Predicting Finger Flexion

Everyday We Litt: Arjun Kanthawar, Desmond Young
University of Pennsylvania
April 28, 2022

Algorithm Summary

Our algorithm can be broken down into four steps: Pre-Processing, Feature Extraction, Modeling/Prediction, and Post Processing. We first eliminated noisy channels and used a common average reference montage for each patient. We then extracted 9 features across 100ms time windows in our electrocorticography (ECoG) data.[1] After extracting the features, we used a ridge regression model to predict finger flexions. Finally, we post-processed our predictions to minimize noise.

Algorithm Details

We first decided to clean our ECoG signal to achieve better model predictions. The first step in our preprocessing pipeline was removing channel 54 in patient 1 and channels 20 and 37 in patient 2. Previous research has shown that these channels have large amounts of noise and are better off being removed from the dataset.[2] Our second pre-processing step involved using a common average reference (CAR) montage for each patient. Simply speaking, (CAR) subtracts the average value at a time point across all channels from each individual channel. This method has been shown to improve model predictions.[2,3] Finally, we filtered the data using a 5th-order bandpass filter with bounds at 1 and 200 Hz and a notch filter at 60 Hz.[2] The bandpass filter has been used in literature, and the notch filter was implemented at 60 Hz because that is the frequency of AC current.[2]

After filtering our data, we extracted nine features across 100ms time windows in our ECoG data. These features were: line length, signal area, signal voltage, bandpower across 5 frequency ranges (5-15 Hz, 20-25 Hz, 75-115 Hz, 125-160 Hz, and 160-175 Hz) and signal variance. Each of these features has been shown to be helpful in decoding finger flexion in literature or is a commonly extracted feature in BCI applications.[4]

After feature extraction, we created a response matrix.[5] In this approach, our response matrix is composed of row vectors corresponding to each time window where a given row vector contains features for all of the ECoG channels over the preceding 14 time bins. 14 was chosen as the number of

preceding time bins through cross-validation.

After calculating our response matrix, we scaled our feature matrix. The results of ridge regression are heavily dependent on feature magnitudes. Therefore, we scale the features to zero mean and unit variance to avoid incorrect results that can arise due to differences in magnitude across different features. Finally, we used ridge regression on the scaled features to predict finger flexion in the five fingers. This involved training our ridge model on the scaled training ECoG and finger flexion data and then predicting finger flexion on a scaled testing set.

To tune the regularization hyperparameter (alpha) we used 5-fold cross-validation. The average correlation found from the validation was plotted for various alpha values.

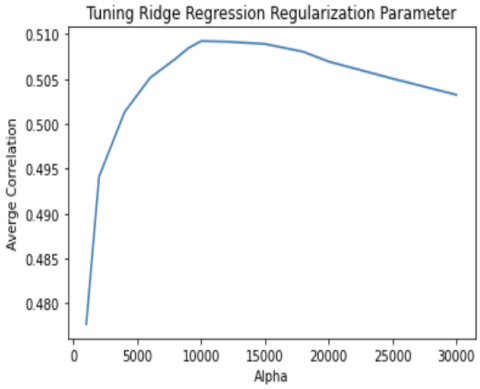
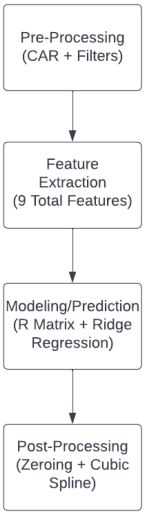


Figure 1: Tuning Curve for Ridge Regression Regularization Parameter

As shown in the plot above, an alpha value of 10000 gave us the highest validation score. Therefore, this was set as the hyperparameter for ridge regression. Note that the plot above is just an example of one of our hyperparameter tuning plots. More values for alpha were tested and 10000 was still shown to be the optimal value.

Finally, post-processing involved up-sampling our flexion data to fit our export needs using cubic splines and setting all negative flexion values equal to zero. Setting negative values equal to zero seemed odd to us, but we consistently saw an improvement in our results when we implemented this. We believe that this technique reduced noise in our predictions. Overall, this algorithm allowed us to achieve a maximum average correlation of 0.5182 with the leaderboard data.

Flow Chart of Algorithm



Other Attempted Methods

One preprocessing technique used in literature involves separating the ECoG signal into three different frequency bands (0-60 Hz, 60-100 Hz, 100-200 Hz) based on different ECoG classifications of signals in those bands.[6] After implementing this, the results were not as promising as we had hoped. We were only able to achieve correlations of ~0.48, slightly lower than what our other models had produced. After this result, we abandoned this technique in favor of other preprocessing steps.

Although our final model contained 9 features, we tried using a subset of those features. Namely, we tried 2 different sets of features. The first set had six features: line length, area, voltage, and bandpower across three frequency ranges (75-115 Hz, 125-160 Hz, and 160-175 Hz). The second set had seven features: signal voltage, variance, and bandpower across five frequency ranges (5-15 Hz, 20-25 Hz, 75-115 Hz, 125-160 Hz, and 160-175 Hz).[4] Both of these models gave slightly worse results than the nine feature model.

To optimize our feature matrix, we tried to use principal component analysis (PCA) to eliminate unnecessary features. We set thresholds of 95% and 99% variance for our model features and then ran our regression models. We consistently obtained better results on our leaderboard data when we used our full feature set rather than the reduced set from either 95% or 99% PCA, so we discarded this method.

Numerous supervised learning models were tested before we settled on ridge regression. We started with the optimal linear decoder method that was outlined in Warland et. al., (1997) which gave us average correlations of ~0.35. To improve on this model, we implemented lasso, ridge, and elastic net regression. For all three models, a grid search with 5-fold cross-validation was used to tune hyperparameters (alpha for all three models and also L_1 ratio for elastic net). Similar to ridge regression, both lasso regression and elastic net are heavily dependent on the magnitude of the features so the data was scaled before using these methods. Lasso regression provided only slightly better results than the linear filter. We were surprised to see that the elastic net model also provided only marginally better results than the linear filter and far worse results than ridge regression.

We next attempted to use k-nearest neighbors (k-nn), support vector regression, and random forest regression. K-nn gave us extremely large correlations (>0.6) when we trained on a subset of our raw data and then tested on a different subset. However, our correlation scores were very small on the leaderboard data (<0.1). K-nn assumes that our testing data is extremely similar to the

training data. Since we were unsure of what the leaderboard data looked like and our preliminary results were not promising, we decided to abandon this method.

The regularization parameter for support vector regression (SVR) was tuned through a grid search 5-fold cross validation. Although this method showed a very high correlation after training and testing on subsets of the raw data (~0.5), the correlation on the leaderboard data was ~0.35. A major issue with SVR is that it is only able to predict one variable at a time, so we were forced to predict the flexion for each finger individually. This does not allow us to take into account correlations between different fingers (ex. thumb moves slightly when index finger moves). Because of this, we abandoned SVR.

One major issue with random forest regression was the amount of time it took to tune the hyperparameters. It would take close to two hours to run a grid search on just a few hyperparameters in the model. After running a few of these validation searches, the model performed at the same level as lasso and elastic net regression. Due to these computational issues and the lack of promising results, we chose to abandon this method.

After modeling, we saw that our predictions had a smaller amplitude than the true flexions. In addition, it seemed that our predictions had more noise. We created thresholds to both increase spike amplitudes and decrease noise through scaling factors. This would drastically improve our results when we trained and tested on different subsets of the raw data, but our results always became worse when we tested on the leaderboard data. We later attempted to use a smoothing filter to reduce the noise in our data but this also proved to only worsen our scores.

Fourth Finger Correlation

The muscular anatomy and neural anatomy of the hand provide insight into why the fourth (ring) finger had highly correlated flexion with the third (middle) and fifth (little) fingers. Taking a look at the muscular anatomy of the finger reveals that the little finger, pointer finger and thumb all have separate extensor muscles allowing them to move independently from the rest of the hand. The ring and middle finger, however, share the same extensor called the extensor digitorum. This makes it hard for the ring and middle finger to move independently of each other, and therefore, their flexion is highly correlated.

In addition to muscular anatomy, neural anatomy can also explain the correlation between these fingers. The ulnar nerve connects with the little, the ring and one side of the middle finger. The nerve branches off into each individual finger, but

the fact that they are intertwined with each other increases the probability that nerve signals heading toward one finger might cause movement in the other two fingers.

Conclusion

Overall, we were happy with the results of our project. We were able to achieve a relatively good score with the leaderboard data (correlation = 0.5182, 7th place), and we felt that we had tried many different techniques to improve our models. In the areas of pre-processing, post-processing and feature extraction, we feel that there may have been more options and techniques that we could have explored. We conducted thorough research in these areas, but there was definitely some room for improvement. From a feature extraction perspective, we used the response matrix throughout almost all of our techniques. It would have been interesting to see what our results would have been if we did not use this method. Finally, for modeling, we feel that we exhausted a lot of options but it would have been interesting to see the results of combining different models and potentially using deep learning if we had more time.

We were slightly frustrated to see that we did not get a prediction on the hidden dataset. We believe it is because we had hardcoded the length of the leaderboard data when we did our cubic splines upsampling at the very end of our algorithm. However, we are still pleased with this project and feel that we have learned a significant amount about model prediction. Going forward, we will be able to better prepare a plan for projects like this to be more efficient and deliver better results.

References

1) Schalk, G., Kubanek, J., Miller, K.J., Anderson, N.R.,Leuthardt, E.C., Ojemann, J.G., Limbrick, D., Moran, D.W., Gerhardt, L.A., and Wolpaw, J.R. (2007). Decoding Two-Dimensional Movement Trajectories Using Electrocorticographic Signals in Humans, *J Neural Eng,* 4: 264-275,
2) Xie, Z., Schwartz, O., & Prasad, A. (2018). Decoding of finger trajectory from ECoG using deep learning. *Journal of neural engineering*, *15*(3), 036009. https://doi.org/10.1088/1741-2552/aa9dbe
3) Chen, W., Liu, X., Litt, B. (2014). Logistic-weighted regression improves decoding of finger flexion from electrocorticographic signals. 2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2014. 2014. 2629-32. 10.1109/EMBC.2014.6944162.
4) Kubánek, J., Miller, K. J., Ojemann, J. G., Wolpaw, J. R., & Schalk, G. (2009). Decoding flexion of individual fingers using electrocorticographic signals in humans. *Journal of neural engineering*, *6*(6), 066001. https://doi.org/10.1088/1741-2560/6/6/066001
5) Warland D.K., Reinagel, P., and Meister, M. (1997) Decoding visual information from a population of retinal ganglion cells. *J Neurophysiol 78:2336-2350.*
6) Liang, N., & Bougrain, L. (2012). Decoding Finger Flexion from Band-Specific ECoG Signals in Humans. Frontiers in neuroscience, 6, 91. https://doi.org/10.3389/fnins.2012.00091

Appendix

```
# 1. Load Packages and Data
```

```
#Set up the notebook environment
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pickle
from scipy.stats import pearsonr
from scipy import signal as sig
import scipy.io
import zipfile
import pickle
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
```

```
with zipfile.ZipFile('/content/final_project.zip', 'r') as zip_ref:
   zip_ref.extractall('/content')
```

```
with open('final_project/Ridge_patient1.pkl', 'rb') as f:
   ridge_pat1 = pickle.load(f)

with open('final_project/Ridge_patient2.pkl', 'rb') as f:
   ridge_pat2 = pickle.load(f)

with open('final_project/Ridge_patient3.pkl', 'rb') as f:
   ridge_pat3 = pickle.load(f)

with open('final_project/R1_raw.pkl', 'rb') as f:
   R1_raw = pickle.load(f)

with open('final_project/R2_raw.pkl', 'rb') as f:
   R2_raw = pickle.load(f)

with open('final_project/R3_raw.pkl', 'rb') as f:
   R3_raw = pickle.load(f)
```

```
# Leaderboard Data
dataLeader = scipy.io.loadmat('truetest_data.mat')
patient1 = np.delete(dataLeader['truetest_data'][0][0],54,axis=1)
patient2 = np.delete(dataLeader['truetest_data'][1][0],[20,37],axis=1)
patient3 = dataLeader['truetest_data'][2][0]
```

2. Data Pre-Processing and Filtering

- Pre-process data with Common Average Reference (CAR)
- Filter Data

```
def CAR(patient):
 for i in range(len(patient)):

   commonAverage = np.mean(patient[i,:])
   patient[i,:] = [j-commonAverage for j in patient[i,:]]

 return patient
```

```
patient1_CAR = CAR(patient1)
patient2_CAR = CAR(patient2)
patient3_CAR = CAR(patient3)
```

```
def SigFilter(raw,fs):
 ze = np.zeros((len(raw[:,0]),len(raw[0,:])))
 fin = np.zeros((len(raw[:,0]),len(raw[0,:])))
 b,a = sig.butter(5,(1,200),btype='bandpass',fs=fs)
 b1, a1 = sig.iirnotch(60, 30, fs)
 for i in range(len(raw[0,:])):
   ze[:,i] = sig.filtfilt(b,a,raw[:,i])
   fin[:,i] = sig.filtfilt(b1,a1,ze[:,i])
 return fin
```

```
patient1_CAR = SigFilter(patient1_CAR,1000)
patient2_CAR = SigFilter(patient2_CAR,1000)
```

```
patient3_CAR = SigFilter(patient3_CAR,1000)
```

3. Define Features

9 features used:

- Line Length
- Area
- Average Voltage
- Variance
- Bandpower (5-15 Hz)
- Bandpower (20-25 Hz)
- Bandpower (75-115 Hz)
- Bandpower (125-160 Hz)
- Bandpower (160-175 Hz)

get_features function calculates features in a given time window

```
def NumWins(x,fs,winLen,winDisp):
 siglength = len(x) / fs
 winNums, _ = divmod((siglength - winLen), winDisp)
 return winNums + 2
```

```
def LL(x):
 return np.sum(np.absolute(np.ediff1d(x)))
```

```
def Area(x):
 return np.sum(np.abs(x))
```

```
def Var(x):
 return np.var(x)
```

```
def Amp_5_15(x,fs):
 b,a = sig.butter(5,(5,15),btype='bandpass',fs=fs)
 signal1 = sig.filtfilt(b,a,x)
 yy = scipy.fft.fft(signal1)
 return np.sum(np.abs(yy))
```

```
def Amp_20_25(x,fs):
 b,a = sig.butter(5,(20,25),btype='bandpass',fs=fs)
 signal1 = sig.filtfilt(b,a,x)
```

```
 yy = scipy.fft.fft(signal1)
 return np.sum(np.abs(yy))
```

```
def Amp_75_115(x,fs):
 b,a = sig.butter(5,(75,115),btype='bandpass',fs=fs)
 signal1 = sig.filtfilt(b,a,x)
 yy = scipy.fft.fft(signal1)
 return np.sum(np.abs(yy))
```

```
def Amp_125_160(x,fs):
 b,a = sig.butter(5,(125,160),btype='bandpass',fs=fs)
 signal1 = sig.filtfilt(b,a,x)
 yy = scipy.fft.fft(signal1)
 return np.sum(np.abs(yy))
```

```
def Amp_160_175(x,fs):
 b,a = sig.butter(5,(160,175),btype='bandpass',fs=fs)
 signal1 = sig.filtfilt(b,a,x)
 yy = scipy.fft.fft(signal1)
 return np.sum(np.abs(yy))
```

```
def get_features(filtered_window, fs=1000):
 """
   Write a function that calculates features for a given filtered window.
   Feel free to use features you have seen before in this class, features that
   have been used in the literature, or design your own!

   Input:
     filtered_window (window_samples x channels): the window of the filtered ecog
signal
     fs: sampling rate
   Output:
     features (channels x num_features): the features calculated on each channel for
the window
 """
 window_mat = np.zeros((1,len(filtered_window[0,:])*9))
 for i in range(len(filtered_window[0,:])):
   window = filtered_window[:,i]

   window_mat[0,i*6] = LL(window)
   window_mat[0,i*6+1] = Area(window)
   window_mat[0,i*6 + 2] = Amp_75_115(window,fs)
```

```
    window_mat[0,i*6 + 3] = Amp_125_160(window,fs)
    window_mat[0,i*6 + 4] = Amp_160_175(window,fs)
    window_mat[0,i*6+5] = Voltage(window)
    window_mat[0,i*6+6] = Amp_5_15(window,fs)
    window_mat[0,i*6 + 7] = Amp_20_25(window,fs)
    window_mat[0,i*6 + 8] = Var(window)



    #window_mat[0,i*7 + 5] = LMP(window)
    #window_mat[0,i*7 + 6] = Var(window)
  return window_mat
```

4. Compute Features

```
def get_windowed_feats(raw_ecog, fs, window_length, window_overlap):
  """
    Write a function which processes data through the steps of filtering and
    feature calculation and returns features. Points will be awarded for completing
    each step appropriately (note that if one of the functions you call within this
script
    returns a bad output, you won't be double penalized). Note that you will need
    to run the filter_data and get_features functions within this function.

    Inputs:
      raw_eeg (samples x channels): the raw signal
      fs: the sampling rate (1000 for this dataset)
      window_length: the window's length
      window_overlap: the window's overlap
    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.
  """
  windows = NumWins(raw_ecog,fs,window_length,window_overlap)
  window_features = np.zeros((int(windows),len(raw_ecog[0,:]) * 9))
  for i in range(0,int(windows)):
    front = int(i * window_overlap * fs)
    end = int(((i * window_overlap) + window_length) * fs)
    win = raw_ecog[front:end,:]
    feats = get_features(win, fs = fs)
    window_features[i,:] = feats
  return window_features
```

```
data1_CAR = get_windowed_feats(patient1_CAR,1000,0.1,0.05)
data2_CAR = get_windowed_feats(patient2_CAR,1000,0.1,0.05)
data3_CAR = get_windowed_feats(patient3_CAR,1000,0.1,0.05)
```

5. Response 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))
 """
 newFeat  = np.vstack([features[0:N_wind-1],features])
 samples = len(features)
 R = np.zeros((samples,len(newFeat[0])*N_wind))
 gap = 0
 chan = 0

 for i in range(len(newFeat[0])*N_wind):
   ft_samp = newFeat[:,chan]
   R[:,i] = ft_samp[gap:samples+gap]
   gap = gap+1

   if gap == N_wind:
     gap = 0
     chan = chan + 1 # new channel

  R = np.hstack([np.ones((samples,1)), R])

 return R
```

```
# Leaderboard data
R1 = create_R_matrix(data1_CAR,14)
R2 = create_R_matrix(data2_CAR,14)
R3 = create_R_matrix(data3_CAR,14)
```

6. ML Training and Testing

Here, we will use the R matrix with Ridge regression to predict finger flexions.

Ridge Regression

```
# Pre-processing for Ridge Regression

sc1 = StandardScaler()
sc1.fit(R1_raw)
test1_transform = sc1.transform(R1)



sc2 = StandardScaler()
sc2.fit(R2_raw)
test2_transform = sc2.transform(R2)

sc3 = StandardScaler()
sc3.fit(R3_raw)
test3_transform = sc3.transform(R3)
```

```
# Predictions
ypred1 = ridge_pat1.predict(test1_transform)
ypred2 = ridge_pat2.predict(test2_transform)
ypred3 = ridge_pat3.predict(test3_transform)
```

Post-Processing and Export Predictions

```
# Set all negative values to 0
for i in range(len(ypred1)):
 for j in range(5):
   if ypred1[i,j]<0:
     ypred1[i,j] = 0
   if ypred2[i,j]<0:
     ypred2[i,j] = 0
   if ypred3[i,j]<0:
     ypred3[i,j] = 0
```

```
subs = [ypred1,ypred2,ypred3]
```

```
subs_clean = np.zeros((115500,15))
for i in range(len(subs)):
 for j in range(len(subs[0][0,:])):
   #  subs_clean[:,j + i*5] = scipy.signal.resample(subs[i][:,j],115500)
    splined =
scipy.interpolate.CubicSpline(np.arange(0,len(subs[i][:,j]),1),subs[i][:,j])
    x = np.arange(0,2949,2949/115500)
    subs_clean[:,j + i*5] = splined(x)
```

```
import numpy as np
from scipy.io import savemat
#create an example submission array
predictions = np.zeros((3,1), dtype=object)
predictions[0,0] = subs_clean[:,0:5]
predictions[1,0] = subs_clean[:,5:10]
predictions[2,0] = subs_clean[:,10:15]
#save the array using the right format
savemat('predictions.mat', {'predicted_dg':predictions})
```