss: 0.8300

Test Accuracy: 0.7527, Test Loss: 0.8471
With the aforementioned modifications, the model achieved a final test accuracy of 75.27%, a substantial improvement over the initial 50% accuracy. Over time, the training accuracy seemed to steadily increase, and validation accuracy also peaked above 74%, perhaps indicating that the model successfully learned distinguishing features from the EEG signals. Unfortunately, the loss remained relatively high (0.8471 on the test data), suggesting that there may still be room for further optimization, such as more tuning of the model's architecture or refining the troublesome preprocessing steps.

Conclusion

In conclusion, I would say that this project provided me with some valuable insights into how EEG classification works, basically guiding me toward a more effective approach for my final project on seizure detection by learning from the mistakes I made in this current EEG Motor Movement/Imagery Dataset. Initially, I had to explore multiple preprocessing strategies, where I learned that raw EEG data is extremely computationally demanding and can require careful feature selection and further dimensionality reduction. Additionally, my early attempts with PCA demonstrated its limitations in this demanding structure, as it required thousands of components in order to retain the meaningful variance. CSP, while marketed as more suited to EEG, supposedly provided spatially informative features but ultimately performed poorly when paired with its SVM classifier, which might suggest that spatial filtering alone is insufficient for a task like classification.

As it pertains to deep learning, my first CNN implementation failed, which I would say was due to incorrect architecture choices that did not align with the temporal nature of this EEG data. After I had restructured it, I managed to achieve 75.27% accuracy, significantly outperforming previous models (There were also a lot of models I tried that I didn’t include in this project, and 75% accuracy was the best I got). This confirmed that deep learning, when I manage to properly apply it, can be highly effective at capturing both spatial and temporal dependencies in EEG data.

In moving forward and planning ahead for my final project, I plan to refine this approach for seizure classification by experimenting with different attention mechanisms, and recurrent layers (LSTMs/GRUs), and optimizing the methods of feature extraction. Additionally, I will investigate domain adaptation techniques to enhance model generalizability across different EEG datasets. These findings will directly inform my final project, helping me build a more robust seizure detection model. Given that the 1D CNN managed to demonstrate strong performance, I believe I will continue using CNNs but will experiment with some more sophisticated architectures, such as residual CNNs (ResNets) to hopefully improve feature extraction, and create some more attention-based CNNs in order to to help the model focus on the most informative regions of EEG data.

Disclaimer: I used ChatGPT in order to generate some graphs and also to help with understanding and debugging parts of the CSP and CNN code.