BE 521: Homework 3 Questions

Spring 2025

68 points

Due: Feb 13th, 2025

Objective: Extract features from data and build a simple detector

AI Usage Notice

The use of artificial intelligence tools (e.g., large language models, code assistants) is permitted. However, students must explicitly state the specific ways AI was used in completing their work. Failure to disclose AI usage may result in an oral examination to assess understanding, at the discretion of Dr. Litt.

If AI was used in the completion of this assignment, please provide a statement below:

[Enter your statement here]

1. Features and Simulations (39 pts)

As you learned in class, features are the backbone of almost all detection strategies, from seizures in EEG to faces in images. Features are usually defined in journal articles as an equation or set of equations, and the task for the reader - if she wants to use that feature - is to implement that feature in code. In this section, you will explore and implement some features commonly used in EEG analysis and test them on simulated time-series data.

In [ ]:
```
!jupyter nbconvert --to html /Prakriti_HW1.ipynb
```

In [1]:
```
#Set up the notebook environment
!pip install git+https://github.com/ieeg-portal/ieegpy.git # Install ieegpy toolbox directly from github
from ieeg.auth import Session
from IPython.display import Image
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal as sig
import datetime
import time
```

```
Collecting git+https://github.com/ieeg-portal/ieegpy.git
  Cloning https://github.com/ieeg-portal/ieegpy.git to /tmp/pip-req-build-5yg4z21i
  Running command git clone --filter=blob:none --quiet https://github.com/ieeg-portal/ieegpy.git /tmp/pip-req-bu
ild-5yg4z21i
  Resolved https://github.com/ieeg-portal/ieegpy.git to commit 080bfa42a8503380ef164b5e7b116613f75073bb
  Preparing metadata (setup.py) ... done
Requirement already satisfied: deprecation in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.32.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (1.26.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.2)
Requirement already satisfied: pennprov==2.2.4 in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.
4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2
.4->ieeg==1.6) (2025.1.31)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.11/dist-packages (from pennprov==2
.2.4->ieeg==1.6) (2.8.2)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg=
=1.6) (1.17.0)
Requirement already satisfied: urllib3>=1.23 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->i
eeg==1.6) (2.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from deprecation->ieeg==1.6
) (24.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6)
(2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6
) (2025.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from request
s->ieeg==1.6) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->ieeg==1.6
) (3.10)
```

1

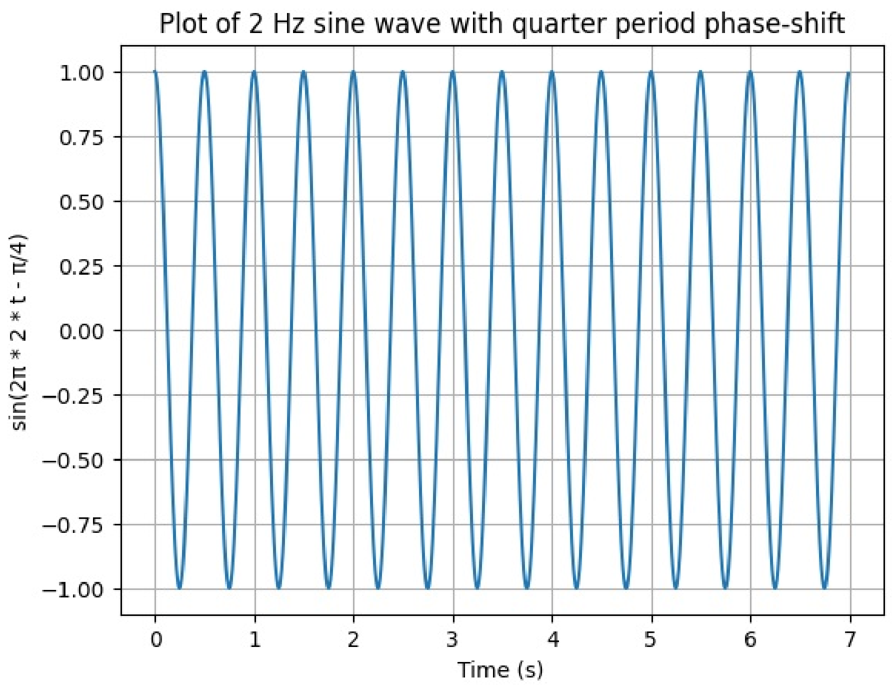
Consider the following toy signal: 7 seconds of a 2 Hz sine wave with a quarter period phase-shift, sampled at 100 Hz.

1a

Plot the signal (2 pts)

```
In [2]: #your code here

fs = 100 # sampling frequency
time = np.arange(0, 7, 1/fs)
frequency = 2 # freq

#create sin wave - with quarter period phase-shift, period = 2pi, shift = 2pi * 1/4
# y = sin( x - pi /2) right shift

y = np.sin(2* np.pi* frequency* time + (np.pi/2))
print(y.size)

# Create the plot
plt.plot(time, y)

# Add labels and title
plt.xlabel('Time (s)')
plt.ylabel('sin(2π * 2 * t - π/4)')
plt.title('Plot of 2 Hz sine wave with quarter period phase-shift')

# Add grid for better readability
plt.grid(True)

# Display the plot
plt.show()
```

```
700
```



Your answer here

1b

Create a function for the line-length feature $LL(x) = \sum_{i=1}^{n} |x_i - x_{i-1}|$. This function can be implemented as a (cheeky) one-liner using Numpy. You may find `np.ediff1d`, `np.sum`, and `np.absolute` to be helpful. (4 pts)

```
In [3]: #It measures the cumulative length of a signal over a given time window,
# providing an estimate of the signal's complexity or activity.

# n = 700 # number of samples
# x = []
# for i in range(0,n):
#   xi = y[i]
#   x.append(xi)

# print(x)

def line_length(x):
  line_length = np.sum(np.abs(np.ediff1d(y)))
  return line_length

print(line_length(y))
```

```
55.99211470131448
```

Your answer here

1c

What is the line length of this signal? (2 pts)

In [4]:
```python
#your code here

print(line_length(y))
```

```
55.99211470131448
```

Your answer here

2

Consider line length of the signal using a sliding window with a certain amount of window overlap (or, to think of it another way, displacement with each "slide"). Now, instead of having just one value for the line length, you will have a number of values.

2a

Given a signal `x` with sampling frequency `fs` and windows of length `winLen` and displacement `winDisp` (both in seconds), create a function called `NumWins` that calculates the number of possible (full) windows in your signal of length `xLen` (in samples). You may assume that `winDisp` is a factor of both `winLen` and the length (in seconds) of `x`. (4 pts)

In [5]:
```python
# 0 - 4
# 4 -2 = 2, 2 + 4 = 6,  2 - 6
# 6 - 2 = 4 , 4 + 4 = 8, 4 - 8
# 6 - 10

def NumWins( signal, fs, winlen, winDisp):

    xlen = signal.size  # duration * fs , but the signals size gives us that as well
    # Convert window length and displacement from seconds to samples
    winLen_samples = (winlen * fs)
    winDisp_samples = (winDisp * fs)

    # Calculate the number of full windows
    num_windows = (xlen - winLen_samples) // winDisp_samples
    num_windows = num_windows + 1

    return int(num_windows)
```

Your answer here

2b

Use this function to calculate the number of windows for the signal described in Question 1.1 for a 400ms window with 200ms displacement. (1 pts)

In [6]:
```python
#your code here

a = NumWins(y, 100, 0.4, 0.2)
print(a)
```

```
34
```

Your answer here

2c

Repeat the above calculation for 50 ms window displacement. (1 pt)

In [7]:
```python
#your code here
# 133

a = NumWins(y, 100, 0.4, 0.05)
print(a)
```

```
133
```

Your answer here

2d

Repeat the above calculation for 100 ms window displacement. (1 pts)

In [8]:
```
#your code here
# 67

a = NumWins(y, 100, 0.4, 0.1)
print(a)
```

```
67
```

Your answer here

3

3a

Create a function called `MovingWinFeats(x, fs, winLen, winDisp, featFn)` that returns a vector of the values of the feature on the signal `x` in all possible windows, where `featFn` is a feature function like the one you wrote in Question 1.1.b. You may find it useful to use your `NumWins` function. You may assume that the product of `winDisp` and the sampling rate `fs` is an integer. (6 pts)

In [9]:
```
#your code here
# 0 - 4
# 4 -2 = 2, 2 + 4 = 6,  2 - 6
# 6 - 2 = 4 , 4 + 4 = 8, 4 - 8
# 6 - 10
def featFn(window):
    return np.sum(np.abs(np.ediff1d(window)))

def MovingWinFeats(x, fs, winLen, winDisp, featFn):

    xlen = x.size  # duration * fs , but the signals size gives us that as well
    winLen_samples = int(winLen * fs)
    winDisp_samples = int(winDisp * fs)

    n = NumWins(x, fs, winLen, winDisp)
    # print(n)
    vector = []
    # vector.append(np.sum(np.abs(np.ediff1d(x[0:winLen_samples]))))

    for i in range(n):

      start_win = i * winDisp_samples
      end_win = start_win + winLen_samples
      x_in_win = x[start_win:end_win]

      feature_value = featFn(x_in_win)

        # Store the computed feature
      vector.append(feature_value)

      # winLen_samples = end_win

    return vector
```

Your answer here

3b

Using the signal you defined in Question 1.1 and the function you created in Question 1.1.b, calculate the line-length over windows of length 400 ms and displacement 200 ms. (2 pts)

In [10]:
```
b = MovingWinFeats(y, fs , 0.4 , 0.2, featFn)

# print(len(b))
print(b)
```

```
[3.1873813145857244, 2.7652037140599783, 2.81467632558119, 3.3011316956894254, 3.5379856217963583, 3.18738131458
57257, 2.765203714059981, 2.8146763255811886, 3.301131695689424, 3.537985621796359, 3.187381314585726, 2.7652037
1405998, 2.814676325581189, 3.301131695689425, 3.537985621796359, 3.1873813145857253, 2.7652037140599806, 2.8146
76325581189, 3.3011316956894254, 3.5379856217963646, 3.1873813145857177, 2.76520371405998, 2.8146763255811824, 3
.301131695689419, 3.5379856217963592, 3.187381314585711, 2.765203714059971, 2.8146763255811895, 3.30113169568943
25, 3.5379856217963632, 3.187381314585724, 2.7652037140599717, 2.814676325581189, 3.3011316956894334]
```

Your answer here

3c

Add a unit-amplitude 10 Hz signal (in the form of a sine wave) to your original signal and again calculate the line length over the same window and displacement. (2 pts)

In [11]:
```python
#your code here

fs = 100 # sampling frequency
time = np.arange(0, 7, 1/fs)
frequency = 2 # freq

#create sin wave - with quarter period phase-shift, period = 2pi, shift = 2pi * 1/4
# y = sin( x - pi /4) right shift

x = np.sin(2* np.pi* frequency* time + (np.pi/2)) + np.sin(2* np.pi* 10 * time)

b = MovingWinFeats(x, fs , 0.4 , 0.2, featFn)

# print(len(b))
print(b)
```

```
[15.213982873452375, 15.276263833971537, 15.202645443068086, 15.259481541227782, 15.444331583060304, 15.21398287
3452373, 15.276263833971534, 15.202645443068066, 15.259481541227787, 15.44433158306033, 15.21398287345241, 15.27
6263833971544, 15.202645443068064, 15.259481541227736, 15.444331583060237, 15.213982873452352, 15.27626383397153
6, 15.20264544306813 9, 15.259481541227832, 15.44433158306032, 15.213982873452288, 15.276263833971502, 15.2026454
4306809, 15.259481541227792, 15.44433158306036, 15.213982873452364, 15.276263833971512, 15.202645443068052, 15.2
59481541227766, 15.44433158306034, 15.21398287345247, 15.276263833971578, 15.202645443068121, 15.259481541227835
]
```

Your answer here

4

Code the following 3 additional features. You only need one line for each.

4a

Area, $A(x) = \sum_{i=1}^{n} |x_i|$ (2 pts)

In [12]:
```python
#your code here

def Area(x):
  area = np.sum(np.abs(x))
  # print(area)
  return area
```

Your answer here

4b

Energy, $E(x) = \sum_{i=1}^{n} x_i^2$ (2 pts)

In [13]:
```python
#your code here
def Energy(x):
  energy = np.sum((np.square(x)))
  return energy
```

Your answer here

4c

Zero-Crossings around mean, $ZX(x) = \sum_{i=2}^{n} [\Phi(\text{FromAbove}) \text{ OR } \Phi(\text{FromBelow})]$.

> Φ is the indicator function which returns 0 if its argument is false, and 1 if it is true.
>
> FromAbove denotes $(x_{i-1} - \bar{x} > 0)\text{AND}(x_i - \bar{x} < 0)$.
>
> FromBelow denotes $(x_{i-1} - \bar{x} < 0)\text{AND}(x_i - \bar{x} > 0)$.
>
> $\bar{x}$ denotes the mean of x. (4 pts)

In [14]:
```python
#your code here

def zero_crossings_mean(x):
    x_mean = np.mean(x)  # Mean of the signal
    zero_crossings = 0  # To store the number of zero-crossings around the mean

    # Loop through the signal from the second element to the last element
    for i in range(1, len(x)):
        # FromAbove condition
        if (x[i-1] - x_mean) > 0 and (x[i] - x_mean) < 0:
            zero_crossings += 1
        # FromBelow condition
        elif (x[i-1] - x_mean) < 0 and (x[i] - x_mean) > 0:
            zero_crossings += 1

    return zero_crossings
```

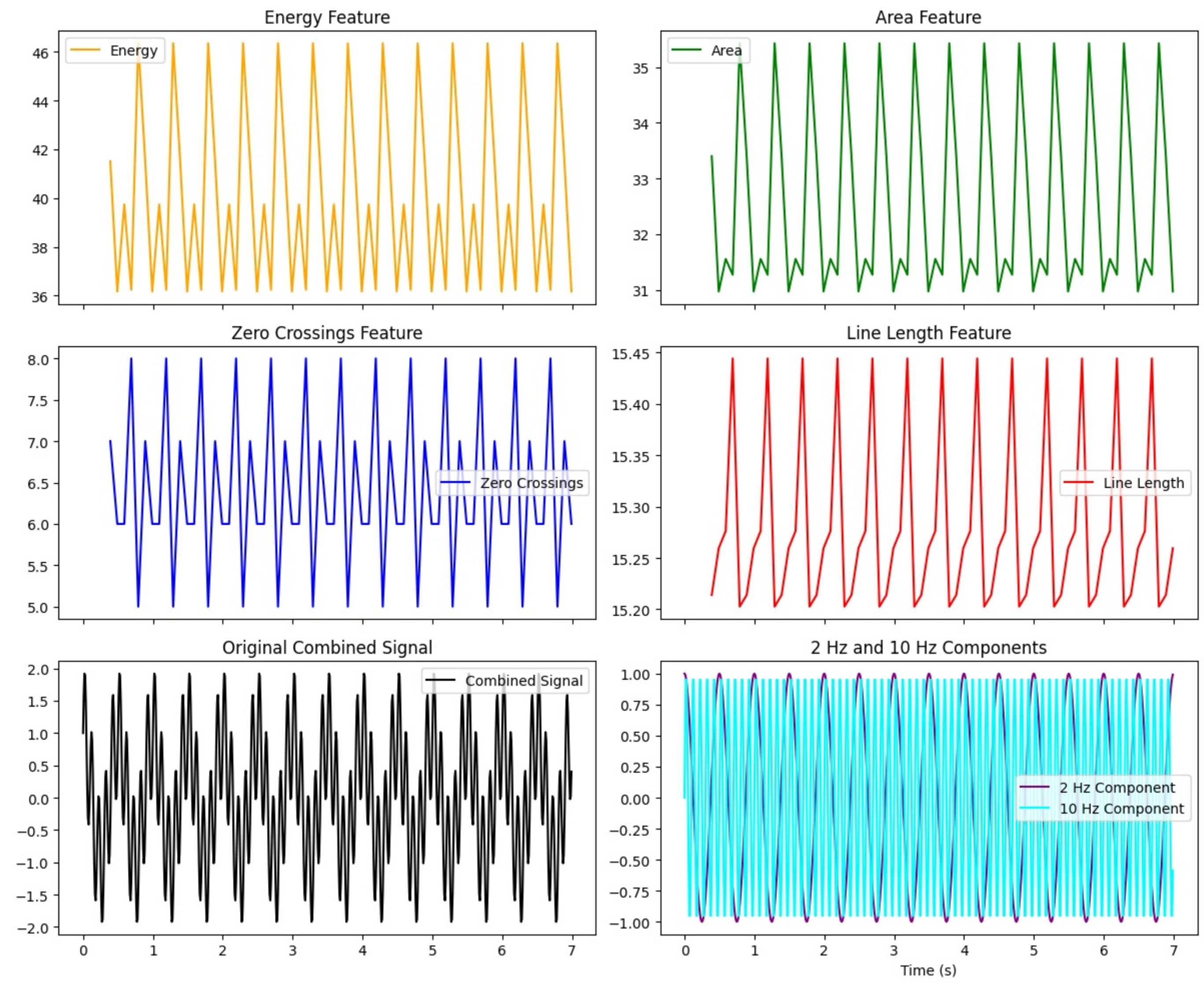
Your answer here

4d

Plot the values of the four features on the combined signal in the first four cells of a 3x2 subplot. Use a 400ms window with 100ms displacement. Using the right-aligned convention (where the official time of the feature is that of the last data point in the window), give the appropriate time axis for each window point. In addition, plot the original signal with the 2Hz and 10Hz components in the last two cells of the 3x2 subplot (to make comparing down the column easy). Ensure that the time axis in all of your plots is the same. (6 pts)

In [34]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generate the time axis
fs = 100  # Sampling frequency in Hz
t_end = 7  # Duration in seconds
time = np.linspace(0,t_end, int(t_end * fs),endpoint=False)

# Define the original 2 Hz signal

original_signal = np.sin((2 * np.pi * 2 * time) + (np.pi/2))

# Define the 10 Hz component
high_freq_component = np.sin(2 * np.pi * 10 * time)

# Create the combined signal
combined_signal = original_signal + high_freq_component

# Feature extraction functions
def area(x):
    return np.sum(np.abs(x))

def energy(x):
    return np.sum(x**2)

# def zero_crossings_mean(x):
#     x_mean = np.mean(x)
#     return np.sum((x[:-1] - x_mean) * (x[1:] - x_mean) < 0)

def line_length(x):
    return np.sum(np.abs(np.diff(x)))


# # Define window parameters
win_len = 0.4  # 400 ms
win_disp = 0.1  # 100 ms



# Compute features
energy_values = MovingWinFeats(combined_signal, fs, win_len, win_disp, energy)
area_values = MovingWinFeats(combined_signal, fs, win_len, win_disp, area)
zero_crossings  = MovingWinFeats(combined_signal, fs, win_len, win_disp, zero_crossings_mean)
line_length = MovingWinFeats(combined_signal, fs, win_len, win_disp, line_length)


# Time axis
num_windows = len(line_length)
time_axis_features = time[int(win_len * fs) - 1 : len(time) : int(win_disp* fs)]


# Plot the features and signals in a 3x2 grid
fig, axes = plt.subplots(3, 2, figsize=(12, 10), sharex=True)

axes[0, 0].plot(time_axis_features, energy_values, label="Energy", color='orange')
axes[0, 0].set_title("Energy Feature")
axes[0, 0].legend()

axes[0, 1].plot(time_axis_features, area_values, label="Area", color='green')
axes[0, 1].set_title("Area Feature")
axes[0, 1].legend()

axes[1, 0].plot(time_axis_features, zero_crossings, label="Zero Crossings", color='blue')
axes[1, 0].set_title("Zero Crossings Feature")
axes[1, 0].legend()
```

```
axes[1, 1].plot(time_axis_features, line_length, label="Line Length", color='red')
axes[1, 1].set_title("Line Length Feature")
axes[1, 1].legend()

axes[2, 0].plot(time[:len(combined_signal)], combined_signal, label="Combined Signal", color='black')
axes[2, 0].set_title("Original Combined Signal")
axes[2, 0].legend()

axes[2, 1].plot(time[:len(original_signal)], original_signal, label="2 Hz Component", color='purple')
axes[2, 1].plot(time[:len(high_freq_component)], high_freq_component, label="10 Hz Component", linestyle='solid
axes[2, 1].set_title("2 Hz and 10 Hz Components")
axes[2, 1].legend()

plt.xlabel("Time (s)")
plt.tight_layout()
plt.show()
```



Your answer here

2. Feature Overlays (17 pts)

In this section, you will use a line-length feature overlay on a segment of EEG containing a seizure. This data is stored in I521_A0003_D001.

Note that IEEG can only send so much data at once (about 600,000 datapoints per channel at a time). To help you import the longer datasets we'll be using in this homework (and in some of the future ones), we've included the following loading function. Feel free to use it.

In [35]:
```
def load_full_channels(dataset, duration_secs, sampling_rate, chn_idx):
  """
  Loads the entire channel from IEEG.org
  Input:-
    dataset: the IEEG dataset object
    duration_secs: the duration of the channel, in seconds
    sampling_rate: the sampling rate of the channel, in Hz
    chn_idx: the indicies of the m channels you want to load, as an array-like object
  Returns:
    [n, m] ndarry of the channels' values.
  """
  #stores the segments of the channel's data
  chn_segments = []

  #how many segments do we expect?
  num_segments = int(np.ceil(duration_secs * sampling_rate / 6e5))

  #segment start times and the step
  seg_start, step = np.linspace(1, duration_secs*1e6, num_segments, endpoint=False, retstep=True)

  #get the segments
  for start in seg_start:
    chn_segments.append(dataset.get_data(start, step, chn_idx))

  #concatenate the segments vertically
  return np.vstack(chn_segments)
```

1

What is the length using hrs:min:sec.ms of the recording? You may find `divmod` to be helpful. Also, the `time` and `datetime` packages are imported and can also be used. (2 pts)

In [46]:
```
#your code here

# get channel label and detials of time series

with open('/content/pra_ieeglogin(9).bin', 'r') as f:
  session = Session('prasadpr', f.read())

dataset= session.open_dataset('I521_A0003_D001')
dataset.get_channel_labels()
key = list(dataset.ts_details.keys())[0]

time = dataset.get_time_series_details(key)
print(time)
print("Sampling rate of this data: ", time.sample_rate)

key = list(dataset.ts_details.keys())[0]
channels = dataset.get_channel_indices([key])[0]

duration_secs = 4880.39
hours, remainder = divmod(duration_secs, 3600)
minutes, seconds = divmod(remainder, 60)
milliseconds = (seconds - int(seconds)) * 1000  # Fractional seconds to milliseconds

print(f"Duration: {int(hours)}:{int(minutes)}:{int(seconds)}.{int(milliseconds)}")
```

```
None(szChirps) spans 4880390000.0 usec, range [-204300-204700] in 976079 samples. Starts @1 uUTC, ends @48803900
01 uUTC with sample rate 200.0 Hz and voltage conv factor 0.01
Sampling rate of this data:  200.0
Duration: 1:21:20.390
```

Your answer here

2

How many data points should we discard at the end if we want to clip the recording to the last full second? Do this clipping. (1 pt)

In [47]:
```
# compute the number of data points per second:
# samples per second=sampling rate
# Compute how many extra samples are in the 0.39 seconds:
# samples to remove=0.39×sampling rate
# Convert this to an integer (since we deal with whole samples).
# Remove these samples from the data.

# Get the sampling rate from the dataset
sampling_rate = time.sample_rate  # Hz (samples per second)

# Compute how many samples to remove
extra_time = 0.395  # Fractional seconds
samples_to_remove = (extra_time * sampling_rate)
# Get total number of samples in the dataset
# number_of_samples = (duration_secs * sampling_rate)  # Convert duration to total samples

number_of_samples = 976079

# Clip the recording using dataset.get_data()
I_clipped = dataset.get_data(0, number_of_samples - samples_to_remove, [channels])
remaining_samples = number_of_samples - samples_to_remove
```

```
# Print results
print(f"Number of samples removed: {samples_to_remove}")
print(f"New number of samples: {len(I_clipped)}")
print(f"Number of samples remaining: {(remaining_samples)}")
print(f"New dataset shape: {I_clipped.shape}")
```

```
Number of samples removed: 79.0
New number of samples: 196
Number of samples remaining: 976000.0
New dataset shape: (196, 1)
```

Your answer here

3

If we want to overlay a feature trace on the original signal, we have to interpolate that feature (which has been calculated over windows) for each data point of the original signal. One of the simplest methods of doing this is called zero-order interpolation, where we just hold the value constant until we get to the next calculated value. For example, if we had a moving window of 1 second with 1 second displacement, the zero-order interpolated feature vector would have the same value the entire 1s5 second, then the same for the entire 2nd second, etc, where each second contains the same number of points as the sampling frequency of the original signal.

3a

Using the `np.tile` and `np.reshape` functions, create an external function `zoInterp(x, numInterp)` that copies each value of `x` `numInterp` times. For `np.reshape`, setting `order='F'` may be necessary. You can implement this function in one line. (2 pts)

In [48]:
```
# overlay a feature trace, interpolate the feature

import numpy as np

def zoInterp(x, numInterp):
    return np.tile(x, (numInterp, 1)).reshape(-1, order='F')
```

Your answer here

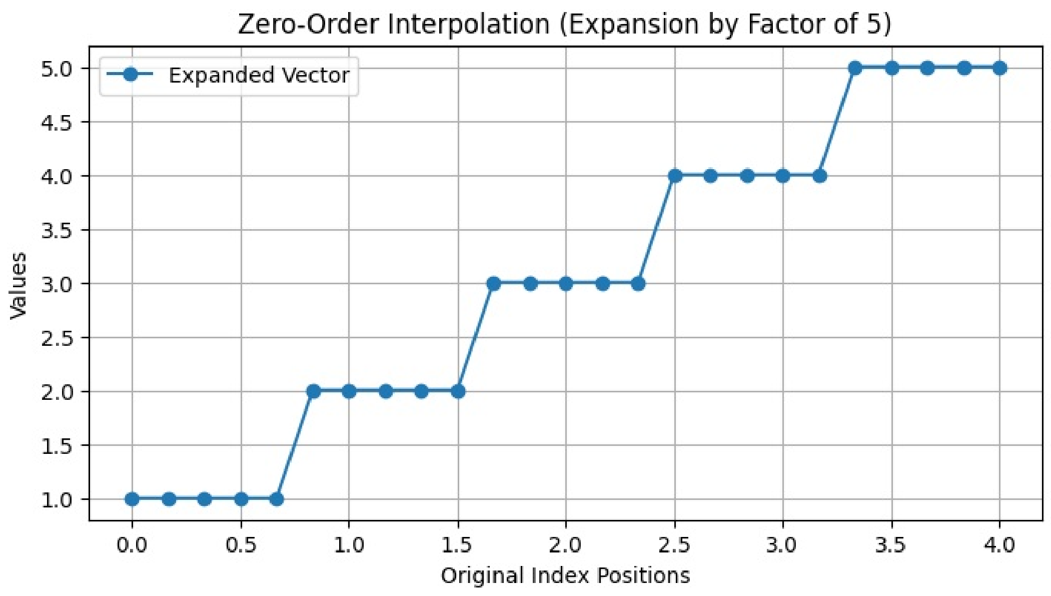
3b

Confirm that this function works correctly by expanding the length of the vector `[1, 2, 3, 4, 5]` by a factor of 5. Plot your results using the `-o` plotting option to let us see the individual points and the line that connects them. (2 pts)

> Hint: your resulting vector should be a column vector of [1,1,1,1,1,2,2,2,...]

In [51]:
```
#your code here

a = [1, 2, 3, 4, 5]
b = zoInterp(a, 5)
print(b)


# Generate x-axis indices for plotting
original_indices = np.arange(len(a))
expanded_indices = np.linspace(0, len(a) - 1, len(b))

# Plot results
plt.figure(figsize=(8, 4))
plt.plot(expanded_indices, b, '-o', label="Expanded Vector")
# plt.xticks(original_indices, labels=x)  # Label original indices
plt.xlabel("Original Index Positions")
plt.ylabel("Values")
plt.title("Zero-Order Interpolation (Expansion by Factor of 5)")
plt.legend()
plt.grid()
plt.show()

# Print the expanded vector for verification
# print("Expanded Vector:", b.reshape(-1, 1))  # Column format
```

```
[1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5]
```



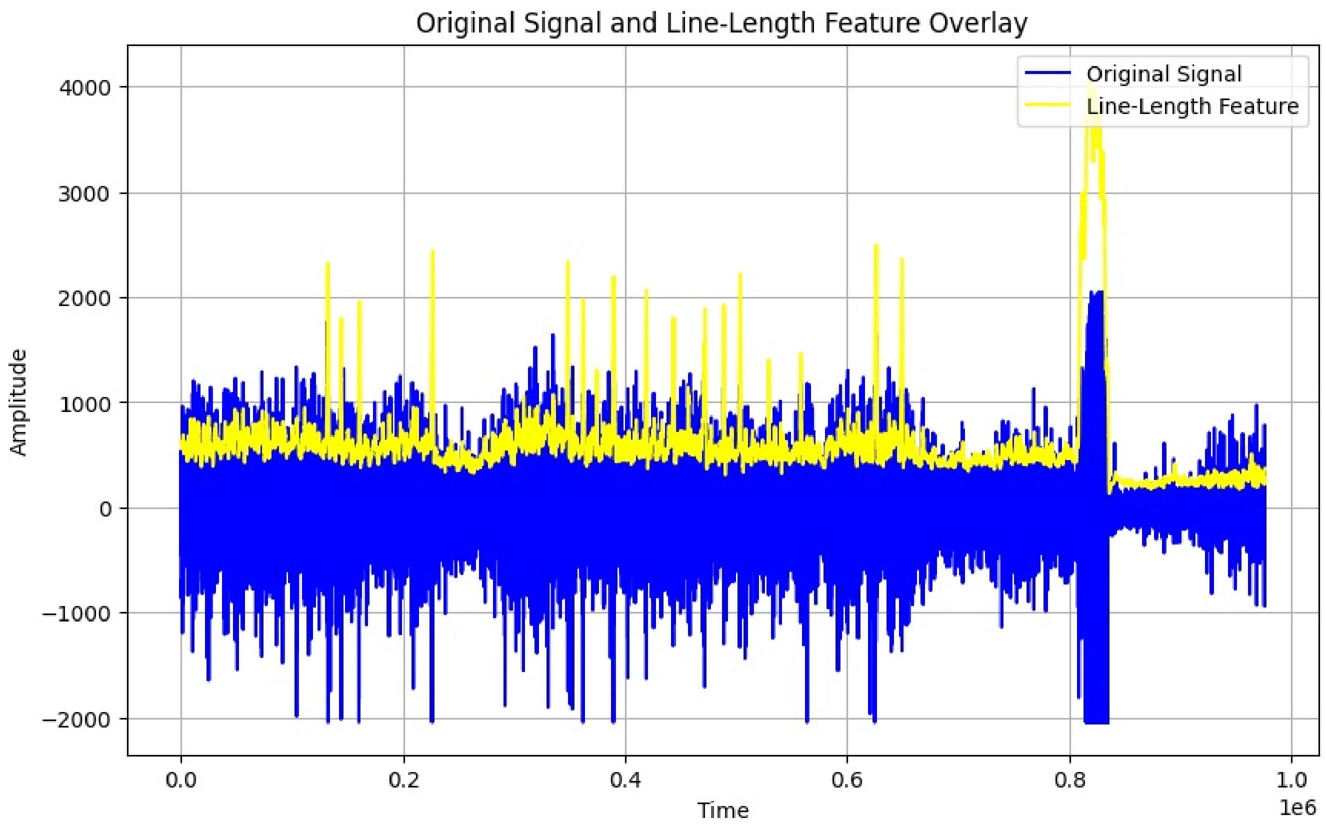
Your answer here

4

Using a 5-second sliding window with 1-second displacement, calculate the line-length feature over the entire signal. Normalize the line-length feature values to have a maximum twice that of the original EEG signal maximum. Plot the signal in blue and overlay the right-aligned line-length feature in yellow. Note: you will need to pad your signal in order to get them to line up correctly and be the same length. Be sure to add a legend in a location in the plot that does not cover up any signal or feature. (6 pts)

In [52]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Load the EEG signal
duration = 4880
I = load_full_channels(dataset, duration, 200, [channels])

# Compute the line-length feature
ll = MovingWinFeats(I, 200, 5, 1, featFn)

# Normalize the feature
ll_normalized = (ll / np.max(ll)) * (2 * np.max(I))

# Check interpolation factor
interp_factor = len(I) // len(ll)
print(f"Interpolation factor: {interp_factor}")

# Correct zero-order interpolation
def zoInterp(x, numInterp, target_length):
    return np.repeat(x, numInterp)[:target_length]

ll_interpolated = zoInterp(ll_normalized, interp_factor, len(I))

if len(ll_interpolated) < len(I):
    padding_needed = len(I) - len(ll_interpolated)
    ll_padded = np.pad(ll_interpolated, (padding_needed, 0), mode='edge')
else:
    ll_padded = ll_interpolated[:len(I)]  # Trim excess if necessary

# Pad correctly using window length
# padding = int(5 * 200)  # 5-second window * sampling rate (200Hz)
# ll_padded = np.pad(ll_interpolated, (padding, 0), mode='edge')
#
# Plot signal + feature overlay
plt.figure(figsize=(10, 6))
plt.plot(I, label='Original Signal', color='blue')
plt.plot(ll_padded, label='Line-Length Feature', color='yellow')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Original Signal and Line-Length Feature Overlay')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

```
Interpolation factor: 200
```



Your answer here

5

What threshold might you use on the raw line-length feature vector (not the normalized one used for plotting) in order to capture the 17 largest pre-seizure chirps that occur? (1 pt)

```
In [55]: #your code here

# print(ll)
print(np.sort(ll))
a = np.sort(ll)
print(a[-17])
# I would use the threshold to be the index of this number 129347.
```

```
[  4399.   4413.   4453. ... 134558. 135295. 136537.]
129347.0
```

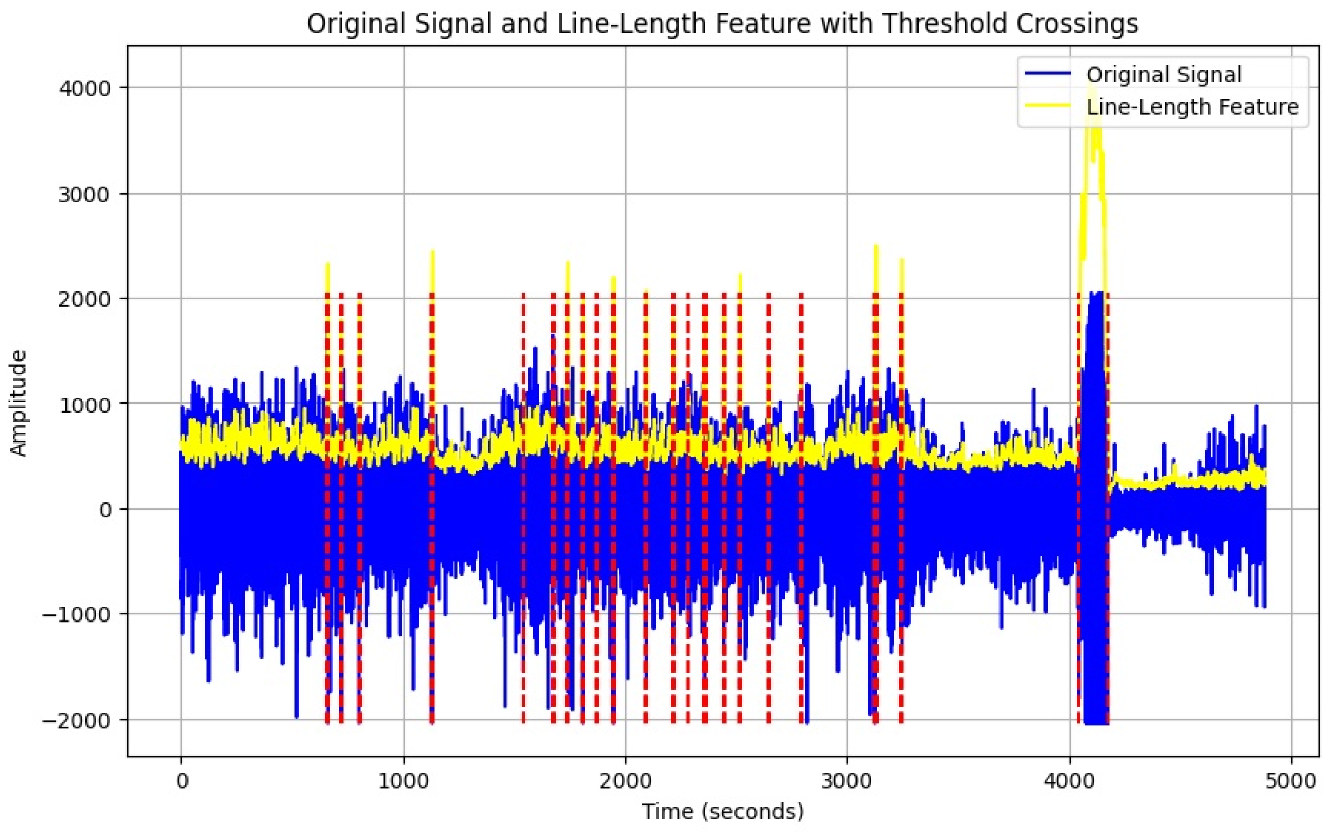
I would use the threshold to be the index of this number 129347 or this number depending on how I compute my it

6

Using this threshold value, in another plot draw red vertical dashed lines at the leading point in time where the threshold is crossed. Add these vertical lines on top of the plot you made in Question 2.4. These events should capture the pre-seizure chirps, the seizure onset, and some flickering during the end of the seizure. (3 pts)

```
In [56]: import numpy as np
import matplotlib.pyplot as plt

threshold = 1000  # Apply threshold on raw line-length feature
time_axis = np.linspace(0, duration, len(I))

# Detect where line-length crosses the threshold
crossing_indices = np.where(np.diff(np.sign(ll_padded - threshold)))[0] + 1
crossing_times = time_axis[crossing_indices]  # Convert to time

# print(f"Crossing times: {crossing_times}")

# Plot signal + feature overlay
```

```
plt.figure(figsize=(10, 6))
plt.plot(time_axis, I, label='Original Signal', color='blue')
plt.plot(time_axis, ll_padded, label='Line-Length Feature', color='yellow')

for time in crossing_times:
    plt.vlines(time, ymin=np.min(I), ymax=np.max(I), color='red', linestyle='--')

plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('Original Signal and Line-Length Feature with Threshold Crossings')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



Your answer here

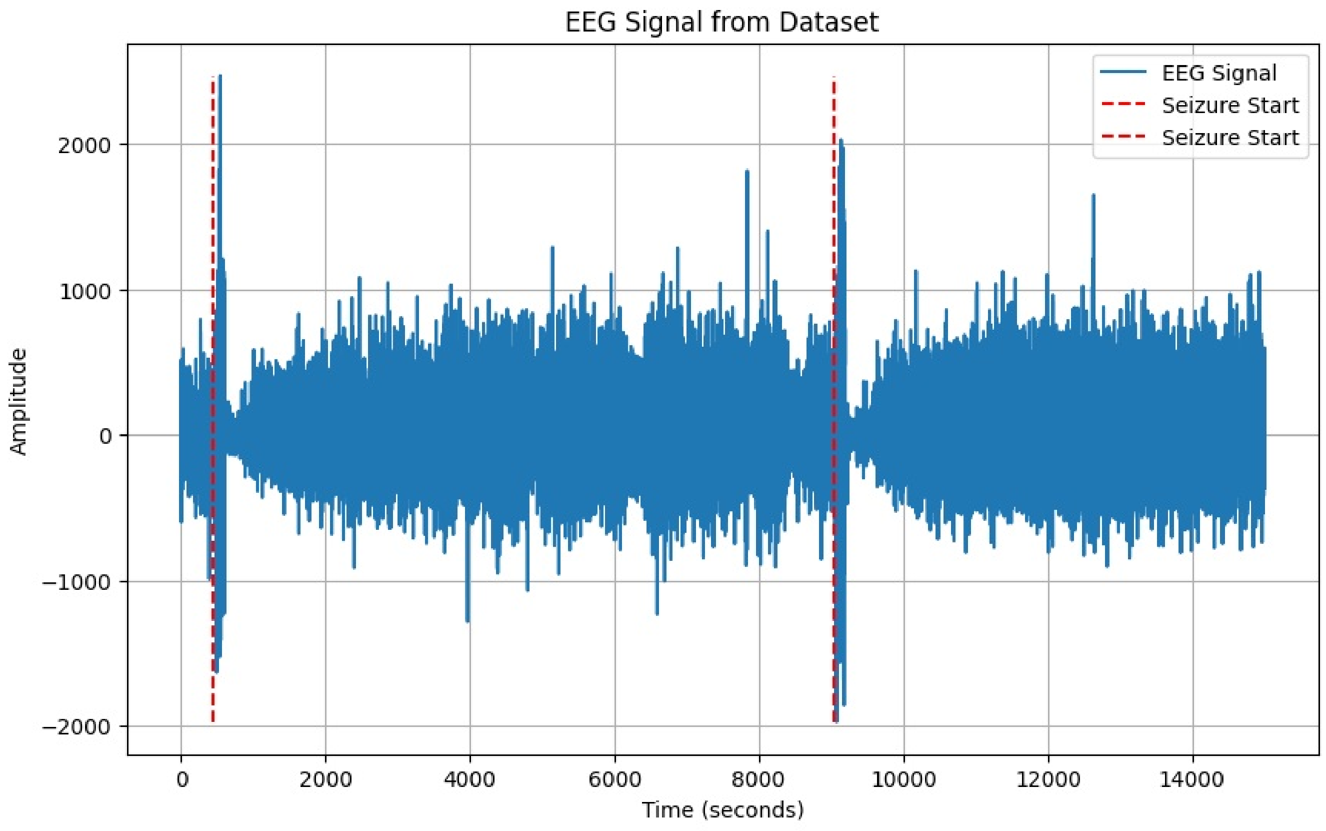
3. Building a Detector (12 pts)

In this section, you will use the features you defined previously to build a seizure detector. Use the EEG data in the file I521_A0003_D002 with channels multiSz_1, and multiSz_2.

1

Plot the signal in multiSz_1 and draw vertical dashed red lines at the times when you think the two seizures begin. You should be able to do this without the need of any features. (2 pts)

In [57]:
```
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
  session = Session('prasadpr', f.read())

dataset= session.open_dataset('I521_A0003_D002')

key = list(dataset.ts_details.keys())[0]
# channel = dataset.get_channel_indices([key])[0]
# time = dataset.get_time_series_details(key)
# print(time)
# multiSz_1 = dataset.get_data(0, 14999995000/1000000, [0])
multiSz_1 = load_full_channels(dataset, 14999995000/1000000, 200,[0])
# print(multiSz_1)


key1 = list(dataset.ts_details.keys())[1]
channel2 = dataset.get_channel_indices([key1])[0]
multiSz_2 = dataset.get_data(0, 488039, [channel2])

time_axis = np.linspace(0, 14999995000/1000000, multiSz_1.shape[0])
# Plot the data
plt.figure(figsize=(10, 6))
plt.plot(time_axis, multiSz_1, label="EEG Signal")
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.title('EEG Signal from Dataset')

seizure_start_times = [450, 9050]
for seizure_time in seizure_start_times:
    plt.vlines(seizure_time, ymin=np.min(multiSz_1), ymax=np.max(multiSz_1), color='red', linestyle='--', label=


plt.legend()
plt.grid(True)
plt.show()



# print(dataset.get_channel_labels())
# a = dataset.get_channel_indices(['multiSz_1'])
# b = dataset.get_channel_indices(['multiSz_2'])
# print(a)
# print(b)
```



Your answer here

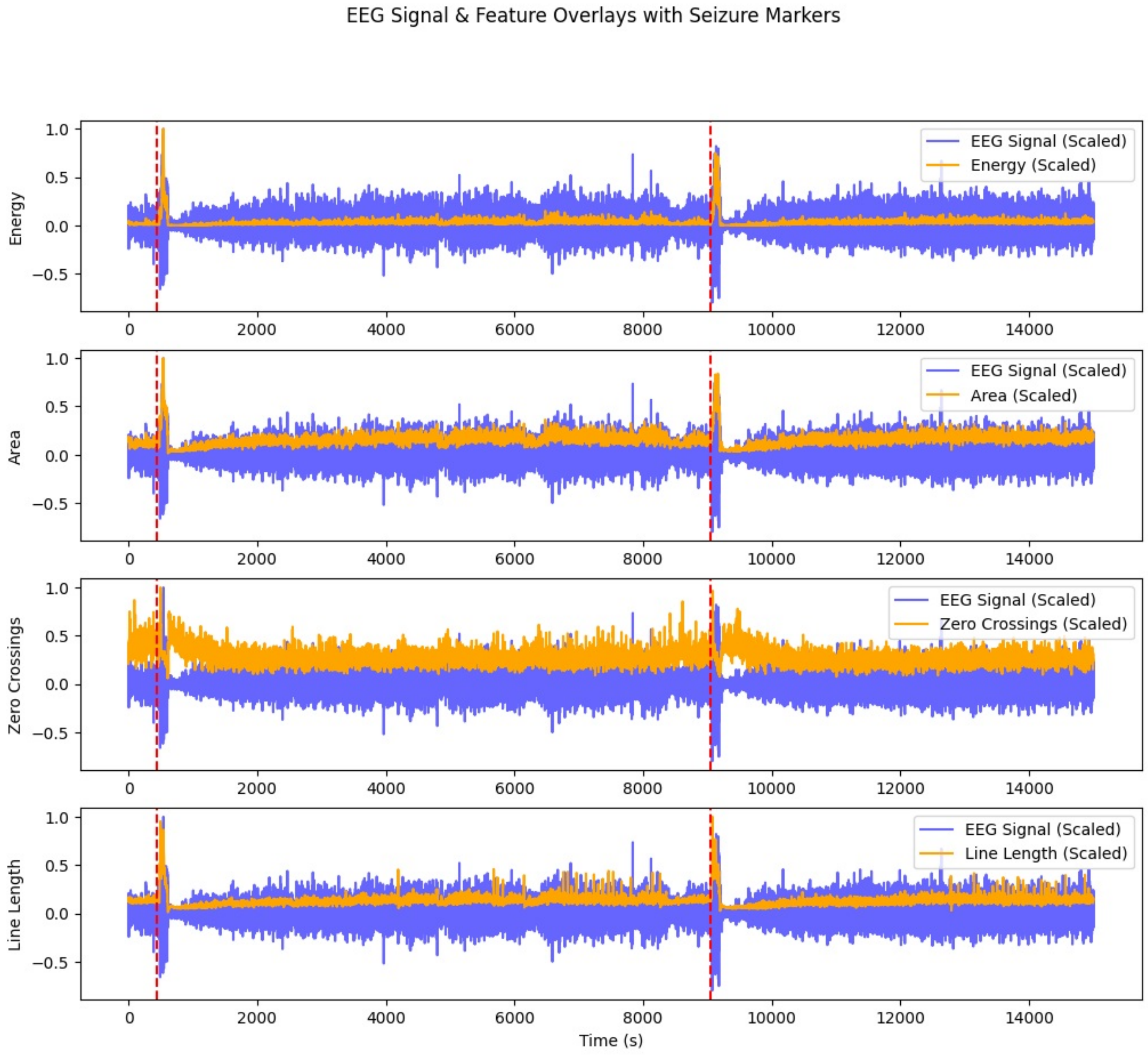
2

Produce feature overlay plots similar to that of Question 2.4 for each of the four features you have implemented along with the red vertical dashed lines at each seizure. Use a 4-second sliding window with 1 second displacement. (4 pts)

In [25]:
```
import numpy as np
import matplotlib.pyplot as plt

# Feature Functions
def Area(x):
    return np.sum(np.abs(x))

def Energy(x):
    return np.sum(np.square(x))

def zero_crossings_mean(x):
    x_mean = np.mean(x)
    zero_crossings = 0
    for i in range(1, len(x)):
        if (x[i-1] - x_mean) > 0 and (x[i] - x_mean) < 0:
            zero_crossings += 1
        elif (x[i-1] - x_mean) < 0 and (x[i] - x_mean) > 0:
            zero_crossings += 1
    return zero_crossings
```

```python
def line_length(x):
    return np.sum(np.abs(np.ediff1d(x)))  # Line Length (Mean Abs Diff)

# Compute number of windows
def NumWins(x, fs, winLen, winDisp):
    return int((len(x) - winLen * fs) / (winDisp * fs)) + 1

# Moving Window Feature Extraction
def MovingWinFeats(x, fs, winLen, winDisp, featFn):
    winLen_samples = int(winLen * fs)
    winDisp_samples = int(winDisp * fs)

    n = NumWins(x, fs, winLen, winDisp)
    features = []

    for i in range(n):
        start_win = i * winDisp_samples
        end_win = start_win + winLen_samples
        x_in_win = x[start_win:end_win]
        features.append(featFn(x_in_win))

    return np.array(features)

# Load EEG Data
fs = 200  # Assuming a sampling rate of 200 Hz
winLen = 4  # 4-second window
winDisp = 1  # 1-second displacement

# Use EEG data from `multiSz_1`
signal = multiSz_1[:, 0]  # Assuming single-channel EEG data

# Compute Features
energy_feat = MovingWinFeats(signal, fs, winLen, winDisp, Energy)
area_feat = MovingWinFeats(signal, fs, winLen, winDisp, Area)
zc_feat = MovingWinFeats(signal, fs, winLen, winDisp, zero_crossings_mean)
ll_feat = MovingWinFeats(signal, fs, winLen, winDisp, line_length)

# Generate Time Axis
time_axis = np.linspace(0, len(signal) / fs, len(signal))
feature_time_axis = np.linspace(winLen / 2, (len(signal) / fs) - (winLen / 2), len(energy_feat))

# Seizure timestamps (example values, replace with actual seizure times)
seizure_times = [450, 9050]   # In seconds

# Plot Features with Original EEG Overlay
plt.figure(figsize=(12, 10))

plt.subplot(4, 1, 1)
plt.plot(time_axis, signal / np.max(np.abs(signal)), color='blue', alpha=0.6, label="EEG Signal (Scaled)")
plt.plot(feature_time_axis, energy_feat / np.max(energy_feat), color='orange', label="Energy (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Energy")
plt.legend()

plt.subplot(4, 1, 2)
plt.plot(time_axis, signal / np.max(np.abs(signal)), color='blue', alpha=0.6, label="EEG Signal (Scaled)")
plt.plot(feature_time_axis, area_feat / np.max(area_feat), color='orange', label="Area (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Area")
plt.legend()

plt.subplot(4, 1, 3)
plt.plot(time_axis, signal / np.max(np.abs(signal)), color='blue', alpha=0.6, label="EEG Signal (Scaled)")
plt.plot(feature_time_axis, zc_feat / np.max(zc_feat), color='orange', label="Zero Crossings (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Zero Crossings")
plt.legend()

plt.subplot(4, 1, 4)
plt.plot(time_axis, signal / np.max(np.abs(signal)), color='blue', alpha=0.6, label="EEG Signal (Scaled)")
plt.plot(feature_time_axis, ll_feat / np.max(ll_feat), color='orange', label="Line Length (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Line Length")
plt.legend()

plt.xlabel("Time (s)")
plt.suptitle("EEG Signal & Feature Overlays with Seizure Markers")
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Creating legend with loc="b
est" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```



Your answer here

3

3a

Based on your plots in the previous question, which of the four features seems to give the largest signal (relative to the background) for when a seizure occurs? Explain why you think this feature is the best. (3 pts)

Based on the corrected plots, Energy appears to give the largest signal relative to the background during a seizure. We could also use Area and line lengths.

Largest Contrast Between Seizure and Baseline

```
The energy feature shows a significant increase during seizure periods compared to the non-
seizure background
```

Energy Captures High-Amplitude Changes

```
Seizures are often characterized by high-amplitude and high-frequency activity, both of which
contribute to a large energy value.
```

Low Baseline During Non-Seizure Periods

```
Unlike Zero Crossings, which may remain noisy even in non-seizure states, the Energy feature
stays relatively low in normal EEG activity.
```

Your answer here

3b

What threshold would you use to determine if a seizure is occurring? (1 pt)

- I would use 1000 if I need to capture pre-seizure chirps, the seizure onset, and some flickering during the end of the seizure.
- But if I want to detect only seizure then I would use 4000 or above

4

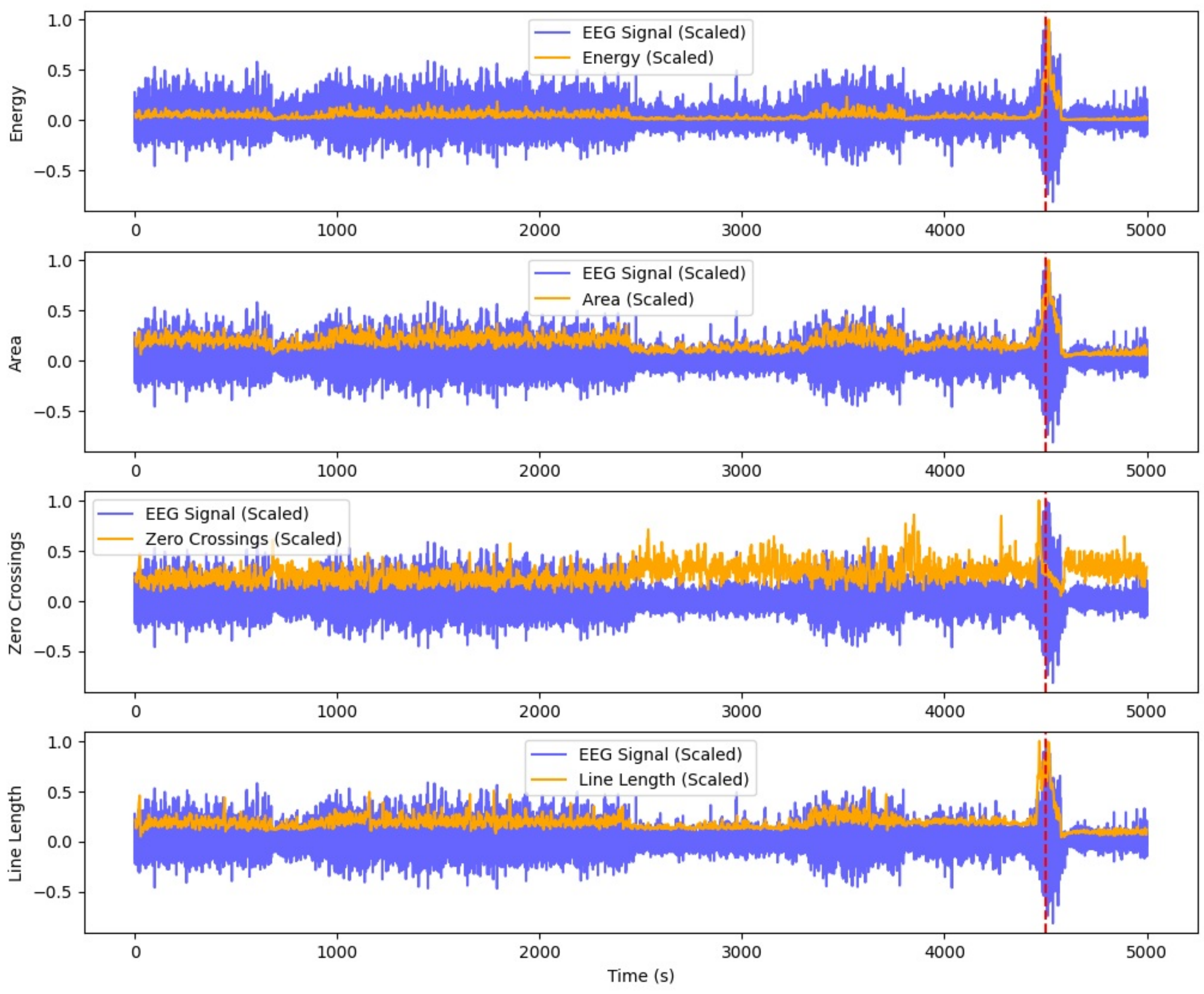
The signal in multiSz_2 contains another seizure (whose location should again be fairly obvious). Plot the data along with the feature and threshold (horizontal black line, with correct normalization for the signal in data2) you determined in the previous question. (2 pts)

**Considering only the energy graph based on my above answer, I think energy works pretty well. I set the threshold to 4500. The threshold 4000 worked for the above graphs. **

In [66]:

```python
import numpy as np
import matplotlib.pyplot as plt

# Load EEG Data (restricting to 100,000 samples for performance)
multiSz_2 = load_full_channels(dataset, 14999995000 / 1000000, 200, [1])
multiSz_2 = multiSz_2[:1000000]  # Limit dataset for efficiency

# Feature Functions
def Area(x):
    return np.sum(np.abs(x))

def Energy(x):
    return np.sum(np.square(x))

def zero_crossings_mean(x):
    x_mean = np.mean(x)
    return np.sum((x[:-1] - x_mean) * (x[1:] - x_mean) < 0)

def line_length(x):
    return np.sum(np.abs(np.ediff1d(x)))  # Line Length (Mean Abs Diff)

# Compute number of windows
def NumWins(x, fs, winLen, winDisp):
    return int((len(x) - winLen * fs) / (winDisp * fs)) + 1

# Moving Window Feature Extraction
def MovingWinFeats(x, fs, winLen, winDisp, featFn):
    winLen_samples = int(winLen * fs)
    winDisp_samples = int(winDisp * fs)

    n = NumWins(x, fs, winLen, winDisp)
    features = np.zeros(n)

    for i in range(n):
        start_win = i * winDisp_samples
        end_win = start_win + winLen_samples
        features[i] = featFn(x[start_win:end_win])

    return features

# Sampling rate and window parameters
fs = 200  # 200Hz sampling rate
winLen = 4  # 4-second window
winDisp = 1  # 1-second displacement

# Compute Features
energy_feat = MovingWinFeats(multiSz_2, fs, winLen, winDisp, Energy)
area_feat = MovingWinFeats(multiSz_2, fs, winLen, winDisp, Area)
zc_feat = MovingWinFeats(multiSz_2, fs, winLen, winDisp, zero_crossings_mean)
ll_feat = MovingWinFeats(multiSz_2, fs, winLen, winDisp, line_length)

# Define time axes (right-aligned convention)
time_axis = np.linspace(0, len(multiSz_2) / fs, len(multiSz_2))
feature_time_axis = np.linspace(winLen, (len(multiSz_2) / fs), len(energy_feat))  # Adjusted to right-align

# Example seizure timestamps (modify accordingly)
seizure_times = [4500]  # Adjust based on actual seizure locations in seconds

# Plot Features with Original EEG Overlay
plt.figure(figsize=(12, 10))
plt.subplot(4, 1, 1)
plt.plot(time_axis, multiSz_2 / np.max(np.abs(multiSz_2)), color='blue', alpha=0.6, label="EEG Signal (Scaled)"
plt.plot(feature_time_axis, energy_feat / np.max(energy_feat), color='orange', label="Energy (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Energy")
plt.legend()

plt.subplot(4, 1, 2)
plt.plot(time_axis, multiSz_2 / np.max(np.abs(multiSz_2)), color='blue', alpha=0.6, label="EEG Signal (Scaled)"
plt.plot(feature_time_axis, area_feat / np.max(area_feat), color='orange', label="Area (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Area")
plt.legend()

plt.subplot(4, 1, 3)
plt.plot(time_axis, multiSz_2 / np.max(np.abs(multiSz_2)), color='blue', alpha=0.6, label="EEG Signal (Scaled)"
plt.plot(feature_time_axis, zc_feat / np.max(zc_feat), color='orange', label="Zero Crossings (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Zero Crossings")
plt.legend()

plt.subplot(4, 1, 4)
plt.plot(time_axis, multiSz_2 / np.max(np.abs(multiSz_2)), color='blue', alpha=0.6, label="EEG Signal (Scaled)"
plt.plot(feature_time_axis, ll_feat / np.max(ll_feat), color='orange', label="Line Length (Scaled)")
for seizure in seizure_times:
    plt.axvline(x=seizure, color='r', linestyle='dashed')
plt.ylabel("Line Length")
plt.legend()

plt.xlabel("Time (s)")
plt.suptitle("EEG Signal & Feature Overlays with Seizure Markers")
plt.show()
```

EEG Signal & Feature Overlays with Seizure Markers



Your answer here