# Part I
# Supervised Machine Learning

## 1 Overview

Our goals for this portion of the lecture is to introduce supervised learning concepts and demonstrate how they can be implemented programatically. In particular, we will discuss both regression and classification and introduce standard models used to solve both types of problems.

### 1.1 Regression vs. Classification

In general, supervised problems can be broken down into two categories: regression or classification. It is important to distinguish your problem type so the correct method can be implemented.

When one is solving regression problem, they are typically interested in finding a function, $f : \mathbf{x} \to y$, where $\mathbf{x} \in \mathbb{R}^p$ is the set of features in the data and $y$ is the target. For regression problems, it is assumed that $y$ is a continuous output. Conversely for classification problems, the objective is to find a function, $f : \mathbf{x} \to y$ where $\mathbf{x}$ is again the feature space, but $y \in \{1, \ldots, C\}$ where there are $C \geq 2$ labels in the data. An example of a regression problem would be using demographics to predict the unemployment rate (a continuous variable) in a given location, and a classification problem would be using health care data to determine whether an individual will have a stroke (binary variable).

For notational simplicity, assume for the rest of this handout that we have data, $\mathbf{X} \in \mathbb{R}^{n \times p}$ where $n$ the number of samples and $p$ the number of features, the target, $\mathbf{y} \in \mathbb{R}^n$, where the values are continuous if we are solving a regression problem or discrete for classification.

## 2 Regression

To introduce regression, we will start by discussing the simplest regression model: linear regression.

### 2.1 Linear Regression

As we mentioned previously, the goal of regression problems is to find a function, $f$, which maps $\mathbf{x}$ to $y$. Ordinary Least Squares (OLS) regression states this function has the form:

$$f(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p \tag{1}$$

where $\boldsymbol{\beta} \in \mathbb{R}^p$ are inferred by solving the optimization problem:

$$\min_{\boldsymbol{\beta}} \quad \sum_{i=1}^{n} (y_i - \boldsymbol{\beta}^T \mathbf{x}_i)^2. \tag{2}$$

This optimization problem might look complicated, but fortunately there are excellent functions written in R that will do it for us. To solve OLS (and other types) of linear regression, we will use the `lm` function.

Moreover, although it is a simple model, linear regression is probably the most common type of machine learning algorithm that is used in the real-world. This is the case because it easy to compute and it is easy to understand the model's output. However, whenever one is working with linear regression (or any model for that matter) it is important to validate whether its assumptions hold true. For linear regression, these are known as the "Gauss-Markov" assumptions and they are as follows:

1. The relationship between $\mathbf{X}$ and $\mathbf{y}$ is linear
2. The residuals, $\epsilon_i$, have the distribution $\epsilon_i \overset{iid}{\sim} \mathcal{N}(0, \sigma^2)$

If these assumptions hold, then linear regression is the "best linear unbiased estimator“ (BLUE). In practice, it is often not necessary for these assumptions to hold exactly, but it is important to check them.

Before we start coding we have to introduce two more quick ideas: training vs testing data sets and model evaluation metrics.

## 2.2 Training vs. Testing Data

When using machine learning to solve your data-related problems, you should **always** have training and testing data. What does this mean? In practical terms it means that you need to split your data into two separate groups – one you use to help learn the function (training set) and the other used exclusively to evaluate its performance (test set). Why do we need two data sets?

Formally, the error one receives on the training set is a biased estimate of future performance, whereas out-of-sample (test) error is a better estimate the model's error. From an intuitive standpoint, I think the following story helps. Suppose you have two professors: Profs A and B. Professors A and B teach the same course except they have one critical difference – Prof A only gives test questions from previous problem sets, and Prof B makes test questions that are similar, but *not* exactly the same as old homeworks. Which one of these professors do you think will have a better idea of how well their class understands the material? Since Prof A only uses old questions a student could just memorize the answers and regurgitate them on the test – in essence, we would have no idea how well that student understood the material. They could just be really good at memorizing things. On the other hand, since Prof B writes different questions from the problem sets, this requires the students to understand the general idea of what is being taught so it can be applied on the test rather than memorizing quirks of the specific problems.

In general, Prof A can be viewed as someone who uses their training set to evaluate the model's performance. Since you are using the same "questions" to both learn the concepts (model parameters) and evaluate performance, we have no idea how much the model has actually inferred about the problem space – it could just be learning quirks of the training data. On the other hand Prof B is someone who uses training data to infer model parameters, but then has *similar*, but not exactly the same data to evaluate the performance. It is necessary that the test data be similar to the training data because otherwise you will not be getting a good estimate of your model's performance. To determine if the training and testing data are close to one another, a standard thing to do is to compute the mean and variance of both the training and testing data. If the values are close to one another then you ought to be fine, but if they are quite different, then there are other questions arise. Namely, you need to consider why your training and testing sets are distributionally different from one another. There could be physical processes from the data collection phase that is causing this or there might have been some error in the data wrangling. In either case, it is important to determine the cause because otherwise you will not be able to correctly evaluate your model's performance.

## 2.3 Model Evaluation

We've alluded to a mysterious "error" of the model, but we have not formally defined what this means. We will briefly highlight some common evaluation metrics to determine model performance and then we will implement all of the ideas into code.

One of the most common measures of fit for a linear regression is known $R^2$. This metric is computed by solving

$$R^2 = \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{3}$$

where $\bar{y} = \frac{1}{n}\sum_i y_i$ and $\hat{y}_i$ is the model's prediction for the $i^{th}$ sample. In sample (i.e. in the training set), this value is defined between the range zero and one where zero would correspond to the situation where your model is no better than guessing the average value for $\mathbf{y}$ and one indicates that the model predicts $\mathbf{y}$ perfectly. The formal definition of $R^2$ is that is measure of the "... proportion of variance in the dependent variable that is predictable from the independent variables."

We've introduced all the main ideas we need to solve a linear regression problem so let's translate these ideas into code!

## 2.4 Regularized Linear Regression

When working with OLS regression, we sometimes face situations when our model is too simple or is too complex. When the model is too simple – typically when you include too few features – both the training and test error will be high. Conversely, when your model is too complex, it "over-fits" the training data. As a result, the in-sample error is low, but on the test set we have poor performance. In general, what we are discussing is known as the bias-variance trade-off. Figure 1 displays what we're saying in a condensed format

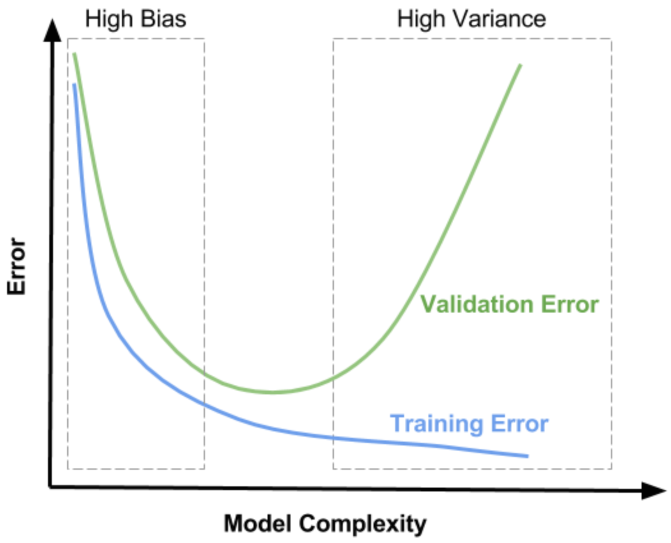

Figure 1: Bias-Variance Trade-Off

Ideally we would like to be somewhere in the middle of that curve – a model with just the right amount of bias and variance so that we can learn the training data, but no under or over-fit. One way to accomplish this in the regression domain is to use regularization.

At a high level, what regularization does is add some penalty to the regression optimization problem so that the model does not include all of the coefficients in the model. For ridge regression the model is

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} (y_i - \boldsymbol{\beta}^T \mathbf{x}_i)^2 + \lambda \|\boldsymbol{\beta}\|_2 \tag{4}$$

and for LASSO

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} (y_i - \boldsymbol{\beta}^T \mathbf{x}_i)^2 + \lambda \|\boldsymbol{\beta}\|_1 \tag{5}$$

where the amount of regularization is controlled by the hyper-parameter, $\lambda$. What both of these approaches attempt to do is penalize the model for having too many coefficients included in the final output. Typically what ridge regression does is shrink the coefficient values whereas LASSO tends to drop features out of the model entirely. However, how do we select an appropriate value for $\lambda$?

## 2.5 Hyper-Parameter Search

One of the key additions to more complex models, like regularized linear regression, is that they contain hyper-parameters. These items control how the model fits the data and finding good values for them is an essential part of the training process.

To find a good value for the hyper-parameter(s) in the model we have to use a "validation" set. The validation set is separate from the the training and testing sets and it used to help us evaluate how a combination of hyper-parameters fits the data. Thus the "training process" has the following steps:

1. Split the data into training, validation, and testing
2. For each unique combination of hyper-parameter(s), build a separate model on the *training* set
3. Evaluate the performance of that model on the *validation* set
4. Using the best model from validation, evaluate the final performance on the *test* set.

Notice again how we do not ever use the test set to help us fit a better model – we only use it as a final metric of how our model is performing. It is essential that you completely separate the test set so that we can get an unbiased estimate of the model's fit to the data.

Now that we've introduce regularized regression, let's implement them in code and compare how they do versus OLS regression.

# 3 Classification

Thus far we have introduced the regression component of supervised learning. Now we are going to talk about classification. In particular, we will start with logistic regression, and then get onto more advanced topics like trees.

For this lecture, we are going to focus on classification when $y \in \{0, 1\}$ – binary classification. This is the simplest starting point and covers a wide range of useful problems. There are generalizations of the models we introduce, but they are beyond the scope of this course.

## 3.1 Logistic Regression

Since we are dealing with a case where $y = 0$ or $y = 1$, it would be nice to say what the probability is that a given $\mathbf{x}$ belongs to either the first or second class. A nice starting point is to use the "logistic" function which is defined as

$$g(t) = \frac{1}{1 + \exp(-t)}. \tag{6}$$

The output of the function $g : \mathbb{R} \rightarrow [0, 1)$ and can be thought of as the probability that a sample with a single feature, $t$, belongs to the class $y = 1$. Generalizing this to the case where $\mathbf{x} \in \mathbb{R}^P$, the logistic function equals

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\beta}^T \mathbf{x})}. \tag{7}$$

Logistic regression is similar to linear regression in the sense that it tries to find coefficients, $\boldsymbol{\beta}$ which allows us to map to a domain which is either zeros or ones. To find the coefficients, the problem can be formulated a maximum likelihood estimation problem and with some algebraic manipulation we get

$$\max_{\boldsymbol{\beta}} \quad -\sum_i \Big(1 + \exp(-y_i \cdot \boldsymbol{\beta}^T \mathbf{x}_i)\Big) \tag{8}$$

where we can add either an $L_1$ or $L_2$ penalty to the coefficients to introduce regularization.

Let's try this out with some examples.

## 3.2 CART

Classification and Regression Tress (CART) is a machine learning model that can handle both classification and regression settings. Nevertheless, CART is more often used in classification settings because regression trees tend to perform poorly.

One benefit of using CART for classification is that can automatically handle multi-class classification without changing the underlying optimization algorithm. Additionally, sometimes they can be more understandable than logistic regression or other popular classification methods which can make it easier to explain what the model is doing to people who are less familiar with how the algorithms work. For example, on the classic Titanic data set which contains the outcomes for passengers on the HMS Titanic, the final tree is shown in Figure 2.

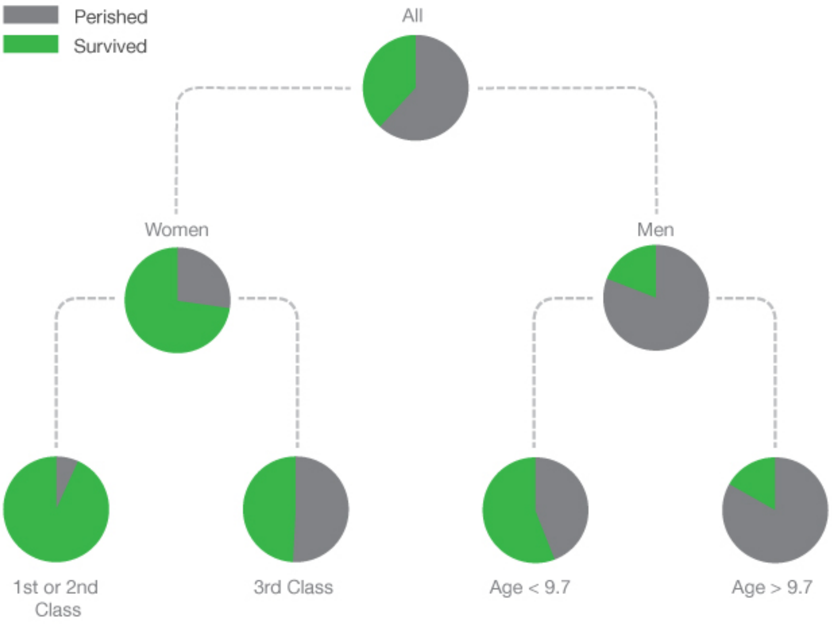

Figure 2: HMC Titanic CART Model

In the tree shown, the algorithm first splits on the sex of the passenger, and then for each leaf it generate its own splits. Unlike the optimization problem formulated for linear and logistic regression, trees do not have a clean *parametric* function which can describe them a-prori. Therefore we have to define how we compute the final tree.

At a high level, the CART algorithm makes sequential splits on the features to make the resulting nodes as "pure" as possible. Let me show a picture to make this a bit clearer. First, we start with the root of the tree where all the data resides. For example, suppose that we have a data set that looks like

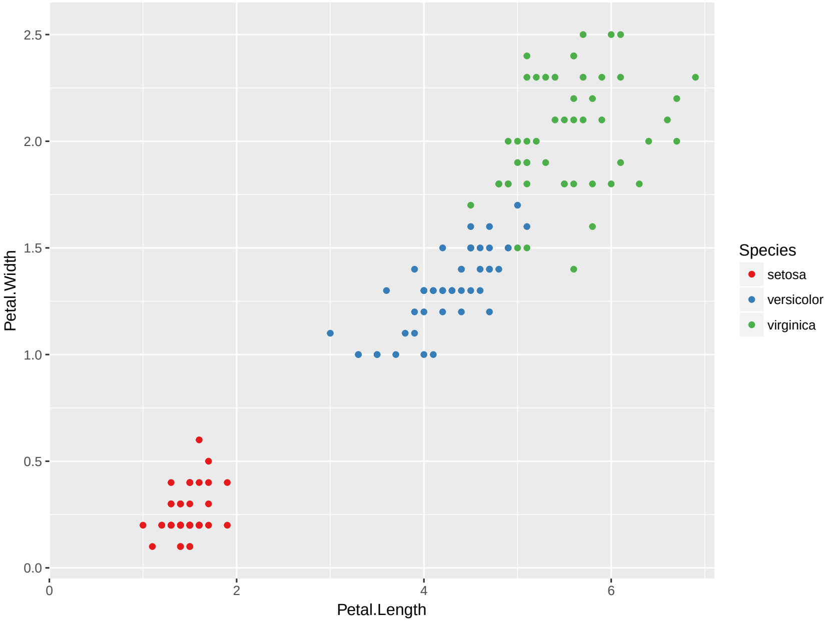

Figure 3: Iris Data

Then, the algorithm splits along a feature which gives the greatest reduction in impurity. In the data above, this looks like

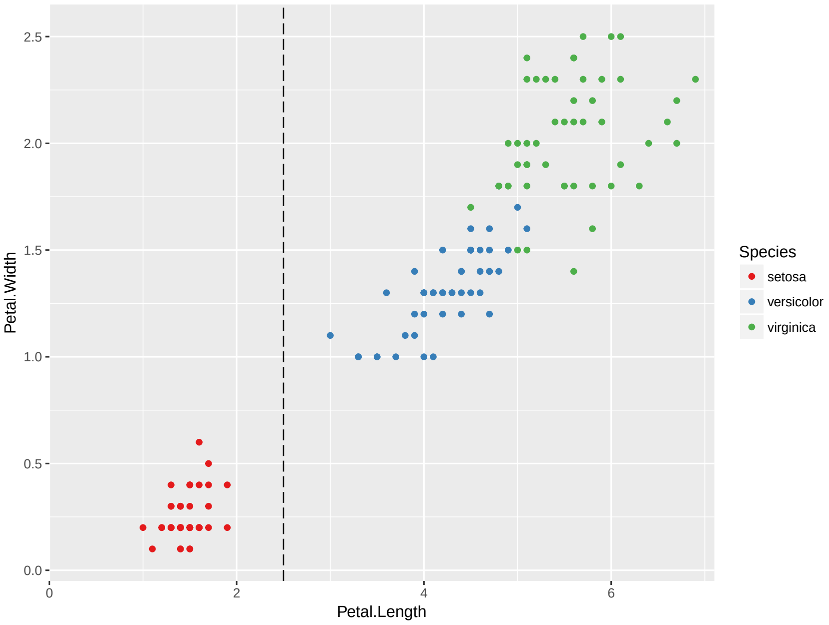

Figure 4: CART First Cut

Next, the algorithm will continue splitting the feature space until either the node is pure or until some complexity condition is met. For our data this looks like

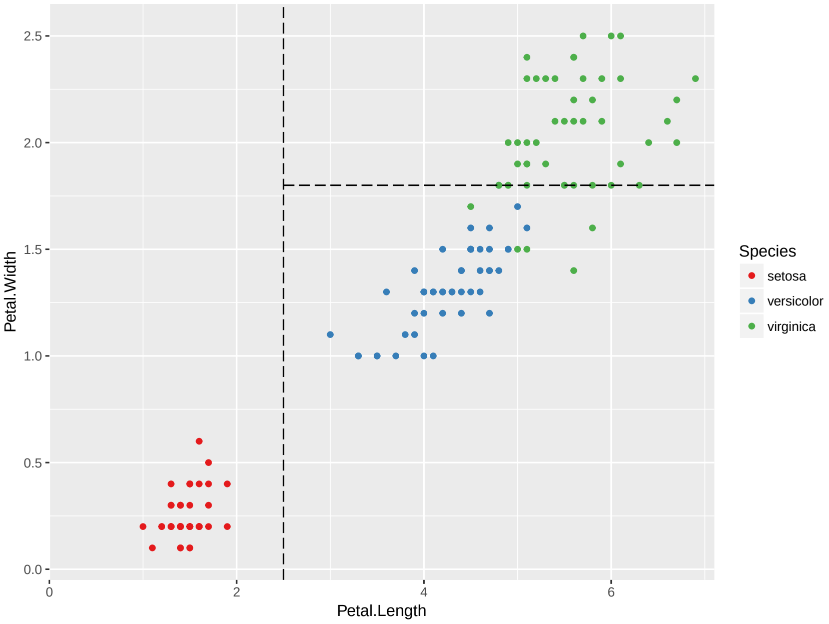

Figure 5: CART Second Cut

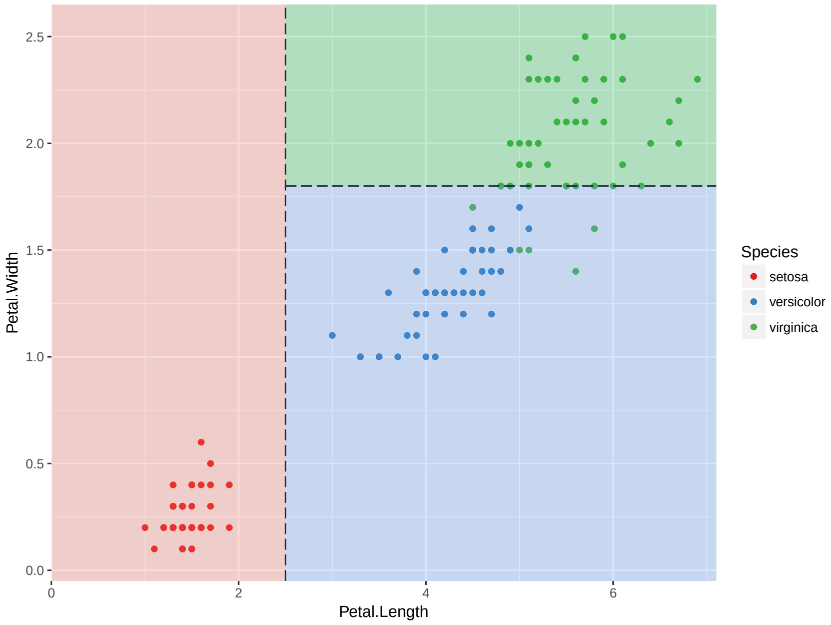

Figure 6: CART Final Model

How CART measures the ”impurity” of the node is through a metric known as the “Gini Index.” This value is defined as

$$\mathrm{GI} = 1 - \sum_{j=1}^{C} p_j^2 \tag{9}$$

where $C$ is the total number of labels in the data and $p_j$ is the relative frequency of class $j$ in a particular node. Thus for each ”bucket”, the algorithm splits on the feature which gives the greatest reduction in the Gini index. Moreover, this splitting process is greedy; it only optimizes for that particular steps then fixes the solution for future steps. This makes the CART algorithm very quick, but can lead to splits which are not globally optimal.

CART, given that it is a more complex model, has many hyper-parameters. In R they are:

- `minsplit`: The minimum number of data points that a node much have in order to be considered for a split
- `minbucket`: The minimum number of data points in each leaf node
- `cp`: The threshold complexity parameter which the algorithm uses to determine whether or not to split at each node (this also indirectly controls the depth of the tree).

If these thresholds are not in place then the CART algorithm will continue splitting until all the points are correctly classified. This will lead to perfect training error, but also a high chance of over-fitting. We will again use validation to find the correct values for these hyper-parameters.

Let’s dive into the code to see how trees work in action.