BIOMEDE 517 - Neural Engineering - Labs 7, 8, and 9 - Dr. Cindy Chestek
Machine Learning and Computational Analysis of Neural Data
Kushal Jaligama

1 - Introduction

Patients that are diagnosed with ALS slowly lose motor function, leading to an inability to interact with their environment. This can be aided using computer systems and electronics that detect neural signals and translate them to commands for an external device such as a prosthetic or as rudimentary as motion of a cursor on a screen. In general, it is useful to gather, decode, and analyze signals from the brain in order to provide an alternative interaction method for the user and also aid in the prediction of future actions. These neural signals can be classified as various types of activity using multiple machine learning methods, all of which generate varying degrees of accuracy in predicting and identifying what a certain neural signal represents. This paper compares a plethora of supervised machine learning methods, which draw conclusions about labeled data, to analyze monkey target reaches and human finger movements. Comparing these helps reveal which type of classification or decoding mechanisms are the best for a given situation.

---

2 - Methods

2.1 - Assumptions of Algorithms

2.1.1 - Naïve Bayes Classification

When observations in a dataset are known to be independent, one can utilize the Naïve Bayes Classifier, a generative classifier, to create a probabilistic model of the distributions within the data. This technique assumes that in a given class, different features of a sample are independent from one another and can natively decode a set of input features into an output class given the distributions of the features.

2.1.2 - Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis separates clusters of data with a hyperplane, with each side of the hyperplane representing a different class. Data samples are assigned to the cluster with the nearest mean that is weighted by the prior probability of the class and normalized to the covariance of the samples. Essentially, this classifier gathers the multivariate distribution of each class and estimates the likelihood of a sample falling into that distribution.

2.1.3 - Support-Vector Machines (SVM)

Support Vector Machines work similarly to LDA, in that they also separate data with hyperplanes, but do not rely on probability distributions. Linear SVMs can separate data based on the locations of data points closest to the maximal separation of binary data clusters. These are known as support vectors, and the SVM finds the largest separation between two margins defined by these vectors. Linear SVMs are useful for binary classification, whereas non-Linear SVMs can embed the data into higher dimensions using the kernel trick to find linear separations between multiple classes, but still only provide binary outputs. To classify multiple data into multiple classes, one can test multiple binary SVMs against each other to determine which one class a sample falls into and which ones it does not.

2.1.4 - Linear Regression

Linear regression assumes that the dataset being decoded has a linear relationship

between its observation and outcome of observation. These solve the objective function (apdx: Table 4, #1). This model works for data that have independent and identically distributed observations, have outcomes are normally distributed, and has a constant variance across observations.

2.1.5 - Ridge Regression

Ridge regression works similarly to linear regression in that both try to create a linear fit on the data, but in its objective function, ridge regression utilizes a normalization term, lambda, that penalizes large values in the weight matrix (apdx: Table 5, #1). This ensures that the algorithm does not overfit to outliers in the data, such as cells that fire spontaneously.

2.1.6 - LASSO

The LASSO algorithm decodes a stream of data similarly to ridge regression, making the change that a L1 norm of the weights matrix instead of L2 norm. The L1 norm of weights results in a non-differentiable form and causes a change in the decoder output.

2.1.7 - Kalman Filter

Decoders such as linear regression, ridge regression, and LASSO make predictions based on the input data. Kalman filters utilize a weighted combination of neural data and simple physics, taking into account the physics parameters of an object at the current and immediately previous time point to make a prediction of a particle's motion.

2.2 - Implementation of Algorithms

All of the machine learning methods used in this laboratory were implemented using MATLAB. All mean squared errors described in this section are sums of the errors across each parameter to simplify interpretation.

2.2.1 - Naïve Bayes Classifier

To analyze monkey reaches with Naïve Bayes, one begins by splitting the data into a training and test dataset. This data describes 95 neurons that have varying firing rates based on one of 8 directions that a monkey reaches, with 182 samples for each neuron-direction combination. Assuming a Poisson distribution, the mean of each feature-class pair was used to calculate the probability density of a sample corresponding to all the reach directions. Summing the logarithmic probabilities for each reach direction (apdx: Table 1, #1) mitigates floating point errors and yields a scalar value, the largest of which corresponds to the predicted direction.

2.2.2 - Linear Discriminant Analysis (LDA)

This classifier utilized ECoG data containing average power values labeled with finger movements. This data was classified with a built-in MATLAB function that made an LDA prediction about a test trial, which utilize the prior probability of a class andthe probability density of a trial in a multivariate Gaussian distribution (apdx: Table 2, #1). Leave-one-out cross-validation was used to assess the accuracy of this model.

2.2.3 - Support-Vector Machines (SVM)

Initially, the classes of ECoG data were restructured manually to create an Nx5 matrix of binary labels (apdx, Table 3, #1). Individual columns of this matrix were used as labels to classify the data using leave-one-out cross validation and input into a MATLAB function that created a

linear SVM model. A model was created for each column to gather classifications for each group. This process was repeated with non-linear SVMs. After manually classifying multiple classes, several different multi-class SVMs built into MATLAB were used, comparing one binary SVM versus all others, comparing one binary SVM to each other SVM, a ternary complete comparison ((3^n - 2^(n-1) + 1) / 2 binary SVMs), and an ordinal comparison (n - 1 binary SVMs).

2.2.4 - Linear Regression

Linear regression utilizes a decoder matrix to decode data and make estimates about the input. This matrix can be solved for using (apdx: Table 4, #2), where x represents hand position and y represents neural firing rates. Multiplying this matrix by the neural firing rate test dataset results in predicted hand position and velocities, and the accuracy of this model can be assessed by calculating the mean squared error (apdx: Table 4, #3) and correlation coefficients of the predictions (apdx: Table 4, #4).

2.2.5 - Ridge Regression

Ridge regression also utilizes a decoder matrix to make estimates about an input dataset and includes a lambda term multiplied by the identity matrix to the transpose of neural data multiplied by itself (apdx: Table 5, #1, #2). This term helps reduce overfitting. In this section, 15 different lambda values were used to determine which value would provide the best decoding accuracy, and the best lambda value produces the lowest mean squared error (apdx: Table 5, #4).

2.2.6 - LASSO

A decoder was trained using MATLAB's built-in lasso() function and the lambda value that provided the most accurate results with a ridge regression decoder.

2.2.7 - Kalman Filters

Kalman filters make estimations of the future based on what was happening just before to the variables involved. The predict, innovate, and update process was implemented in MATLAB, using an estimation of the physics of a particle along with a linear decoder and their associated noises. Predictions were made using a Kalman gain, which describes the best proportion of neural firing rate estimates compared to the best guess of the physics of the particle. The parameters of the Kalman filter were gathered using training data and predictions were made on test data.

---

3 - Results

3.1 - Outcomes of Algorithms

3.1.1 - Naïve Bayes Classifier

Using a Poisson distribution with the mean of a 91 sample training dataset causes the classifier to yield a classification accuracy of 97.39% correct predictions on the test data. This means that Naïve Bayes gives results far better than assigning 1 of 8 classes to a sample by chance (12.5%). To confirm the accuracy of Naïve Bayes, the classifier was also run on a spoofed dataset with equal size and distribution of the original. Using the mean of the entire original dataset to calculate probabilities for feature-class pairs resulted in a 99.66% accuracy of predicting which class spoofed firing rates belong to.

3.1.2 - Linear Discriminant Analysis (LDA)

The LDA classifier produced predictions that were 71.79% accurate, over the ten leave-one-out predictions. The accuracy and error of the classifier can be visualized by a confusion matrix (apdx: Figure 1), which shows on its diagonal entries when a finger was correctly classified as itself. The off-diagonal entries indicate occurrences of finger movements being classified as another finger.

3.1.3 - Support-Vector Machines (SVM)

Each method of classification using SVMs results in varying degrees of accuracy. These are listed in (apdx: Table 3), and the most accurate method of classification over the entire dataset arose with manually separated binary labels and a linear SVM at 88.72%. However, the best accuracy for a single class comes from manually separated binary labels running through a non-linear SVM at 94.87% for class 5 (ring and pinkie finger movements). The performance of SVMs is visualized using confusion matrices (apdx: Figure 2) that display the number of occurrences a finger was classified as itself along the diagonal and occurences of incorrect classifications in off-diagonal cells.

3.1.4 - Linear Regression

The mean squared error gathered from linear regression is shown in (apdx: Table 3, #3). The correlation coefficients for this data (apdx: Table 4, #4) reveal that the linear decoder's predictions for hand position and velocity of a test dataset based on a training dataset of neural firing rates and hand position and velocity are highly correlated with the actual position and velocities of the test dataset.

3.1.5 - Ridge Regression

Ridge regression utilizes a lambda term as a normalization step to eliminate outliers, and the optimal lambda value that produced the least error in predictions was 0.0286. Ridge regression produced similar correlation coefficients to that of linear regression (apdx: Table 5, #5), and the mean squared error was slightly lower than that of linear regression (apdx: Table 5, #4). One can see that this lambda value arises when the mean squared error is at its minimum (apdx: Figure 3).

3.1.6 - LASSO

LASSO produced results (apdx: Table 6) that were similar to that of Ridge Regression, as indicated by the similar correlation coefficients. The intermediate calculations of LASSO cause predictions to be calculated differently, but the accuracy of this model is extremely similar to Ridge Regression.

3.1.7 - Kalman Filters

Kalman filtering of neural firing rates produced predictions (apdx: Table 7) that were slightly worse than that of LASSO, as indicated by the lower correlation coefficients. Kalman Filtering utilizes accumulation of a particle's state to make its estimations, so it is a fair hypothesis to say that Kalman Filtering could be an extremely accurate way to decode neural signals, given enough training data.

---

4 - Discussion

Machine learning is an extremely powerful tool to analyze neural data, as making accurate and fast predictions is vital to provide the correct types of neural stimulation or other treatment for patients with neurological disorders. Kalman filters

and LASSO provided very accurate results,
but neural networks may outshine the rest.

References

Chestek C A, Gilja V, Blabe C H, Foster B L, Shenoy K V, Parvizi J and Henderson J M 2013 Hand posture classification using electrocorticography signals in the gamma band over human sensorimotor brain areas *Journal of Neural Engineering* **10** 026002

Pistohl, Tobias, et al. “Decoding Natural Grasp Types from Human ECoG.” *NeuroImage*, vol. 59, no. 1, 2012, pp. 248–260., doi:10.1016/j.neuroimage.2011.06.084.

Williams, Justin C, et al. “Complex Impedance Spectroscopy for Monitoring Tissue Responses to Inserted Neural Implants.” *Journal of Neural Engineering*, vol. 4, no. 4, 2007, pp. 410–423., doi:10.1088/1741-2560/4/4/007.

BIOMEDE 517 - Neural Engineering - Labs 7, 8, and 9 - Dr. Cindy Chestek

Appendix

Kushal Jaligama

Table 1 - Naïve Bayes Classification Equations and Results

#	Equation Name	Equation
1	Sum of Logarithmic Probabilities for Feature-Class Pairs	$\sum_{j=1}^{D} \log \hat{g}_{k=1}^{(j)}(X_i)$
#	Result	Value
2	Classification Accuracy of Test Data using Training Data Mean	97.39%
3	Classification Accuracy of Spoofed Data using Full Dataset Mean	99.66%

Table 2 - Linear Discriminant Analysis Equations and Results

#	Equation Name	Equation
1	Objective Function for Linear Discriminant Analysis	$\hat{f}(x) = \arg\max_k [\hat{\pi}_k * \phi(x; \hat{\mu}_k, \hat{\Sigma})]$
#	Result	Value
2	Classification Accuracy with Leave-One-Out Cross Validation	71.79%

Figure 1 - Confusion Matrix for Linear Discriminant Analysis (LDA)

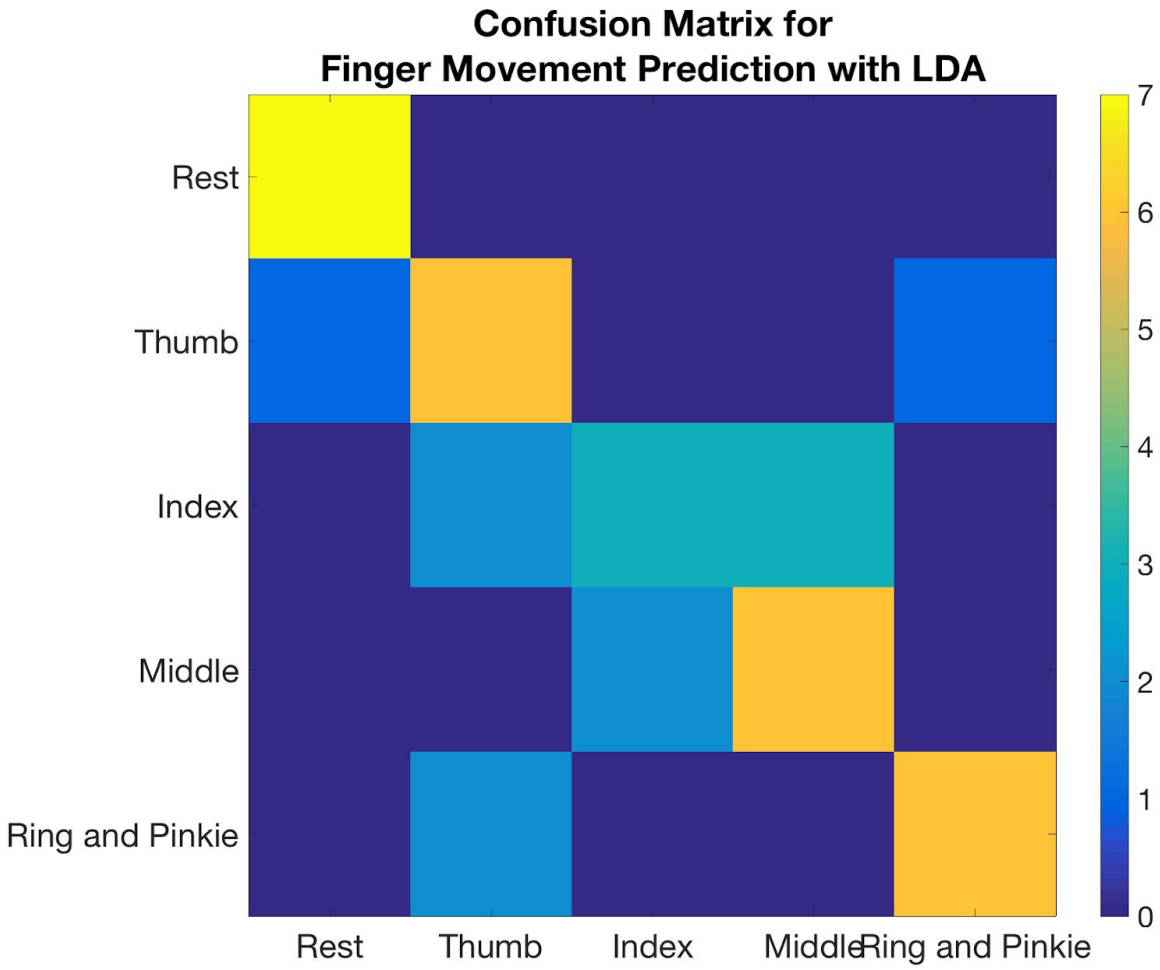
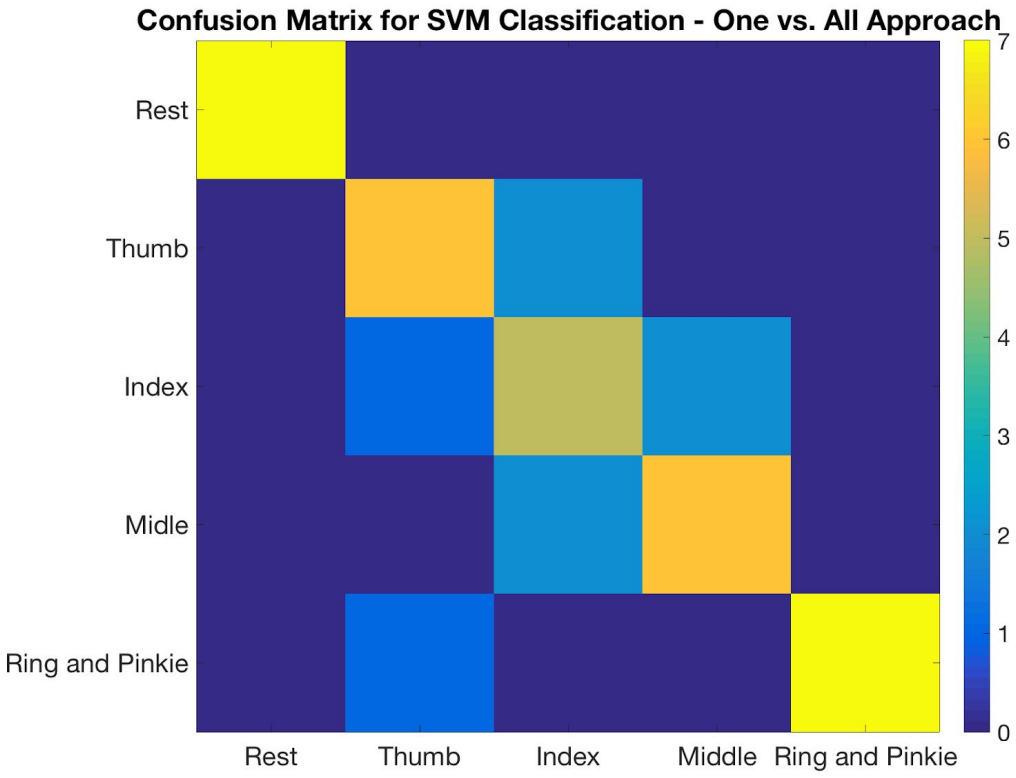


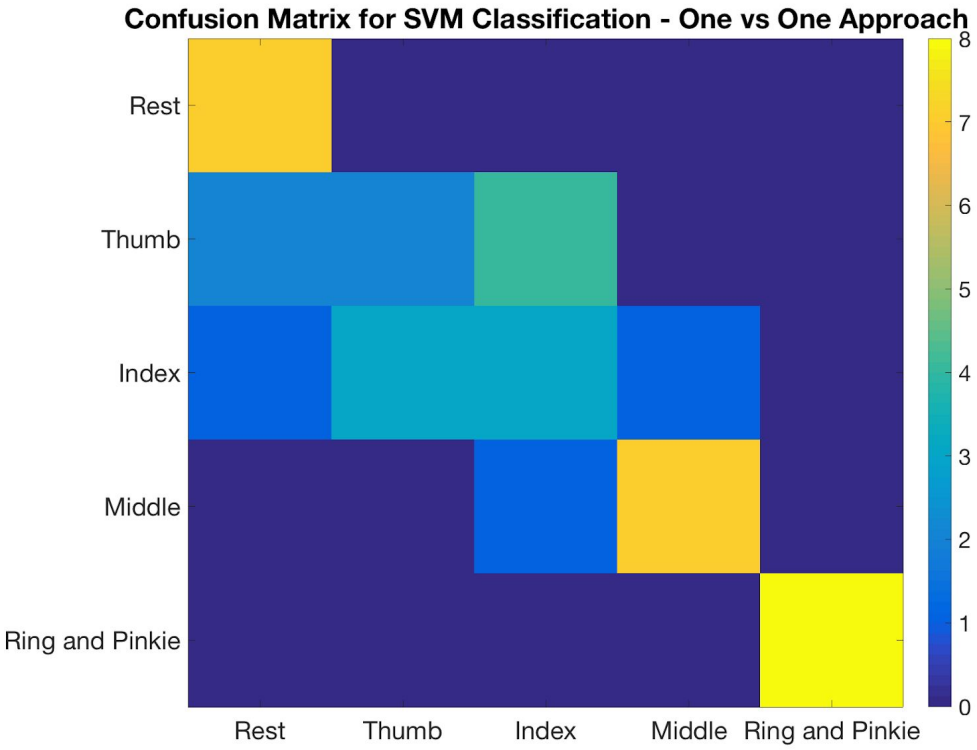
Table 3 - Support Vector Machines Equations and Results

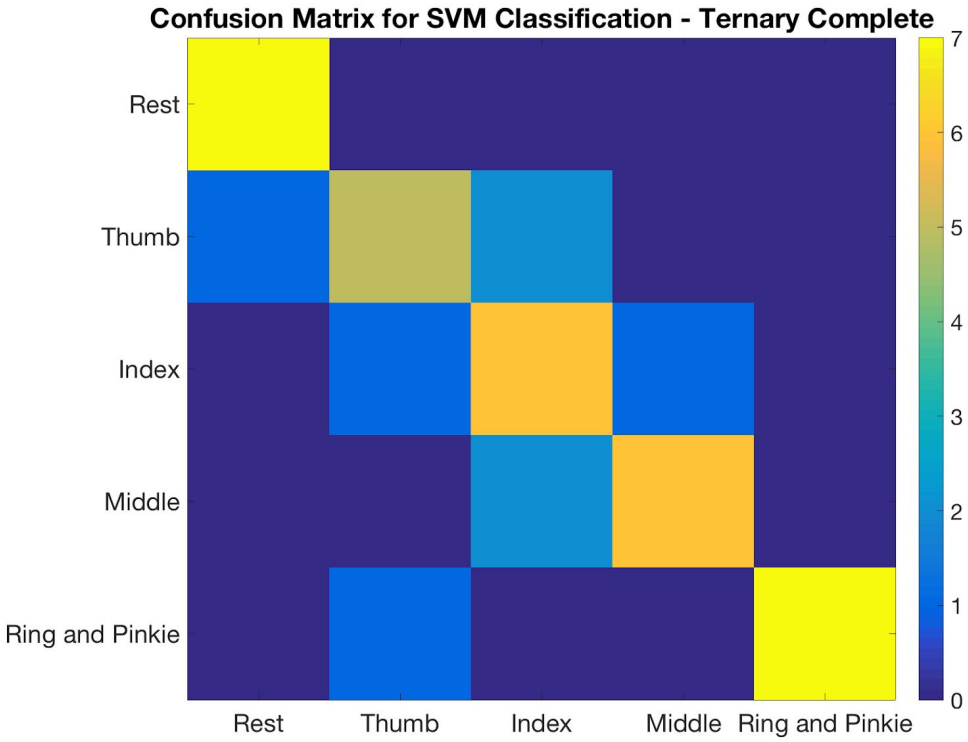
#	Equation Name	Equation
1	Restructuring of Multiple Classes to Binary Labels	$Y_{i,j} = \begin{cases} 1, & if\ observation\ i\ belongs\ to\ group\ j \\ 0, & if\ observation\ i\ does\ not\ belong\ to\ group\ j \end{cases}$
#	SVM Configuration	Accuracy
2	Manually Separated Binary Labels - Linear SVM	By Class Accuracy {class:accuracy} = {1:100%}, {2:79.49%}, {3:76.92%}, {4:89.74%}, {5:9744%}; Overall Accuracy = 88.72%

3	Manually Separated Binary Labels - Non-Linear SVM	By Class Accuracy {class:accuracy} = {1:89.74%}, {2:79.49%}, {3:79.49%}, {4:87.18%}, {5:94.87%}; Overall Accuracy = 85.15%
4	Multiclass SVM Model - One vs. All	Accuracy = 69.23%
5	Multiclass SVM Model - One vs. One	Accuracy = 79.49%
6	Multiclass SVM Model - Ternary Complete	Accuracy = 79.49%
7	Multiclass SVM Model - Ordinal	Accuracy = 79.49%

Figure 2 - Confusion Matrices for Multiclass Support Vector Machines




Confusion Matrix for SVM Classification - One vs One Approach
Rest
Thumb
Index
Middle
Ring and Pinkie
Rest
Thumb
Index
Middle
Ring and Pinkie
8
7
6
5
4
3
2
1
0


Confusion Matrix for SVM Classification - Ternary Complete
Rest
Thumb
Index
Middle
Ring and Pinkie
Rest
Thumb
Index
Middle
Ring and Pinkie
7
6
5
4
3
2
1
0

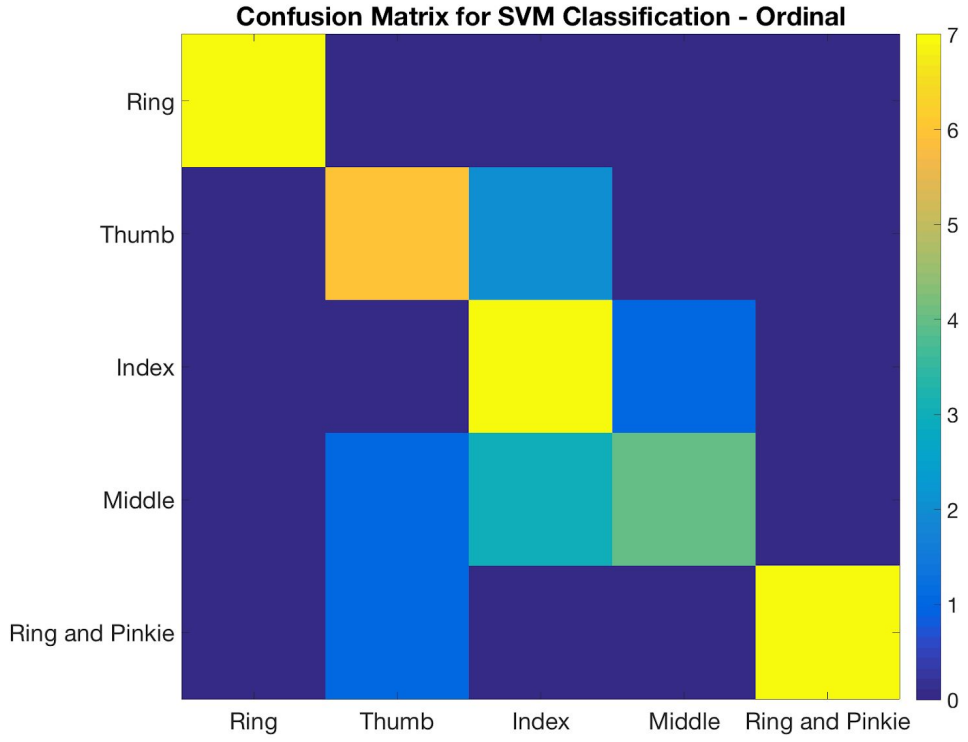

Confusion Matrix for SVM Classification - Ordinal
Ring
Thumb
Index
Middle
Ring and Pinkie
Ring
Thumb
Index
Middle
Ring and Pinkie
7
6
5
4
3
2
1
0

Table 4 - Linear Regression Equations and Results

#	Equation Name	Equation
1	Objective Function	$\min_{\vec{w},b}\left[\frac{1}{n}\sum_{i=1}^{n}(\vec{w}^T\vec{x}_i + b - y_i)^2\right]$
2	Decoder Matrix	$B = (transpose(y)y)^{-1}transpose(y)x$
#	Result Name	Value
3	Sum of Mean Squared Error of Predictions on Test Data	5.8726e+03
4	Correlation Coefficients of Prediction to Actual Motion	0.9467, 0.9193, 0.8762, 0.8450 Position X, Position Y, Velocity X, Velocity Y

Table 5 - Ridge Regression Equations and Results

#	Equation Name	Equation
1	Objective Function	$\min_{\vec{w},b}\left[\frac{1}{n}\sum_{i=1}^{n}(\vec{w}^T\vec{x}_i + b - y_i)^2 + \lambda\left\|\vec{w}\right\|_2^2\right]$
2	Decoder Matrix	$B = (transpose(y)y \ + \ n\ lambda\ I)^{-1}transpose(y)\ x$
#	Result Name	Value
3	Optimal Lambda	0.0286
4	Sum of Mean Squared Error of Predictions on Test Data	5.8390e+03
5	Correlation Coefficients of Prediction to Actual Motion	0.9469, 0.9194, 0.8766, 0.8461 Position X, Position Y, Velocity X, Velocity Y

Figure 3 - Sum of Mean Squared Errors versus Lambda

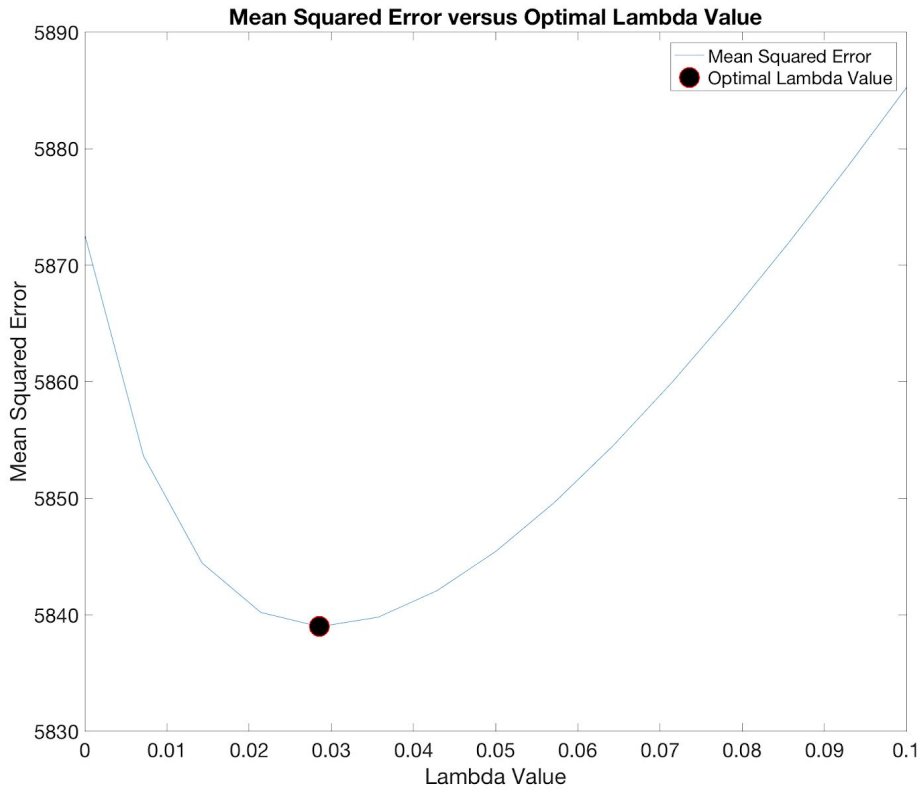


Table 6 - LASSO Results

#	Result Name	Value
1	Optimal Lambda	0.0286
2	Sum of Mean Squared Error of Predictions on Test Data	6.1863e+03
3	Correlation Coefficients of Prediction to Actual Motion	0.9475, 0.9204, 0.8769, 0.8459 Position X, Position Y, Velocity X, Velocity Y

Table 7 - Kalman Filter Results

#	Result Name	Value
1	Sum of Mean Squared Error of Predictions on Test Data	1.0170e+07

2	Correlation Coefficients of Prediction to Actual Motion	0.8955, 0.8928, 0.7893, 0.8031 Position X, Position Y, Velocity X, Velocity Y

Code Utilized for This Lab Report

```
% BIOMEDE 517 - Neural Engineering
% Lab 7 Part 1 - Naïve Bayes Classifier with Poisson Distribution
% Kushal Jaligama

% Predicting reach direction of monkey using 95 neurons

clearvars
close all

% Load data from firingrate.mat
% 95 neurons, 8 reach dirs, 182 samples for each neuron-dir comobo
load('firingrate.mat')

training_samples = 91;
total_samples = 182;
test_samples = total_samples - training_samples;
% Split data in half, training and testing test data sets
% Since we have 182 samples, use training_samples samples for each half
training_data = firingrate(:, 1:training_samples, :);
test_data = firingrate(:, training_samples+1:total_samples, :);

% Training Step - Parameters for Poisson Distribution
% Calculate lambda indexed by neuron, target
lambda = zeros(95, 8); % n x m matrix
% mean firing rate of neuron n when reaching to direction m
for dir = 1:8
   for j = 1:95
      lambda(j, dir) = mean(training_data(j, :, dir));
   end
end

% Prediction Step - Testing the Classifier
% Loop through all individual trials of test data
```

```
predictions = zeros(test_samples, 8); % This is to assign the features

for class = 1:8
   for i = 1:test_samples
      % Get the feature vector (95 neurons, i'th sample, class)
      X_i = test_data(:, i, class);
      for k = 1:8
         % Calculate log prob density for the feature vector
         g_hat = poisspdf(X_i, lambda(:,k));
         % We will assign sample based on sum of probabilities
         arg_k(k) = sum(log(g_hat));
      end
      % Assign feature to maximal k class
      % (best direction for this set of neuron firing rates)
      [maximal_k, index] = max(arg_k);
      predictions(i, class) = index;
   end
end

% Calculate accuracy of class assignments
number_correct = 0;
number_total = 0;
for class = 1:8
   for i = 1:test_samples
      if predictions(i, class) == class
         number_correct = number_correct + 1;
      end
      number_total = number_total + 1;
   end
end
accuracy = number_correct / number_total

% Applying classifier to spoofed dataset
% Calculate lambda indexed by neuron, target for full dataset
lambda_full = zeros(95, 8); % n x m matrix
for dir = 1:8
   for j = 1:95
      lambda_full(j, dir) = mean(firingrate(j, :, dir));
   end
```

```
end

% Generate random data along a Poisson Distribution
spoof_data = zeros(95, total_samples, 8);
for i = 1:total_samples
   spoof_data(:,i,:) = poissrnd(lambda_full);
end

% Prediction Step - Testing the Classifier
% Loop through all individual trials of test data
predictions_full = zeros(total_samples, 8); % This is to assign the features

for class = 1:8
   for i = 1:total_samples % samples
      % Get the feature vector
      X_i_LDA = spoof_data(:, i, class);
      for k = 1:8
         % Calculate log prob density for the feature vector
         g_hat_LDA = poisspdf(X_i_LDA, lambda_full(:,k));
         % We will assign sample based on sum of probabilities
         arg_k_LDA(k) = sum(log(g_hat_LDA));
      end
      % Assign feature to maximal k class
      % (best direction for this set of neuron firing rates)
      [maximal_k, index] = max(arg_k_LDA);
      predictions_full(i, class) = index;
   end
end

% Calculate accuracy of class assignments
number_correct = 0;
number_total = 0;
for class = 1:8
   for i = 1:total_samples
      if predictions_full(i, class) == class
         number_correct = number_correct + 1;
      end
      number_total = number_total + 1;
   end
```

```
end
accuracy_full = number_correct / number_total

% BIOMEDE 517 - Neural Engineering
% Lab 7 Part 2 - Linear Discriminant Analysis
% Kushal Jaligama

clearvars
close all

% Load data from ecogclassifydata.mat
load('ecogclassifydata.mat')
% Average power values from 0.5s prior to movement to 1.5s after
% movement in 60-120 Hz band
numTrials = 39;
% 27 electrode pairs, 'group' indicates which finger
% 1 is rest, 2 is thumb, 3 is index, 4 is middle
% 5 is ring and pinkie.

% Make LDA prediction of test sample's class
% Iterate through data and give each sample a turn at being test data
% Rest of samples should be training
for i = 1:numTrials
   test = powervals(i, :);
   training = powervals(1:i, :);
   training(end:numTrials - 1, :) = powervals(i + 1:end, :);
   train_groups = group(1:i, :);
   train_groups(end:numTrials - 1, :) = group(i + 1:end, :);
   predictions(i) = classify(test, training, train_groups, 'linear');
end

predictions = transpose(predictions);
num_correct = 0;

for i = 1:numTrials
   if predictions(i) == group(i)
      num_correct = num_correct + 1;
   end
end
```

```
accuracy = num_correct / numTrials

% This tells you what points were classified correctly and not
conf = confusionmat(group, predictions);
% A value of 1 at (2,1) in conf means one value that was supposed
% to be in class 2 was misclassified to group 1

% Plot this for visualization
imagesc(conf)

% Calculate percent correct by class
for i = 1:5
   total_in_class = sum(conf(i,:));
   percent_correct(i) = conf(i, i) / total_in_class;
end

percent_correct

% BIOMEDE 517 - Neural Engineering
% Lab 7 Part 3 - Support-Vector Machine
% Kushal Jaligama

clearvars
close all

% Load ECoG data
load('ecogclassifydata.mat');

numTrials = 39;
numGroups = 5;

% Restructure group to establish binary labels
Y = zeros(numTrials, numGroups);
for i = 1:numTrials
   for j = 1:numGroups
      if group(i) == j
         Y(i, j) = 1;
      end
```

```
    end
end

% Make predictions on the model with an SVM using leave-one-out
% cross validation
for i = 1:numGroups
    y = Y(:, i);
    SVMmodel = fitcsvm(powervals, y, 'KernelFunction', 'linear', 'Leaveout', 'on');
    predictions(:, i) = kfoldPredict(SVMmodel);
end

% Establish total number of correct predictions for accuracy testing
num_correct = 0;
% Establish number of correct preds in each class
by_class = zeros(1, 5);
for j = 1:numGroups
    for i = 1:numTrials
        if predictions(i, j) == Y(i, j)
            num_correct = num_correct + 1;
            by_class(j) = by_class(j) + 1;
        end
    end
    % Calculate the accuracy for each class
    by_class_accuracy(j) = by_class(j) ./ numTrials;
end

overall_accuracy = num_correct / (numTrials * numGroups)

by_class_accuracy

% Non-linear SVM with radial basis kernel function
for i = 1:numGroups
    y = Y(:, i);
    SVMmodel = fitcsvm(powervals, y, 'KernelFunction', 'rbf', 'Leaveout', 'on');
    nonlin_predic(:,i) = kfoldPredict(SVMmodel);
end

% Calculate overall prediction accuracy and by class
nonlin_num_correct = 0;
```

```
nonlin_by_class = zeros(1, 5);
for i = 1:numGroups
   for j = 1:numTrials
      if nonlin_predic(j, i) == Y(j, i)
         nonlin_num_correct = nonlin_num_correct + 1;
         nonlin_by_class(i) = nonlin_by_class(i) + 1;
      end
   end
   nonlin_by_class_accuracy(i) = nonlin_by_class(i) ./ numTrials;
end

nonlin_overall_accuracy = nonlin_num_correct / (numTrials * numGroups)

nonlin_by_class_accuracy

% BIOMEDE 517 - Neural Engineering
% Lab 7 Part 4 - Multi-Class Support-Vector Machine
% Kushal Jaligama

clearvars
close all

% Load ECoG data
load('ecogclassifydata.mat');

numTrials = 39;
numGroups = 5;

% Create a multi-class SVM model that can use int v
SVMmodel = fitcecoc(powervals, group, 'Leaveout', 'on', 'Coding', 'onevsall');
predictions = kfoldPredict(SVMmodel);

num_correct = 0;
for i = 1:numTrials
   if predictions(i) == group(i)
      num_correct = num_correct + 1;
   end
end
```

```
disp('onevsall')
accuracy = num_correct / numTrials
conf1 = confusionmat(group, predictions);

% onevsone - force each SVM to be tested against all non-target
% classes separately in the model
SVMmodel = fitcecoc(powervals, group, 'Leaveout', 'on', 'Coding', 'onevsone');
predictions = kfoldPredict(SVMmodel);

num_correct = 0;
for i = 1:numTrials
   if predictions(i) == group(i)
      num_correct = num_correct + 1;
   end
end

disp('onevsone')
accuracy = num_correct / numTrials
conf2 = confusionmat(group, predictions);

% ternarycomplete - partitions n classes into positive, negative,
% and zero valued classes that are cycled during training
SVMmodel = fitcecoc(powervals, group, 'Leaveout', 'on', 'Coding', 'ternarycomplete');
predictions = kfoldPredict(SVMmodel);

num_correct = 0;
for i = 1:numTrials
   if predictions(i) == group(i)
      num_correct = num_correct + 1;
   end
end

disp('ternarycomplete')
accuracy = num_correct / numTrials
conf3 = confusionmat(group, predictions);

% ordinal - uses n-1 binary SVMs for n classes. sampling space is
% partitioned at each class threshold
SVMmodel = fitcecoc(powervals, group, 'Leaveout', 'on', 'Coding', 'ordinal');
```

```
predictions = kfoldPredict(SVMmodel);

num_correct = 0;
for i = 1:numTrials
   if predictions(i) == group(i)
      num_correct = num_correct + 1;
   end
end

disp('ordinal')
accuracy = num_correct / numTrials
conf4 = confusionmat(group, predictions);

figure(1)
imagesc(conf1)
figure(2)
imagesc(conf2)
figure(3)
imagesc(conf3)
figure(4)
imagesc(conf4)

% BIOMEDE 517 - Neural Engineering
% Lab 8 All Parts - Continuous Decoders
% Kushal Jaligama

% Real-time decoding algorithms starting with linear regression
% Using data from a reach task

clearvars
close all

% Part 0 - Process Data
load('contdata.mat')
% Columns of X contain X position, Y position, X velocity, Y velocity
% Columns of Y contain firing rates of 950 recorded units
numObservations = 31413; % Number of time points

% Split data into training and test 50/50
```

```
training_rows = floor(numObservations / 2);
training_x = X(1:training_rows, :); % Position X, Y, Velocity X, Y
training_y = Y(1:training_rows, :); % Firing Rates of 950 units

test_rows = ceil(numObservations / 2);
test_x = X(test_rows:end, :);
test_y = Y(test_rows:end, :);

% Add a column of ones to the neural data to calculate an intercept
% term in the regression models
training_y = [ones(training_rows, 1) training_y];
test_y = [ones(test_rows, 1) test_y];

% Part 1 - Linear Regression

% Solve a linear regression equation with training data
% x = yB => B is the linear decoder matrix we want to find
training_y_t = transpose(training_y);
B = inv(training_y_t * training_y) * (training_y_t * training_x);

% To predict new data, multiply it by your linear decoder matrix, B
linear_predictions = test_y * B;

% Measure the mean squared error of predictions on the test data
linear_mean_squared_error = sum(mean((linear_predictions - test_x).^2))
% Correlation of predictions to actual motion
for i = 1:4
   linear_corr_coeffs(i) = corr2(test_x(:, i), linear_predictions(:, i));
end
linear_corr_coeffs

% Part 2 - Ridge Regression
least_error = intmax;
lambda_vals = linspace(0, 0.1, 15);
ridge_errors = zeros(1, 15);
% Perform ridge regression on all of the lambda values
for i = 1:15
   lambda = lambda_vals(i);
   square = training_y_t * training_y;
```

```
   B_ridge = inv(square + training_rows * lambda * eye(size(square))) * training_y_t *
training_x;
   prediction_ridge = test_y * B_ridge;
   % Calculate the mean squared errors of predictions on test data
   ridge_errors(i) = sum(mean((test_x - prediction_ridge).^2));
   % Use error terms to find the best lambda value
   if ridge_errors(i) <= least_error
      optimal_lambda = lambda;
      least_error = ridge_errors(i);
   end
end

optimal_lambda
least_error

plot(lambda_vals, ridge_errors)
hold on
plot(optimal_lambda, least_error, 'ro')

% Use the best lambda value to get best prediction
square = (training_y_t * training_y);
B_ridge = inv(square + training_rows * optimal_lambda * eye(size(square))) * training_y_t *
training_x;
best_ridge_predictions = test_y * B_ridge;
best_ridge_mean_squared_error = sum(mean((test_x - best_ridge_predictions).^2))
% Correlation of predictions to actual motion
for i = 1:4
   ridge_corr_coeffs(i) = corr2(test_x(:, i), best_ridge_predictions(:, i));
end
ridge_corr_coeffs

% BIOMEDE 517 - Neural Engineering
% Lab 9 Part 1 - LASSO
% Kushal Jaligama

close all

% Part 0 - Process Data
load('contdata.mat')
```

```
% Columns of X contain X position, Y position, X velocity, Y velocity
% Columns of Y contain firing rates of 950 recorded units
numObservations = 31413; % Number of time points

% Split data into training and test 50/50
training_rows = floor(numObservations / 2);
training_x = X(1:training_rows, :); % Position X, Y, Velocity X, Y
training_y = Y(1:training_rows, :); % Firing Rates of 950 units

test_rows = ceil(numObservations / 2);
test_x = X(test_rows:end, :);
test_y = Y(test_rows:end, :);

% Add a column of ones to the neural data to calculate an intercept
% term in the regression models
training_y = [ones(training_rows, 1) training_y];
test_y = [ones(test_rows, 1) test_y];


% LASSO

optimal_lambda = 0.0286

% Set up the B matrix
B = zeros(951, 4);
for i = 1:4
   % Get the weights for each recorded unit (neurons)
   B(:, i) = lasso(training_y, training_x(:,i), 'Lambda', optimal_lambda);
end

lasso_predictions = test_y * B;

% Now calculate the mean squared error and correlation coefficients
% Measure the mean squared error of predictions on the test data
lasso_mean_squared_error = sum(mean((lasso_predictions - test_x).^2))
% Correlation of predictions to actual motion
for i = 1:4
   lasso_corr_coeffs(i) = corr2(test_x(:, i), lasso_predictions(:, i));
end
```

lasso_corr_coeffs

```
% BIOMEDE 517 - Neural Engineering
% Lab 9 All Parts - LASSO and Kalman Filters
% Kushal Jaligama

clearvars
close all

% Part 0 - Process Data
load('contdata.mat')
% Columns of X contain X position, Y position, X velocity, Y velocity
% Columns of Y contain firing rates of 950 recorded units
numObservations = 31413; % Number of time points

% Split data into training and test 50/50
training_rows = floor(numObservations / 2);
training_x = X(1:training_rows, :); % Position X, Y, Velocity X, Y
training_y = Y(1:training_rows, :); % Firing Rates of 950 units

test_rows = ceil(numObservations / 2);
test_x = X(test_rows:end, :);
test_y = Y(test_rows:end, :);

% Add a column of ones to the neural data to calculate an intercept
% term in the regression models
training_y = [ones(training_rows, 1) training_y];
test_y = [ones(test_rows, 1) test_y];

% Kalman Filter

% The physics is represented as:
% x_t is a state ("position, velocity")
% x_t = Ax_(t-1) + w_t
%       physics    noise

% y_t is neural firing rates at time step
% y_t = C_x(t) + q_t
%       LinFilt  noise
```

```
% Transpose the matrices to put time domain as columns
training_x = transpose(training_x);
training_y = transpose(training_y);
test_x = transpose(test_x);
test_y = transpose(test_y);

% Set up the time step variables
training_x_prev = training_x;
% Remove first element of training data (ones)
training_x(:, 1) = [];
training_y(:, 1) = [];
% Remove last element of "prev" matrix for prediction purposes
training_x_prev(:, training_rows) = [];

% Get the physics and linear filter matrices
C = (training_y * transpose(training_x)) / (training_x * transpose(training_x));
A = (training_x * transpose(training_x_prev)) / (training_x_prev* transpose(training_x_prev));

% Get the noise
W = (1 / (numObservations - 1)) * (training_x - A * training_x_prev) * transpose(training_x - A *
training_x_prev);
Q = (1 / (numObservations - 1)) * (training_y - C * training_x) * transpose(training_y - C *
training_x);

% Start making predictions
xhat = zeros(4, test_rows);
xhat(:, 1) = test_x(:, 1);

% a posterior covariance of x, how accuracte is the estimate
postcov = W;

% Predict, innovate, update
for i = 2:test_rows
   % Prediction given last time step
   xhatprev = A * test_x(:, i - 1);
   postcovprev = A * postcov * transpose(A) + W;
   Kt = postcovprev * transpose(C) / (C * postcovprev * transpose(C) + Q);
   xhat(:, i) = xhatprev + Kt * (test_y(:, i) - C * xhat(:, i-1));
```

```
    postcov = (eye(4) - Kt * C) * postcovprev;
end

% Now calculate the accuracy of this entire thing
kalman_mean_squared_error = sum(mean(test_x - xhat).^2)
% Correlation of predictions to actual motion
for i = 1:4
    kalman_corr_coeffs(i) = corr2(test_x(i, :), xhat(i, :));
end
kalman_corr_coeffs
```