NeuroNinjas: Elena Grajeles, Nova Meng, Pavan Raghupathy
University of Pennsylvania
April 28, 2024

Algorithm Summary

Our algorithm consists of five main steps: Data Loading, Preprocessing, Feature Extraction, Modeling/Prediction, and Post-Processing. Initially, ECoG data from five participants is loaded. This data then undergoes rigorous preprocessing, including bandpass and notch filtering to eliminate unwanted frequencies and noise, ensuring the integrity of the data for accurate model predictions. Next, the data is segmented using a moving window technique to extract features such as average voltage, signal energy, and Shannon entropy from both the time-domain and frequency-domain. An R feature matrix that encapsulates temporal features from multiple windows, is then fed into XGBoost models for predicting finger movements. Finally, the model outputs are refined through cubic spline interpolation and low-pass filtering to produce smooth and realistic predictions of finger movements.

Algorithm Details

In the final algorithm, the procedure begins with the loading of electrocorticography (ECoG) data from a MATLAB file, which contains neural recordings from three different participants. Each participant's ECoG data is separately loaded into distinct variables for isolated processing. Following data acquisition, the algorithm proceeds to load three pre-trained XGBoost regression models from pickle files. These models are specifically trained to predict finger movements from the ECoG signals, with each model corresponding to one of the three participants.

The preprocessing stage involves multiple steps to refine the signal for further analysis. Initially, a bandpass filter is applied to retain frequencies between 2 Hz and 160 Hz, which is critical to capture the brain's electrical activity while removing noise. Additionally, several notch filters eliminate specific frequencies known to represent noise, including the standard 50 Hz power line interference and other sporadic spike frequencies identified in the data.

After the filtering process has been completed, the algorithm segments the ECoG data using a moving window technique, where each window undergoes feature extraction. This step

computes various signal characteristics such as average voltage, signal energy, and Shannon entropy, as well as frequency-domain features within defined bands.

Next, using the function MovingWinFeats, features from each data segment are compiled. A composite feature matrix (R Matrix) integrated these features across multiple windows, enabling the algorithm to capture temporal dependencies essential for dynamic predictions. During the algorithm testing phase, the feature matrices are input into their corresponding XGBoost models to generate predictions for finger movements. To ensure continuity in the predictions, cubic spline interpolation is employed, followed by a low-pass filter to smooth the prediction trajectories. Finally, the algorithm structures the predictions for each participant into a comprehensive output matrix. The process of the algorithm is visualized below in Figure 1.

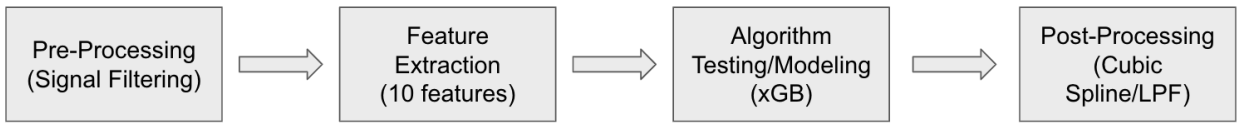

Fig. 1: Algorithm Flowchart

Figure 2 highlights the motivation behind the post-processing steps we employed on the calculated finger movement prediction. We experimented with varying low-pass filters on interpolated predictions from an XGBoost model, aimed at smoothing the predictions of finger movements. This step is crucial as it helps eliminate high-frequency noise that might distort the realism of the predictions. By plotting these two datasets, we can visually assess how the low-pass filtering enhances the alignment of the predicted movements with the actual movements. The correlation metrics computed in this process quantify the improvement, providing a direct link between the filtering technique and the increased prediction accuracy.

This visualization was pivotal in motivating the choice of a 0.5 Hz cutoff for our low-pass filter in the post-processing phase. It clearly illustrates the benefits of filtering, showing a significant reduction in noise and an increase in the smoothness of the output, which more closely mimics physiological finger movements. Thus, this figure not only supports our post-processing decisions but also highlights the practical impact of these choices on the model's performance.

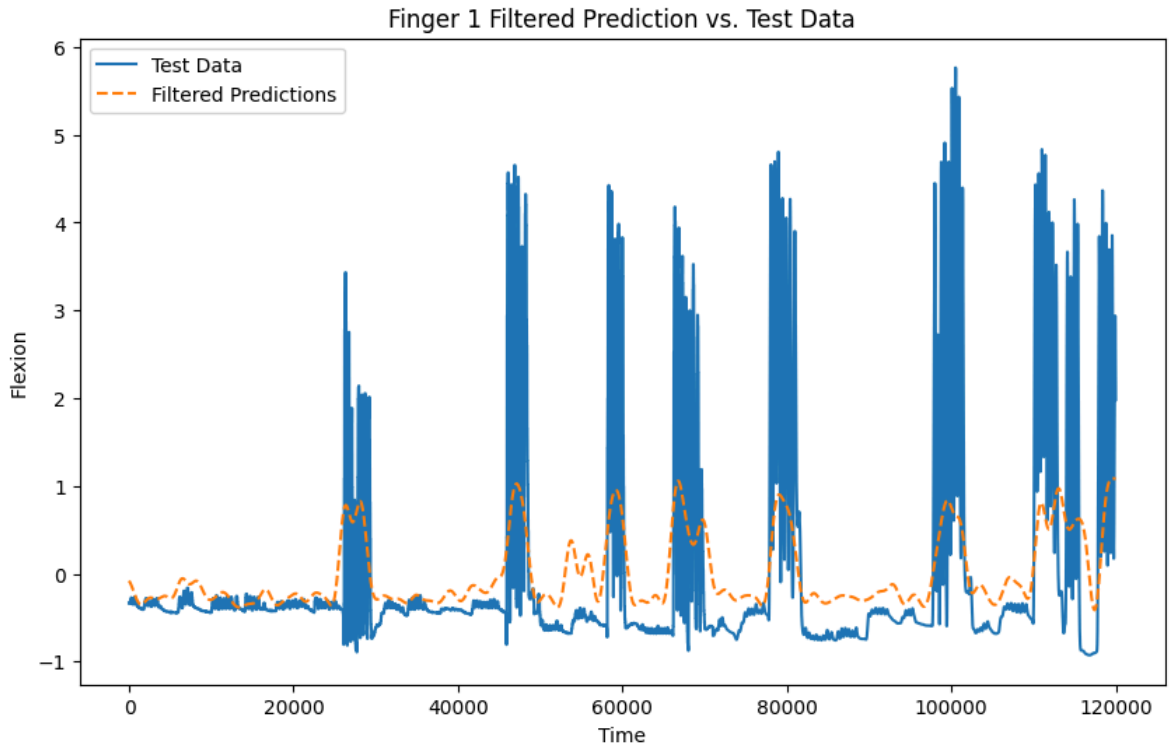

Fig 2: Example of Filtered Predictions vs Test Data (Patient 1, Finger 1)

Other Attempted Methods:

Throughout the development of our final algorithm, we evaluated several regression techniques. One approach that initially seemed promising was the use of the AdaBoostRegressor, an ensemble method that combines multiple weak models (XGBoost regressors, in this case) into a robust predictor. We found the training process proved excessively time-consuming, ultimately deemed inefficient for our needs. This inefficiency was a significant drawback since it hindered rapid iteration and testing, which are crucial in a research and development setting.

Furthermore, we experimented with Gradient Boosting Regression. Despite its reputation for high performance in regression tasks, the results were underwhelming, with an average correlation across multiple trials only around 0.36. This modest performance highlighted the challenges of tuning gradient boosting models for highly specific biomedical data, where the signal complexity and noise can substantially affect performance.

We also explored Neural Network Regression using an MLPRegressor, aiming to leverage deep learning's capability for handling non-linear relationships and complex patterns.

Despite working extensively on tuning the configuration of the network architecture and learning parameters, the results were largely disappointing. The correlations were generally low, with an average correlation peaking at just under 0.10. We believe the neural network's performance might have been hampered by the need for extensive data scaling, reflecting the practical challenges of applying deep learning to smaller datasets.

These experiences underscored the necessity of selecting and configuring regression models that are not only theoretically capable but also practically efficient and well-suited to the specific characteristics of the data being analyzed.

Fourth Finger Correlation:

The anatomy of the muscles and nerves in the hand explains why the flexion of the ring finger often correlates strongly with the movements of the middle and little fingers. Specifically, the ring and middle fingers cannot move as independently as the others because they share a common extensor muscle, the extensor digitorum. This shared muscle structure naturally leads to synchronized movements between these two fingers, resulting in highly correlated flexion patterns. Furthermore, the neural structure of the hand contributes to this phenomenon. The ulnar nerve, which services the little finger, the ring finger, and one half of the middle finger, splits into branches that extend to each of these fingers. The interconnected nature of these nerve branches means that signals intended for one finger can influence movement in the others, enhancing the likelihood of correlated motion between them.

Conclusion:

We were happy with the performance of our algorithm overall. We were able to pass both checkpoints with relatively good scores on the leaderboard data (correlation = 0.4853). However, there were several areas that we felt could have been improved upon in our algorithm. Primarily, we felt that we could have done more to experiment with different types of modeling. We ended up just using the xGB method after experimenting with different constants, but we felt that, given additional time, there could have been more of an opportunity for us to try combining several different machine learning methods together and testing the performance of this combined algorithm. Additionally, there may have been different options in the areas of pre-processing, post-processing, and feature extraction. While we did conduct thorough research

in these areas, there was definitely room for improvement in our execution of these methods. In the future, we hope to prepare a plan that combines several different types of algorithms and novel methods for similar projects in order to deliver better results.

References:

Korik, A., et al. “3D hand motion trajectory prediction from EEG MU and beta Bandpower.” Progress in Brain Research (2016), pp. 71–105

Merk, Timon, et al. "Machine learning based brain signal decoding for intelligent adaptive deep brain stimulation." Experimental Neurology 351 (2022): 113993.

Xie, Ziqian, Odelia Schwartz, and Abhishek Prasad. "Decoding of finger trajectory from ECoG using deep learning." Journal of neural engineering 15.3 (2018): 036009.

Yao, Lin, and Mahsa Shoaran. "Enhanced classification of individual finger movements with ECoG." 2019 53rd Asilomar Conference on Signals, Systems, and Computers. IEEE, 2019.

Appendix:

```
#load data
all_data = loadmat('truetest_data.mat')

leader1_ecog = all_data['truetest_data'][0][0]
leader2_ecog = all_data['truetest_data'][1][0]
leader3_ecog = all_data['truetest_data'][2][0]

# load models
with open('xgb_regressors1.pkl', 'rb') as f:
 xgb_regressors = pickle.load(f)

with open('xgb_regressors2.pkl', 'rb') as f:
 xgb_regressors2 = pickle.load(f)

with open('xgb_regressors3.pkl', 'rb') as f:
 xgb_regressors3 = pickle.load(f)

# hardcode shape of the data
shape_n = np.shape(leader1_ecog)[0]
```

```
from scipy.signal import filtfilt, iirnotch

def firwin_me(lowcut, highcut, fs, order=5):
   nyquist = 0.5 * fs
   low = lowcut / nyquist
   high = highcut / nyquist
   b = firwin(order, [low, high], pass_zero=False)
   return b

def filter_data(raw_eeg, fs=1000):
   lowcut = 2  # Lower cutoff frequency
   highcut = 160  # Higher cutoff frequency
   order = 5  # Filter order

   # Apply bandpass filter
   b = firwin_me(lowcut, highcut, fs, order=order)
```

```
    filtered_data = filtfilt(b, [1.0], raw_eeg, axis=0)  # Ensure filtering
across time

    # Notch filter to remove 50 Hz power line noise
    notch_freq = 50  # Frequency to remove (Hz)
    quality = 50
    b_notch, a_notch = iirnotch(notch_freq, quality, fs)
    filtered_data = filtfilt(b_notch, a_notch, filtered_data, axis=0)

    # Notch filter to remove spike frequencies
    spike_frequencies = [110.076, 0.02095, 74.1995, 38.51574774775,
101.9315, 79.65909302326]  # Spike frequencies in Hz
    for freq in spike_frequencies:
        b_notch, a_notch = iirnotch(freq, quality, fs)
        filtered_data = filtfilt(b_notch, a_notch, filtered_data, axis=0)

    return filtered_data
def NumWins(x_len, fs, winLen, winDisp):
    # Calculate how long winLen and winDisp are in samples
    winDispSamples = int(fs * winDisp)
    winLenSamples = int(fs * winLen)

    # Calculate how many windows there will be with overlap
    numWinsWithOverlap = 1 + (x_len - winLenSamples) // winDispSamples

    return numWinsWithOverlap



def avg_t_voltage(signal):
    avg_voltage = np.mean(signal)

    return avg_voltage

def signal_energy(signal):

    energy = np.sum(np.square(signal))
    return energy

def shannon_entropy(signal, fs=1000):
```

```
    # Calculate the PSD using the Welch method
    frequencies, psd = welch(signal, fs=fs)

    # Calculate the probability mass function for the signal
    prob_mass_func = psd / np.sum(psd)

    # Calculate Shannon entropy
    entropy = -np.sum(prob_mass_func * np.log2(prob_mass_func))

    return entropy
```

```
def avg_f_bandpower_and_entropy(signal, fs=1000):
    # Define frequency bands of interest
    bands = [(8, 12), (12, 30), (75, 115), (115, 150)]

    # Initialize arrays to store band power and entropy
    avg_bandpower = np.zeros(len(bands))
    band_entropy = np.zeros(len(bands))

    # Calculate the PSD using the Welch method
    frequencies, psd = welch(signal, fs=fs)

    # Iterate over each frequency band
    for i, (f_min, f_max) in enumerate(bands):
        # Find indices corresponding to the frequency range
        ind_min = np.where(frequencies >= f_min)[0][0]
        ind_max = np.where(frequencies >= f_max)[0][0]

        # Calculate and store average power in this band
        avg_bandpower[i] = np.mean(psd[ind_min:ind_max])

        # Calculate Shannon entropy in this band
        band_signal = signal[ind_min:ind_max]
        band_entropy[i] = shannon_entropy(band_signal, fs)

    return avg_bandpower, band_entropy
```

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

```
    # Determine number of windows for input x
    num_windows = NumWins(len(x), fs, winLen, winDisp)

    # Initialize empty array to contain feature data
    features = []

    # For all windows...
    for ich in range(num_windows):
        # Extract windowed segment
        window_start = int(ich * winDisp * fs)
        window_end = window_start + int(winLen * fs)
        window_segment = x[window_start:window_end]

        # Calculate feature for this windowed segment and append to
features list
        features.append(featFn(window_segment))

    # Convert features list to numpy array
    features_final = np.array(features)

    # Return determined features
    return features_final
```

```
def get_features(filtered_window, fs=1000, num_features = 10):
    """
    Calculate features for a given filtered window.

    Args:
        filtered_window (window_samples x channels): The window of the
filtered ecog signal.
        fs (int): Sampling rate.

    Returns:
        features (channels x num_features): The features calculated on each
channel for the window.
    """

    # Determine number of channels in data
    num_channels = filtered_window.shape[1]
```

```
    # Initialize empty array to contain feature data
    features = np.zeros((num_channels, num_features))

    # For each channel...
    for ich in range(num_channels):
        # Get current window
        window = filtered_window[:, ich]

        # Determine & store average time domain voltage
        avg_voltage = avg_t_voltage(window)
        features[ich, 0] = avg_voltage

        energy = signal_energy(window)
        features[ich, 1] = energy

        # cfc_value = compute_cfc(window, (8, 12), (70, 120), fs)
        # features[ich, 3] = cfc_value

        # Determine & store average frequency domain bandpowers
        avg_bandpowers, band_entropy = avg_f_bandpower_and_entropy(window,
fs)
        features[ich, 2:6] = avg_bandpowers
        features[ich, 6:10] = band_entropy

    # Return array of features
    return features
```

```
def get_windowed_feats(raw_ecog, fs, window_length, window_overlap,
num_features):
    """
    Process filtered ECoG data through the steps of filtering and feature
calculation and return features.

    Inputs:
      filtered_ecog (samples x channels): the filtered ECoG signal
      fs: the sampling rate (e.g., 1000 for this dataset)
      window_length: the window's length in seconds
      window_overlap: the window's overlap in seconds
      numFeat: the number of features calculated for each channel
```

```
    Output:
      all_feats (num_windows x (channels x features)): the features for
each channel for each time window
        note that this is a 2D array.
    """
    # Determine number of channels
    filtered_ecog = filter_data(raw_ecog)
    num_channels = filtered_ecog.shape[1]

    # Determine the number of windows
    num_windows = NumWins(len(filtered_ecog), fs, window_length,
window_overlap)

    # Determine the features for each channel for each time window
    unformatted_features = MovingWinFeats(filtered_ecog, fs, window_length,
window_overlap, get_features)

    # Initialize an empty array to contain feature data
    features = np.zeros((num_windows, num_channels * num_features))

    # Reformat determined features to correct shape
    for ich in range(num_channels):
        features[:, ich * num_features: (ich + 1) * num_features] =
unformatted_features[:, ich, :]

    # Return determined features
    return features
```

```
# define funtion to create R matrix
def create_R_matrix(features, N_wind):
    """
    Write a function to calculate the R matrix

    Input:
      features (samples (number of windows in the signal) x channels x
features):
        the features you calculated using get_windowed_feats
      N_wind: number of windows to use in the R matrix

    Output:
```

```
    R (samples x (N_wind*channels*features))
  """

  # determine number of windows
  num_windows = features.shape[0]

  # add first N_wind-1 rows to start of features matrix
  adding_rows = features[0:N_wind-1,:]
  features_modified = np.concatenate((adding_rows,features),axis=0)

  # initialize empty matrix to contain R
  R = np.zeros((num_windows,features.shape[1]*N_wind+1))
  # populate first column with value=1
  R[:,0] = 1

  # for each time window...
  for ich in range(num_windows):
    # get start and end indices for the window
    window_start = ich
    window_end = ich + N_wind
    # for every feature...
    for jch in range(features.shape[1]):
      preced_feat = features_modified[window_start:window_end,jch]
      # save features to R matrix
      R[ich,jch*N_wind+1:jch*N_wind+N_wind+1] = np.transpose(preced_feat)

  # return determined matrix containing R
  return R
```

```
def downsample_flexion_data(data_glove, num_windows):
  num_samples_per_window = len(data_glove) // num_windows
  downsampled =
np.array([np.mean(data_glove[i*num_samples_per_window:(i+1)*num_samples_pe
r_window], axis=0)
                          for i in range(num_windows)])
  return downsampled
```

```
# Define window params
```

```
fs = 1000
win_len = .9
overlap = .1
num_feat = 10
N_wind = 3
```

```
# Create feature matrices for each patient

fs1_test_final =
get_windowed_feats(leader1_ecog,fs,win_len,overlap,num_feat)
RT1_final = create_R_matrix(fs1_test_final, N_wind)

fs2_test_final = get_windowed_feats(leader2_ecog,fs,win_len,
overlap,num_feat)
RT2_final = create_R_matrix(fs2_test_final, N_wind)

fs3_test_final = get_windowed_feats(leader3_ecog,fs,win_len,
overlap,num_feat)
RT3_final = create_R_matrix(fs3_test_final, N_wind)
```

```
# Define the low-pass filter parameters
def butter_lowpass(cutoff, fs, order=5):
   nyquist = 0.5 * fs
   normal_cutoff = cutoff / nyquist
   b, a = butter(order, normal_cutoff, btype='low', analog=False)
   return b, a

def lowpass_filter(data, cutoff, fs, order=5):
   b, a = butter_lowpass(cutoff, fs, order=order)
   filtered_data = filtfilt(b, a, data)
   return filtered_data
highcut_frequency = .5

p1_XGB = np.zeros((shape_n, 5))

for finger_index, xgb_regressor in enumerate(xgb_regressors):
   predictions = xgb_regressor.predict(RT1_final)
   original_time_points = np.linspace(0, len(predictions) / 1000,
len(predictions))
```

```
   interpolation_function = CubicSpline(original_time_points, predictions)
   interpolated_predictions =
lowpass_filter(interpolation_function(np.linspace(0, len(predictions) /
1000, shape_n)),highcut_frequency,fs)
   p1_XGB[:, finger_index] = interpolated_predictions
```

```
p2_XGB = np.zeros((shape_n, 5))

for finger_index, xgb_regressor in enumerate(xgb_regressors2):
   predictions = xgb_regressor.predict(RT2_final)
   original_time_points = np.linspace(0, len(predictions) / 1000,
len(predictions))
   interpolation_function = CubicSpline(original_time_points, predictions)
   interpolated_predictions =
lowpass_filter(interpolation_function(np.linspace(0, len(predictions) /
1000, shape_n)),highcut_frequency,fs)
   p2_XGB[:, finger_index] = interpolated_predictions
```

```
p3_XGB = np.zeros((shape_n, 5))

for finger_index, xgb_regressor in enumerate(xgb_regressors3):
   predictions = xgb_regressor.predict(RT3_final)
   original_time_points = np.linspace(0, len(predictions) / 1000,
len(predictions))
   interpolation_function = CubicSpline(original_time_points, predictions)
   interpolated_predictions =
lowpass_filter(interpolation_function(np.linspace(0, len(predictions) /
1000, shape_n)),highcut_frequency,fs)
   p3_XGB[:, finger_index] = interpolated_predictions
```

```
pred_XGB = np.zeros((3,1), dtype=object)
pred_XGB[0,0] = p1_XGB
pred_XGB[1,0] = p2_XGB
pred_XGB[2,0] = p3_XGB

savemat('predictions.mat', {'predicted_dg':pred_XGB})
```