**Lab 7: Supervised classifier algorithms**

# Introduction

In the last lab, we learned how to use *unsupervised* algorithms to automagically sort samples from large datasets into different groups. In this lab, we will use *supervised* algorithms to classify data into groups. These algorithms are ‘supervised’ because they need to be trained on labeled data. (Note that none of the Lab 5 data needed to be labeled for the clustering algorithms to work.) This lab introduces three supervised classifier algorithms: Naïve Bayes, linear discriminate analysis, and support-vector machines. By the end of this lab, you should be familiar with how to use all three and have an understanding of how they are related.

Note: For calculating overall classification accuracy, take the sum of accurate classifications divided by the number of predictions.

# Software

This lab must be completed using MATLAB.

# Part 1) Naïve Bayes

Naïve Bayes classifiers are a family of classifiers that are natively able to decode a set of input features into one of several different output classes, given some assumptions about the distribution of features. For example, a Naïve Bayes classifier could take firing rates from different neurons and predict whether a monkey is pointing up, down, left, or right. (Note that output classes are distinct, discrete, and exclusive.) These classifiers take the form

where is a vector of features (e.g., firing rates of different neurons), is the number of features in , and is the estimated probability distribution of feature for class (e.g., the distribution of neuron #14’s firing rates when the monkey is pointing left). (If you’re unfamiliar with this notation, the output takes the value of associated with the largest value in the operation.)

Note that Naïve Bayes classifiers assume that the different features of sample are independent from one another, for a given class. For example, if firing rates are recorded from 30 different neurons on a Utah array of a monkey performing a grasp, the algorithm assumes that each of the neurons’ firing rates are completely independent of one another. This is usually untrue, but turns out not to matter too much for decoder performance.

Here, we’re going to build a Naïve Bayes classifier from scratch and measure its performance. We’ll be using a set of spike data recorded from a monkey performing reach tasks.

In order to train and test our classifier, we need to establish training and test datasets.

1. Load data from firingrate.mat. The dataset contains firing rates recorded from 95 different neurons and 8 different reach directions, with 182 samples for each neuron-direction combination.
2. Split the data in half, with the first half labeled for training and the second reserved for testing.

We begin by *training* our classifier. Using the training data, we will estimate key parameters of the model that the classifier will use to make predictions.

1. Use the training dataset to estimate the mean, , for each feature-class (neuron-direction) pair. We will use this to estimate the distribution of spike rates for each feature-class pair, , by assuming a Poisson distribution. The Poisson distribution is a reasonable assumption for spike data, as count variables very often fall under Poisson distributions.

Now that we’ve estimated the necessary parameters for the classifier, we can *test* the classifier using the test data. We do this to make sure we’re not overfitting the classifier to the training data.

1. To predict the reach direction of a given sample in the test dataset:
   1. For each feature, (the firing rate of neuron in sample ), use to find the probability density of observing when the monkey is reaching in direction #1. The MATLAB function poisspdf() may be useful here.
   2. Once you’ve calculated for every feature, we would normally multiply them together get . Note that this is the argument that needs to be evaluated for each class in the equation shown in the introduction for Part 1. However, this ends up being a very, very small value that can cause floating point errors. To avoid that issue, we’ll take the log of each probability and add them instead to get

This creates much more reasonable values.

* 1. Calculate this argument for each class for sample .
  2. Assign sample to class , where is the class for which is the largest.

1. Repeat this to predict the reach direction of every sample in the test dataset.
2. Calculate the accuracy of the decoder. How does its performance compare to chance?

To make doubly sure that we’re not fooling ourselves with the classifier’s performance, we can also apply it to a spoofed dataset that looks very similar to the original, but contains no meaningful information.

1. Find the overall mean of the full dataset’s spike rates.
2. Create a second set of data with the same size and overall distribution (assume Poisson) as the original.
3. Test your classifier on this fabricated dataset. What is the performance? Is this what you expect?
4. What would happen if for some in the training data? (This is possible for Poisson-distributed data.) What are the implications for neural decoding performance?

# Part 2) Linear discriminant analysis

Linear discriminant analysis (LDA) is a supervised classifier algorithm that can be thought of as a hyperplane separating two clusters of data. Everything that falls on one side of the hyperplane is in one class, and everything falling on the other side is in the other class.

Although this may sound different from Naïve Bayes, it operates on a very similar principal. While Naïve Bayes looks at each feature independently, LDA determines the classification of a sample by estimating the likelihood of a sample falling within the multivariate distribution of samples within each class. Specifically, it assigns samples to the class that has the nearest mean, weighted by the prior probability of that class and normalized to the covariance of samples. (This should sound very similar to a specific clustering algorithm you examined in the previous lab.) The classifier takes the form

where is the prior probability of class (just like in Naïve Bayes) and is the probability density of in the multivariate Gaussian distribution with estimated mean and covariance . Note that is estimated from the entire data set, not just from data in class .

Instead of writing an LDA algorithm from scratch, we’ll use the MATLAB built-in classify(). For this part of the lab, we’ll use a set of ECoG data from a finger movement task.

1. Load data from ecogclassifydata.mat. The dataset contains average power values from 0.5 sec prior to movement to 1.5 sec after movement in the 60-120 Hz band. There are 38 trials with data recorded from 27 electrode pairs. The ‘group’ variable indicates which finger was squeezing (1 is rest, 2 is thumb, 3 is index, 4 is middle, 5 is ring and pinkie).
2. Since we don’t have a lot of trials, creating independent training and test sets is impractical. Instead, we will use leave-one-out cross-validation. To start, designate the first trial as the test data and the rest of the data as training data.
3. Use the MATLAB built-in testClass = classify(testData, trainingData, trainingClasses, ’linear’) to make an LDA prediction of the test sample’s class.
4. Repeat this process such that every sample takes a turn at being the test data, with all other samples acting as training data. Keep track of the prediction for each test case.
5. What is the overall percentage accuracy of this algorithm?
6. In addition to overall accuracy, it is also important to know where errors occur. Create a confusion matrix showing what percentage of the time a finger is classified as itself (on diagonal entries) and what percentage of the time it was classified as another finger (on off-diagonal entries). See Figure 3 of Cindy’s 2013 paper, ‘Hand posture classification using electrocorticography signals in the gamma band over human sensorimotor’, for reference. MATLAB’s pcolor() may be useful here.

# Part 3) Support-vector machine

Support-vector machines (SVM) extend linear classification with hyperplane separators, but do not rely on probability distributions. While LDA classification can determine a binary separation of data using distances to means, the separating hyperplane is not optimized to variation in the data membership itself. Linear SVMs create a hyperplane based on the locations of “support-vectors,” which are the data points that lie closest to the maximal separation of binary data clusters. In other words, an SVM classifier finds the greatest separation of the data clusters, and then determines how best to fit a hyperplane between them using the data points that sit closest to their opposite clusters. This is called the hard-margin solution for SVMs, and relies on solving the optimization problem

where is the vector normal to the hyperplane separator, is the class of the data, is a parameter such that is the translation of the hyperplane from the origin, and is the *n*-dimensional coordinates of the data point with *n* being the dimension of the sampling space. The hyperplane is then determined to be the midpoint of the two margins defined by the data points that lie closest to the opposing cluster with normalized length 1, and is known as the maximal-margin hyperplane.

The solution to this problem generates a classifier that maps the sampling space to binary outputs, as we saw with LDA. However, what about the case of linearly inseparable data where you cannot draw a line between two clusters with good classification performance? We saw in the previous lab that linear transformations of the data, such as principal component decomposition, can help to rearrange the information into clumps. Sometimes that doesn’t work too well, and the decomposition is uninformative for classification. Nonlinear SVMs account for this problem by embedding the sampling data into higher- or infinite-dimensional space in an attempt to identify linearly separable clusters (using the “kernel trick”). This process uses the hinge loss function to determine if a data point is on the correct side of the margin that determines the hyperplane, and if found to be on the incorrect side solves the optimization problem

where the parameter dictates the tradeoff between margin size and classification accuracy near the margins, in order to optimize the width of the margins of the hyperplane. (This is also known as regularization, which helps avoid overfitting.) This is called the soft-margin solution for SVMs. This optimization problem is solved by the objective functions

where is a classification slack variable. The first function is known as the L1-SVM, is non-differentiable, and generally inferior to the second function. The second is known as the L2-SVM, is differentiable, and is more forgiving to non-generalized data and robust against outliers. We will be using the L2 form for classifying our neural data.

We will use MATLAB built-ins for this portion of the lab as well, given that coding an SVM from scratch is well beyond the scope of this class. Similarly, there are many parameters that can be optimized with an SVM, but we will not be exploring those due to time. We will begin by looking at a linear SVM.

1. Load the ECoG data (ecogclassifydata.mat). We’ll first need to restructure the ‘group’ variable to establish binary labels. Create an matrix , such that
2. Note that we need to classify five different classes, but SVMs can only classify two. To get around this, we simply design an SVM for each class. Each SVM determines whether the sample belongs to that SVM’s class or not. To start, create a vector corresponding to the first column of .
3. Now create an SVM classification model for the first group using the MATLAB built-in

SVMmodel = fitcsvm(X, y, ‘KernelFunction’, ‘linear’, ‘Leaveout’, ‘on’)

where X is the “powervals” data and y is the vector corresponding to the partitioned labels for the given class. There will be one SVM model per class that you are classifying. Since we’re still using a fairly small dataset, we will use cross-validation again, so use the “Leaveout” option to dictate leave-one-out cross validation.

Note: You can look at the model object and see how many support-vectors were used to create the hyperplane. Optimized algorithms should use about 50% of the data points as support-vectors.

1. Now classify the cross-validated models using

predictions = kfoldPredict(SVMmodel);

1. Repeat steps 2 through 4 for all classes. Keep track of the predictions.
2. Determine the overall performance of the classifier. Compare it to the performance of your LDA algorithm by comparing percent correct by class. Why do you think the linear SVM worked better or worse than the LDA algorithm?

Now we will look at a non-linear SVM, as these are the primary types of SVM’s in use today.

1. Create a non-linear SVM model using the radial basis kernel function with leave-one-out cross validation:

SVMmodel = fitcsvm(X, Y, ‘KernelFunction’, ‘rbf’, ‘Leaveout’, ‘on’)

Test it using the same method you used for the linear SVM.

1. Compare the performance of the non-linear SVM to the linear SVM and LDA algorithms. Did one work better than another for a certain class? What about the data may have led to such results?
2. Now try this process with a kernel function of your choosing. (Look in the documentation for fitcsvm().) Many SVMs are optimized by the choice of kernel function and how it interacts with the data you are processing. You may use any established kernel function except “linear” and “radial basis/Gaussian,” and you may even try to determine your own function. Compare the performance of this kernel function to the previous tests. How did this new kernel function change the decoding performance for each class with respect to the data in that class?

# Part 4) Mutli-classification support-vector machines

You may be fed up with managing all of these multiple binary classifiers at this point. So is everyone else. As of 2015, MATLAB has multi-classification SVM support so that you can manage things with a single model. We will explore several methods for applying multi-classification SVMs, but beware that this is a front-end technique that has a great deal of back-end overhead. SVMs are strictly binary, so how they are combined for multi-classification can alter the overall system performance.

1. Using the original variables in ecogclassifydata.mat (you don’t need to partition anything or create binary labels), use

SVMmodel = fitcecoc(X, Y, ‘Leaveout’, ‘on’, ‘Coding’, ‘onevsall’)

This will create a multi-class SVM model that can take the integer values in “group” and use them as class definitions. The ‘Coding’ option determines how a collection of binary classifiers are combined to conduct multi-classification. In this case, the One-vs.-All method is used, which should be similar to how you manually conducted classification on the five different classes in groups earlier. This tells the system that the target class has one binary value, while all other classes have the other binary value (i.e., {-1, 1}). Compare the performance of this system to your previous tests.

We now want to explore how classification performance changes based on how you combine binary classifiers. MATLAB offers several options for how to multi-classify. For each of the methods below, save the predictions to create a confusion matrix like you did in Part 2.

1. A common method for combining binary SVMs is One-vs.-All as we used before. That technique creates *n* SVM models for *n* classes. A more robust version of this is One-vs.-One, which creates binary SVMs for *n* classes. This technique forces each SVM to be tested against all non-target classes separately in the model. Use the tag below and compare the results to One-vs.-All.

SVMmodel = fitcecoc(X, Y, ‘Leaveout’, ‘on’, ‘Coding’, ‘onevsone’)

1. The next method uses the largest number of SVMs for multi-classification, with binary SVMs for *n* classes. Ternary Complete partitions *n* classes into positive-, negative-, and zero- valued ({-1, 0, 1}) classes that are cycled during training.

SVMmodel = fitcecoc(X,Y, ‘Leaveout’, ‘on’, ‘Coding’, ‘ternarycomplete’)

1. The final method we will look at uses binary SVMs for *n* classes. Ordinal partitions the sampling space at each class threshold.

SVMmodel = fitcecoc(X, Y, ‘Leaveout’, ‘on’, ‘Coding’, ‘ordinal’)

1. Plot confusion matrices for each multi-classification SVM model you test (4 plots), and report the total classification accuracy for each. Examine how the performance changes based on parameter or method changes. Are there any surprising results compared to the binary classification systems? Are there consistent errors across methods? Why might there be more misclassifications using multi-classifiers compared to binary classifiers?

# Guidelines for Lab Report (on Labs 7, 8, and 9 together)

*Introduction:* The introduction should be one paragraph long summarizing the motivation for developing the tools used in this lab and what they can be used for, along with a brief summary of everything you will show in this lab report.

*Methods:* From Lab 7, there should be methods paragraphs (and diagrams where necessary) on:

1. Assumptions of the algorithms used
2. How the algorithms were implemented

Include the code as an Appendix to your report. Cite sources for any values used in your models.

*Results:* You should include the following in your Results:

1. The outcome of each algorithm (including 1 plot from part 2 and 4 plots from part 4).
2. The limitations of each algorithm.
3. How the algorithms compare to each other.

Include all figures produced by MATLAB that could help explain and illustrate your findings.

*Discussion:* Should be 2-3 paragraphs long describing what you could use these tools for in the future.

This report will be combined with Labs 8 and 9 to create one cohesive report. The report (not including Appendix) should be no longer than 4 pages. Use 12 pt. font and 1.15-1.5 line spacing. If your text is over the 4-page limit with figures, you can move your figures to an appendix section that goes beyond the 4-page limit. However, any text that goes beyond this limit will not be graded, except for figures, figure titles (no captions), and your code.

Please upload your report to Canvas and leave a hard-copy with your GSI in lab. The hard-copy will be graded, so be sure different lines on your plots are distinguishable (using color or different line styles).