What Is Multicollinearity?

Multicollinearity is a problem that many a data scientist will come across in the problems they solve. It's a situation where the explanatory variables are nearly perfectly correlated with other. In this situation, it becomes difficult to use linear regression via OLS or gradient descent because the technique cannot accurately estimate the regression coefficients, often resulting in inflated values for these parameters. This is because it's hard to distinguish the effect of one explanatory variable from another, and subsequently each explanatory variable's effect on the response variable. As a product of multicollinearity, we observe the value of the regression coefficients changing, sometimes drastically, each time we initialize a linear regression algorithm. Ultimately, this renders traditional linear regression as a less preferable method for handling data that exhibits these types of patterns.

Testing for Multicollinearity

Very highly positive regression coefficients are one of the first tell-tale signs of multicollinearity. In addition to this, we should calculate the correlation of all the explanatory variables with each other. Correlation coefficients of $.95 \leq \rho \leq 1$ should also raise red flags in the mind of a data scientist. Specifically, though, there is a statistic that we can use to determine whether we most definitely have multicollinearity in our data set, called variance inflation factor.

Variance Inflation Factor (VIF)

The VIF statistic is calculated on a range from $0 \leq VIF \leq \infty$. Typically, the rule of thumb is that any VIF score that is > 5 indicates multicollinearity, and any score above 10 indicates severe multicollinearity. The statistic is calculated by regressing a given explanatory variable against the others and then using the result to calculate the coefficient of determination, yielding the following:

$$VIF_j = \frac{1}{1 - R_j^2}, \text{ Where } j = 1, \ldots, k$$

Ridge Regression

To combat multicollinearity specifically, ridge regression was developed and is a useful technique. Relevant to our discussions of norms earlier (L1 versus L2), ridge regression uses an L2 norm to achieve an optimal solution. Here is the equation for ridge regression:

$$\arg\min_{\beta} \left\| y - X\beta \right\|_2^2 + \lambda \left\| \beta \right\|_2$$

One of the key distinctions in ridge regression is the tuning parameter λ, which determines the degree to which the regression coefficients shrink. The technique gets the name *ridge* due to the fact that the L2 norm forms a spherical or circular shaped region where the optimal solutions for the regression coefficients exist are chosen along the "ridge" of this shape. Visually, this often looks like Figure 3-9.

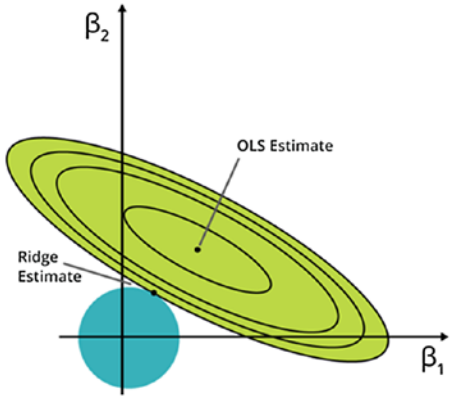

Figure 3-9. *Ridge regression OLS estimates*

Least Absolute Shrinkage and Selection Operator (LASSO)

Lasso is very similar to ridge regression except LASSO performs variable selection while regressing the explanatory and response variables. The key differentiation between LASSO and ridge regression is the fact that LASSO uses the L1 norm rather than the L2 norm, giving the selection region a square or cubic shape depending on the dimensionality of the data. In Figure 3-10, we can see the LASSO OLS estimate:

$$\arg\min_{\beta} \|y - X\beta\| + \lambda \|\beta\|$$

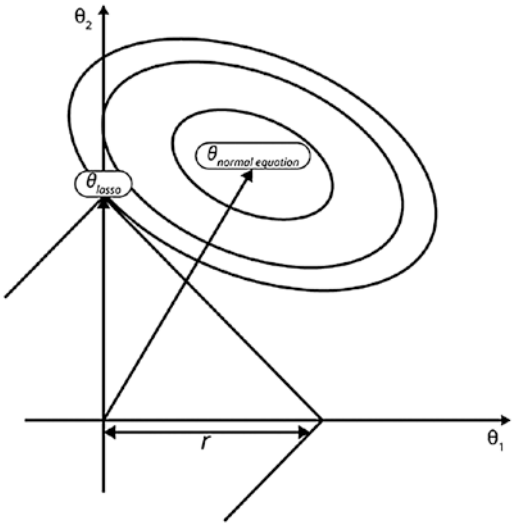

Figure 3-10. *LASSO regression*

Comparing Ridge Regression and LASSO

Both methods are highly useful for instances in which your data suffers from multicollinearity, but in instances where you're seeking to fit data as you would in a simple linear regression, these methods should be avoided, and the gradient methods along with the OLS method given earlier should be used. If you don't have more than one explanatory variable, these methods won't be of use to you. Although that's unlikely to be the case in practical terms most times, it's important to remember nonetheless.

Evaluating Regression Models

Beyond just building regression models, we need to find a way to determine how accurately the results yielded from a model are, and ultimately choose the best one on a case-by-case basis. In the case of regression, a useful method of evaluating machine learning models is by bootstrapping. Typically, *bootstrapping* involves running different regression models over several iterations using a data set that's smaller than the original, and with the original observations in randomized order, and then sampling several statistics and comparing their values relative to the other models' values. The process is as follows:

1. Build several models.
2. Collect sample statistics that we use as evaluators of each model over N iterations of the experiment.

3. Sample each of these evaluators and collect statistics upon each iteration, such as:
    a. Mean
    b. Standard deviation
    c. Max
    d. Min
4. Evaluate the results and pick the model that's most effective given your objectives and situational constraints.

The rest of this section covers the evaluators you should pick during bootstrapping.

Coefficient of Determination (R^2)

As described in Chapter 2, the coefficient of determination is what we use to evaluate how accurately a model explains variability in y through the variability in x. The higher the R^2 value, the better. That said, generally speaking, "good" R^2 values should be in the following range: $.70 \leq R^2 \leq .95$. Anything lower than .70 should be viewed as generally unacceptable, and anything higher than .95 should be examined to see if there is overfitting in the model. Although this won't change across a given iteration very much, we still should evaluate this objectively across models.

Mean Squared Error (MSE)

The MSE measures the distance of a given predicted value of y from the average value of the actual response variable. Our objective with any regression model is to minimize this statistic as much as possible, so we will want to pick the model that has the lowest MSE relative to the others being examined. This will be the evaluator that shows the most variance across models and should be the one that gives us the most inferential power with respect to which model we should choose.

Standard Error (SE)

In the case of a regression model, we would probably measure the standard error of a given model. The objective we should have should be to have a standard error that is as close to 0 as possible. Highly negative or highly positive standard error values are generally undesirable.

Classification

Moving beyond the case of predicting specific values, our data observations often belong to some class that we would like to label them as such. We refer to this paradigm of problems as *classification problems*. To introduce readers to these types of problems, we begin by addressing the most elementary of these algorithms: logistic regression.

Logistic Regression

In addition to regression, one of the important tasks of machine learning is classification of an observation. Although there are multinomial classification algorithms, we will start by examining a *binary classifier*, a method often used as a baseline for the remainder of the classifiers. Logistic regression gets its name from the function that powers it, known as the *logistic function*, illustrated in Figure 3-11.

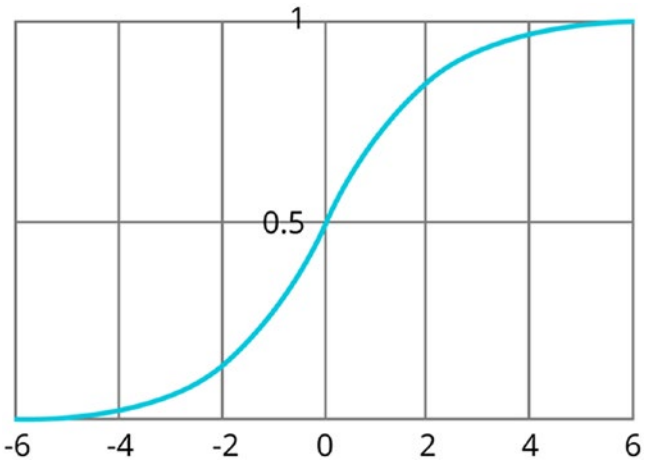

Figure 3-11. *Visualization of logistic function*

The function itself reads this this:

$$f(x)=\left(\frac{1}{1+e^{-x}}\right)$$

The intuition behind how we classify an observation is simple: we set a threshold for a given *f(x)* value and then classify it as a 1 if it meets or exceeds this threshold and a 0 if otherwise. In many contexts, the *x* variable will be replaced by a linear regression formula, in which we model the data. As such, the equation for *f(x),* or the log odds, will be

$$\pi = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

where π = log odds and π* is the given threshold. As for the threshold we establish, that depends on what we would like to maximize: accuracy, sensitivity, or specificity.

- *Sensitivity/recall*: The ability of a binary classifier to detect true positives:

$$True\ Positive\ Rate = \frac{True\ Positives}{Positives} = \frac{True\ Positives}{True\ Positive + False\ Negatives}$$

- *Specificity*: The ability of a binary classifier to detect true negatives:

$$Specificity = \frac{True\ Negative}{Negatives} = \frac{True\ Negatives}{True\ Negatives + False\ Positives}$$

- *Accuracy*: The ability of a binary classifier to accurately classify both positives and negatives:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Positives + Negatives}$$

In certain contexts, it may be more advantageous to magnify any of these statistics, but that's all relative to where these algorithms are being applied. For example, if you were testing the probability of a phone battery combusting, you would probably want to be certain that false negatives are minimized as much as possible. But if you were trying to detect the probability that someone is going to find a match on a dating website, you probably would want to maximize true positives. The relationship between the tradeoff of these predictive abilities is most easily exemplified using an ROC curve, which shows how altering the value of π^* affects the classification statistics of the model.

Receiver Operating Characteristic (ROC) Curve

The ROC curve initially was used during World War II for the purposes of radar detection, but its uses were soon considered for other fields, statistics being one of them. The ROC curve displays the ability of a binary classifier to accurately detect true positives and simultaneously check how inaccurate it is by displaying its false positive rate. This is shown in Figure 3-12.

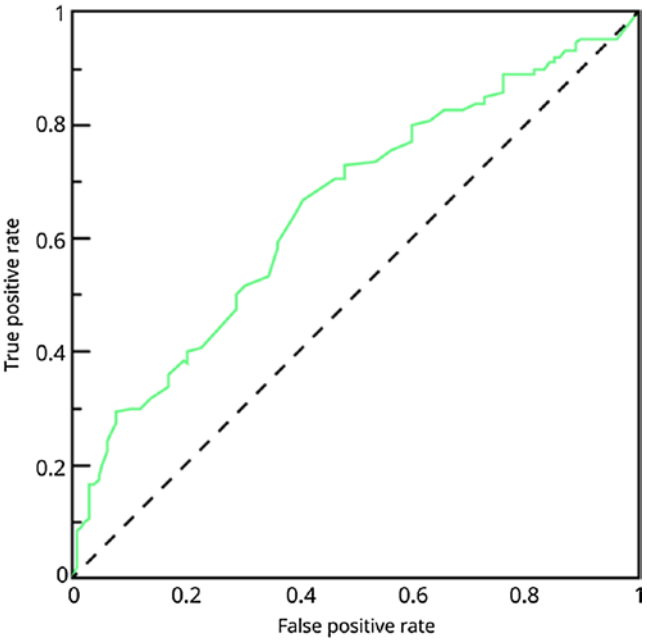

Figure 3-12. *Example ROC curve plot*

In the context of logistic regression, the evaluation of any specific model, given a specific threshold for π, is ultimately determined by the *area under the curve*, or AUC. The vertex of the plot is the .50 AUC score, which indicates that the model, should its score be this, is no better at classifying than a random guess. Ideally, this AUC score would be as close to one, but we generally accept anything $\geq .70$ as acceptable.

Confusion Matrix

Another method of evaluating classification models is the *confusion matrix*, a graphical representation of the classifiers predictions against the actual labels of a given observations. From this visualization, we derive the values for the statistics listed previously that ultimately help us accurately evaluate a classification model's performance. Figure 3-13 shows a visual example of a confusion matrix.

	P′ (Predicted)	n′ (Predicted)
P (Actual)	True Positive	False Negative
n (Actual)	False Positive	True Negative

Figure 3-13. *Confusion matrix*

Interpreting the values within a confusion matrix often is a subjective task that is up to the reader to determine. In some instances, false positives, such as determining whether users should buy a product or not, will not be as detrimental to solving the problem at hand. In other cases, such as determining whether a car engine is faulty, false positives may actually be detrimental. Readers should be conscious of the task they are performing and tune the model to limit the false positives and/or false negatives accordingly.

Limitations to Logistic Regression

Logistic regression can only predict discrete outcomes. It requires many of the assumptions necessary for ordinary linear regression, and overfitting of data can become quite common. In addition to this, classification with logistic regression works best when we have data that is clearly separable. For these reasons, in addition to the fact that there are more sophisticated techniques available, it is a common modeling practice to consider the logistic regression model to be the baseline by which we juxtapose other classification methods and observe the nuances.

Moving forward, we will look at a simple example using logistic regression. This data set will be referenced in later chapters, and in Chapter 10 in detail, for those who are curious about the process by which this model was produced.:

```
#Code Redacted, please check github!
#Logistic Regression Model
lr1  <- glm(data[,1] ~ data[,2] + data[,3] + data[,4] + data[,5] + data[,6]
+ data[,7],
            family = binomial(link = "logit"), data = data)

#Building Random Threshold
y_h <- ifelse(lr1$fitted.values >= .40, 1, 0)
```

```
#Construct ROC Curve
roc(response = data[,1], predictor = y_h, plot=TRUE, las=TRUE,
    legacy.axes=TRUE, lwd=5,
    main="ROC for Speed Dating Analysis", cex.main=1.6, cex.axis=1.3,
    cex.lab=1.3)
```

The ROC curve for our model is shown in Figure 3-14.

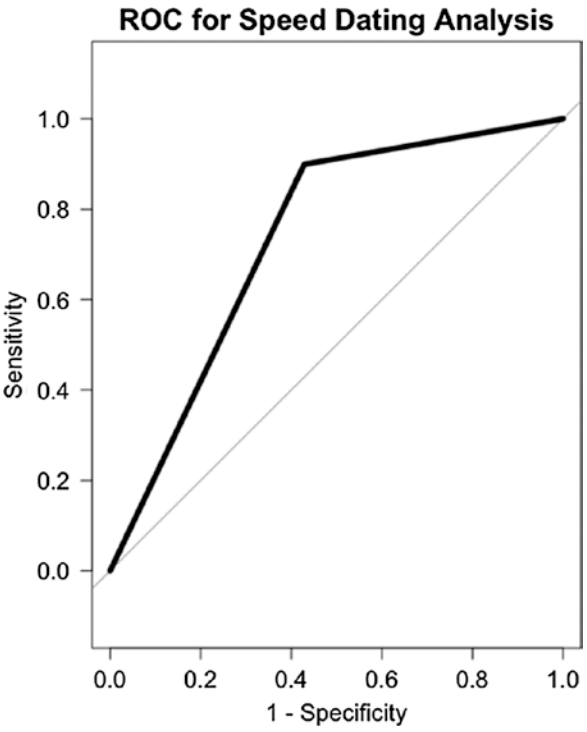

Figure 3-14. *ROC curve for logistic regression example*

Using the preceding code, we have an area under the curve of .7353. Given the threshold that we set before, this model's performance would be considered acceptable, but it should likely undergo more tuning.

Support Vector Machine (SVM)

Among the more sophisticated machine learning models available, *support vector machines* are a binary classification method that has more flexibility than the logistic regression model in that they can perform non-linear classification. This is performed via its kernel functions, which are equations that orthogonally project the data onto a new feature space, and the classification of the objects are performed as a product of two hyperplanes constructed by a norm (See Chapter 2).

In the case of liner SVMs, we take in as our inputs a response variable, Y, and an explanatory variable(s), x. We orthogonally project this data into feature space, such that we form the hyperplane that separates the data points. The size of these hyperplanes is determined by the Euclidean norm of the weights, or w, vector in addition to an upper bound and lower bound, respectively denoted as the following:

$$wx + b = 1$$

$$wx + b = -1$$

We keep reiterating this process until we have reached a norm of w that maximizes the separation between the two classes. The separation of the classes is maximized by minimizing ||w||, being that the size of the hyperplane is given by the following:

$$\frac{2}{\|w\|}$$

The following constraints also prevent us from allowing observations to fall in between the two hyperplanes:

$$wx + b \geq 1, \text{ if } y = 1$$

$$wx + b \leq -1, \text{ if } y = -1$$

The observations that ultimately fall on the boundaries of the hyperplane are the most important, as they are the "support vectors" that define the separation between classifications. This transformation is shown in Figure 3-15.

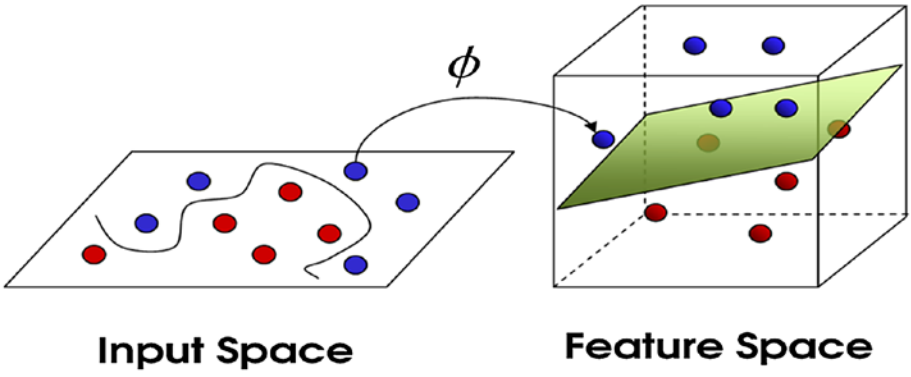

Figure 3-15. *Orthogonal transformation of data via kernel function*

The optimization problem formed from these constraints in addition to objective of the SVM is given by the following:

$$\text{Minimize} \|\text{w}\| \text{subject to } y_i(wx+b) \geq 1, \text{ for i} = 1, \ldots, \text{n}$$

Types of Kernels

To expand the flexibility of SVMs, different kernels have been developed. Among them are the following:

- Polynomial
- Gaussian radial basis function
- Hyperbolic tangent

Sub-Gradient Method Applied to SVMs

A sub-gradient of a function is defined as a generalization of a derivative to a function that are not differentiable. Simply stated, it is the slope of a line that goes through the derivative of a function, but falls below the derivative. Modern modifications to the SVM algorithms have yielded better performing classification models when dealing with data with more than 10^5 features and 10^5 observations. We define the optimization problem as

$$f(w,b) = \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(wx+b)\right)\right] + \lambda\|w\|^2$$

where *f* is a convex function of w and b. Moreover, this allows us to use gradient decent methods because they work particularly well on convex sets. Given a cost function C(w), defined as the actual classification minus the predicted classification, we use the gradient descent formula therefore as follows:

$$\min_{w\in\mathbb{R}^d} \mathcal{C}\sum_{i=1}^{n}\max\left(0, 1 - y_i f(x_i)\right) + \|w\|^2$$

$$\min_{w} \mathcal{C}(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i f(x_i)\right)$$

The sub-gradient step size selection method is similar to the bold driver approach described earlier in this chapter. As always, the context in which an algorithm is being applied should ultimately decide which method is used, not just which performs better.

Extensions of Support Vector Machines

A regression method proposed in 1996 by Vladimir Vapnik, Harris Druck, Christopher Burges, Linda Kaugman, and Alexander Smola is among the more popular extensions of SVMs. The difference is that in SVR, we don't care about the observations that fall within the hyperplane. Instead, we only modify the shape of the hyperplane in response to points that fall outside the loss tolerance zone, with the objective of minimizing the amount of these points that fall out of this zone. The type of regression performed can be altered again by the same kernel functions listed earlier. In addition to this, an alternative to K-means clustering which uses the Gaussian kernel as the activation function for orthogonal projection. Here, the algorithm searches to make the hyperplane such that the smallest sphere that encloses the image of the data defines a given cluster. Clustering algorithms are covered later in this chapter.

Limitations Associated with SVMs

The main problem with SVMs arises from what is arguably the key to why they are so powerful: kernel functions. Determining the proper kernel to use is often stated as the largest drawback to this technique. When delving deeper into this aspect of algorithm training, specifically the selection of the loss parameter and the Gaussian kernel's width parameter is not apparently obvious and is highly subject to the context in which the algorithm is being used. Second, although SVMs do perform well on large data sets, they are a computationally expensive method and require sufficiently good hardware when applied in an industry setting. As such, it does not always make sense to use SVMs in contexts outside of research, or any context where real-time data would be analyzed.

The following is a quick example of SVMs used on the iris data set:

```
#Code Redacted, please check github!
require(LiblineaR)
require(e1071)

#SVM Classification
output  <- LiblineaR(data=s, target=y_train, type = 3, cost = heuristicC(s))

#Predicted Y Values
y_h <- predict(output, s, decisionValues = TRUE)$predictions

#Confusion Matrix
confusion_matrix <- table(y_h, y_train)
print(confusion_matrix)
```

When executing our code, it yields the confusion matrix shown in Figure 3-16.

```
   y_train
y_h  1  2  3
  1 34  0  0
  2  1 18  8
  3  0 15 24
```

Figure 3-16. *Confusion matrix for support vector machine*

Machine Learning Methods: Unsupervised Learning

Moving beyond the paradigm in which we know the answers we're trying to predict is the more ambiguous section of deep learning, in which we are trying to make inferences based off of our algorithms. This specific subset of problems is known as being a part of *unsupervised learning*, or problems where we don't know a priori what the answers should be.

K-Means Clustering

Until now, we've spoken primarily about supervised learning, but another important aspect of machine learning is the use of algorithms in unsupervised learning cases. Typically, unsupervised learning can be performed as an exploratory research method, or as a preliminary step prior to the primary component of the experiment. One of the best examples of unsupervised learning is the K-means clustering algorithm. The motivation behind this algorithm is to find observations that are similar based on the distance they are away a cluster center.

Assignment Step

Here, we take the observations of data and give an initial set of k means by calculating the means of three random observations within the data. From this point, we assign each observation to the cluster centers based on which assignment yields the smallest within cluster sum of squares, determined by the Euclidean norm between the observation's mean and the cluster center mean

$$S_i^t = \left\{ x_p : \left\| x_p - m_i^t \right\|^2 \le \left\| x_p - m_j^t \right\|^2 \ \forall j, 1 \le j \le k \right\}$$

where S = cluster center and x_p is an observations mean.

Update Step

We then recalculate the cluster mean by taking the mean of the observations within the center and then reiterate over these two steps until reassignments stop:

$$m_i^{t+1} = \frac{1}{\left|S_i^t\right|} \sum_{x_j \in S_i^t} x_j$$

Limitations of K-Means Clustering

The major problem with K-means clustering is that the solution reached is often dependent on where the means are initialized, and therefore convergence upon a global minimum isn't guaranteed. Also, depending on the variation of K-means chosen, the time taken until convergence may also not be particularly fast.

Here's a brief example of K-means clustering:

```
#Upload data
data  <- read.table("http://statweb.stanford.edu/~tibs/ElemStatLearn/
datasets/nci.data", sep ="", header = FALSE)
data <- t(data)
k_means  <- c()
k  <-  seq(2, 10, 1)
for (i in k){
  k_means[i]  <- kmeans(data, i, iter.max = 100, nstart = i)$tot.withinss
}

clus <- kmeans(data, 10)$cluster
summ  <- table(clus)
#Removing NA Values
k_means  <- k_means[!is.na(k_means)]
#Plotting Sum of Squares over K
plot(k, k_means,  main ="Sum of Squares Over K-Clusters", xlab = "K
Clusters", ylab= "Sum of Squares",
     type = "b", col = "red")
```

Typically when performing K-means clustering, the most difficult part is determining which value of k we should pick. Typically, the more clusters one has, the lower the sum of squares within a cluster between its observations and the cluster centers will be. However, the more clusters that are present, the less informative these clusters are. Therefore, the challenge becomes a tradeoff between sum of squares over the K clusters and as least clusters as possible to make the observations reasonably differentiable. Figure 3-17 shows a plot that aids us in that effort.

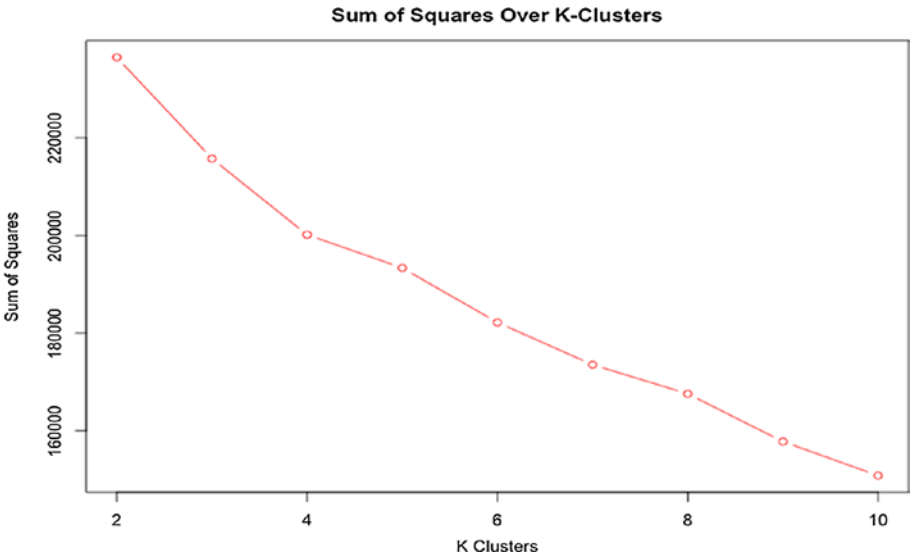

Figure 3-17. *Within cluster sum of squares over K clusters*

In the plot in Figure 3-17, we notice that our sum of square decreases dramatically in the beginning, but we see that a tapering off toward the end in the value changes. As such, it's reasonable for us to choose a value between 6 and 8, preferably closer to, if not, 6. This follows the objectives laid out in the prior paragraph and would yield us with actionable insights, or create a feature for a data sight that contains significant differences for a classification or regression algorithm to detect.

Expectation Maximization (EM) Algorithm

Popular within the paradigm of unsupervised learning, EM algorithms can be used for a multitude of purposes such as classification or regression. Most specifically of use to the user, it can be used to impute values that are missing within a data set. We will show this capability in Chapter 11. Regardless, the EM algorithm is a probabilistic model, which distinguishes it from many machine learning models, which often tend to be deterministic. The algorithm uses the log-likelihood function to estimate the parameter and then maximizes the expected log-likelihood found.

Expectation Step

Consider a set of unknown values Z, which is a subset of the data set X. We calculate the log-likelihood of a parameter with regard to the conditional distribution of Z given X. The following equation yields the expected value of the maximum likelihood estimate of the parameter:

$$L(\theta;X)=p(X|\theta)=\sum_z p(X,Z|\theta)$$

$$Q(\theta|\theta^t)=E_{(Z|X,\theta^t)}\left[\log L(\theta;X,Z)\right]$$

Maximization Step

In this step, we seek to maximize the probability of the given parameter we are analyzing. The equation is given by the following:

$$\theta^{t+1}=\arg\max_{\theta} Q(\theta|\theta^t)$$

Limitations to Expectation Maximization Algorithm

The EM algorithm also tends to be very slow to converge and doesn't yield the asymptotic variance-covariance matrix of the MLE. In addition to this—similar to the same limitation with naïve Bayes classifiers, because the MLE estimator assumes feature independence—it would be ill advised to use this method if the features being analyzed are in fact not independent. The following is an example of the EM algorithm used for classification via clustering:

```
#Expectation-Maximization Algorithm for Clustering
require(MASS)
require(mclust)
y_h <- Mclust(x_train, G = 3)$classification
print(table(y_h, y_train))
plot(Mclust(x_train, G = 3), what = c("classification"), dimens=c(1,3))
```

When executing our code, it yields the plot shown in Figure 3-18.

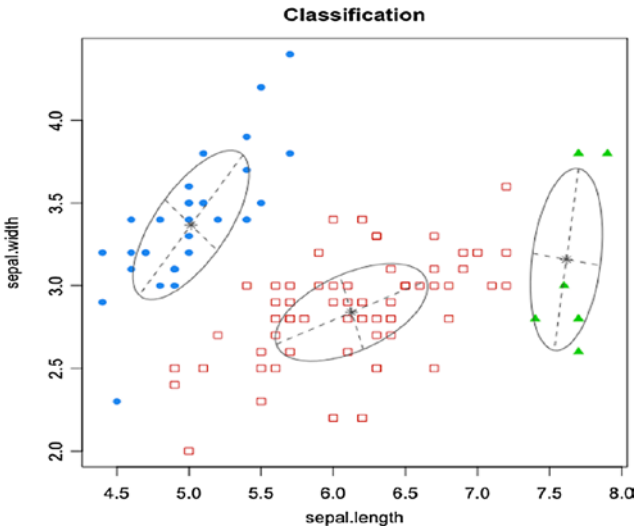

Figure 3-18. *Iris data clusters from EM algorithm clustering*

Decision Tree Learning

Commonly used across a variety of fields for the purpose of data mining, *decision trees* yield a relatively simple method of uncovering insights hidden below the surface of data. There are broadly two types of decision tress, and they typically are used for regression and classification. Decision trees are constructed by creating a rule that determines which direction the decision flows. The idea is that you use a funnel methodology in which the first rule is the broadest and you break down the questions into subsets until the final "leaf" is the most granular aspect determined.

The benefits often associated with decision trees is that overall, they are relatively easy to understand and generally quite effective. In addition, decision trees can handle missing data better than some machine algorithms can without replacing or changing the data (we can just average the values or classifications), and they are quick to compute final values relative to other modeling techniques. Above all, there are varieties of methods that can be used to help the trees learn effectively, and they can model data well when traditional regression methods cannot.

Classification Trees

Classification trees are similar to regression trees. The splits are usually determined by binary variables, but they can be both numerical and categorical. In addition to this, classification trees can make two types of predictions: 1) *point prediction*, which simply denotes the class, and 2) *distributional prediction*, which gives a probability for each class. For probability forecasts, each terminal node in the tree yields a distribution over the classes. If the leaf corresponds to the sequence of answers, given by $A = a$, $B = b$, … $Q = q$, then the following equation yields the probability:

$$\Pr(Y = y \mid A = a, B = b, \ldots, Q = q)$$

To evaluate the classification tree, the same methods of evaluating different classification models as described earlier are used. But we also introduce the concept of average loss. Simply stated, some errors are likely to cause greater "damage" toward accurately reaching the correct classification. The *average loss* formula is given by the following:

$$\text{Loss}(\text{Y} = \text{j} \mid \text{X} = \text{x}) = \sum_i L_{ij} \Pr(Y = i \mid X = x)$$

Moving beyond this, we can determine whether the model made an incorrect classification in cases where it was or was not uncertain using the normalized negative log-likelihood. The formula for it is given by

$$L(data, Q) = -\frac{1}{n} \sum_i^n logQ(Y = y_i \mid X = x_i)$$

where $Q(Y = y|X = x)$ is the conditional probability the model predicts. In this context, L is also referred to as *cross-entropy*. If perfect classification were possible, L would be 0. If there is some irreducible uncertainty in the classification, the best possible classifier would give $L = H[Y|X]$, the conditional entropy of Y given X. Less than ideal predictors have $L > H[Y|X]$. Here is an example of a classification tree:

```
require(rpart)
#Classification Tree
classification_tree <- rpart(y_train ~ x_train[,1] + x_train[,2] + x_
train[,3] + x_train[,4]
                             +x_train[,5] + x_train[,6], method = "class")
pruned_tree <- prune(classification_tree, cp = .01)

#Data Plot
plot(pruned_tree, uniform = TRUE, branch  = .7, margin = .1, cex = .08)
text(pruned_tree, all = TRUE, use.n = TRUE)
```

When executing our code, we yield the plot shown in Figure 3-19.

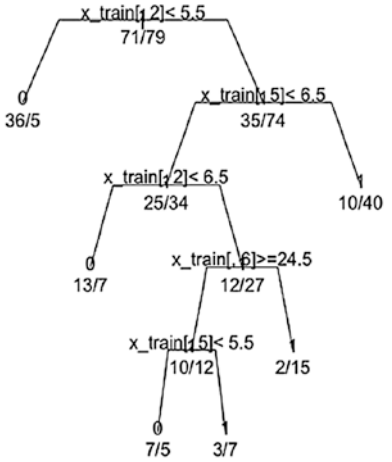

Figure 3-19. *Classification tree splits based on Classification tree model fitted above*

The confusion matrix accompanying this model is shown in Figure 3-20.

```
   y_train
y_h  0  1
  0 56 17
  1 15 62
```

Figure 3-20. *Confusion matrix for classification tree*

Regression Trees

The primary goal in this model is to maximize the probability of landing up at a given leaf as a product of the variables being analyzed. We seek to maximize the information we get about the response variable upon each split we approach. This is modeled by

$$I[C:Y]$$

where I is information, C is the variable that determines the leaf we move toward, and Y is the response variable

$$I[Y;A]=\sum_a \Pr(A=a)I[Y;A=a]$$

where,

$$I[Y;A=a]=H[Y]-H[Y\mid A=a]$$

where

$$H(X) = E\left[I(X)\right] = E\left[-\ln\left(P(X)\right)\right]$$

Regardless of whether we are looking at continuous or discrete variables, we calculate the sum of squares the same way

$$S = \sum_{x \in leaves(T)} \sum_{i \in c} \left(y_i - m_c\right)^2$$

where $m_c = \frac{1}{n_c} \sum y_i, i \in c,$ the prediction for leaf c.

Uncertainty in prediction using regression trees, similar to the uncertainty seen in classification trees, is an issue worth considering when employing these models. Primarily, these uncertainties are imprecise estimates of the conditional probabilities. The tree is also actively changing as the response values shift. We would ideally like a measure of how different the tree could have been if we drew a different sample from the same distribution. This can be estimated using *non-parametric bootstrapping*. Assuming data (x_1, y_1), (x_2, y_2), …, (x_n, y_n), we draw a random set of integers J_1, J_2, …, J_n, independently and uniformly from the numbers 1:n, with replacement. Then we set

$$\left(X_i', Y_i'\right) = \left(x_{J_i}, y_{J_i}\right)$$

where we treat this bootstrapped sample just like the original data and fit a tree to it. Repeated over many iterations, we get a bootstrap sampling distribution of trees. This approximates the actual sampling distribution of regression trees. The spread of the prediction of our bootstrapped trees around the original indicates the distribution.

Limitations of Decision Trees

Typically, the most difficult parts of building a decision tree are choosing the rule that creates the best decision tree and choosing a tree size that isn't overly complex, which leads to overfitting in the training set, or one that doesn't yield any actionable insights at all. To make things worse, it's difficult to tell when exactly overfitting occurs just from the training error alone. To mitigate these problems, it is generally encouraged that decision trees have a sufficient training example size. Ideally, the model fits to the data reasonably well, and the rules employed to determine splits in direction should not be overly complex. The stopping criterion ultimately controls when we reach a leaf. Examples of often-used rules are to stop when the information yielded decreases below a certain user-determined threshold or when the "child" of the "parent" node yields a sufficiently small enough set of data points. Moving forward from this however, decision trees are relatively simple models that don't always perform very well on complex data with respect to regression problems, and also don't perform well on categorical data where there are multiple levels for each category.

Ensemble Methods and Other Heuristics

For instances in which standard machine learning algorithms fail, a significant boost in accuracy can be achieved from algorithms that are in actuality combinations of multiple algorithms. We refer to these as *ensemble methods*.

Gradient Boosting

Originally developed by Leo Breiman, *gradient boosting* is a technique used on regression and classification problems for the purposes of producing a superior model from weaker models. It builds the model iteratively, and the optimization problem is to minimize the gradient of this function. Let's take a model F, which we expect to predict a value y_h, with the objective of minimizing the squared error. Let M be the number of boosting iterations we want to go under, where $1 \le m \le M$. We assume that at the outset of our experiment we will have a model F_m, which we seek to improve. Therefore:

$$F(x)_{m+1} = F(x)_m + h(x) = y$$

$$h(x) = y - F(x)_m$$

Gradient boosting seeks to make F_{m+1} more correct than the previous model. Other loss functions that have been proposed are the squared error loss function given by

$$h(x) = \frac{1}{2n} \Sigma \left(h_{\theta_x}^i - y^i \right)^2$$

where

$$\nabla h(x) = \frac{1}{n} \Sigma \left(h_{\theta_x}^i - y \right)$$

where n = the number of observations within data set X.

Gradient Boosting Algorithm

1. Define the optimization problem as

$$F^* = \arg\min_F E_{x,y}[L(y, F(x))$$

where $L(y, F(x))$ is some differentiation loss function, such as the gradient of the squared loss as shown earlier.

2. Calculate the residuals as given by the following equation for $m = 1, \ldots, M$:

$$r_{im} = -\left[\frac{\partial L\left(y_i, F\left(x_i\right)\right)}{\partial F\left(x_i\right)}\right]_{F(x)=F(x)_{m-1}}$$

3. Use the initial model with a training set to iteratively improve the performance.

4. Calculate γ_m via the following equation:

$$\gamma_m \arg\min_{\gamma} \sum_i^n L\left(y_i, F\left(x_i\right)_{m-1} + \gamma h_m\left(x_i\right)\right)$$

5. Repeat 2–4 until convergence.

Random Forest

The final ensemble method I will address in this chapter is that of the random forest. Simply stated, *random forests* are combinations of several decision trees, such that each decision tree can be considered unique from the others with respect to the features it evaluates at a given branch. Although the length of these trees is homogenous, each tree's decision is independent from one another. The value we choose for a given observation typically is the average value with respect to all the trees in the case of regression, or it's the average (or most prevalent) observation with respect to all of the trees in classification.

Limitations to Random Forests

Random forests' main limitations is the fact they, similar to the trees they are made of, have a tendency to overfitting. The same techniques I recommended for use on decision trees, such as pruning and preemptively limiting growth, should be used here to limit the probability of a tree overfitting.

Bayesian Learning

Built off of Bayes' theorem, and ultimately employed in many machine learning and natural language processing models, Bayesian learning uses representations of random variables and their conditional dependencies via a directed graph. Bayesian learning is used in situations such as determining the sentiment of a word given the context it is within, and finding the probability of a name being that of a female or a male based on the genders it typically is prescribed to within a test set.

Naïve Bayes Classifier

A simple application of Bayes' theorem is to the case of classification. Naïve Bayes classifier uses conditional probability to determine the likelihood of an event. Let's say we have a vector $z = [z_1, z_1, \ldots, z_3]$ and we want to determine the probability of event A. We would model this equation as

$$P(A|z) = \frac{P(z|A)P(A)}{P(z)}$$

where *P(A|Z)* is defined as the posterior probability, *P(z|A)* is the prior probability, *P(A)* is the likelihood, and *P(z)* is the probability of the instance occurring (this can almost always be ignored). Now we want to use this formula to properly be able to classify observations. To this, we turn this into an optimization problem, given by the following equation:

$$\hat{y} = \arg\max_{k \in \{1,\ldots,K\}} P(A_k) \prod P(z_i | A_k)$$

We assign a value to *y* based on the value that maximizes the probability of some event *A*. Although this isn't the only way to use a naïve Bayes classifier, it's an example of one of the more common ways Bayes' theorem is applied for the purpose of classification.

Limitations Associated with Bayesian Classifiers

Bayesian classifiers' biggest limitation lies mostly the fact that it assumes the independent nature of features, which won't always be the case in many contexts in which we're analyzing data. Once it's established that feature independency doesn't exist, we can't use this classifier at all:

```
#Bayesian Classifier
require(e1071)

#Fitting Model
bayes_classifier <- naiveBayes(y = y_train, x = x_train , data = x_train)
y_h <- predict(bayes_classifier, x_train, type = c("class"))

#Evaluating Model
confusion_matrix <- table(y_h, y_train)
print(confusion_matrix)
```

When executing the preceding code, the confusion matrix shown in Figure 3-21 is yielded.

```
   y_train
y_h  0  1
  0 56 17
  1 15 62
```

Figure 3-21. *Confusion matrix for Bayesian classifier example*

Final Comments on Tuning Machine Learning Algorithms

One of the more difficult parts of practicing machine learning algorithms that I've not addressed yet is the concept of *parameter tuning*. The amount of parameters one can tune depends on which algorithm is being employed, but nonetheless this is a challenge that is noted across the entire discipline. We have discussed why it's important to ensure that overfitting doesn't occur so we achieve as robust a solution as possible. Generally speaking, robustness is reflected by stability of prediction power from one data set to the next, and overfitting is reflected by stark drop-offs in predictive power from on data set to the next. I'll now discuss how to achieve this robustness via methods in the following sections.

50/25/25 Cross-Validation

Users should use a validation set to do the parameter tuning against, which should be 50% of the size of the total data set. Then the users should create two training sets: one will be used to train their tuned algorithm, and the other to test the degree of robustness/check for overfitting. Other percentage splits can be examined to see the difference in performance.

Tune One Parameter at a Time

Should the reader be using the packages and not a custom implementation of an algorithm, there will likely be parameters that are set to default values. Trying to change more than one parameter at a time is difficult not only for the sake of the results of the algorithm being yielded in a timely fashion, but also due to the fact that it's hard to separate the contribution of specific parameters from the degree of change in the output. For example, random forests get a great deal of their power from the largeness of the individual trees as well as from the amount of trees allowed to have within a given model. Augmenting both of these at the same time distorts the ability to which we can properly tune the algorithm as a whole and ultimately can lead to under or overfitting.

Using Search Algorithms to Tune Machine Learning Parameters

Readers who want to take a more advanced approach are advised to pay close attention in Chapter 8 where we discuss in depth search algorithms that can be used to choose machine learning algorithms. Although still a developing area of research, there has been significant success achieved with using GridSearch and other local search algorithms to choose better algorithms by lowering an error statistic of a regression algorithm or increasing an AUC score yielded by a classification algorithm.

Reinforcement Learning

Reinforcement learning differs from supervised learning in that the labels we are fitting against in supervised learning problems are never given. Instead, there is a focus on finding the proper balance between leveraging existing knowledge in the model and knowledge that we want the model to find from the environment which is not already known. Integral to the field of reinforcement learning is this subtopic of *probability theory*. In these type of problems, we assume there is a gambler near a group of slot machines who has to decide which machines to play, how many times to play each machine, and in which order to play the machines. When played, each machine provides a random reward from a probability distribution specific to a given machine. The objective is to maximize the amount of money that the gambler will have taken from this period of gambling. Moving forward, we can generally describe the reinforcement learning problem as one that requires an intelligent exploration of an environment, in reference to the same objectives described in the multi-armed gambler approach.

Distinguishing reinforcement learning from supervised and unsupervised methods is the fact that the actions we take significantly affect the subsequent information we get, hence the emphasis on making the best possible decision upon each iteration of a given algorithm. The basic algorithm is described as following:

- Agent
    1. Execute a given action
    2. Observe a certain outcome
    3. Receive a reward, usually modeled in the form of a scalar
- Environment
    1. Receives action performed by the agent
    2. Outputs an observation as well as a scalar reward