Statistics

Statistical Learning

1. What is statistical learning?

1.0. Notation used in the book

- Random variables are represented by upper case letters (Y, X)
- E as in "expectation" is the expected value of a *random variable* - i.e., it is the long-run average value of repetitions.
    - $E(Y - \hat{Y})^2$ evaluates the exact same way as $(Y - \hat{Y})^2$, but the *E* reminds us that the calculated value is a long-run approximation.
- *n* represents the number of cases (i.e., distinct observations)
- *p* represents the number of variables that can be used in prediction

Matrix notation

- Bold capital letters (**Y**) are used to represent matrices
- Lowercase lettters (y) are used to represent vectors
- $x_{i,j}$ where *i* represents the observation (row) and *j* represents the variable (column) in a matrix.

  A row vector (i.e., one observation), x_i, is described by $x_{i1}, x_{i2} \ldots x_{ip}$, where *p* is the p^{th} variable

  A column vector, x_j (i.e., all the values for one variable), is described by $x_{1j}, x_{2j} \ldots x_{nj}$, where *n* is the n^{th} observation
    - This is like a vector in a data frame in R.
    - Note the column vector x_j is not *italicized*
    - In general, both column and row vectors are represented as columns.
- $\boldsymbol{X^T}$ or x^T represents the transpose of a matrix or vector
    - In the case of a vector, recall that vectors are usually represented as columns; if that vector is a row vector then it may be written as x^T in order to represent it as a column, such that (As far as I can tell, you can just ignore the *T*)

1.1. Statistical Learning

The relationship between two related variables, Y and X, can be represented simply as:

$$Y = f(X) + \epsilon$$

Where f is a function of X,

and ϵ is irreducible, innate error

- We cannot remove *irreducible error*, but we can model them. Different statistical models take different approaches to model the errors; for example, linear regression assumes that the error terms are normally distributed.
- We don't know what *f(X)* is exactly, so we often try to estimate it.

Why estimate *f*?

- We want to use X to predict Y.
- We want to make inference about (for example):
    - Whether there is relationship between Y and X, and in what direction.

- How the relationship between Y and X can be summarized mathematically.
- Which variables are most strongly associated with Y?

1.1.1. Parametric vs. non-parametric methods

We estimate *f* using training data. Our goal is to identify a model such that: $Y \approx \hat{f}(X)$. Broadly speaking, most statistical learning methods can be summarized as either *parametric* or *nonparametric*.

Parametric methods

- Parametric methods make an assumption about the functional form/shape of *f*. For example, linear regression assumes that *f* is linear.
  - This greatly simplifies the procedure, because it means that we don't need to estimate as many parameters.
  - However, often the parametric form that we choose does not meet the actual form of *f*.Our estimate of *f* may be poor if the model that we assume is too far from the true form of *f*.
    - Additional parameters can be included to increase the flexibility of your model, but this method requires additional estimation and can lead to overfitting. For smaller data sets, parametric methods may be more appropriate because they are less susceptible to noise.
- After a model has been selected, we still need to choose which procedure we will use to train the model. For example, the ordinary least squares (OLS) method is a common approach to fitting a linear model, but other possibilities exist.

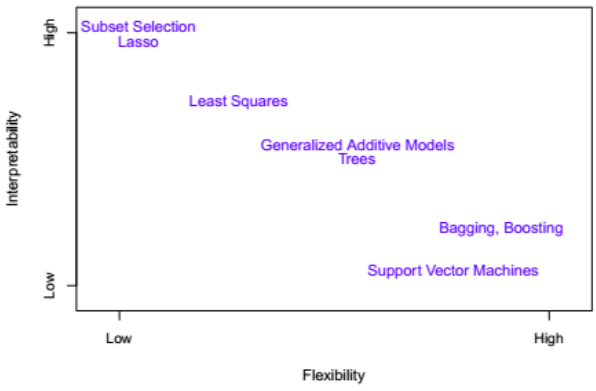
Non-parametric methods

- Non-parametric methods do not make explicit assumptions about the form of *f*. Instead, they seek to estimate *f* as closely as possible based solely on the data.
  - By avoiding assumptions about the form of *f*, these models have the potential to accurately fit *f* over a wide possibility of shapes.
  - Because non-parametric approaches do not simplify the model to a small number of parameters, we need a much larger number of observations to accurately estimate *f*. By extension, non-parametric methods are more susceptible to overfitting.
  - Non-parametric methods require more computational power.

Prediction accuracy vs. model interpretability

In general, more restrictive models are easier to interpret, while more flexible models, are harder to interpret.

- Restrictive models are often used in inference because it is easier to make sense of the output.
- Flexible models are often used in prediction because they have the potential to be very accurate (though this is not always the case).



1.1.2. Supervised vs. unsupervised learning

Most statistical learning problems can be classified as supervised or unsupervised.

Supervised

- For each observation there is an associated explanatory variable (X) and response variable (Y); we attempt to fit a model that will accurately predict future Y based on X.
- Statistical learning methods falling into this category include: linear regression, GAM, boosting

Unsupervised

- For each observation there is an associated explanatory variable (X), but no response variable (Y); we cannot perform methods like linear regression because there is no response variable that can 'supervise' the analysis.
- Without a response variable, we can still seek to understand the relationships between the variables or between the observations.
    - An example of this is cluster analysis.
- There is also 'semi-supervised' learning, which may occur if you have some observations with Y and some observations missing Y. There exist statistical learning methods (beyond this book) that can incorporate both the observations with Y and those missing Y.

1.1.3. Regression vs. classification

- We tend to refer to problems with a quantitative response variable as *regression* problems, whereas we refer to problems with a qualitative response variable as *classification* problems.
    - The distinction between them is not always that crisp. For example, logistic regression is often used for classification, but it also estimates class probabilities, and so can be thought of as a regression method as well.
    - Other methods such as K-nearest neighbours (KNN) and boosting are used in either context.
- In general, we choose a statistcal learning method based on the class of the response variable, rather than the predictors.

1.2. Assessing model accuracy

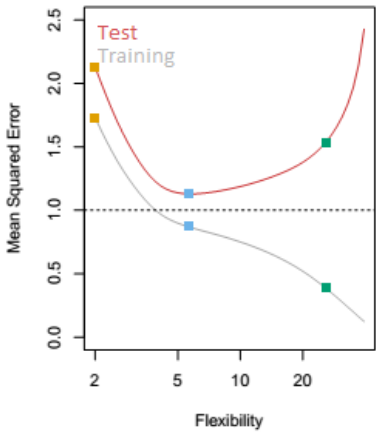
There is no 'best' method that dominates all other methods over all possible data sets. One model may perform very well on one data set, but worse on a similar data set. Thus, it is important to decide which method produces the best results for any given set of data.

In order to evaluate the performance of a statistical model, we need a metric to inform us on how well the predictions match the observed data. This is dicussed in more detail specific to each method later, but this section communicates important points which are common to all of these metrics.

In regression, we frequently use the *mean squared error* (MSE):

$$MSE = mean\ of\ squared\ error\ terms = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

By necessity, the training MSE for your model will be low. However, we are concerned with the test MSE. The relationship between the training MSE and test MSE is 'U' shaped: As we increase the flexibility of the model, the training MSE decreases; the test MSE decreases to a point before increasing again (due to over-fitting).



1.2.1. The bias-variance trade-off

The 'U-shaped' curve observed in the test MSE is a result of two competing properties of statistical learning methods: bias and variance.

It is possible to prove mathemetically (though outside the scope of this book) that the expected test MSE for a given observation can always be decomposed into three fundamental quantities:

- The variance of $\hat{f}(x_i)$
- The squared *bias* of $\hat{f}(x_i)$
- The variance of the error terms ϵ.

$$E((y_0 - \hat{f}(x_i))^2 = Expected\ MSE_{test} = Var\left(\hat{f}(x_i)\right) + Bias(\hat{f}(x_i))^2 + Var(\epsilon)$$

In order to minimize the test MSE, our statistical learning method must minimize *variance* and *bias*. Note that both of these terms are squared, so they are inherently non-negative.

Variance of $\hat{f}(x_i)$

- The variance of $\hat{f}$ is a measure of how much $\hat{f}$ would change if we estimated it using a different training data set. Since the training data are used to fit $\hat{f}$, different data sets will result in a different $\hat{f}$. Ideally, $\hat{f}$ should not change dramatically between training sets.
    - A model with high variance will change drastically with small changes in the training data.
    - An inflexible model will generally produce lower variance than a very flexible one.
    - A data set with too few observations or too many variables will generally produce a model with higher variance (as a result of overfitting from a flexible model), so inflexible models generally outperform flexible ones in these cases.

Bias of $\hat{f}(x_i)$

- Bias is the error that is introduced by approximating a real-life problem using a model. Simpler models that make broader approximations are more likely to introduce high bias.
    - An inflexible model will general have higher bias than a flexible one.

Variance of ϵ

- For reasons beyond what I can prove, is the irreducible error – therefore, the expected test MSE can never lie below this value. From what I can understand, you can't really do much with this term.
    - If Variance of ϵ is high, highly flexible models will overfit the data, which is largely noise.

In general, as we increase flexibility, we decrease bias and increase variance. Within a class of methods, as we increase the flexibility, the bias will decrease faster than the variance increases, so the test MSE will decrease. At a certain point, increasing flexibility will have less of an effect on variance and more of an effect on bias, so the test MSE will increase.

In real life, we lack knowledge of the true *f*, so it is not possible to explicitly compute the test MSE, bias, or variance for a particular method. Therefore, it is hard to know what level of flexibility will minimize these parameters.

1.2.2. Model accuracy in the classification setting

The most common metric for model accuracy in classification is the *error rate*, given by the proportion of false classifications (false positives + false negatives).

$$Error\ rate = \frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$$

Where $\hat{y}_i$ is the predicted class for the *i*th observation using $\hat{f}$,

And *I* is an indicator variable that equals 1 if $y_i \neq \hat{y}_i$ and 0 otherwise.

As with the MSE in regression, we are easily able to calculate the training error rate, but we are really interested in the test error rate.

It is possible to show through black magic that the test error rate is minimized, on average, by a very simple classifier that assigns each observation to the most likely class, given its predictor values. In other words, we should assign a test observation with predictor x_i to the class *j* for which the conditional probability that Y = *j* given X = x_i is largest:

$$max(P(Y = j|X = x_i))$$

This simple classifier is called the *Bayes classifier*. In a simple two-class problem with only two possible response values (1 or 2), the Bayes classifier corresponds to predicting class 1 if $P(Y = 1|X = x_i) > 0.5$, and class 2 otherwise. Classification using the Bayes classifier produces the lowest possible test error rate, called the *Bayes error rate*. The error rate at X = x_i is equal to 1 minus the Bayes classifier:

$$Bayes\ error\ rate = E(max(P(Y = j|X = x_i)))$$

The Bayes error rate is equivalent to the irreducible error discussed in the regression context; this is because the populations have some overlap which cannot be correctly classified with the available data.

In real life, as lack knowledge of the conditional distribution of Y given X, so computing the Bayes classifier is impossible; therefore, the Bayes classifier acts as an unattainable goal against which to compare other methods. Some approaches, such as KNN attempt to estimate the conditional distribution of Y given X.

2. Linear Regression

2.1. Least-squares regression and correlation

Review of notation:

X Parameter. The true value for the random variable *X*. This is always unknown, and typically we try to estimate it from the data.

$\bar{x}$ Statistic. The mean of the observed data, and our best estimate for the parameter *X*.

x_i The '*i*'th observation from our sample of the variable *x*.

$\hat{x}$ The predicted value of *x* calculated from our model, based on observed data (i.e., $x_1, x_2, \dots, x_n$)

You should recall some basics about regression, so I will not mention such details. When selecting a model in our linear regression model, we want to minimize error - that is, the distance between our predicted value and the observed value. There are many ways of doing this, but the least-squares method is the most common. In the least-squares method, we select a line that minimizes the sum of the squares of the vertical distances of all of the data points from the prediction.

$$"Sums\ of\ squares_{error}" = SS_E = \sum (error)^2 = \sum (y_i - \hat{y}_i)^2$$

Where $\hat{y}_i = \hat{b}_0 + \hat{b}_1 x_i$

We use the least-squares method for several reason:

- Squared errors (rather than any other exponent) have special properties, including the relationship between r^2 as a quantity that describes the amount of variance explained by the model, and additivity of variance (the sum of the variance of a set of numbers is equal to the variance of the sum of those numbers).
- When the error terms ∈ (i.e., residuals) are normally distributed, it is the best method to use.

2.2. Simple Linear Regression

2.2.1. Intuition and assumptions

Simple linear regression (SLR) studies the relationship between a response variable, *Y*, and a single explanatory variable, *x*. We expect that different values of *x* will produce different values of *Y*. This is similar to when comparing the means of two groups. The only difference is that in linear regression, the explanatory variable is continuous instead of categorical.

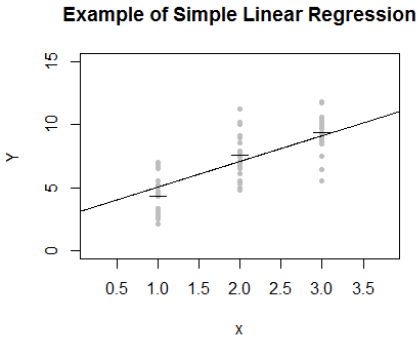
In SLR, we create a subpopulation for every value of *x*. Effectively, we are examine the distribution over *every* possible value of *x*. When doing this, we make several assumptions:

1) The relationship between *Y* and *x* is linear.

2) The error, $\in$, is a random variable that is normally distributed with a mean of 0 and a constant variance of σ^2.
3) It follows from the second assumption that:

$$Y \sim N(\beta_0 + \beta_1 x_i, \sigma^2)$$

That is to say that: *Y* is normally distributed over every value of *x*.



These assumptions may or may not be true, and so they should be tested wherever possible. This can be done by looking at observed residual plots (covered later).

The figure on the right is a simple illustration of the assumptions just mentioned. Given a value of *x*, *Y* follows a normal distribution with mean μ_y.

2.2.2. The linear regression model

If we assume that the true relationship between *Y* and *x* can be approximated by a linear function, then we can write this relationship as:

$$Y_i = \beta_0 + \beta_1 x_i + \in_i$$

with intercept β_0, slope β_1, and error $\in_i$

$\in_i$ is the random error term (i.e., residuals), which is the result of things like other variables that cause variaiton in *Y*, or mesasurement error. We typically assume that $\in_i$ is independent of *x*. β_0, β_1, and σ are the model parameters. However, in almost all cases, we don't actually know what they are. We estimate them from the data. The model that we choose will use $\hat{b}_0$ and $\hat{b}_1$ that minimize the residual sums of squares (SS_E). Using calculus to minimize SS_E, we can show how to calculate these statistics. Luckily, we will never have to do this by hand because statistical software will do it for us.

$$\hat{\boldsymbol{b}}_{\boldsymbol{1}} = \frac{\boldsymbol{Covariance}(\boldsymbol{x},\boldsymbol{y})}{\boldsymbol{variance}(\boldsymbol{x})} = \frac{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}}(\boldsymbol{x_i}-\bar{\boldsymbol{x}})(\boldsymbol{y_i}-\bar{\boldsymbol{y}})}{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}}(\boldsymbol{x_i}-\bar{\boldsymbol{x}})}$$

$$\hat{\boldsymbol{b}}_{\boldsymbol{0}} = \bar{\boldsymbol{y}} - \hat{\boldsymbol{b}}_{\boldsymbol{1}}\bar{\boldsymbol{x}}$$

Where $\bar{y}$ and $\bar{x}$ are the sample means for those variables.

2.2.3. Assessing accuracy of the co-efficients

$\hat{b}_o$ and $\hat{b}_1$ are unbiased estimators of β_0 and β_1, respectively. These variables are normally distributed with a mean $\hat{b}_o$ and $\hat{b}_1$. We can estimate the accuracy of these statistics from the data:

$$\sigma^2_{\beta_1} \cong Var(\hat{b}_1) = \frac{s^2}{\sum(x_i-\bar{x})^2} \qquad SE(\hat{b}_1) = \frac{s}{\sqrt{\sum(x_i-\bar{x})^2}}$$

$$\sigma^2_{\beta_0} \cong Var(\hat{b}_0) = s^2 * \left(\frac{1}{n} + \frac{\bar{x}^2}{\sum(x_i-\bar{x})^2}\right) \qquad SE(\hat{b}_0) = s * \sqrt{\frac{1}{n} + \frac{\bar{x}^2}{\sum(x_i-\bar{x})^2}}$$

Where: $Sample\ variance = s^2 = Var(\epsilon) = \frac{\sum(y_i-\bar{y})^2}{n-1}$

Multiple linear regression (MLR) follows the same principles as SLR, except multiple explanatory variables are used to construct a linear function.

Notation for multiple regression

Because there are multiple variables and multiple cases in each variable, we represent them with the notation x_{ij}, where *i* denotes the case and *j* denotes which explanatory variable. Ex.,

Case 1: $Y \approx \beta_0 + \beta_1 x_{1,2} + \beta_2 x_{1,2} + \beta_3 x_{1,3} + \beta_4 x_{1,4}$

2.3. Multiple Linear Regression

The multiple regression model is the same as SLR, except there is an additional term for each variable. Likewise, there is an additional 'slope' parameter (β_j) for each variable. These are estimated the same was as in SLR; see Section **2.2. Simple Linear** Regression. Because we are estimating additional parameters, we have fewer degrees of freedom. This is given by:

$$df = n - p - 1$$

Where *p* is the number of parameters in the model.

Note that we use *n - 2* in SLR simply because *p = 1*. The *n - p - 1* form is simply the more generalizeable form. Consequently, the variance, σ^2, is estimated using *df = n - p - 1*.

*Note: Minimizing the error using the least-squares method for multiple variables is computationally heavy (you're still just adding up all the terms to get $\hat{y}$), so we invariably allow software to perform these calculations for us.

2.4. Inference for Regression

This assumes you are familiar with Sections **2.2. Simple Linear Regression** and **2.3. Multiple Linear Regression**. Otherwise, refresh yourself on that material.

Analysis of variance summarizes the different sources of variation in the data using the sums of squares.

$$DATA = FIT + RESIDUAL$$

$$SS_T = SS_M + SS_E$$

$SS_M = \sum(\hat{y}_i - \bar{y})^2$ - Differences between the observed values of $\hat{y}$ and the mean $\bar{y}$ because *y* varies over *x*. This type of variance is explained by a linear equation.

$SS_E = \sum(y_i - \hat{y}_i)^2$ - Differences between the predicted value of *y* and the observed value of $\hat{y}$. This type of variance is not explained by a linear equation and is thus attributed to "error."

$SS_T = \sum(y_i - \hat{y})^2$ - The sum of the squared errors from both SS_M and SS_E terms.

If we are testing the null hypothesis that $\beta_1 = 0$, we are assuming that all values of *y* come from the same population with mean μ_y, and that there are no "subpopulations" of *y* that vary with *x*. In this case, any variation in *y* would be explained by the sample variance. Recall that this is:

$$Sample\ variance = s^2 = \frac{\sum(y_i - \bar{y})^2}{n-1}$$

Note that the numerator of this expression is equal to SS_T, and recall that the denominator is the total degrees of freedom in the sample, df_T. Just like the sums of squares, the degrees of freedom are also given by:

$$df_T = df_M + df_E$$

$df_T = n - 1$	- The total degrees of freedom is estimated by *n - 1*.
$df_M = 1$	- For a simple linear model, there is one explanatory variable, and so 1 *df*.
$df_E = n - df_M$	- The remaining degrees of freedom go do df_E. For SLR, this is *n - 2*.

The ratio between the sums of squares (SS) and the degrees of freedom *df* is called the **mean squared error**. The mean squared error and this provides us with our estimator for σ:

$$MS_E = s^2 = \frac{\sum(y_i - \bar{y})^2}{n-2} \qquad \sigma \cong s = \sqrt{s^2}$$

2.4.1. Assessing model accuracy: R^2 and *r*

2.4.1.1. Correlation co-efficient: *r*

Correlation is defined as:

$$\rho = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \approx r = \frac{cov(x,y)}{\sigma_x \sigma_y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \times \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

In simple linear regression, the slope $\hat{b}_1$ can be expressed as the ratio between the sd of *y* to the sd of *x*, multiplied by a co-efficient '*r*'. Thus:

$$\hat{b}_1 = r\frac{s_y}{s_x} = \frac{Covariance(x,y)}{variance(x)} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})}$$

The slope is an important part of interpreting the regression line because it provides some measure of *effect size*. <u>However, the slope is dependent on the units of measurement</u>. The slope can be interpreted as *"Each change of one standard deviation in x corresponds to a change of r standard deviaions in y."* Therefore, as *r* approaches -1 or 1 *more* of the variation in *y* can be explained by variation in *x* (and vice-versa). Note however that <u>*r* is very sensitive to outliers</u> with high values.

It follows that *r* is a measure of the strength of the relationship between two variables. In simple linear regression using the least-squares method, $R^2 = r^2$. However, for other methods, this is not true. Additionally, in multiple-linear regression, *r* does not apply, since it only quantifies the association between a single pair of variables.

2.4.1.2. Residual standard error and R^2

When performing both inference and prediction, we often want to quantify the extent to which the model fits the data.

The quality of a linear regression fit is typically assessed using two related quantities:

- *residual standard error (RSE)*
- R^2 statistic.

The RSE is a measure of the *absolute lack of fit*. It is simply the square-root of the average SS_E, and describes the average amount that the response will deviate from the true regression line:

$$RSE = \sqrt{\frac{1}{n-2} \times RSS}$$

The R^2 statistic is another measure of fit, but takes the form of a *proportion* – the proportion of variance in Y explained by X.

$$R^2 = \frac{variance\ of\ predicted\ values\ of\ \hat{y}}{variance\ of\ observed\ values\ of\ y} = \frac{SS_M}{SS_T}$$

- R^2 and its connection to the correlation co-efficient *r* are special properties of the least-squares method of regression and is not true for other methods of model-fitting.
- The variance in the observed data *will always be greater than or equal to* the variance of the model.
- R^2 is generally used more often than *r* because it can be applied to multiple regression.

2.4.2. ANOVA *F* test for slope

The null hypothesis: $\beta_1 = 0$ can be tested by comparing the amount of variability that can be explained by the model versus variability that cannot. The anova test statistic uses an *F* statistic (note that this is not the same *F* statistic that is used in *equality of spread*).

$$F = \frac{MS_M}{MS_E}$$

We calculate this test under the assumption that H_0 is true, so we use $df_M = 1$ and $df_E = n - 2$. When much of the variability can be explained by the MS_M, *F* will be large.

The *F* statistic is simply t^2. This means that the *F* distribution is right-skewed, and that only two-sided hypotheses can be tested. For linear regression with a single explanatory variable, *t* procedures are typically used because they more easily allow for one-sided alternatives and are closely related to confidence intervals. However, the *F* statistic is necessary for comparing multiple means.

In SLR, the anova *F* test is equivalent to a two-sided *t* test of the hypothesis that the slope = 0. For MLR, the anova *F* test will test the hypothesis that *all* of the regression co-efficients, except the intercept, are equal to 0. In this case, the alternate hypothesis is that *at least one* of the regression co-efficients is not 0. A large *F* will provide evidence against the null hypothesis that *together* the variables in the model are associated with the response variable; a post-hoc test is necessary to determine *which* variables are driving this relationship.

Anova tables:

The anova *F*-test can be summarized in an anova table. This method can be used to calculate anova *F*-tests by hand.

Source	Degrees of freedom	Sum of squares	Mean square	*F*
Model	p	$\sum(\hat{y}_i - \bar{y})^2$	SSM/DFM	MSM/MSE
Error	$n-p-1$	$\sum(y_i - \hat{y}_i)^2$	SSE/DFE	
Total	$n-1$	$\sum(y_i - \bar{y})^2$	SST/DFT	

2.4.3. t-test and confidence intervals for slope

In general, this is done by software automatically, so it is not critical to remember the math. * It should be noted that R calculates two-sided *p-values*.

When we perform inference on a slope, we are testing the hypothesis that *the slope differs from 0*, against the null hypothesis that *the slope is 0*. That is to say, we are testing the hypothesis that relationship between *x* and *y* is real.

To do this, we simply perform a *t*-test:

- We need to check that the SLR assumptions are met.
- If the assumptions for SLR are met, the only assumption that we need to check for a *t*-test is the independence of observations.

$$t_{df} = \frac{\hat{b}_1 - 0}{SE(\hat{b}_1)}$$

with $df = n - 2$

We can also compute confidence intervals just as with a regular *t*-procedure:

$$CI = \hat{b}_1 \pm t_{df} \times SE(\hat{b}_1)$$

* Note that the same procedures can be carried out on the intercept to test the hypothesis that $\hat{b}_0$ is not 0; however, this is generally not meaningful.

2.4.4. Prediction - confidence intervals and prediction intervals

We can use our regression model to predict $\hat{y}$ for any given value of *x*, represented by x^*. More importantly, we need to construct confidence or prediction intervals to convey how accurate we can expect our prediction to be. The two are calculated in a similar fashion:

$$Confidence\ interval = \hat{y} \pm\ t_{df} \times SE_{\mu}$$

$$Prediction\ interval = \hat{y} \pm\ t_{df} \times SE_{\hat{y}}$$

Where:

$$\hat{y} = \hat{b}_0 + \hat{b}_1 x$$

$$SE_{\mu} = s * \sqrt{\frac{1}{n} + \frac{(x^* - \bar{x})}{\sum(x_i - \bar{x})^2}} \qquad SE_{\hat{y}} = s * \sqrt{1 + \frac{1}{n} + \frac{(x^* - \bar{x})}{\sum(x_i - \bar{x})^2}}$$

The difference between confidence intervals and prediction intervals is that $SE_{\hat{y}}$, which is used in the prediction interval, includes both the variability from the error in the least-squares line *and* variability from (future) sampling error in $\hat{y}$. *This is given by the '+1' term inside the radical*. Thus, prediction intervals are *always* larger than confidence intervals. The 95% prediction interval will contain the predicted value of *y* for an additional observation 95% of the time.

2.4.5. Test for non-zero correlation

The population-level correlation between *x* and *Y* is given by the Greek letter rho: ρ. When *x* and *Y* are jointly Normally distributed (i.e. if they are plotted against each other they are both Normal) *and* $\rho = 0$, it is said that the two variables are independent.

We will often want to testthe null hypothesis that $\rho = 0$. To do this, we will compute the *t*-statistic:

$$t_{df=n-2} = \frac{r * \sqrt{n-2}}{\sqrt{1-r^2}}$$

Again, most software packages will automatically calculate this test for non-zero correlation. Most frequently Pearson's *r* is calculated, but other correlation co-efficients can also be tested this way.

2.4.6. Significance tests and confidence intervals for β_j

In multiple regression, we calculate CIs and significance tests the same way as SLR, but we do so for each regression co-efficient, β_j, in the model. The procedures are simple and are given in **2.4.** The only difference is that SE_{β_j} is calculated in a slightly different way. However, we may ignore this and allow the software to do the computations for us.

2.4.7. Practical approaches to multiple linear regression for inference

This is not an exhaustive guide. Update as needed.

1. Summarize the data for each variable
    - Note the class of each variable, and missing values.
    - Examine the distribution and note any strange observations.
2. Examine the relationship between every pair of variables
    - Plot the response variable against each explanatory variable *AND* all explanatory variables against each other to identify relationships and outliers. You will need to make *p(p+1)/2* plots.
    - Compute correlation co-efficients for each pair of continuous variables. This will provide some sense of which variables will be the strongest predictors.
        i. Note: Simple correlation does not predict usefulness in multiple regression. Just because a particular variable does not have a significant correlation with a response variable does not mean that it will not be a useful (and statistically significant) predictor in a multiple regression.
3. Perform an initial regression analysis
    - This is done in R using `lm()`.
    - This may be performed on all of the variables or on some reasonable subset of them. This is simply to provide a simple gauge of the predictive power / significance of the model. The output will automatically perform anova followed by a *t*-test.
        ii. Note: Check *df*. When performing anova on a multivariate linear model, it is a good idea to calculate the *df* by hand and check that this matches the software. This is to check that the software is using the correct number of cases and variables.
        iii. Note: Strong predictors may appear to not be significant. When two explanatory variables are highly correlated, they may have overlap in the predictive information they bring to the model. Significance tests for individual regression co-efficients are calculated with all the other predictors in the equation held constant; if two variables provides similar information (i.e., they are correlated), one may not appear to be significant.
        iv. Note: Individual regression coefficients, SE, and *p*-values can only be interpreted in context. This follows from the previous point. These statistics are calculated assuming the other explanatory variables are still in the model. Thus, if any part of the model changes, these statistics will change.
    - Examine the residuals. It is standard to plot the residuals versus the predicted values $\hat{y}$ and also the residuals versus each explanatory variable. Variables that do not meet the assumptions should be re-evaluated or excluded from the model.
4. Refine the regression model
    - We can examine out output from earlier and drop the variable with the lowest *p*-value. This can be assessed using `drop1()` in R.
    - After dropping, re-assess the model and see if the loss in R^2 is acceptable. This can be assessed objetively using AIC or cross-validation, etc.

Logistic regression is concerned with modeling the mean of the response variable *p* in terms of an explanatory variable *x*. The response variable *p* cannot be modeled using linar regression because *p* must fall between 0 and 1. Instead, we use the *logistic function*:

$$\boldsymbol{p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}}$$

I don't really understand this equation, but re-arrangement of the logistic function gives rise to the odds:

$$\boldsymbol{odds} = \frac{\boldsymbol{p(X)}}{\boldsymbol{1-p(X)}} = e\beta 0 + \beta_1 X$$

Where $\hat{p}$ is the observed proportion for one of two mutually exclusive outcomes.

The odds are simply the ratio of the proportions for two possible outcomes. Ex.,

- Odds of 1/4 mean that 1 in 5 people will have the outcome.
- Odds of 99/111 mean that 99 in 210 people will have the outcome.
- Odds are most often used in gambling, because they relate more naturally to the correct betting strategy.

By taking the natural log of both sides, we get the *logit*, which are the *log-odds* (on the left):

$$\boldsymbol{ln}\left(\frac{\boldsymbol{p}}{\boldsymbol{1-p}} = \boldsymbol{\beta_0} + \boldsymbol{\beta_1 x}\right)$$

where *p* is the proportion from a binomial distribution

x is the explanatory variable (which may be categorical or numerical)

The parameters for the logistic model are β_0 and β_1.

2.4.8. Estimating regression coefficients

β_0 and β_1 are unknown, and must be estimated based on available data. For linear regression, we generally use the least squares approach to estimate these coefficients. While we could use a non-linear least squares approach to fit the model, the more general *maximum likelihood* method is preferred because it has "better statistical properties." The basic intuition behind ML is that we are seeking estimates of β_0 and β_1 that result in predicted probabilities that are as close to the true observed outcome. I.e., we want estimates close to 0 for outcome 1 and estimates close to 1 for outcome 2.
ML can be formalized into an equation called a *likelihood function*:

$$\beta 0, \beta 1 = i{:}yi = 1 p(xi) i'{:}yi' = 0 (1 - pxi')$$

How this function 'works':

- We have a set of observed data.
- $\prod_{i:y_i=1} p(x_i)$ corresponds to the probability that *xi* = 1.
- Because the observations are independent, we can add together all of the $p(x_i)$ and $(1 - pxi')$, we *get the conditional probabilities of* P(*yi = 1|xi) and P*($y_i' = 2|x_i'$)*
    - (The prime ′ symbol just denotes a separate observation)
- We then choose $\hat{b}_0$ and $\hat{b}_1$ to maximize the likelihood function, *based on the observed data*.
- In reality, we always use software to calculate this.

Making predictions

Once the coefficients have been estimated, computing the probability of a given outcome becomes fairly easy. The only thing to be aware of is that linear coefficients are usually calculated, but these are inputs for a non-linear function:

2.4.9. Unbalanced groups in classification

Often in classification, one outcome is much less likely than another outcome.

Case-control sampling

- Often investigators will just take a bunch of cases and then match them with controls, so that the rate of case = 1 is much higher in the sample compared to the overall population.
- In case-control samples, we estimate the slope parameters, $\hat{b}_j$, accurately, but the constant term, $\hat{b}_0$, is incorrect.
    - We can correct this with a simple transformation:

      $\hat{b}_0^* = \hat{b}_0 + \log\frac{\pi}{1-\pi} - \log\ \pi⍰ ⍰1 - \tilde{\pi}⍰$

      Where π is the true population prevalence of case = 1, and

      $\tilde{\pi}$ is the prevalence of case = 1 in the sample

- Note that the $log\ \pi\ ⍰1-\pi$ terms are just the logit.

Issues with unbalanced data

- Sampling more controls than cases reduces the variance of the parameter estimates (in a good way)
    - However, there are diminishing returns here. Above a ratio of about 5 controls : 1 case, the variance refuction flattens out, so there is little more to gain by sampling more contorls.

2.4.10. Interpreting logistic regression with a simple example

We will investigate a simple example of logistic regression to demonstrate:

- How to interpret 'odds'
- How to estimate *b0* and *b1*
- Interpreting the slope, *b1*
- Interpreting 'odds ratios'

'We are comparing the proportion of frequent binge drinkers among men and women college students. The proportion of men who are binge drinkers is **0.260,** and the proportion for women is **0.206.** We'll assign *x* = 1 for male and *x* = 0 for female.'

The odds are given by:

$odds male = 0.2601 - 0.260 = 0.351$ $odds female = 0.2061 - 0.206 = 0.259$

How to interpret 'odds':

- *"The odds of a man being a binge drinker are 0.351, approximately 1/3 (to 1) ."*
- *"The odds of a man being a binge drinker are 0.351 to 1."*

- *"The odds of a man being a binge drinker are 1 to 2.85."*
- *"The odds of a man <u>not</u> being a binge drinker are 2.85 to 1."*

<u>Estimating *b0* and *b1*</u>:

In general, the calculations needed to find the estimates *b0* and *b1* are complex and require software. However, we can do it in this simple example. First, we take the log odds (logisticmale/female). $x = 0$ for females, so logitfemale = *b0*:

$logitmale = lnpmale1-pmale = b0+b1x$

$logit_{male} = -1.05$

$logitfemale = lnpfemale1-pfemale = b0+b1x$

$logit_{female} = -1.35 = b_0 + b_1 * (0)$

$logit_{female} = \boldsymbol{b_0} = \boldsymbol{-1.35}$

Estimate *b1*

$$b_1 = logit_{male} - logit_{female}$$

$$b_1 = -1.05-(0.135)$$

$$\boldsymbol{b_1 = 0.30}$$

We now have our estimates for *b1* and *b0*:

$p1-p = -1.35+0.30x$

<u>Interpreting the slope, *b1*</u>

The slope for the logistic regression model is 0.30, but thinking about log(odds) is awkward. We typically apply a transformation. With some algebra, we can demonstrate that:

$$\frac{odds_{men}}{odds_{women}} = e^{0.30} = 1.34$$

This transformation, e^{b_1}, gives us the odds ratio. (This is also easily done by dividing one of the odds by the other).

<u>Interpreting odds ratios</u>:

- *"The odds that a man is a binge drinker to the odds that a woman is a binge drinker is 1.34 (or 4/3)."*
- *"The odds for men are 1.34 the odds for women" OR "The odds for women are 0.74 the odds for men"*
- The odds ratio × the odds for men = the odds for women
- *"Over a large number of trials, there will be 1.34 male binge drinkers for every 1 female binge drinker."* - (is this correct?)

2.4.11. Inference for logistic regression

Statistical inference for logistic regression is very similar to inference for linear regression. For reasons unknown to me, we use the Normal *z* statistic instead of the *t* statistic when we calculate our estimates for the model parameters.

* Note: Some software will output the significance test results a the "Wald statistic." This is the same as *z*. Other software will report the results as *z2*, which is *chi-squared*. Because the square of a standard Normal random variable has the $\chi 2$ with 1 degree of freedom, these two distributions give the exact same test results.

2.4.11.1. Inference and confidence intervals for slope

All the inference is the same for logistic regression as linear regression, except we use the *z* statistic.

To test the null hypothesis that slope is 0, we compure the *z* statistic:

$$z = \frac{b_1}{SE_{b_1}}$$

The confidence interval for the slope $\beta 1$ is calculated the same as usual:

$$CI_{\beta_1} = b_1 \pm z \times SE_{b_1}$$

2.4.11.2. Inference for odds ratios

The confidence interval for the odds ratio, e^{β_1} is obtained by transforming the confidence interval for the slope by *e*.

$$CI_{OR} = e^{b_1 \pm z * SE_{b_1}}$$

The odds ratio is sometimes used in inference, since it provides a measure of effect size. When the 95% CI fails to include 1, we reject *H0* that the odds for the two groups are the same.

2.4.11.3. Inference for Multiple Logistic Regression

Making inference for both the slope and odds ratios for multiple logistic regression is the same as simple logistic regression, except that we use the χ^2 statistic instead of the *z* statistic.

$$\chi^2 = z^2$$

with 1 degree of freedom.

The χ^2 test assesses if the combination of all of the variables can be used to predict the response variable. As with anova for multiple linear regression, this test doesn't tell you which of the explanatory variables is a suitable (i.e., significant) predictor. In the case where the explanatory variables are correlated, it is possible that the χ^2 returns a significant result but no individual variable does.

2.5. Assessing model assumptions

2.5.1. Limitations of the linear model

Linear regression models suffer from two main limitations:

- Linearity – The model approximates the true relationship between variables as linear, which introduces bias. There are methods to allow for more flexibility in a linear model to reduce this bias, such as *polynomial regression*.
- Additivity – The model treats the addition of new terms additively. Adding an *interaction term* into the model between two variables may help account for the effect of two variables, when combined, on the response.

2.5.2. Model assumptions to check

Potential problems with linear regression include:

1. Non-linearity of the data
    - Linear regression assumes there is a linear relationships between the reponse and predictor variables. If the true relationship is non-linear, thenany inference we make is suspect, and predictive accuracy is limited.
    - Residual plots are a useful tool to assess if the linearity assumption is appropriate.
    - Simple non-linear transformations may remedy non-linearity problems to a certain extent.

2. Correlation of error terms
    - Another assumption is that the observations are independent; i.e., the error terms are uncorrelated – this means that knowing one error term, ϵ_1, provides no information about the value of ϵ_2. Standard errors from the estimate co-efficients and fitted values are calculated based on the assumption of uncorrelated error terms. If this is not true, the standard errors will be underestimated.
    - If error terms are correlated, we will have narrows CIs, PIs, and artificially small *p-values*.
        - For example, if you accidently duplicate your data, calculations would be performed with a sample size of *2n*, but with the same value for all of the paramters; our CIs would be narrower by a factor of $\sqrt{2}$.
    - Correlation of error terms frequently appears in time-series data, but also arises in family studies, or if people have similar diets or have been exposed to the same environmental factor.
    - We can identify correlation of error-terms by looking at residual plots for *tracking* – that is, adjacent residuals tend to take on similar values. If errors are uncorrelated, we expect to see erratic movement of the residuals with no discernible pattern.

3. Non-constant variance
    - Linear regression also assumes constant variance.
    - This can be identified in a residual plot because the residuals have a funnel shape (i.e., wider on one end than the other). Most often, variance increases as the response variable increases.

- Heteroscedasticity can be corrected in part by applying a concave non-linear transformation such as a *log* or $\sqrt{x}$, which shrink larger values in comparison to smaller ones (or a convex function if the variance is larger at the bottom).
- If we have multiple raw observations per response, we may already have an idea of variance at each response. In this case, we can fit our model by *weighted least squares regression*, which weighs each response proportional to the inverse variances – this assigns a greater weight to observations with greater precision.

4. Outliers
   - Outliers typically have little effect on the least squares line (i.e., model), but it does reduce the confidence in the model – confidence intervals and *p-values* will be larger than otherwise, and R^2 values will be lower.
   - Outliers can be identified by residual plots. R will generally highlight outliers in these plots.
   - In practice, it can be difficult to decide what constitutes an outlier. One way to identify *possible* outliers are the *studentized residuals*, which are computed by dividing each residual by its estimated standard error: e_i/SE (I don't understand this). Studentized residuals greater than 3 possible outliers.

5. High leverage points
   - While outliers have unusual values of y_i, high leverage points are those which have unusual values of x_i.
     - Extreme values of x_i have a larger influence on the model than other points – this can be a problem because the model may be affected heavily by only a few points; any problems in these points may invalidat the entire fit.
   - In SLR, it is easy to identify high leverage points by looking at a scatterplot. In multiple linear regression, it is possible to have points that have normal values of x_1 and x_2, but abnormal in terms of the combination of x_1 and x_2.
   - In order to quantify an observation's leverage, we compute the *leverage statistic*. For simple linear regression, this is given as:

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{1'=1}^{n}(x_i - \bar{x})^2}$$

     - We can se here that h_i increases as the distance between x_i and $\bar{x}$ increases.
     - h_i always falls between *1/n* and *1*; with a mean of *(p + 1)/n*. A point with a leverage that greatly exceeds this may be a high leverage point.
     - There is a simple calculation for leverage that extends to MLR, but it is not covered in this book.

6. Collinearity
   - Collinearity is when two or more predictor variables are closely related. Because they are closely related, it is difficult to estimate the individual contribution (i.e., coefficient) of each predictor with all other variables held constant (because they tend to move together). This makes the standard error estimates for the coefficients larger, resulting in a higher probability of a false negative *F* test.
   - Pairwise collinearity can be detected by through scatterplots, but collinearity occuring between 3 or more variables is harder to detect.
     - Large changes in the co-efficients of certain variables when one variable is added are removed are indicative of collinearity.

- If the *F* test for non-zero slope is positive, but the individual model *p-values* are large, there may be collinearity.
- The *variance inflation factor* (VIF):

$$VIF(\hat{\beta}_j) = \frac{1}{1 - R^2_{X_J|X_{-j}}}$$

Where: $R^2_{X_J|X_{-j}}$ is the R^2 from a regression of X_j onto all other predictors (X_{-j})

- If $R^2_{X_J|X_{-j}}$ is close to 1, collinearity is present, and VIF will be large.
- A VIF > 5 or 10 (indicating $R^2_{X_J|X_{-j}} > 0.8$ indicates problematic collinearity.

- There are two ways of dealing with collinearity:
  - Drop one variable; since they are highly correlated, you don't expect to lose that much information.
  - Combine the two colinear variables into a single variable.

2.5.3. Checking assumptions with residual plots

This section briefly describes how different violations of our assumptions of SLR may manifest. See Section 2.6.2**.** Graphical diagnostics for residuals to carry out these diagnostics in R.

1) The relationship between *Y* and *x* is linear:

   The residual plot displays curvature

2) The error term, ϵ, has a constant variance (i.e., homoscedastic)

   The residuals become wider apart as *x* increases (or decreases)

3) The error term, ϵ, is normally distributed

   The QQ plot and/or histogram for the residuals do not look normal

4) The error terms, $\epsilon_1, \epsilon_2, \ldots \epsilon_n$, are correlated

   The residual plots exhibit *tracking* in which adjacent error terms have similar values

5) Outliers / high leverage points

   There are extreme observations in the residual plot.

   There are extreme observations in the studentized residual vs. leverage plot.

2.6. `Linear regression in R`

2.6.1. `lm()`

`lm()` is a powerful function that fits a general linear model to the data.

- <u>The model does not necessarily need to be linear - only the co-efficients</u>. Many non-linear functions (such as polynomials or exponentials) can easily be modeled using `lm()` as well.
- Note: Using `lm()` appropriately requires that the assumptions for a general linear model are met. Review the Linear Regression section in the Statistics Manual to see if a general linear model can be applied.

`lm()`

`formula` - R object that specifies the linear model.

- Note that the formula can be created and saved as an object using the `formula()` function. This allows you to manipulate the model through the formula object without having to invoke the `lm()` function. This apparently saves processing time.
- Review formulas if you are having trouble expressing the desired formula.

`data` - Data frame or list containing the variables found in `formula`

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, they are ignored.

The object is a list of length 12 containing information about the model. Ex.,

```
names(m1)
[1] "coefficients"  "residuals"     "effects"
[4] "rank"          "fitted.values" "assign"
[7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

2.6.2. Graphical diagnostics for residuals

`plot(m1)` can be used to display 4 diagnostic graphs for the model. Additional diagnostic plots can also be called. See `?plot.lm`.

It is recommended to first apply `par(mfrow = c(2,2))`.

- Residuals vs. Fitted
- Normal Q-Q for the Residuals
- $\sqrt{\text{Standardized residuals}}$ vs. Fitted
- Standardized residuals vs. Leverage

For all practical purposes, the `plot()` function discussed above will work. However, the graphs can also be constructed interactively for better overview. This constructs:

- A scatter plot of Y ~ X, along with the best-fit line (i.e, the model)
- Scatterplot of Residuals vs. Index
- Histogram of Residuals
- Q-Q plot for the Residuals

```
m1 <- lm(y ~ x)
sim.residuals <- function(x) dnorm(x, mean = 0, sd = sd(residuals(m1)))
    ## Note the estimation here that the variance of the residuals is equal to the
variance in the error terms.

par(mfrow = c(2,2))
plot(y~x); abline(m1)  # Scatterplot
```

```
plot(residuals(m1)); abline( h = 0, lty = 3)  # Residual plot
hist(residuals(m1), prob = T); curve(sim.residuals, add = T, lwd = 3)  # Histogram
qqnorm(residuals(m1)); qqline(residuals(m1))  # QQ plot
```

2.6.3. Extraction functions

A number of extraction functions can be applied to `lm()` and `glm()` to extract additional information from the model. This is list of some of the most useful, but there are many more. Many of these functions appear redundant because they are built-into the `lm()` object, but they can also be applied to other fitted model objects.

`anova()` - ANOVA test table for model reduction. Tests which variables account for the most variance.

`object` - fitted model object

`test` - Character string. Which test statistic should be used to test the model?

- Possible inputs: `"F", "Chisq", "LRT", "Rao"`

`drop1()` - Test the effect of dropping each individual term from the model. You may remove the 'least' significant' term and repeat the function step-wise to improve the model. `drop1()` will only test to drop terms that are not part of an interaction term. Note that this differs from anova in that

`object` - A fitted model object

`test` - Character string. Which test statistic should be used to test the model?

- Possible inputs: `"F", "Chisq", "LRT", "Rao"`
- Note: `"F"` should be used for `lm()`

- For usage, see Error! Reference source not found.. Error! Reference source not found..

`coef()` - Returns the estimated model of parameters.

`object` - `lm` or other model object. The model you just made.

`cofint()` - Confidence intervals for estimated model parameters.

`object` - `lm` or other model object. The model you just made.

`contrasts()` - Set or view the contrasts (i.e., numerical labels) associated with a factor (needed to interpret co-efficients for categorical variables).

`x` - A factor or logical variable

`hatvalues()` - Calculates leverage statistics for the variables in a model.

`model.matrix()` - The design matrix (*I'm not sure what this does*).

`object` - `lm` or other model object. The model you just made.

`predict()` - Returns predicted values from the model. Can also be applied to a new data set to make predictions using the same model.

- `object` - `lm` or other model object. The model you just made.
- `newdata` - Optional data frame. Data set used to make new predictions based on the model in `object`. If omitted, fitted values are used.
- `se.fit` - logical. Default = FALSE. If TRUE, the output of `predict()` becomes a list containing the predicted values, '`fit`,' and their standard errors, '`se.fit`.' Thus, `se.fit` can be used to construct confidence intervals.
- `interval` - string. Type of interval to calculate: "`none`", "`confidence`", "`prediction`".
- `level` - numeric. Confidence level (ex., 0.95).

`residuals()` - Returns the residuals

- `object` - `lm` or other model object. The model you just made.

`rstandard()` - Returns the studentized (standardized) residuls

- `object` - `lm` or other model object. The model you just made.

`summary()` - A summary printout, and access to summary statistics. This summary is a list object itself, with sub-elements that can be accessed. Ex.,

```
names(summary(m1))
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled"
```

`update()` - Allows you to refit a model.

- `object` - The model object to be updated. This function can be used with only this argument (Ex., if the formula is to remain the same, but the data frame used to construct the model has changed)
- `formula` - The new formula to be used to construct the model.
  - ' `~.` ' indicates 'the same formula as before'
  - Terms may be added or removed terms using the '+' and '−' operators
- Note: in many cases it is better practice to simply create a new object rather than replace the existing one.

`vcov()` - Covariance matrix for estimated model parameters

`vif()` - Calculates the *variance inflation factor* for each variable in the model. Not included in base R, but is part of the `cars` library.

- `mod` - `lm` or other model object. The model you just made.

`which.max()` - related to `hatvalues()`; identifies the index of the observation with the largest leverage.

3. *K*-Nearest Neighbours

3.1. The KNN regression model

K-nearest neighbours (KNN) *regression* is one of the simplest and best known non-parametric methods. KNN regression is closely related to *KNN classification*.

Given a value *K* and a prediction point, x_0, KNN regression identifies the closest *K* training observations to x_0, represented by N_0. It then estimates $f(x_0)$ using the average of all of the training responses, N_0. In other words:

$$\hat{f}(x_0) = \frac{1}{K} \sum_{x_i \epsilon N_0} y_i$$

Where x_0 is a chosen prediction point, and

N_0 are the K-nearest neighbours

"the estimated $f(x_o)$ is equal to the mean of y_i for all of x_i belonging to N_0"

Selecting *K*

- The optimal value for *K* will depend on the bias-variance trade-off:
  - Small values of *K* provide the most flexible fit, but have high variance. For example, when *K = 1*, the prediction for a given region is based entirely off of a single observation.
  - Larger values of *K* provide smoother, less flexible fits. Smoothing reduces variance, but may increase bias by masking some of the structure in *f(X)*.

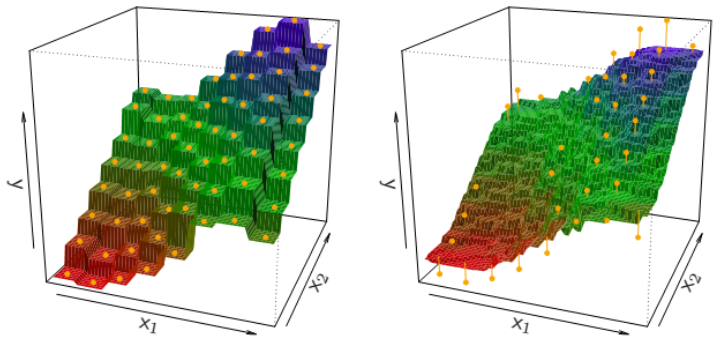


FIGURE 3.16. *Plots of $\hat{f}(X)$ using KNN regression on a two-dimensional data set with 64 observations (orange dots).* Left: $K = 1$ *results in a rough step function fit.* Right: $K = 9$ *produces a much smoother fit.*

3.1.1. KNN regression versus linear regression

When do parametric approaches out-perform non-parametric approaches?

- **When the parametric form that has been selected is close to the true form of *f*.**
  - If the parametric function chosen *perfectly* matches the true form of *f* then there is *no bias*. Therefore, any non-parametric function will incur a variance cost with zero improvement in bias.
  - As long as the parametric form is close to the true form of *f*, the variance cost will outweigh the bias gain.
- KNN may be expected to outperform linear regression (for example) if:
  - *f* is substantially non-linear

    - When there is a large *n* and small *p*.
- KNN often performs worse than linear regression if:
    - When there is large *p* and small *n*.
        - With a small *n* and large *p*, it is more likely that a given observation will have no nearby neighbours, leading to poor predications.
    - There is a lot of noise
    - Interpretability is an important factor

3.2. The KNN classifier

The KNN classidier is a simple non-parametric classification method. In the case of two classes, the KNN classifier and KNN regression appear to be mathematically equivalent, only the interpretation changes.

As with regression, given a value *K* and a prediction point, x_0, KNN regression identifies the closest *K* training observations to x_0, represented by N_0. It then estimates the *conditional probability* for class *j* as the fraction of points in N_0 where $Y = j$. In other words, it simply takes the proportion of points in the neighbourhood that fall into each class:

$$P(Y = j|X = x_0) = \frac{1}{K} \sum_{x_i \epsilon N_0} I(y_i = j)$$

Next, KNN applies Bayes rule and classifies the observation, x_o, to the class with the largest probability (i.e., whichever point appears most frequently in the neighbourhood). There are no parameters in the model, but the user must decide on a value of *K*.

As with KNN regression, small values of *K* provide more flexible (less biased) fits, while larger values of *K* provide less flexible (lower variance) fits.

3.3. Limitations in KNN

- KNN can only be applied to numeric data
    - Categorical variables can be binary coded, but this is also subject to another limitation:
- KNN weighs the importance of values based on Euclidean distance.
    - This means that dummy coding a variable as 100 and 0 vs. 1 and 0 will make a difference.
    - This also means that the scale of the units is critical. Ex., A difference of 10,000 cents will be weighted more than a difference of $100, or that a difference in income of $10 is weighted equally as a difference in age of 10 years.
        - In general, you will need to find a distance function that allows you to weight the importance of your variables properly. For example, you can standardize the values using z-scores, but this also has severe limitations (ex., if the data are on a non-linear scale, or you have dummy variables)

The process of coming up with a distance function that allows you to accurately weigh every variable can be *extremely* complex and makes up the majority of the work when making a KNN model. The only 'easy' way to apply a KNN model is if all of the variables are in the same units and are considered to be of equal importance.

4. Classification

4.1. Logistic Regression

While linear regression models the response of *Y* on *X* directly, logistic regression models the *proability* that *Y* belongs to a particular category based on *X*. In a simple scenario with binary outcomes and one predictor, we can express this as a conditional probability:

$$\boldsymbol{P(Y = 1|X)}$$

We shorten this to:

$$\boldsymbol{p(X)}$$

In general, we assign *Y* to whichever outcome has the highest probability; with binary outcomes, we assign outcome has the highest probability; with binary outcomes, we assign $Y = 1$ if $P(Y|X) > 0.5$. However, you can change assignment rules; if you want to be more (or less) conservative, for example, you may only make a decision if $P(Y|X) > 0.9$.

4.1.1. The logistic model

Logistic regression is concerned with modeling the mean of the response variable *p* in terms of an explanatory variable *x*. The response variable *p* cannot be modeled using linar regression because *p* must fall between 0 and 1. Instead, we use the *logistic function*:

$$\boldsymbol{p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}}$$

I don't really understand this equation, but re-arrangement of the logistic function gives rise to the odds:

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

Where $\hat{p}$ is the observed proportion for one of two mutually exclusive outcomes.

The odds are simply the ratio of the proportions for two possible outcomes. Ex.,

- Odds of 1/4 mean that 1 in 5 people will have the outcome.
- Odds of 99/111 mean that 99 in 210 people will have the outcome.
- Odds are most often used in gambling, because they relate more naturally to the correct betting strategy.

By taking the natural log of both sides, we get the *logit*, which are the *log-odds* (on the left):

$$\boldsymbol{ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x}$$

where *p* is the proportion from a binomial distribution

x is the explanatory variable (which may be categorical or numerical)

The parameters for the logistic model are β_0 and β_1.

4.1.2. Estimating regression coefficients

β_0 and β_1 are unknown, and must be estimated based on available data. For linear regression, we generally use the least squares approach to estimate these coefficients. While we could use a non-linear least squares approach to fit the model, the more general *maximum likelihood* method is preferred because it has "better statistical properties." The basic intuition behind ML is that we are seeking estimates of β_0 and β_1 that result in predicted probabilities that are as close to the true observed outcome. I.e., we want estimates close to 0 for outcome 1 and estimates close to 1 for outcome 2.
ML can be formalized into an equation called a *likelihood function*:

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

How this function 'works':

- We have a set of observed data.
- $\prod_{i:y_i=1} p(x_i)$ corresponds to the probability that $x_i = 1$.
- Because the observations are independent, we can add together all of the $p(x_i)$ and $(1 - p(x_{i'}))$, we get the conditional probabilities of $P(y_i = 1|x_i)$ and $P(y_i' = 2|x_i')$
    - (The prime $'$ symbol just denotes a separate observation)
- We then choose $\hat{b}_0$ and $\hat{b}_1$ to maximize the likelihood function, *based on the observed data*.
- In reality, we always use software to calculate this.

Making predictions

Once the coefficients have been estimated, computing the probability of a given outcome becomes fairly easy. The only thing to be aware of is that linear coefficients are usually calculated, but these are inputs for a non-linear function:

4.1.3. Unbalanced groups in classification

Often in classification, one outcome is much less likely than another outcome.

Case-control sampling

- Often investigators will just take a bunch of cases and then match them with controls, so that the rate of case = 1 is much higher in the sample compared to the overall population.

- In case-control samples, we estimate the slope parameters, $\hat{b}_j$, accurately, but the constant term, $\hat{b}_0$, is incorrect.
    - We can correct this with a simple transformation:

$$\hat{b}_0^* = \hat{b}_0 + \log\frac{\pi}{1-\pi} - \log\frac{\tilde{\pi}}{1-\tilde{\pi}}$$

Where π is the true population prevalence of case = 1, and

$\tilde{\pi}$ is the prevalence of case = 1 in the sample

- Note that the $log\frac{\pi}{1-\pi}$ terms are just the logit.

Issues with unbalanced data

- Sampling more controls than cases reduces the variance of the parameter estimates (in a good way)
    - However, there are diminishing returns here. Above a ratio of about 5 controls : 1 case, the variance refuction flattens out, so there is little more to gain by sampling more contorls.

4.1.4. Interpreting logistic regression with a simple example

We will investigate a simple example of logistic regression to demonstrate:

- How to interpret 'odds'
- How to estimate b_0 and b_1
- Interpreting the slope, b_1
- Interpreting 'odds ratios'

'We are comparing the proportion of frequent binge drinkers among men and women college students. The proportion of men who are binge drinkers is **0.260**, and the proportion for women is **0.206**. We'll assign x = 1 for male and x = 0 for female.'

The odds are given by:

$odds_{male} = \frac{0.260}{1-0.260} = 0.351$ $\qquad$ $odds_{female} = \frac{0.206}{1-0.206} = 0.259$

How to interpret 'odds':

- *"The odds of a man being a binge drinker are 0.351, approximately 1/3 (to 1) ."*
- *"The odds of a man being a binge drinker are 0.351 to 1."*
- *"The odds of a man being a binge drinker are 1 to 2.85."*
- *"The odds of a man not being a binge drinker are 2.85 to 1."*

Estimating b_0 and b_1:

In general, the calculations needed to find the estimates b_0 and b_1 are complex and require software. However, we can do it in this simple example. First, we take the log odds ($\text{logistic}_{\text{male/female}}$). x = 0 for females, so $\text{logit}_{\text{female}} = b_0$:

$$logit_{male} = ln\left(\frac{\hat{p}_{male}}{1-\hat{p}_{male}}\right) = b_0 + b_1 x$$

$$logit_{male} = -1.05$$

$$logit_{female} = ln\left(\frac{\hat{p}_{female}}{1-\hat{p}_{female}}\right) = b_0 + b_1 x$$

$$logit_{female} = -1.35 = b_0 + b_1 * (0)$$

$$logit_{female} = \boldsymbol{b_0 = -1.35}$$

Estimate b_1

$$b_1 = logit_{male} - logit_{female}$$

$$b_1 = -1.05 - (0.135)$$

$$\boldsymbol{b_1 = 0.30}$$

We now have our estimates for b_1 and b_0:

$$ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = -1.35 + 0.30x$$

Interpreting the slope, b_1

The slope for the logistic regression model is 0.30, but thinking about log(odds) is awkward. We typically apply a transformation. With some algebra, we can demonstrate that:

$$\frac{odds_{men}}{odds_{women}} = e^{0.30} = 1.34$$

This transformation, e^{b_1}, gives us the odds ratio. (This is also easily done by dividing one of the odds by the other).

Interpreting odds ratios:

- *"The odds that a man is a binge drinker to the odds that a woman is a binge drinker is 1.34 (or 4/3)."*
- *"The odds for men are 1.34 the odds for women" OR "The odds for women are 0.74 the odds for men"*
- The odds ratio × the odds for men = the odds for women
- *"Over a large number of trials, there will be 1.34 male binge drinkers for every 1 female binge drinker."* - (is this correct?)

4.1.5. Inference for logistic regression

Statistical inference for logistic regression is very similar to inference for linear regression. For reasons unknown to me, we use the Normal *z* statistic instead of the *t* statistic when we calculate our estimates for the model parameters.

* Note: Some software will output the significance test results a the "Wald statistic." This is the same as *z*. Other software will report the results as z^2, which is *chi-squared*. Because the square of a standard Normal random variable has the χ^2 with 1 degree of freedom, these two distributions give the exact same test results.

4.1.5.1. Inference and confidence intervals for slope

All the inference is the same for logistic regression as linear regression, except we use the *z* statistic.

To test the null hypothesis that slope is 0, we compure the *z* statistic:

$$z = \frac{b_1}{SE_{b_1}}$$

The confidence interval for the slope β_1 is calculated the same as usual:

$$CI_{\beta_1} = b_1 \pm z \times SE_{b_1}$$

4.1.5.2. Inference for odds ratios

The confidence interval for the odds ratio, e^{β_1} is obtained by transforming the confidence interval for the slope by *e*.

$$CI_{OR} = e^{b_1 \pm z * SE_{b_1}}$$

The odds ratio is sometimes used in inference, since it provides a measure of effect size. When the 95% CI fails to include 1, we reject H_0 that the odds for the two groups are the same.

4.1.5.3. Inference for Multiple Logistic Regression

Making inference for both the slope and odds ratios for multiple logistic regression is the same as simple logistic regression, except that we use the χ^2 statistic instead of the *z* statistic.

$$\chi^2 = z^2$$

with 1 degree of freedom.

The χ^2 test assesses if the combination of all of the variables can be used to predict the response variable. As with anova for multiple linear regression, this test doesn't tell you which of the explanatory variables is a suitable (i.e., significant) predictor. In the case where the explanatory variables are correlated, it is possible that the χ^2 returns a significant result but no individual variable does.

4.2. Linear Discriminant Analysis

Linear discriminant analysis is based on Bayes' Theorem. While logistic regression attempts to estimate $P(Y = k|X = x)$, LDA aims to model the distribution of the predictors *X* separately in each of the response classes (i.e., *X* given *Y*) and then use Bayes' Therorem to these around into estimates for $P(Y = k|X = x)$.

When the distributions are assumed to be normal, LDA turns out to be very similar to logistic regression. LDA can also be generalized to other distributions, but most commonly normal or quadratic distributions are used.

Why use LDA over logistic regression?

- When the classes are well-separated, parameter estimates for the logistic regression model are surprisingly unstable; LDA does not suffer from this problem.
- If *n* is small and the distribution of the predictors, *X*, is approximately normal in each of the classes, the LD model is again more stable than the logistic model.
- LDA is popular when we have more than two response classes.

4.2.1. Bayes' Theorem for classification

We use Bayes' Theorem when we wish to classify an observation into one of *K* classes, where $K \geq 2$.

$$P(Y = k|X = x) = \frac{P(X = x|Y = k) \times P(Y = k)}{P(X = x)}$$

- Where $P(Y = k|X = x)$ is the posterior probability that $Y = k$, given that $X = x$,
- $P(Y = k)$ and $P(X = x)$ are the marginal or *prior* probabilities that $Y = k$ and $X = x$, respectively.

When we perform LDA, we typically rearrange Bayes' Theorem into the following form:

$$\boldsymbol{p_k(X) = P(Y = k|X = x) = \frac{\pi_k \times f_k(x)}{\sum_{l=i}^{K} \pi_l \times f_l(x)}}$$

- Where π_k is the marginal or *prior* probability that $Y = k$ for a randomly selected observation (it is not related to the mathematical constant in any way).
- π_l is the marginal/prior probability that $Y = l$ for a randomly selected observation (the bottom term sums all of the marginal probabilities for all possible classes 1 through *K*).
- Let $f_k(x) \equiv P(X = x|Y = k)$ denote the *density function* of X for an observation that comes from the k^{th} class.
- $p_k(X)$ is simply shorthand for $P(Y = k|X = x)$.

- $f_k(x)$ is the conditional probability that $X = x$, given that $Y = k$. In other words, $f_k(x)$ is large if there is a high probability that $X = x$ if we know that $Y = k$.
    - It is often difficult to estimate this without assuming a simpler form for the density function.
- We can easily estimate π_k from the training data; we compute the fraction of training observations that belong to the k^{th} class.

Recall that the Bayes classifier, which classifies an observation to the class for which $p_k(X)$ is largerst, has the lowest possible error rate out of all classifiers (of course, this is only true when π_k and $f_k(x)$ are correctly specified. However, if we can accurately estimate $f_k(x)$, then the LDA classifier approximates the Bayes classifier.

4.2.2. Simple LDA for $p = 1$

For simplicity, we will assume that $p = 1$. We would like to estimate $f_k(x)$ in order to estimate $p_k(x)$. In the simplest scenario, we assume that $f_k(x)$ is *Normal*. When $p = 1$, the normal density function takes the forn:

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k}\exp\left(-\frac{1}{2\sigma_k^2}(x-\mu_k)^2\right)$$

Where μ_k and σ_k^2 are the mean and variance parameters for the k^{th} class.

We also assume that $\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2$; that is, the variance terms are equal across all *K* classes, so we can simply denote the variance term by σ^2. If we plug the normal approximation of $f_k(x)$ into the equation above, and perform some black magic (take logs and discard terms that do not vary with *k*), we arrive at:

$$\boldsymbol{\delta_k(x) = x \times \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{\sigma^2} + \log(\pi_k)}$$

Where $\delta_k(x)$ is the linear discriminant function.

- Assigning observations for the class for which $\delta_k(x)$ is maximized is equivalent to the Bayes classifer.
- To identify the decision boundary, we simply set $\delta_1(x) = \delta_2(x)$ and evaluate the equation. To classify an observation, we calculate each of $\delta_1(x), \delta_2(x), \ldots \delta_K(x)$ and classify *x* to the class for which $\delta_k(x)$ is largest.
    - For a case where $K = 2$ and $\pi_1 = \pi_2 = 0.5$, we simplify this and find *decision boundary* when:

$$x = \frac{\mu_1^2 - \mu_2^2}{2(\mu_1 - \mu_2)} = \frac{\mu_1 + \mu_2}{2}$$

Estimating the parameters

- In the case where we don't know the population parameters μ_k, σ^2, and π_k, we estimate them from our data:

$$\boldsymbol{\bar{x}_k = \frac{1}{n_k}\sum_{i:y_i=k} x_i}$$

Where $n_k$is the number of observations where $y = k$

- "The mean of *x* for all observations where $y = k$."

$$s^2 = \frac{1}{n-K}\sum_{k=1}^{K}\sum_{i:y_i=k}(x_i - \bar{x}_k)^2$$

$$\boldsymbol{s^2 = \sum_{k=1}^{K}\frac{n_k - 1}{n - K} \times s_k^2}$$

- "The pooled variance estimate: sum the squared differences in *x* for all observations where for all observations where $y = k$, then sum all of these variances across all of the *K* classes."

- OR "The weighted sample variance for each class summed across all of the *K* classes."

$$\hat{\pi}_k = \frac{n_k}{n}$$

- "The proportion of observatoins where $y = k$ (i.e., the marginal probability of $y = k$)."

4.2.3. LDA for $p > 1$

LDA can be extended to the case of multiple predictors. To do this, we assume that al l of the variables, X_1, X_2,.. X_p, are drawn from a *multivariate Normal* distribution, with a *class-specific mean* and a common *covariance matrix* (I don't understand those words).

The multivariate Normal distribution assumes that each individual predictor follows a one-dimensional normal distribution with some correlation between each pair of predictors.

In the adjacent plot, we have two examples of multivariate normal distributions where $p = 2$ where the predictors are either uncorrelated or correlated. In either panel, the surface can be cut anywhere along X_1 or X_2 and the resulting cross-section will be a 1-dimensional Normal distribution.

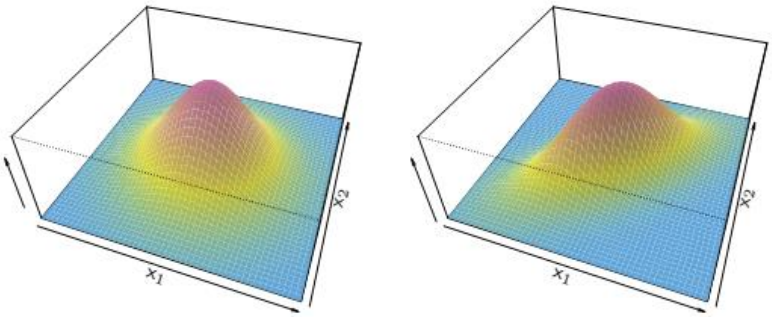


FIGURE 4.5. *Two multivariate Gaussian density functions are shown, with* $p = 2$. Left: *The two predictors are uncorrelated.* Right: *The two variables have a correlation of* 0.7.

We describe a *p*-dimensional random variable, *X*, as: $X \sim N(\mu, \Sigma)$. Here $E(X) = \mu$ is the mean of *X* (a vector with *p* components), and $Cov(X) = \Sigma$ is the $p \times p$ covariance matrix of *X*. I also don't know what these words mean.

Formally, the multivariate normal density function is defined as:

$$f(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2} \times \frac{(x-\mu)^T(\mathrm{x}-\mu)}{\Sigma}\right)$$

If we plug in $f(X = x)$ and do some re-arrangement, we find that the discriminant function is given by:

$$\boldsymbol{\delta_k(x) = \frac{x^T \times \mu_k}{\Sigma} - \frac{1}{2} \times \frac{\mu_k^T}{\Sigma} + \log(\pi_k)}$$

Despite its complex form, $\delta_k(x)$ is a linear function which can be written as:

$$\delta_k(x) = c_{k_0} + c_{k_1}x_1 + c_{k_2}x_2 + \cdots + c_{k_p}x_p$$

4.3. Quadratic discriminant analysis

Recall that we can rewrite Bayes' Theorem in the following form:

$$p_k(X) = P(Y = k|X = x) = \frac{\pi_k \times f_k(x)}{\sum_{l=i}^{K} \pi_l \times f_l(x)}$$

The formula can be applied as long as we know (or approximate) $f_k(x)$. For the LDA classifier, we assume that observations within each class are drawn from a multivariate Normal distribution with a class-specific mean vector and a common covariance matrix. The QDA classifier also assumes that observations come from a Normal distribution but assumes that each class, *k*, has its own covariance matrix. That is, it assumes that an observation from the *k*th class is of the form:

$$X \sim N(\mu_k, \Sigma_k)$$

where Σ_k is a covariance matrix for the *k*th class

Under this assumption, the quadratic discriminant function takes the following form:

$$\boldsymbol{\delta_k(x)} = -\frac{\mathbf{1}}{\mathbf{2}} \times \frac{(\boldsymbol{x} - \boldsymbol{\mu_k})^{\boldsymbol{T}}}{\boldsymbol{\Sigma}} \times (\boldsymbol{x} - \boldsymbol{\mu_k}) + \mathbf{log}(\boldsymbol{\pi_k})$$
$$= -\frac{\boldsymbol{x^T}}{\boldsymbol{2\Sigma_k}}\boldsymbol{x} + \frac{\boldsymbol{x^T}}{\boldsymbol{\Sigma_k}}\boldsymbol{\mu_k} - \frac{\boldsymbol{\mu_k^T}}{\boldsymbol{2\Sigma_k}}\boldsymbol{\mu_k} + \mathbf{log}(\boldsymbol{\pi_k})$$

The QDA classifier requires estimates for Σ_k, μ_k, and π_k, and assigns $X = x$ to the class for which this quantity is largest. Unlike in LDA, the quantity *x* appears as a quadratic function.

Why does it matter whether or not we assume that the *K* classes share a common covariance matrix (i.e., when do we choose QDA vs. LDA)?

- This comes back to the bias-variance trade-off. Estimating a covariance matrix for each class requires estimating $K \times p(p+1)/2$ parameters. With a large number of classes and , we end up having to estimate a lot of parameters.
- If we instead assume that all classes have the same covariance matrix, the LDA model becomes linear in *x*, which mans that there are only $K \times p$ parameters to estimate.
- As a consequence, LDA is a much less flexible classifier than QDA and has much lower variance – but if the assumption of shared covariance matrices is badly off, then LDA may suffer from high bias.

4.4. Classification Error

We will examine a simple examine to shed some light on classification error. We will perform LDA on the data set `Default` in the `ISLR` library to predict whether or not an individual will default on the basis of credit card balance and

student status. The LDA model fit to 10,000 training observations results in a training errorrate of 2.75%. This sounds quite good, but there are several caveats which must be noted.

- First, training error rates will almost always be higher than test error rates. Extremely low training error rates may be equally, or more, indicative of an overfit than a good fit.
    - In general, the higher the ratio of *p/n*, the more we expect overfitting to be a problem.
- Second, since only 3.33% of the individuals in the training set defaulted, a simple but useless classifier that always predicts the majority class will have an error rate of 3.33%.
    - In other words, the trivial *null* classifier will achieve an error rate that is only slightly higher than the LDA training set error rate. Error rates must always be compared to the null classifier

4.4.1. Types of classification errors

With this model, we fail to predict 23/9667 individuals that *did* default, but we only correctly predict 81/333 individuals that *did not* default.

- This gives us a specificity of: $1 - 23/9667 \ = 99.76\%$
- And a sensitivity of: $\frac{81}{333} = 24.32\%$

Why does LDA do such a poor job of classifying the customers who default (i.e., why is the sensitivity so low?) LDA attempts to approximate the Bayes classifier, which has the lowest *total* error rate of all classifiers if the assumptions are correct. However, the Bayes classifier minimizes the error rate irrespective of class – this is not so useful if you are more concerned about identifying individuals that are likely to default than identifying those who won't.

Recall in the discussion about Bayes classifier, by default Bayes classifier aims to minimize total error rate by assigning obsevations to the class for which they have the highest posterior probability. I.e., for a two-class problem:

$$P(default = Yes|X = x) > 0.5$$

However, if we are more concerned about identifying potential defaulters, we can weigh the costs of errors for misclassification to each class. For example:

$$P(default = Yes|X = x) > 0.2$$

This increases the overall error rate, but will decrease the number of false negatives.

A popular way to visualize the effect of changing the threshold is through *receiver operator characteristic (ROC)* curves. ROC curves display the true positive and false positive rates as the threshold slides up and down.

- The overall performance of a model over all thresholds is given by the AUC of the ROC curve.
- An ROC curve may be generated for each classifier (i.e., model).
- ROC curves are useful for comparing different classifiers, since they take into acocunt all possible thresholds.

Below is a confusion matrix, with the terms describing types of classification error.

		predicted condition			
	total population	prediction positive	prediction negative	Prevalence = $\frac{\Sigma \text{ condition positive}}{\Sigma \text{ total population}}$	
true condition	condition positive	**True Positive (TP)**	**False Negative (FN)** (type II error)	True Positive Rate (TPR), Sensitivity, Recall, Probability of Detection = $\frac{\Sigma \text{ TP}}{\Sigma \text{ condition positive}}$	False Negative Rate (FNR), Miss Rate = $\frac{\Sigma \text{ FN}}{\Sigma \text{ condition positive}}$
	condition negative	**False Positive (FP)** (Type I error)	**True Negative (TN)**	False Positive Rate (FPR), Fall-out, Probability of False Alarm = $\frac{\Sigma \text{ FP}}{\Sigma \text{ condition negative}}$	True Negative Rate (TNR), Specificity (SPC) = $\frac{\Sigma \text{ TN}}{\Sigma \text{ condition negative}}$
	Accuracy = $\frac{\Sigma \text{ TP} + \Sigma \text{ TN}}{\Sigma \text{ total population}}$	Positive Predictive Value (PPV), Precision = $\frac{\Sigma \text{ TP}}{\Sigma \text{ prediction positive}}$	False Omission Rate (FOR) = $\frac{\Sigma \text{ FN}}{\Sigma \text{ prediction negative}}$	Positive Likelihood Ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic Odds Ratio (DOR) = $\frac{\text{LR+}}{\text{LR−}}$
		False Discovery Rate (FDR) = $\frac{\Sigma \text{ FP}}{\Sigma \text{ prediction positive}}$	Negative Predictive Value (NPV) = $\frac{\Sigma \text{ TN}}{\Sigma \text{ prediction negative}}$	Negative Likelihood Ratio (LR−) = $\frac{\text{FNR}}{\text{TNR}}$	

4.5. Comparing classification methods

This section has covered logistic regression, LDA, QDA, and KNN. Determining which classification method to use is always an empirical process. ISL provides 6 simulated examples in which different classification models out-perform others. The general idea is: The closer the model matches the process by which the data are generated, the lower the error rates.

Logistic regression and LDA tend to perform better when the decision boundaries are linear, QDA outperforms when the decision boundary is moderately non-linear, and KNN tends to outperform when the underlying function is complex. However, we must carefully choose the smoothness for KNN and other non-parametric approaches. Additionally, by adding polynomial terms to a linear model like LDA or logistic regression, we can accommodate some amount of non-linearity that lies between LDA and QDA.

4.6. `Classification models in R`

4.6.1. `glm()` for logistic regression

`glm()`, 'Generalized Linear Model'

- A *Generalized* linear model (as in `glm()`) is a more generalized general linear model (as in `lm()`), and therefore encompasses `lm()`. Examples of distributions that can be modeled with glm(): **Poisson, binomial, multinomial, normal, gamma.**
- Usage of `glm()` is similar to `lm()` with the exception of the `family` argument, which specifies which distribution and link function should be used to compute the model.

`glm()` - Usage is identical to `lm()`, except it requires that you specify a link function (i.e., `family`).

`formula` - Formula object that specifies the model.

- For logistic regression, the response variable can be given as case-by-case data (a list of 0s and 1s) or as aggregated data, as a 2-column matrix with failures in column 1 and successes in column 2.

`family` - String. A description of the error distribution and the link funciton to be used in the model. I.e.,
"`gaussian`" links to "`identity`", "`log`", "`inverse`"
"`binomial`" links to "`logit`", "`probit`", "`cauchit`", "`log`", "`cloglog`"
(see R help for other families and their respective link functions)

`data` - Data frame or list containing the variables found in `formula`

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, they are ignored.

- Note: The extraction functions described in Section **2.6.3. Extraction functions** also apply to `glm()`.

4.6.2. `lda()` and `qda()` for linear and quadratic discriminant analysis:

LDA and QDA models can be fit using the `lda()` and `qda()` functions as part of the `MASS` library. The syntax for `lda()` is identical to that of `lm()`.

`lda()` - Performs linear discriminant analysis

`formula` - R object that specifies the linear model.

`data` - Data frame or list containing the variables found in `formula`

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, the procedure will fail.

** `qda()` works the exact same way as `lda()`.

Note: the `plot()` function can be applied to `lda` objects.

4.6.2.1. Evaluating `lda()` classification error rates

Confusion matrices

After using `predict()` to create a data frame containing a vector of predictions using the classification model, you can generate a confusion matrix with simple commands:

```
lda.prediction <- predict(lda.model, new.data)   # Make predictions using the lda model (response variable must have the same name as in the formula)
lda.classification <- lda.prediction$class       # Extract the predicted classifications
table(prediction = lda.classification, truth = test.data$classification)      # Generate a confusion matrix that compares your predicted classifications to the true classification of the test data set.
```

```
          truth
prediction   0   1
         0 489  87
         1  17 407
```

The mean is equal to 1 – the overall error rate

```
mean(lda.classification == test.data$classification)
```

```
 [1] 0.896
```

A function for which displays the confusion matrix and prints common classification terms.

```
confusion.matrix <- function(yhat, y) {
        # Inputs must be vectors of 0 and 1 outcomes (and they should be equal length)
    print("Confusion matrix:")
    print(table(prediction = yhat, truth = y))
    print(paste("Sensitivity:", sum(yhat == 1 & yhat == y) / sum(y == 1)))
    print(paste("Specificity:", sum(yhat == 0 & yhat == y) / sum(y == 0)))
    print(paste("PPV:", sum(yhat == 1 & yhat == y) / sum(yhat == 1)))
    print(paste("NPV:", sum(yhat == 0 & yhat == y) / sum(yhat == 0)))
}
```

Adjusting prediction thresholds

By default, predictions are made by assigning observations to the class for which they have the highest posterior probability. However, you can change this threshold manually to generate your own predictions. In this example, we change the threshold to classify a '1' from 0.5 to 0.4.

```
sum(lda.prediction$posterior[, 1] >= 0.40)
[1] 602
```

```
sum(lda.prediction$posterior[, 1] < 0.6)
[1] 448
```

This example uses sums to give us an indication of the overall error rate, but we could just as easily create a new vector of predictions using this rule.

4.6.3. knn()

`knn()` is part of the `FNN` library, though there are many other `knn` packages.

`knn()` - Outputs a vector of factors which are estimates of *y* using the *k* nearest neighbours.

`train` - data frame. Data frame containing all desired predictors, but NOT the response variable. All variables must be numeric.

`cl` - vector. The vector of true classifications in the training data.

`test` - data frame. A second data frame with the same number and name of columns as in the `train` set.

`k` - Integer. The number of nearest neighbours to use

`prob` - logical. Default = FALSE. If TRUE, it will display the posterior probabilities (i.e., the proportion of neighbours that voted for the chosen outcome).

`algorithm` - String. Which algorithm should be used to find the knn: `"kd_tree"`, `"cover_tree"`, `"brute"`

Notes:

- `knn()` can also be used for regression if *y* is continuous, though it can only take on values which already exist in the data set.
- Decisions are made by majority vote, with ties being broken at random. For reproducibility, setting a seed may be a good idea.
- Does not accept NAs.
- The knn model does not know how to properly handle variables with different units (i.e., a difference of $1000 will be weighed more than a difference of $1K). To correct this, apply `scale()` to normalize the data by mean and sd.

Example:

```
library(ISLR); library(dplyr)
set.seed(1)

knn.train.data <- filter(Smarket, set == "train") %>%
    select(Lag1, Lag2, Direction)

knn.test.data <- filter(Smarket, set == "test") %>%
    select(Lag1, Lag2, Direction)

knn.result <- knn(train = knn.train.data[, -3],
                  cl = knn.train.data[, 3],
                  test = knn.test.data[, -3],
                  k = 1)

table(prediction = knn.result, truth = knn.test.data[, 3])
```

```
          truth
prediction Down Up
      Down   43 58
      Up     68 83
```

4.6.3.1. Assessing the knn model

Identify the *k* nearest neighbours selected

```
indices <- attr(knn.result, "nn.index")
indices[1:20]
```

Identify which group the *k* nearest neighbours fall into

`knn.checkclass <- matrix(train.data$classification[indices], ncol = k)` # Finds the classification of the *k* nearest neighbours from the training data. `ncol = k` should be the same `k` as in the original knn model.

I'm not sure what this does:

`attr(knn.result, "nn.dist")`

5. Resampling Methods

Re-sampling procedures are an indispensible tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample - by doing this, we can obtain estimates about the variability of our model. Re-sampling procedures are computationally expensive, but advances in computing power has enabled the use of these procedures.

5.1. Cross-validation

5.1.1. Validation set

In the absence of a very large designated test set that can be used to directly estimate the test error rate, we can still estimate the test error rate by *holding out* a subset of training observations from the fitting process and then applying the model to those observations. While we could simply divide the training set into a training set and a validation set, this has two drawbacks:

- The validation estimate of the test error can be highly variable, depending on which obserations were included in the training vs. test set.
- Using this approach means that we are only using half of our data. As models generally perform better with more observations, the validation set error may over-estimate the test error rate for a model fit on the entire data set.

5.1.2. Leave-One-Out Cross Validation

Leave-one-out cross validation (LOOCV) is similar to the validation set approach, but attempts to address the issues that arise when using the validation set approach. LOOCV has much less bias than the validation set approach; as a result, we tend not to overestimate the test error rate as much. Additionally, unlike the validation set approach, LOOCV yields the same test error estimate every time.

LOOCV is simply performed by fitting a model which includes all of the training set observations except for one (*n* - 1). Then a prediction is made for the excluded observation to obtain the MSE (in the case of a quantitative variable and least-squares regression). We perform this process *n* times, and take the average MSE of all our predictions to obtain the CV error (our estimate for the test error):

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n} MSE_i$$

LOOCV has the potential to be computationally expensive. However, in *least squares linear regression*, there is an elegant shortcut that means that the model need only be fit once:

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - h_i}\right)^2$$

Where $\hat{y}_i$ is the fitted value from the original least squares fit, and
h_i is the leverage statistic for the observation (covered in linear regression under **2.5.2**)

In essence, this equation inflates the error of observations with high leverage, which allows the equality to hold. However, this equation does not hold in general, in which case the model has to be re-fit *n* times.

5.1.3. *k*-Fold cross validation

k-fold CV is a generalized approach for CV, and the most widely applied. KFCV breaks up the training data into *k* groups (or *folds*) of approximately equal size. A model is fit *k* times, each using all but one fold and then using the remaining fold as the validation set. We then average the MSE over all folds to obtain the CV error.

$$CV_{(n)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

LOOCV is a special case of KFCV in which *k* = *n*. We can easily see that this equation is identical to the one given for LOOCV, except we use *k* instead of *n*.

In practice, *k*-fold CV is performed using *k* = 5 or *k* = 10. Obviously, it is less computationally expensive than LOOCV, but there is a non-computational advantage as well. Each model that we fit shares a portion of the information. In the case of LOOCV, each model is almost identical (i.e., they are highly correlated). So while LOOCV eliminates the risk of bias associated with the validation set approach, it results in higher variance. A *k* of 5 or 10 generally does a reasonable job at optimizing this bias-variance trade-off.

We can just as easily apply cross validation to classification problems with a qualitative response variable:

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n} Error_i$$

5.2. The bootstrap

Bootstrapping is a simple, but powerful procedure that allows is to quantify the uncertainty associated with our parameter estimates (i.e., statistics) or statistical learning method. For some statistical learning methods, like linear regressions, calculating standard errors for statistics is trivial and provided automatically in software outputs. However, the bootstrap can be applied to a wide range of statistical learning methods, including some for which a measure of variability is otherwise difficult to obtain.

Bootstrapping uses resampling to simulate a sampling distribution. We draw a large number of samples (with replacement) from the original data set - each acts as an approximation of another sample drawn from the population. We can then compute statistics for each sample (ex., mean, co-efficients), followed by standard deviation for each statistic (i.e., standard error estimates). You can also easily calculate confidence intervals by taking the mean *y* value at the appropriate quantiles.

Why use bootstrapping (from *Introduction to the Practice of Statistics, 6e*)

- Bootstrapping allows us to bypass some of the conditions and assumptions that are required for traditional inference procedures. However, certain assumptions are still required for bootstrapping to work, such as random sampling from the population and independence of observations.
- Theoretical approaches to statistics, such as *t* procedures, are not appropriate when data are highly skewed. Bootstrapping is able to get around this.
- There is no traditional inference procedure for the ratio of two means.

Block bootstrap

Sampling with replacement is only appropriate when the observations are independent of one another (i.e., it cannot be used when observations are correlated, such as in time-series). In this case, it is possible to break the observations into blocks which are independent of one another - instead of sampling individual observations, you sample blocks to construct the bootstrap data sets.

5.3. Cross-validation and Bootstrap in R

5.3.1. Validation set

```
set.seed(1)
size <- length(iris[, 1])
train <- sample(size, size / 2)    # sample half the obs

# Fit a model on the training set
lm.fit <- lm(Sepal.Length ~ Sepal.Width, data = iris, subset = train)

# Calculate the MSE for the validation set
mean( (iris$Sepal.Length - predict(lm.fit, data = iris))[-train]^2 )
```

5.3.2. `cv.glm()` for cross-validation

`cv.glm()` - Calculates the estimated *K* fold cross-validation prediction error for glms.

`data`	- Matrix or data frame.
`glmfit`	-A `glm` object containing the results of a generalized linear model fitted to `data`.
`cost`	- A function of wo vector arguments specifying the cost function fo the cross-validation. By default, it is the mean squared error.
`K`	- The number of folds to use in cross validation. By default, K is equal to the length of the data (i.e., LOOCV)

10-fold CV:

```
glm.fit <- glm(Sepal.Length ~ Sepal.Width, data = iris) #simple linear model
cv.err <- cv.glm(iris, glm.fit, K = 10)

names(cv.err)
[1] "call"  "K"     "delta" "seed"

cv.err$delta
[1] 0.6894815 0.6885508

# The second delta is a bias-adjusted CV estimate of the error.
```

5.3.3. `boot()` for the boostrap

`boot()`

`data`	- Vector, matrix, or data frame from which to sample
`statistic`	- An (often user-defined) function. In general, it must take two arguments: `data` and `index`, which specifies the observations to sample from.
`R`	- The number of bootstrap replicates.

```
coeff <- function(data, index)
    return(lm(Sepal.Length ~ Sepal.Width, data, subset = index)$coeff)
coeff(iris, 1:150)

boot(iris, coeff,  1000)
```

6. Linear Model Selection and Regularization

6.1. Subset selection

6.1.1. Best subset selection

Subset selection is choosing which model(s) we wish to use. Best subset selection is a simple (but expensive) method by which we evaluate every single possible model given p predictors (i.e., 2^p). For example, we select the best model for every number of predictors; we then look at each of these models to identify which model is the *best* (i.e., how many predictors should the model include). This can be described by the following algorithm:

Best subset selection algorithm

1. Let M_0 denote the *null model*, which contains no predictors and simply predicts the sample mean for each observation.
2. For k = 1, 2, ... p:
    a. Fit all $\binom{p}{k}$ models that contain k predictors.
    b. Choose M_k that minimizes RSS / maximizses R^2.
3. Select a single best model from M_0, ..., M_p using some validation method (ex., CV, AIC, BIC, etc.)

Notes

- We discuss RSS here, but the sample principles can be generalized to other models.
- Step 2 is trivial but computationally expensive.
- Step 3 is non-trivial, since training error always decreases monotonically as you add predictors. Therefore, a model containing all p variables will always have the lowest training error (but not necessarily the lowest test error).
- Best subset selection also has non-computational limitations, which similarly fall under the bias-variance trade-off; by looking at every single possible model, you increase your chances of over-fitting compared to if you look at a smaller subset of models to begin with (i.e., it has high variance).

6.1.2. Stepwise selection

Instead of comparing 2^p possible models, we can explore a more restricted set of models using automated approaches. The three classical approaches are:

- Forward selection
- Backward selection
- Mixed/hybrid selection

6.1.2.1. Forward selection

We begin with a *null model* – containing only an intercept. We then compare p and add the one that results in the lowest *RSS*. We continue this process until some stopping rule is satisfied. As a result, we evaluate the following number of models:

$$1 + \sum_{k=0}^{p-1} (p - k) = 1 + \frac{p(p+1)}{2}$$

Where k is the k^{th} predictor (out of the total p predictors)

Which is substantially smaller than 2^p (when $p = 20$, best subset selection requires fitting ~1,000,000 models, while forward selection requires ~200). Forward selection can be summarized in the following algorithm:

Forward stepwise selection algorithm:

1. Let M_0 denote the *null model*.
2. For $k = 0, \ldots p - 1$:
    a. Consider all $p - k$ models containing one more predictor than M_k.
    b. Choose M_{k+1} that minimizes RSS / maximizses R^2.
3. Select a single best model from $M_0, \ldots, M_p$ using some validation method (ex., CV, AIC, BIC, etc.)

Forward selection tends to perform well in practice. However, it may perform worse than best subset selection when variables are highly correlated because this method may hold onto variables that were included early, but later become redundant.

6.1.2.2. Backward selection

We being with a model containing all variables and then perform step-wise drop-offs of the least statistically significant variable (i.e., largest *p-value*) until some stopping rule is satisfied (ex., all *p-values* are below some threshold) or until only 1 predictor remains. Backward selection can be summarize in the following algorithm:

Backward stepwise selection algorithm:

1. Let M_p denote the *full model*.
2. For $k = p, p - 1, \ldots, 1$:
    a. Consider all k models containing one less predictor than M_k.
    b. Choose M_{k-1} that minimizes RSS / maximizses R^2.
3. Select a single best model from $M_0, \ldots, M_p$ using some validation method (ex., CV, AIC, BIC, etc.)

Backward selection requires that $n > p$, while forward selection can be used in this case. Backward selection evaluates more possible models than forward selection, and so is more computationally expensive.

6.1.2.3. Mixed/hybrid selection

Best subset, forward, and backward stepwise selection tend to give similar, but not identical, models. Hybrid versions of forward and backward stepwise selection attempt to mimic best subset selection while still reducing the computational overhead.

We start with a null model and add predictors as in forward selection. However, as we add variables, some old predictors may become redundant – these variables are dropped. This procedure of adding and dropping variables continues until some stopping rule is met (ex., all *p-values* in the model are sufficiently low, and any additional variables would have high *p-values*).

Example hybrid stepwise selection algorithm:

- Note: I made this up; it may be that this doesn't work in practice.

1. Let M_0 denote the *null model*.
2. For k = 0, 1, ..., p:
    a. Consider all 2k models containing one more predictor and one less predictor than M_k.
    b. Continue with the model that adds or removes a variable that minimizes RSS / maximizes R^2.
    c. If an iteration occurs where same variable is added and then removed, stop.
    d. For all models with a given number of predictors, k, choose M_k that minimizes RSS / maximizes R^2.
3. Select a single best model from M_0, ..., M_p using some validation method (ex., CV, AIC, BIC, etc.)

6.1.3. Choosing the optimal model

Adding additional predictors to a model will monotonically decrease the training error, but not necessarily the test error. There are two main approaches by which we can estimate the test error:

- Indirectly estimate the test error by making an adjustment to the training error by including a penalty for additional predictors
- Directly estimate the test error rate using a validation set approach or cross-validation approach.

6.1.3.1. C_p, AIC, BIC, and Adjusted R^2

C_p, Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), and Adjusted R^2 are methods for estimating the optimal number of predictors using the training error, but penalizing for a high number of predictors. All four methods are variaitons of the same concept, just using different penalties.

C_p

$$C_p = \frac{1}{n}(RSS + 2d\hat{\sigma}^2)$$

Where d is the number of predictors in the model, and
$\hat{\sigma}^2$ is an estimate of the variance of the error term, ϵ.

We see from this equation that C_p adds a penalty term to the RSS to compensate for the fact that training MSE tends to be lower than the test MSE. It is possible to show through black magic that if $\hat{\sigma}^2$ is an unbiased estimator of σ^2, then C_p is an unbiased estimator of the test MSE.

AIC

$$AIC = \frac{1}{n\hat{\sigma}^2}(RSS + 2pd)$$

This is a simplified version of the formula given in ISL. In linear regression, AIC is proportional to C_p, so both would minimize to the same model.

BIC

$$BIC = \frac{1}{n}(RSS + \log(n)d\hat{\sigma}^2)$$

The BIC is similar to C_p except that it replaces $2p\hat{\sigma}^2$ with $\log(n)p\hat{\sigma}^2$. Since $\log(n) > 2$ when $n > 7$ (which is almost always in practice), the BIC puts a heavier penalty on the $p\hat{\sigma}^2$ term (i.e., the number of predictors).

Adjusted R^2

$$Adjusted\ R^2 = 1 - \frac{RSS/(n-d-1)}{TSS/(n-1)}$$

Adjusted R^2 is similar to R^2 except that it includes a penalty which increases the RSS term the more predictors are introduced. Unlike the other metrics, a high Adjusted R^2 indicates a model with small test error. Unlike C_p, AIC, and BIC, Adjusted R^2 is not well motivated by statistical theory (i.e., it's the worst of the three).

6.1.3.2. Validation and Cross-validation

Another way to choose an optimal model is to directly estimate the test error using a validation set or cross-validation methods (discussed above). This approach has prefered over indirect estimates like C_p, AIC, and BIC, because they make fewer assumptions about the underlying modeland can be used in a wider range of cases (ex., some times it is hard to estimate the variance of the error).

6.1.3.3. The one-standard-error rule

Both cross-validation, and indirect test error metrics will be minimized by a single model, but often adding additional predictors only decreases the estimated test error very slightly. In these cases, it is hard to justify p predictors compared to $p + 1$ or $p - 1$ predictors.

It is worthwhile to compare the estimated test errors as a function of the number of predictors and apply the *one-standard-error rule*. That is, we calculate the standard error of the estimated test MSE for each model size, and then select the model with the fewest predictors tht is still within one standard error. The rationale is that, if several models all appear to perform just as well, then we should choose the simplest model because it will be more generalizeable and interpretable.

6.2. Shrinkage Methods

Shrinkage methods are an another approach that can be used to select models as alternatives to subset selection methods. Each model has a coefficient 'budget.' Subset selection methods get around this by removing variables that are not worth it. In contrast, shrinkage methods can be used to fit a model that contains all *p* predictors, but contrains the coefficient estimates. While it is not immediately obvious how this is useful, shrinking coefficient estimates can greatly reduce the variance before introducing a large amount of bias.

6.2.1. Ridge Regression

Recall that regression methods typically minimize the residual sums of squares (RSS):

$$RSS = \sum_{i=i}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

Ridge regression is the same as least squares regression, except it introduces a penalty for having a large coefficient - ridge regression chooses coefficient estimates, $\hat{\beta}^R$, that minimize the RSS + the shrinkage penalty:

$$RSS + \lambda \sum_{j=1}^{p} \beta_j^2 = \sum_{i=i}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Where λ is the *tuning parameter*, which is determined by the user.

- Notice that the equation uses the squared coefficient estimates, β_j^2, to calculate the penalty. Thus, as the coefficients increase in magnitude, the mode they are penalized.
- Note: We don't shrink the intercept.

6.2.1.1. Applying Ridge Regression in practice

The tuning parameter is a user defined weight which is important for determining how badly to penalize large coefficients. When the tuning parameter is zero, the ridge regression is equivalent to full lease squares regression.

When applying ridge regression in practice, we are trying to find the value for λ that minimizes the test error. This is similar to the process of finding the optimal number of preditors, *d*, to minimize the test error. Selection of the tuning parameter should be done by cross-validation (we can't use methods like AIC because shrinkage methods treat predictors differently).

Unlike standard least squares regression, ridge regression estimates are *not* scale invariant. In least squares regression, if we multiple X_j by 1000, then β_j will be 1000-times smaller, resulting in the same value of $X_i\beta_j$. In ridge regression, however, the shrinkage penalty term does not incorporate *X*, only β_j^2. Therefore, changing units can substantially affect estimates. A simple solution to this problem is to standardize the predictors by *z-score*:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \bar{x}_j)^2}}$$

The demoninator is simply the estimated standard deviation of the j^{th} variable - the result of this is that all the standardized predictors will have a standard deviation of 1.

One disadvantage of ridge regression is that, while shrinkage penalty shrinks coefficients toward zero, none of the coefficients are exactly set to zero unless $\lambda = \infty$. As a result, ridge regression includes all predictors in the final model - while this does not affect model accuracy, it is difficult to interpret ridge regression outputs when *p* is large

6.2.2. The Lasso

The lasso is another shrinkage technique that is similar to ridge regression, but automatically 'selects' a subset of predictors. This is because the lasso coefficients, β_λ^L, minimize the RSS + a shrinkage penalty that depends on the *absolute* value of β_j:

$$RSS + \lambda \sum_{j=1}^{p} |\beta_j| = \sum_{i=i}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Just like ridge regression, the lasso will shrink coefficients toward zero as $\lambda \to \infty$. However, $|\beta_j|$, unlike β_j^2, will shrink coefficients to exactly zero if λ is large enough. Mathematically, this is because using $|\beta_j|$ yields sharp 'corners' that occur on axes (i.e., where coefficients = 0). Thus, lasso automatically performs variable selection. We say that the lasso yields *sparse* models - i.e., only a subset of the variables are included.

6.2.2.1. Intuition behind shrinkage methods

One way to describe how shrinkage methods work is the re-state the equation above as:

Ridge regression:

$$\min_{\beta} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \right\} \text{ subject to } \sum_{j=1}^{p} \boldsymbol{\beta_j^2} \leq s$$

Lasso:

$$\min_{\beta} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \right\} \text{ subject to } \sum_{j=1}^{p} |\boldsymbol{\beta_j}| \leq s$$

We can think of this equation as follows: When we perform lasso, we are trying to find the set of coefficient estimates that minimize the RSS, subect to the constraint that there is a budget, *s*, for how large the sum of the absolute coefficient estimates.

- When *s* is large enough, there is no constraint on the coefficients, and a full least squares regression can be fitted.
- As *s* decreases, the coefficient estimates must be small in order to avoid violating the budget.

- For every value of λ, there is a value, *s*, that yields an optimal set of ridge regression coefficient estimates.

Note that ridge regression, the lasso, and best subset selection have very similar formulas, except that the constraints are applied differently.

Consider the problem for best subset selection:

$$\min_{\beta} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \right\} \text{ subject to } \sum_{j=1}^{p} \boldsymbol{I(\beta_j \neq 0)} \leq s$$

$I(\beta_j \neq 0)$ is a logical indicator variable, which takes on a value of 1 if true or 0 if false. In this equation, we are constraining the number of predictors with non-zero coefficient estimates to *s*. However, this problem is expensive to solve when *p* is large, since we must solve for all models containing *s* predictors. We can instead interpret ridge regression and lasso as computationally cheaper alternatives to best subset selection, which apply constraints that are easier to solve.

6.2.3. Choosing Lasso vs. Ridge Regression vs. Least Squares

6.2.3.1. Shrinkage methods vs. Least Squares

- Compared to full least squares regression, shrinkage methods reduce variance and increase bias.
- When the true relationship between predictors and the response is close to linear, a linear model will have low bias, but high variance.
    - Particularly when *p* is close to *n*, variations in the training data can lead to large differences in the co-efficient estimates. When *p* > *n*, the least squares estimates do not even have a unique solution.
- Thus, shrinkage methods work best in situations where the least squares estimates (are likely to) have high variance.
- Ridge regression also has computational advantages over subset selection..

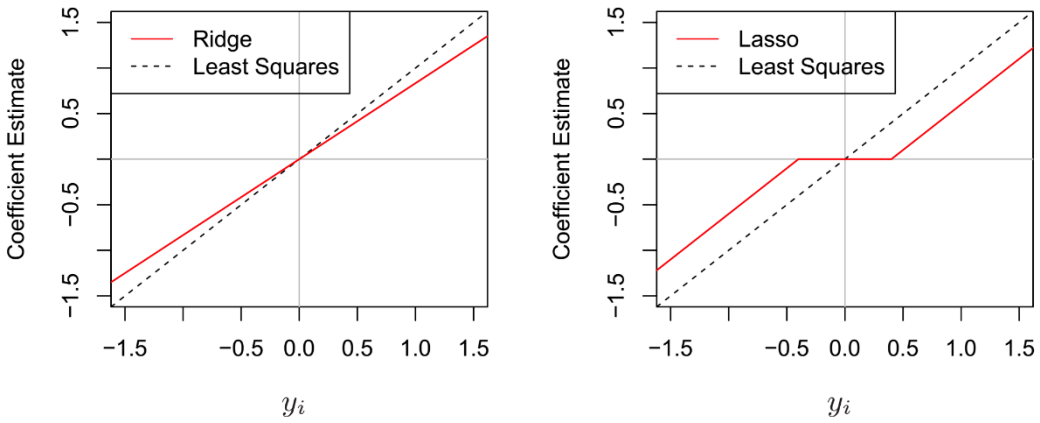
6.2.3.2. Lasso vs. Ridge Regression

- Neither lasso nor ridge regression will dominate the other in all cases.
    - Since lasso performs yields a sparse model, it tends to outperform ridge regression when the number of predictors related to the resposne is *small*.
    - Since ridge regression yeilds a dense model, it tends to outperform lasso regression when the number of predictors related to response is *large*.
- However, it is never known *a priori* how many predictors will be related to the response - so we must cross-validate our models to determine which is approach is most appropriate.
- In general, we might hope that the true model is sparse, since it is easier to interpret these models - thus, all else equal, we may prefer lasso regression.

6.2.4. Mathematical quirks of ridge regression vs. lasso

These sections are a bit over my head, but do shed some insight on how ridge regression and lasso work internally.

Ridge regression and lasso perform two very different types of shrinkage. We see in the figure below that ridge regression (left) shrinks the coefficient estimates proportionally to the full least squares regression estimates. In contrast, lasso (right) aplies shrinkage using soft-thresholding. That is, each coefficient estimate is shrunk by a constant, with thresholds and different values of y_i. From this, we see that sufficiently small coefficients are shrunken exactly to zero. Note that this figure depicts a greatly simplified example, but the general ideas still hold for more complex data.



6.2.4.1. Bayesian interpretation for ridge regression and lasso

(Unfinished)
Ridge regression and lasso can be viewed from a Bayesian perspective

6.3. Dimension Reduction Methods

Dimension reduction methods decrease variance by transforming the predictors and then fitting a least squares model using the transformed predictors. Dimension reduction methods work by reducing the number of predictors, *p*, to a smaller subset, *M*. This is done by combining the information given by multiple variables. For example, instead of height and weight we could compute BMI, which uses pieces of information from each variable. Dimension reduction methods try to do this in ways that don't rely on *a priori* knowledge.

We define $Z_1, ..., Z_M$ as the a *linear combinations* of our original *p* predictors:

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

Where $\phi_{1m}, \phi_{2m}, \ldots, \phi_{jm}$ is a vector of constants for each of $m = 1, ..., M$.

It is important how the ϕ constants are chosen - and different methods will choose these differently (see the sections below). We can fit a linear regression model using the new predictors:

$$y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} \quad , \qquad i = 1, \dots, n$$

Where θ_m are the regression co-efficients for the new z_{im} predictors.

We can substitute z_{im} in the above equation and do some re-arrangement to reveal an interesting property:

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{m=1}^{M} \theta_m \sum_{j=1}^{p} \phi_{jm} X_j = \sum_{m=1}^{M} \sum_{j=1}^{p} \theta_m \phi_{jm} X_j = \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} \boldsymbol{\beta_j X_j}$$

Where

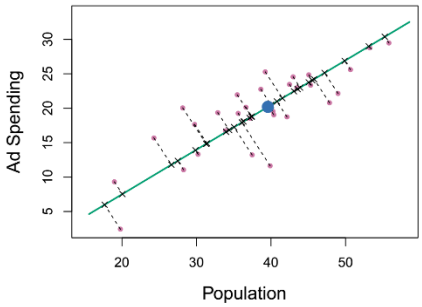
$$\beta_j = \sum_{m=1}^{M} \theta_m \phi_{jm}$$

Thus, we see that dimension regression can be thought of as a special case of the OLS regression where β_j is constrained to the form $\sum_{m=1}^{M} \theta_m \phi_{jm}$. These constraints may bias the coefficient estimates, but choosing a model for which $M << p$ can greatly reduce the variance of the coefficients. If $M = p$ and all Z_m are linearly independent, then these models are equivalent to OLS regression with p predictors.

6.3.1. Principle Component Analysis

PCA is a popular technique for reducing the dimensions of a data set. It is discussed in more detail in a future section. An overview is given here.

The *first principle component* is the 'direction' along which the observations from the data *vary the most*. For example, in an example with two highly correlated variables, we can imagine the first principle component being a line which minimizes the squared errors..



The first principle component in the graph to the right can be summarized by the following equation:

$$Z_1 = 0.839 \times (\text{pop} - \overline{\text{pop}}) + 0.544 \times (\text{ad} - \overline{\text{ad}})$$

Where Z_1 is a vector of length n (i.e., the same length as the variables)

- Here $\phi_{11} = 0.839$ and $\phi_{21} = 0.544$ are the principal component loadings, which define the 'direction' referred to above.
- PCA finds values for ϕ such that $\phi_{11}^2 + \phi_{21}^2 = 1$; this particular linear combination yields the highest variance. That is to say, $Var(0.839 \times (\text{pop} - \overline{\text{pop}}) + 0.544 \times (\text{ad} - \overline{\text{ad}}))$ is maximized under this circumstance.
- A positive z_{i1} value indicates the particular observation has below-average population size and below-average ad spending.

The second principle component finds the direction along which the observations vary the most *but must be uncorrelated to* Z_1. This is equivalent to saying that Z_2 must be *perpendicular or orthogonal* to Z_1. When $p > 2$, we can continue this pattern to yield up to p principal components.

6.3.1.1. Principle components regression

PCR involves constructing the first M principal components, and then using these as predictors in a least squares linear regression. PCR makes an implicit assumption that the directions in which X_1 ..., X_p vary the most are the directions that are associated with Y. While this is not guaranteed to be true, it often turns out to be a reasonable approximation.

If this assumption holds, then PCR will lead to better results than OLS, since most of the information in the data relating to the response is stored in Z_1, ... Z_M; if M is much smaller than p, we can mitigate overfitting.

If the original predictors are not in the same units, it is recommended to standardize the predictors using *z-scores* (as with ridge regression), since variables with a higher variance will have a larger effect on the PCR model. However, if all variables are in the same units, they can be kept that way (and not standardize the variances).

- As we increase M, bias decreases, but variance increases. We generally choose M by cross-validation.
- Because the main benefit of PCR is that it can reduce the number of predictors, it only performs better when the data can be summarized well with a small M.
    - Like the lasso, if all of the predictors are related to the response, then M will have to be large to adequately summarize the data.
- If $M = p$, it is equivalent to OLS. If M is large, it is often better to use OLS or ridge regression.
- Note: PCR is not a feature selection method, since each of the principle components is constructed using all of the original predictors.
    - In this way, PCR is similar to ridge regression. In actuality, PCR and ridge regression are mathematically very similar - and ridge regression can be though of as a continuous version of PCR.

6.3.2. Partial Least Squares

PLS addresses some of the short-comings of PCR, but in practice it does not seem to provide much improvement over PCR. For that reason, it is only discussed briefly here.

An apparent shortcoming of PCR is that the principle components are identified in an *unsupervised way*, since the response to Y is not considered (that is, the response does not *supervise* the identification of principle components). PLS is the same as PCR, except the new predictors, Z_1, ... Z_M, are selected in a *supervised* way (that is, it makes use of the response variable). Broadly speaking, PLS attempts to find 'directions' that help explain both the response and the predictors.

Here is a surface-level description of how PLS directions are computed. After standardizing the p predictors, PLS computes the Z_1 by setting each ϕ_{j1} equal to the coefficient from simple linear regression of Y onto X_j. It is possible to show that this coefficient is proportional to the correlation between Y and X_j. Therefore, PLS puts the highest weights on variables that are the most strongly related to the response.To take the second PLS direction, we first adjust each of the remaining variables for Z_1 by regressing each variable X_1, ... X_p on Z_1 and taking the *residuals*. The residuals are the remaining information that has not been explained by the first PLS direction. We then compute Z_2 or this *orthogonalized* data just like Z_1. Again, we can repeat this up to M PLS directions. Finally, we fit a linear model just as in PCR. M is determined by cross-validation.

6.4. The Problem of High Dimensionality

6.4.1. The Problem with High Dimensions

Traditional statistical methods were not designed for high dimensional data. By 'traditional statistical methods' I am referring to linear regression, logistic regression, and linear discriminant analysis, among many others. For example, when p is as large as n, a linear regression line will always yield a perfect fit, regardless of whether or not there is a true relationship in any of the variables. Consider the case where $p = 1$ and $n = 2$; this will always result in a perfect line through each point; this holds true for any high dimension data set.

High dimensional data yield models that are too flexible. The training error will monotonically decrease as p increases, but the test error will often increase.

6.4.2. Regression in High Dimensions

The methods covered in this section (subset selection, shrinkage, and dimension redution) all serve to reduce the variance in a high dimensional data set. However, even these methods perform poorly when the model is extremely sparse (very large p, but very few are associated with the response).

High dimensional problems are also more susceptible to multi-collinearity. Recall that this is when multiple predictors are correlated with each other and may not be directly related to response. If we perform step-wise subset selection, we will conclude with a model that includes d predictors; while these d predictors may be very useful in predicting response, it would be incorrect to say that they will predict response better than any other set of d predictors. If the predictors are highly correlated, running the same step-wise procedure on an independent data set would likely obtain a different set of predictors.

It is also important to note that traditional statistics of model fit should *never* be used to report model error (i.e., p-values, MSE, or R^2. Cross-validation errors or MSE on an independent validation should be reported instead.

6.5. Model Selection In R

6.5.1. `regsubsets()` for subset selection

`regsubsets()` - Model selection by exhaustive search, forward/backward stepwise, or sequential replacement.

`x`	- Design matrix or formula for the model (like with `lm()`)
`data`	- Optional data frame
`y`	- response vector (not necessary if `x` is a formula)
`method`	- String. "`exhaustive`", "`backward`", "`forward`", "`seqrep`"
`nvmax`	- Integer.Maximum size of subsets to examine (default = 8).

```
names(summary(reg_subsets_object))
[1] "which"  "rsq"    "rss"    "adjr2"  "cp"     "bic"    "outmat" "obj"
```

Example (best subset selection)

- Applying different subset methods is done the same way – we just change the `method` argument.

```
library(ISLR)
library(leaps)    # Contains the regsubsets for best subset

str(Hitters)

regfit.full <- regsubsets(Salary ~ ., na.omit(Hitters), method = "exhaustive", nvmax
= 19)
summary(regfit.full)    # Outputs the best model for each model size. "*" indicates
that the variable was included in the model.

reg.summary <- summary(regfit.full)

par(mfrow = c(2, 2))
plot(reg.summary$rss, type = "b")
plot(reg.summary$cp, type = "b")
plot(reg.summary$bic, type = "b")
plot(reg.summary$adjr2, type = "b")

# Let's say we choose the model with 6 predictors, based on the BIC. We then find
the call for this model:

coef(regfit.full, 6)
```

7. Non-linear Models

7.1. Polynomial Regression

Polynomial regression is a simple extension of linear regression, and technically not even a different thing. If the polynomial is large enough, polynomial regression can produce extremely non-linear curves. Generally speaking, though, it is unlikely to exceed 3 or 4 degrees because the fit becomes overly flexible (particularly at the boundaries of the *X* variable).

Notably, polynomial regression can be used for any type of generalized linear model, including logistic regression.

7.2. Step Functions

One of the main problems with polynomial regression is that imposes a *global* structure on the function of *X*. For example, an even polynomial will dictate that the fit follows a parabola. We can instead use *step functions*, which break up the range of *X* into bins, and then fits a different constant to each bin. This is equivalent to convering a continuous variable into an ordinal variable.

We can create *K* cut points to create *K* + 1 new dummy variables:

$$C_0(X) = I(X < c_1)$$
$$C_1(X) = I(c_1 < X < c_2),$$
$$\vdots$$
$$C_K(X) = I(c_K < X)$$

Where *I* is a logical indicator that returns 1 if true and 0 if false.

This can be easily performed using `I()` or `cut()` in `R`.

7.3. Basis Functions

Polynomial regression and step-functions are really just a special case of a *basis function* approach. The basis function approach is identical to linear regression, but instead of directly using the predictors, *x*, we transform the predictors through a basis function, $b_1(x), \ldots, b_K(x)$, and instead use those. The model can be summarized as follows:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \cdots + \beta_K b_K(x_i) + \epsilon_i$$

Note that the basis function is fixed and must be chosen a head of time. In polynomial regression, for example, we use the basis function: $b_j(x) = x_i^j$. The basis function approach still follows the standard linear regression model, so all of the statistics and inference tools used in linear regression can also be applied to basis functions.

7.4. Regression Splines

Regression splines are a flexible class of basis functions that extend upon the polynomial and piecewise regression approaches discussed above. A spline is defined as: 'A numeric function that is piecewise-defined by a set of functions (usually polynomials), with constraints on the first (and potentially higher order) deriative to ensure continuity/smoothness.'

7.4.1. Piecewise Polynomials

Piecewise polynomials combine polynomial regression with step functions. Instead of fitting a high-degree polynomial over the entire range of *X*, we can fit lower degree polynomials over different regions of *X*. A piecewise cubic polynomial model might loook like:

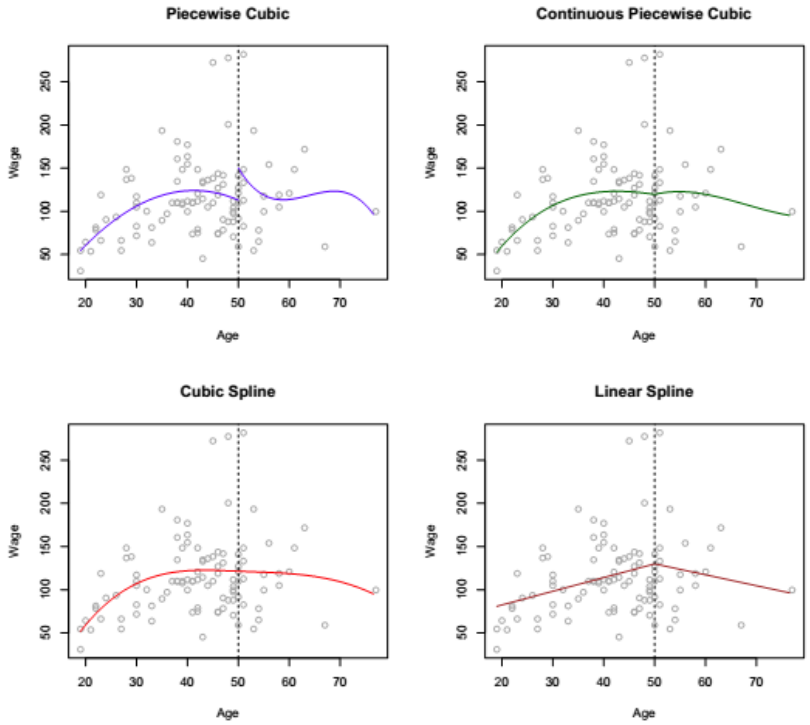
$$y_i = \begin{cases} \beta_{01} + \beta_{11} x_i + \beta_{21} x_i^2 + \beta_{31} x_i^3 + \epsilon_i & \text{if } x_i < c; \\ \beta_{02} + \beta_{12} x_i + \beta_{22} x_i^2 + \beta_{32} x_i^3 + \epsilon_i & \text{if } x_i \geq c. \end{cases}$$

Essentially, we are fitting two different polynomial functions to the data, depending on where it falls in *X*. The point where the co-efficients change, *c*, is called a *knot*. Including more knots leads to a more flexible model. However, without constraints and splines, the two different models yield vastly different estimates for y_i near the knots (*top-left of figure below*).

7.4.2. Constrains and Splines

In order to reduce the variance in a piecewise polynomial, we can add constraints, such as: "the fitted curve must be continuous." (*top-right of figure below*). While this results in a more sensible function, we see an unnatural 'V' shape at the knot.

We can add further constraints: In the lower-left plot the model has the additional constraints that both the first and second derivatives must be continuous. Adding these constraints 'frees up' one degree of freedom by reducing the complexity of the fit.



7.4.3. The Spline Basis Representation

In general, regression splines are cubic (because supposedly humans cannot detect discontinuity at the 3rd derivative). Regression splines can be modeled using the basis model. I don't really follow the math, so I won't write it down here. Essentially, we start off by representing polynomial regression using basis functions as we normally would. i.e.,

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \cdots + \beta_K b_K(x_i) + \epsilon_i$$

...but we add an addition *truncated power basis* function per knot. A truncated power basis function (for a cubic spline) is defined as:

$$h(x,\xi) = h(x,\xi)_+^3 = \begin{cases} (x,\xi)^3 & \text{if x} > \xi \\ 0 & \text{otherwise} \end{cases}$$

Where ξ is the knot.

Again, the math is beyond me, but we can show that a cubic spline with *K* knots uses *K* + 4 degrees of freedom.

Natural Splies

Splines have high variance at the outer range of the predictors, since polynomials are highly variables near the edges. A simple solution to this (and commonly used) is to use *natural splines*. Natural splines are constrained to be linear at the boundary. In general, this results in estimates that are more stable at the boundaries.

7.4.4. Choosing the number and location of knots

Regerssion splines are most flexible in the regions that contain a lot of knots, so one option is to place more knots in places where the function might vary most rapidly. In practice, people tend to place knots in a uniform fashion. Software can often do this automatically if you specify the desired degrees of freedom.

Choosing the *number* of knots is slightly more complex. A subjective approach may be to simply look at the curve and see which one looks the most sensible. A more objective approach is to use cross-validation (often 10-fold).

7.5. Smoothing Splines

7.5.1. Overview of smoothing splines

Regression splines aim to minimize RSS. Smoothing splines are a variation on regression splines that minimize RSS + a penalty term. This is a similar concept to ridge / lasso. The equation for smoothing splines is given below:

$$\sum_{i=1}^{n}(y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt$$

Where $g(x)$ is some function that minimizes the value of the equation,
λ is a non-negative tuning parameter,
$g''(t)$ is the second derivative of $g(x)$.

Recall that the second derivative is a change in slope (i.e., roughness; consider that the second derivative of a line is zero). Thus, the penalty term punishes models that are not smooth. Note that if $\lambda = 0$, we arrive at a function that perfectly interpolates every training data point. If $\lambda = \infty$, we get a least squares line.

It can be shown that the function $g(x)$ that minimizes the above equation is in fact a *shrunken* natural piecewise cubic polynomial with knots at *every observation*, with the level of shrinkage controlled by λ. (It ends up being cubic because we choose the second derivative as the penalty term).

7.5.2. Choosing the smoothing paramter λ

A smoothing spline may therefore have *many* degrees of freedom, resulting in excessive flexibility. However, the tuning paramter, λ, controls the roughness of the smoothing spline, and hence the effective degrees of freedom. It is possible to show that as λ increases from 0 to ∞, the effective degrees of freedom, λ_{df}, decrease from *n* to 2.

Usually degrees of freedom refer to the number of free parameters, such as the number of coefficients fit in a model. Alrhough a smoothing spline has *n* parameters, and hence *n* nominal degrees of freedom, these *n* parameters are heavily shrunken down. Therefore, λ_{df} is used as a measure of flexibility of a smoothing spline.

With smoothing splines, we do not need to choose the number of knots, as there will automatically be one at each observation. Instead, we need to choose λ. This is often done by cross-validation. LOOCV can actually be computed very

efficiently with smoothing splines, with essentially the same cost as computing a single fit, using a formula that I don't bother writing down here because I will never understand it.

7.6. Local Regression

Local regression is a different approach for fitting flxible non-linear functions. Local regression computes a fit for a given x_0, but does so only using nearby training observations. The procedure is as follows:

Algorithm for local regression

1. Gather the fraction $s = k / n$ training observations whose x_i are closest to the point being estimated, x_0.
    - The user must define the span, *s*, to use in the local regression.
2. Assign a weight $K_{i0} = K(x_1, x_0)$ to each point in this neighbourhood such that the furthest point from x_0.
    - The user must define this weighting function (ex., if it is linear, quadratic, etc.).
3. Fit a *weighted least squares regression* of $y_i$on the x_i using the weights in step (2) by identifying the coefficients that minimize:

$$\sum_{i=1}^{n} K_{i0}(y_i - \beta_0 - \beta_1 x_i)^2$$

4. Compute the fitted value at x_0 given by $=\hat{f}(x_0) = \hat{\beta}_0 - \hat{\beta}_1 x_i$

- Note that the weights, K_{i0}, will differ for each value of x_0.
- The selection of *s* is an important decision, and will determine how flexible the fit is (akin to KNN). We may specify *s* directly, or try to determine an appropriate *s* by cross-validation.
- It is possible to fit some predictors using a global coefficient, while fitting others with varying coefficients (i.e., local). This is a useful way to adapt a model for new data. Local regression also generalizes very naturally when we want to fit models that are local in a pair of variables (i.e., fit coefficients using a two-dimensional neighbourhood).
    - The details of either of these are not discussed.
- Local regression can perform poorly fitting to *p*-dimensional neighbourhoods if *p* is larger than ~3, becaues there will generally not be enough training observations close to x_0. This is a problem also encountered by KNN.

7.7. Generalized Additive Models

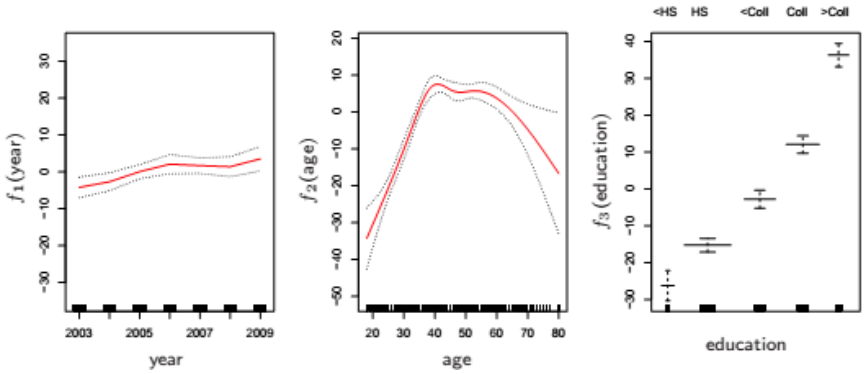
7.7.1. GAMs for regression problems

GAMs are a natural extension of multiple linear regression in order to allow for non-linear relationships between each feature and the response. A key feature of GAMs is that they retain *additivity*, which makes the model much more interpretable. However, as with all additive models, GAMs do not account for interations (though both low-order and complex interactions can be added if they are considered *a priori*).

In essence, GAMs replace the linear coefficient, β_{ij}, with a smooth non-linear function, $f_j(x_{ij})$. We can therefore write the GAM model as:

$$y_i = \beta_0 + f_1(x_{i1}) + \cdots + f_p(x_{ip}) + \epsilon_i$$

When it comes to GAMs, the coefficients are not of interest when interpreting the model; instead, we are interested in the plots. The figure below is an example of a GAM. Each predictor is fit by a function, and the individual terms are then added together to make a prediction.



This particular model was called using:

```
lm(wage ~ ns(year, df = 5) + ns(age, df = 5) + education, data = Wage)
```

This model was fit using natural splines, but GAMs fit with smoothing splines generally yield very similar results. GAMs can easily be fit with local regerssion, polynomial regression, or any of the approaches discussed above.

7.7.2. GAMs for classification

GAMs can also be used in classification settings. In this case, GAMs are an extension of the logistic regression model. Once again, we are able to replace the linear coefficients with a non-linear function. This yields the model:

$$\log\left(\frac{p(X)}{1-p(x)}\right) = \beta_0 + f_1(X_1) + \cdots + f_p(X_p) + \epsilon_i$$

- Logistic regression GAMs behave the same way as linear regression GAMs.
- A key difference is in their interpretation: Because logistic regression GAMs are additive in their *log odds*, but not in their probabilities.

7.7.3. Pros and cons of GAMs

- GAMs allow us to automatically fit non-linear relationships that a standard linear regression model will miss.
- Non-linear fits can potentially make more accurate predictions
- GAMs retain additivity, so we can still examine the effect of each variable, X_j on Y while holding all other variables fixed – therefore, GAMs are useful for inference.
- The smoothness of the function f_j can be summarized by the degrees of freedom.

- The main limitation of GAMs is that the model is restricted to additive – with many variables, interactions can be missed. We can add these interactions in manually, but they must be identified or known *a priori*.

8. Tree-Based Methods

8.1. Regression Decision Trees

The basis for decision trees is simple, and not discussed here. Instead I am writing about how to build decision trees. The process of building a regression tree can be broken down into two steps:

1. Divide the predictor space – that is, the set of possible values for $X_{1,\dots,}X_p$ into J distinct and non-overlapping regions, $R_1,\dots,R_J$.
2. For every observation that falls into the region R_J, we make the same prediction (i.e., the mean of the response values for the training observations in R_J).

- In theory, we could choose any shape to define R_J, but we use (*high dimensional*) rectangles/boxes for the simplicity and easy interpretation of these models. The goal of regression trees is to find 'boxes', $R_1,\dots,R_J$, that minimize the RSS, which is given by:

$$RSS = \sum_{j=1}^{J} \sum_{i \epsilon R_J} (y_i - \hat{y}_{R_J})^2$$

Where $\hat{y}_{R_J}$ is the mean response for the training objects falling inside R_J.

Unfortunately, this quantity is computationally infeasible to calculate (notice that it is a nested loop). Instead, decision trees take a *top-down*, *greedy* approach known as *recursive binary splitting*.

- *Top-down* refers to the way that the decision tree starts at the top node with all observations in a single region, and then splits them *down* the tree.
- *Greedy* refers to the fact that decision trees simply make the best decision *at each node* (i.e., greatest reduction in RSS) without considering future nodes.
    - Ex., given numbers 9, 4, and 1, a decision tree would make 12 using 9 + 1 + 1 + 1, rather than 4 * 3.

8.1.1. Tree Pruning

A limitation of decision trees is that they can easily overfit data, given enough nodes. it is possible to only consider splits that decrease RSS by a certain threshold, but you may miss out of future splits that decrease RSS by much more.

In general, it is best to *prune* a large decision tree in order to arrive at an optimal size. In theory, we might perform cross-validation in order to determine the best decision tree. However, estimating the test error rate for every possible sub-tree is extremely cumbersome.

Cost complexity pruning is used to select a smaller subset of trees for consideration. It is performed by minimizing:

$$\sum_{m=1}^{|T|} \sum_{i \epsilon R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

Where α is a non-negative tuning parameter,

$|T|$ is the number of terminal nodes of the tree *T*, and

R_m is the rectangle corresponding to the m^{th} terminal

For each value of α there is a corresponding subtree $T \subset T_0$ (i.e., the tree *T* which is a subset of the original tree T_0). We can select a value for α based on cross-validation. Then we can return to the full data set and obtain a subtree corresponding to α. This process is summarized below:

Algorithm for building a regression tree

1. Use recursive partitioning to grow a large tree on the training data. Stop when each terminal node has fewer than *w* observations.
2. Apply cost complexity pruning to the tree to obtain a sequence of best subtress as a function of α.
3. Use kfCV to choose α. For each fold:
    a. Repeat step 1 and 2 on the *K – 1 / K*th fraction of the training data (i.e., exclude the *k*th fold).
    b. Evaluate the MSE as a function of α.
4. Select α that minimizes the average MSE across all folds. Return the subtree from step 2 that corresponds to the chosen α.

8.1.2. Classification Trees

Classification trees behave the same way as regression trees; however, instead of making estimates using the mean response within R_J, it uses the most common class.

The process of building a tree is identical to a regression tree, except we can't use RSS as a metric of error. A natural alternative is the classification error rate, but it turns out that classification error is not sufficiently sensitive for tree-growing. Instead we use either the *Gini Index*, or *Cross-Entropy*.

The Gini index is defined by:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Where $\hat{p}_{mk}$ is the proportion of the most commonly occuring class in the m^{th} region that are from the k^{th} class.

- The Gini index is the total variance across *K* classes. The Gini index is used as a measure of node purity -- the Gini becomes minimized as $\hat{p}_{mk}$ approaches to 0 or 1.

A related (and very similar mathematically) alternative to the Gini index is cross-entropy, given by:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk})$$

- since $\hat{p}_{mk}$ is between 0 and 1, $-\hat{p}_{mk}\log(\hat{p}_{mk})$ must be non-negative. Like the Gini index, *D* approaches zero as $\hat{p}_{mk}$ approaches 0 or 1.

Node purity is preferable to classification error rate in *constructing* classification trees because it increases our confidence in the predicted outcome, even if it doesn't affect the overall error rate. For example, *G* and *D* may call for a split that results in same predicted outcome at both ends of the split. While this does nothing to improve the overall error rate, it provides additional information. Suppose that people with less than $1,000 in debt never default, while 5% of people with more than $1,000 in debt do default -- the predicted outcome of both is non-default, but the two different nodes are associated with different levels of certainty.

While cross entropy and Gini index are useful for evaluating the quality of a particular split, the classification rate is generally used to assess the accuracy of the final pruned tree. Any of the three methods may be used when pruning trees.

8.1.3. Pros and Cons of Decision Trees

- Easy to explain and interpret
- Can handle categorical variables with many levels without having to create dummy variables
    - However, factors with many levels might be over-weighted by decision trees (for example, a categorical variable with a unique label for each training observation would yield perfect training classification).

- Without modification, decision trees have low predictive accuracy compared to other approaches.

8.2. Bagging, Random Forests, and Boosting

8.2.1. Bagging

Unpruned decision trees have high variance and low bias. Bagging makes use of bootstrapping in order to reduce the amount of variance in decision trees. Bagging is a general procedure than can be applied to any method that has high variance, but it is particularly useful for decision trees.

The bagging procedure involves sampling data from the original training data (as in bootstrapping) and fitting a new decision tree, $\hat{f}^{*b}(x)$, on each of the *B* 'bagged' training sets. We then take take the average response of each of the individual trees:

$$\hat{f}_{bag} = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

- Bagged trees are not pruned, so each individual tree has highvariance

- Bagging can easily be applied to classification trees; instead of taking the average $\hat{y}_i^{*b}$, we take the most common 'vote' from each of the classification trees.
- The number of trees in bagging is not a critical parameter. Increasing the number of trees may not improve performance, but it does not lead to overfitting. In general, try ~1000 trees, and add more if test error decreases by adding more.

8.2.1.1. *Out of Bag* Error Estimation

There is a straightforward way to estimate the test error of a bagged model without cross-validation. Because bagging makes use of the bootstrap, it already leaves out some observations.

1. On average, about two thirds of the observations will be used in any individual tree - the remaining 1/3 are called *out of bag* (OOB) observations.
2. We can make predictions on the OOB observations, resulting in (on average) $B / 3$ predictions per observation.
3. The average predicted response for each observation is virtually equivalent to LOOCV.

OOB error estimation is a valid measure of test error, and is convenient for large data sets where cross-validation is computationally onerous.

8.2.1.2. Variable Importance in Bagged Models

Bagged models are difficut to interpret. It is still possible to obtain an overall summary of the importance of the predictors using the RSS / Gini index. We can record the amount that the RSS / Gini index decreases due to a given split, averaged overall trees. We can then represent the importance of a particular variable by the relative amount that it decreases the RSS. In general, this is represented graphically with the most important variable given a score of 100%, and all other variables represented as a fraction of that.

8.2.2. Random Forests

Bagged trees are still highly correlated, because strong predictors will likely end up at the top of every tree -- meaning that bagged trees still have reducible variance. Random forests provide another improvement over bagged trees by *decorrelating* the trees.

In addition to boostrapping observations, random forests also bootstrap *predictors*.

- We typically choose $m \approx \sqrt{p}$, though *m* is a critical parameter that needs to be optimized.
    - Kuhn 2013 suggests starting with 5 values ranging from 2 to *p*.
- Random forests where $m = p$ are equivalent to bagging.
- Like bagging, *B* is not a critical parameter - a large *B* will not result in overfitting.
- Random forests give rise to an ensemble of independent trees, which are robust to a noisy response. In contrast, random forests may underfit if the data are not noisy, since we are intentionally censoring data.

8.2.3. Boosting

Boosting is another general approach that can be used to improve regression/classification models. Boosting involves *sequentially* fitting decision trees using information from previously grown trees. Specifically, we fit the first tree to the response, y_i. The second tree is fit to the *residuals*, r_i, of the first tree (and so on). Each tree is incorporated to our overall model, but with a shrinkage parameter, λ -- the shrinkage parameter ensures that the boosted tree learns 'slowly.' Statistical learning approaches that learn slowly tend to perform those that learn 'quickly.'

Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set

Algorithm for boosting regression trees

1. Start with a null model, $\hat{f}(x) = 0$; set the first residual equal to the response: $r_i = y_i$ for all *i* in the training set.
2. For each tree, b = 1,..., B, repeat:
    a. Fit a tree $\hat{f}^b$ with d splits
    b. Update the original model, $\hat{f}$, by adding a shrunken version of the new tree:
    $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
    c. Update the residuals:
    $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$
3. Output the boosted model, which takes the final form of:
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

8.2.3.1. Tuning parameters in boosting

1. The number of trees, B. Unlike bagging and RFs, boosting can overfit if B is too large - though in tends to occur slowly. We use cross-validation to select B.
2. The shrinkage parameter, λ. This controls the how much information can be 'gained' in a single step. Typical values may be 0.01 or 0.001. A very small λ will require a large B.
3. The number of splits per tree, d. This controls the complexity of the tree -- often d of 1 or 2 is used. Note that a tree with a single split ('stump') involves only a single variable. Therefore, an ensemble of trees using d = 1 are fitting an additive model (i.e., there are no interations).

8.3. Decision trees and random forests in R

8.3.1. `rpart` to model data using decision trees

'rpart' stands for "Recursive Partitioning." Recursive partitioning is a tool that allows us to develop easy to visualize decision rules for predicting outcomes by Classification or Regression; this is often called the **C**lassification **A**nd **R**egression **T**ree (CART) method.

A useful guide on using `rpart` can be read here: Reference\rpart.pdf

Limitations

- Decision trees assign priority to variables that explain most of the variance; therefore, they may miss out on other explanatory variables.
    - A simple example is a decision tree which dispenses change: We must dispense 30c in a world where no 5c coins exist.
        - The tree would see that a 25c coin explains most of the variance, and thus perform:

            25 + 1 + 1 + 1 + 1 + 1
        - However, in reality a faster way to dispense this would be:

            10 + 10 + 10
- Again because of the way these trees assign priority, they are biased toward factors with a large number of levels. This is because they so many possible ways of splitting the data that they can do so in ways that perfectly explain the variance, but have no basis in reality.
- Decision trees are prone to over-fitting.

Generating a model

`rpart()`

`formula` - Formula object. Same as other models.

`data` - Data frame from which the formula variables are taken from.

`method`- Specific string. arguments can be used, but "`class`" and "`anova`" are used commonly to do classification and regression, respectively.

`control` - Optional parameters for controlling tree growth, fed into the `rpart.control()` function. Read the help page.

- Ex., `control = rpart.control(minsplit = 30, cp = 0.001)` requires that the minimum number of observations in a node be 30 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted.

`rpart.control()`

`minsplit` - Integer. Minimum number of observations that must exist in a node in order for a split to be attempted. Default = 20.

`minbucket` - Integer. Minimum number of observations that must exist in a terminal node. If only one of `minsplit` or `minbucket` is specified, `minbucket / 3 = minsplit` is automatically specified.

`cp` - "Complexity parameter." Numeric. Any split that does not decrease the overall lack of fit by a factor of `cp` will be deemed unimportant *a priori* and will not be computed. Default = 0.01

- Ex., for `anova` splitting, `cp` takes on a tangible value; each step must decrease R^2 by `cp`.

`maxcompete` - Integer. The number of competitor splits retained in the output. Instead of just informing the user which split was chosen, it may be helpful to know which variable came in second, third, etc. Default = 4.

`maxsurrogate` - Integer. The number of surrogate splits retained in the output. If set to 0, computation time will be ~1/2. Default = 5.

`usesurrogate` - Integer from 0:2. How to use surrogates in the splitting process.

- `0`: Display only; an observation with a missing primary split variable will terminate at that node. Corresponds to the action of `tree`.
- `1`: Use surrogates to split subjects if the primary variable is missing. If the surrogate is missing, the observation will terminate at that node.
- `2`: Use surrogates as in 1. If all surrogates are missing, then send the observation in the majority direction (Default). Corresponds to the recommendations of Breiman *et. al* (1984).

`xval` - Integer. Number of cross validations. Default = 10.

`surrogatestyle` - Integer from 0:1. Controls the selection of the best surrogate.

- `0`: Use the total number of correct classification for a potential surrogate variable (Default). This option more severely penalizes covariates with a large number of missing values.
- `1`: Use the 'percent correct' classifications over the non-missing values of the surrogate.

`maxdepth` - Integer. The maximum depth of the tree (the root counts as depth = 0). Default = 30.

Viewing results

`predict()` - Just like in base R, returns a vector of the predicted responses from a fitted `rpart` object.

`object` - `rpart` object. The model that you wish to extract the predictions from.

`newdata` - Data frame. The "prediction/test" data frame, which contains all of the variables included in the original `rpart` model.

`type` - String denoting the type of predicted value to be returned.

- "`prob`" returns a matrix of probabilities of each outcome of the original outcome variable.
- "`class`" returns a classification (i.e., binary decision).
- "`vector`" - returns a vector of the predicted response. For regression, this is the mean response at the node; for Poisson trees it is the estimated response rate. For classification trees it is the predicted class.

`plotcp()` - Plot cross-validation results

`post()` - Create a postscript plot of the decision tree.

`print()` - Print a text version of the tree. Allows overview in the event of overplotting.

`printcp()` - Display cp table

`prp()` - Plot the `rpart` object. Also allows pruning. See below under "Extracting results"

`fancyRpartPlot()` - Plots a nicer version of the decision tree. Requires the packages `rattle`, and `rpart.plot`

`rsq.part()` - Plot approximate R^2 and relative error for different splits. Labels are only appropriate for the "anova"

`summary()` - Provides a detailed report of the model, including surrogate splits

Pruning results

When selecting the number of splits to use in a tree, the convention is either:

(A) use the best tree possible (i.e., lowest cross-validation erorr)

(B) the simplest tree within one standard error of the best tree (i.e., the '1 - SE' method).

Both of these can be seen using the `printcp()` command. Method (A) provides the best fit, but Method (B) provides some protection against overfitting by using a simpler tree.

Ex.,

```
            CP nsplit rel error    xerror       xstd
1  0.4444444444      0 1.0000000 1.0000000 0.04244576
2  0.0307017544      1 0.5555556 0.5555556 0.03574957
3  0.0233918129      3 0.4941520 0.5233918 0.03497048
4  0.0204678363      4 0.4707602 0.5000000 0.03437157
5  0.0102339181      5 0.4502924 0.5233918 0.03497048
6  0.0065789474      8 0.4181287 0.4970760 0.03429471
7  0.0058479532     14 0.3771930 0.5116959 0.03467453
8  0.0043859649     16 0.3654971 0.5116959 0.03467453
9  0.0029239766     18 0.3567251 0.5146199 0.03474917
10 0.0005847953     20 0.3508772 0.4970760 0.03429471
11 0.0001000000     25 0.3479532 0.5058480 0.03452394
```

(A) Row 11 (i.e., 25 splits) provides the lowest cross-validation error ("`xerror`").

(B) The best tree has `xerror` and standard error ("`xstd`") equal to `0.5058480 + 0.03452394 = 0.54037194`. Therefore, we would select the simplest tree that has an `xerror < 0.54037194`, which is row 3 (i.e., 3 splits). We call `prune()` and specify `cp` to be the same as the `cp` value in row 3. Ex.,

```
prune(tree, cp = 0.0233918129)
```

`prp()` - Plot `rpart`. Allows pruning with the `snip` argument.

- `x` - rpart object.
- `snip` - Makes an interactive plot for an `rpart` object. Click on nodes to delete them. When you quit, the object will automatically update with those nodes removed, provided you specify `y <- prp(x)$obj`.

`prune()` - Prune the tree; uses similar commands to the `control` argument of `rpart()`.

8.3.2. `randomForest` to model data using decision trees

While `rpart` generates a single decision tree, `randomForest` generates multiple trees at random, i.e., a forest. More specifically, a democratic forest; each tree will vote on an outcome and the majority will "win." As such, random forests provide one method of overcoming issues of overfitting in CART modeling.

Random forests are 'random' because trees are grown employing two techniques which ensure that the trees grow differently.

1) The first is *bagging*, which involves taking a random sample (with replacement) of the *observations* to be used to train the model.
    - This prevents the overall model from being overfitted to exceptional observations.
    - At the same time, *Out of bag* observations are used to test accuracy of your data.
2) The second is *subsetting*, which involves taking a random sample of the *variables* to be used to train the model; the available variables is re-sampled at each node. In general, only the square-root of total number of variables is sampled at each node.
    - This prevents a single variable from dominating the first node of every decision tree.

Limitations

- `randomForest` doesn't know how to deal with NA values. As such, it cannot be applied if there are may missing values, and you will have to exclude or assign values (either by prediction, or arbitrarily) for all observations.
    - `rpart` has the advantage that it knows how to employ surrogate variables.

Generating a model

`randomforest()`

`x, formula`	- Formula specifying the response variable and predictor. `x` is a dataframe (to be used with `y`). If the reponse variable is a factor, classification is assumed. Otherwise, regression is assumed.
`y`	- `y` is the response variable, if `x` is used instead of `formula`. If a factor, classification is assumed. Otherwise, regression is assumed.
`data`	- Optional data frame containing the variables in `formula`.
`subset`	- Optional index vector specifying which rows should be used.
`na.action`	- Function to be applied if NA values are found.
`ntree`	- Number of trees to compute. Should only be limited for computational reasons. Default = 500
`importance`	- Logical. Should the importance of each predictor be assessed? Generally, this should be TRUE.

Viewing results

`predict()` - Just like in base R or `rpart`, returns a vector of the predicted responses from a fitted `randomForest`

`object`	- `rpart` object. The model that you wish to extract the predictions from.
`newdata`	- Data frame. The "prediction/test" data frame, which contains all of the variables included in the original `rpart` model.
`type`	- String denoting the type of predicted value to be returned.

- "`prob`" returns a matrix of probabilities of each outcome of the original outcome variable.
- "`response`" (also accepts "`class`") returns predicted values, whether continuous or binary.
- "`votes`" - the number of votes.

`varImpPlot()`

`x` - `randomForest` object.

`sort` - Logical. Should the variables be sorted by order of importance? Default = TRUE.

`n.var` - Integer. How many variables should be stored? Ignored if sort = FALSE.

8.3.3. `party` to model data using conditional inference trees

party is another recursive partitioning package, but its main feature is ctree(), which constructs conditional inference trees, taking into account distributional properties of the variables. Conditional inference trees overcome the main limitations of the CART method, namely overfitting and bias toward factors with many levels. Instead of a "pure" approach, conditional inference trees use statistical tests to determine the importance of each variable.

NB: "*Ensembles of conditional inference trees have not yet been extensively tested, so this routine is meant for the expert user only and its current state is rather experimental. However, there are some things available in cforest that can't be done with randomForest, for example fitting forests to censored response variables (see Hothorn et al., 2006a) or to multivariate and ordered responses."* -- cforest help page.

Roughly, the `ctree` algorithm works as follows:

1) Test the global null hypothesis of independence between any of the input variables and the response.
   - Stop if this hypothesis cannot be rejected.
   - ELSE select the input variable with strongest association to the response. This association is measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response.
2) Implement a binary split in the selected input variable.
3) Recursively repeate steps 1) and 2).

Generating a model

`ctree()` - Make a single tree.

`formula` - formula object. Note that '`:`' and '`-`' will not work.

`data` - Optional data frame containing the variables in `formula`.

`subset` - Optional index vector specifying which rows should be used.

`weights` - Optional vector of weights to be used in the fitting process (*I don't know what this does*).

`controls` - `TreeControl` object. Can be achieved by feeding into the `ctree_control()` function. See below.

`scores` - Optional named list of scores to be attached to ordered factors. By default, the scores are `1:length(x)`. This may be changed to, for example: `list(x = c(1, 5, 6)`.

`ctree_control()`

`teststat` - Character specifying the test statistic to be applied.

`"quad"` `"max"`

`testtype` - Character specifying how to compute the distribution of the test statistic.
`"Bonferroni"` `"MonteCarlo"` `"Univariate"` `"Teststatistic"`

`mincriterion` - The cut-off alpha level that must be exceed to implement a split. Takes on the form of `1 - p.` Default = 0.95.

`minsplit` - Integer. Minimum number of observations that must exist in a node in order for a split to be attempted. Default = 20.

`minbucket` - Integer. Minimum number of observations that must exist in a terminal node. Default = 7.

`stump` - Logical. Should only stumps (a tree with only 3 nodes) be computed? Default = FALSE.

`nresample` - Integer. Number of Monte-Carlo replications to use when the distribution of the test statistic is simulated. Default = 9999

`maxsurrogate` - Integer. The number of surrogate splits to investigate. Note that only surrogate splits for ordered covariables have been implemented (version 1.1-2). Default = 0.

`mtry` - Integer. Number of input variables to be randomly sampled as candidates at each node for random forest-like algorithms. Default = 0 (indicating no random selection)

`savesplitstats` - Logical. Should two-sample statistics be saved for each node? Default = TRUE

`maxdepth` - Integer. Maximum depth of the tree. Default = 0 (indicating no limit)

`remove_weights` - Logical. Should weights attached to nodes be removed after fitting? Default = FALSE

`cforest()` - Make a forest. Inputs are identical to `ctree()` with the exception of `controls` and `newdata`.

`controls` - `TreeControl` object. Can be achieved by feeding into the `cforest_control()` function. Some notable parameters that should be considered:

- `mtry = 5` Change depending the number of predictor variables (usually $\sqrt{n}$)
- `ntree = 500` Increase if you have more variables.
- `mincriterion = 0.95` Regulates tree depth. To grow large trees, decrease this.

`cforest_control()` - Similar to `ctree_control()` with a few additional arguments and different defaults.

`teststat` - Character specifying the test statistic to be applied. Default = `"quad"`

`testtype` - Character specifying how to compute the distribution of the test statistic.
Default = `"Teststatistic"`

`mincriterion` - The cut-off alpha level that must be exceed to implement a split. Takes on the form of `1 - p.` Default = 0.95.

`mtry` - Integer. Number of input variables to be randomly sampled as candidates at each node for random forest-like algorithms. Default = 5.

`savesplitstats` - Logical. Should two-sample statistics be saved for each node? Default = TRUE.

`ntree` - Integer. Number of trees to grow. Default = 500.

`replace` - Logical. Should observations be sampled with or without replacement? Default = TRUE.

`fraction` - Numeric. What fraction of observations should be drawn if replace = FALSE? Default = 0.632.

`trace` - Logical. Should a progress bar be printing while the forest grows?

Viewing results

`predict()` - Uses `predict()` from base R.

`newdata` - Data frame to test the predictions. If NULL, will generate predictions on the original data.

9. Other considerations when building a model

9.1. Class Imbalance in Classification

9.2. Categorical Predictors

9.2.1. Nominal predictors

To use categorical data as a predictor, the data must be converted to quantitative data. If there are only two levels, it is easy to simply assign one as 1 (and therefore the other as 0) and then treat it as a continous variable. If there are more than two levels the process becomes more complicated.

Dummy coding

- Used when there is a control or comparison group in mind.
- The variable is broken into $n_{categories}$ - *1* variables. The reference group is assigned a value of 0 for each code variable. The comparison group at each variable is assigned a value of 1. All other groups are assigned 0 for that variable.
- The b_1 values should be interpreted such that the experimental group is being compared against the control.
- *I think this is what R does with multi-level factors.*
- Limitations:
    - Relies on some *a priori* hypothesis about which group will have the highest score on each variable. As such, it is not suitable for exploratory analyses or when there are many variables to test.

- The comparator group should be well defined and not some "Other" category.
- The comparator group should not be small compared to the other groups.

Effects coding

- Similar to dummy coding, except one group is compared to all other groups. Unlike dummy coding, there is no control group. Instead, the data are compared against the mean of all groups (i.e., grand mean).
- May be weighted by calculating a weighted grand mean, which accounts for sample size in each variable.
  - Weighted analysis should be used when the sample is representative of the population.
  - Unweighted analysis should be used when differences in sample size arise by chance *(although this seems backward*)

9.2.2. Ordinal predictors

Ordinal variables are common predictors in regression analysis but they are pesky. Some common strategies for including ordinal predictors are:

1) Encode the ordinal variable as a nominal categorical variable.
   - This results in the loss of "rank" information and introduces many dummy variables.
2) Encode the ordinal variable as a continuous variable by rank.
   - This artificially imposes a linear relationship.
3) Encode the ordinal variable as a continuous variable using the mid-point of each level.
   - This only works if a truly continous variable is being cut into categories (i.e., income)
   - This method is challenged by variables with open lower- or upper- bounds, which have no mid-point.

Penalized regression with ordinal predictors

Gertheiss & Tutz (2008) propose a technique for incorporating ordinal variables using co-efficient smoothing. Although the math is beyond my level, the basic concept is as follows: The variables are dummy coded; instead of simply estimating the (slope) parameters by maximum likelihood methods, they instead penalize the differences between co-efficients of adjacent categories in the estimation procedures. This method assumes that the response variable *y* will change slowly when moving between two adjacent categories of the independent variable - this method therefore avoids large jumps between adjacent categories in favour of a smoother co-efficient vector.

This method is used in the package `ordPens`.

9.3. Missing Data

9.3.1. Types of missing data

Missing completely at random (MCAR)

- As it implies, values are missing completely at random. I.e., missingnes is not related to any property of the observation, any property of the variable, the value of other variables, or the missingness of other variables. That is to say, missingness is due only to noise.
    - This almost never happens.
- If you drop these cases you will not introduce bias, but the efficiency of your estimate will be lower (likely by increasing variance from lost data). You can model these variables using a random model to improve efficiency.

Missing at random (MAR)

- The occurence of missing values is random but associated with other variables that have been measured. That is to say, missingness can be modeled.
    - In a way, MAR is a conditional form of MCAR. Once MAR has been modeled, it is essentially MCAR in that only noise remains.
- If you drop these observations you will introduce bias (selection bias) and reduce efficiency. However, if you model this you can eliminate bias and improve efficiency.

Missing not at random (NMAR or MNAR)

- The occurence of missing values is associated with variables that are unknown or unmeasured.
- You can't do anything about this. Unfortunately, it is not even possible to distinguish NMAR from MAR *post hoc* because the data required to test this are missing - by definition.
    - You can attempt to re-gather data on missing entries and test if these entries differ significantly from non-missing entries (ex., following up with survey non-respondents).

9.3.2. Ways of dealing with missing data

This section deals primarily with how to handle missingness for data MAR. MCAR is dealt with essentially the same way (but simpler), and NMAR data cannot be modeled. Long-story short: multiple imputation is generally the best way to estimate missing values to improve efficiency and reduce bias.

1. Listwise deletion: drop all observations with any missing data
    - Default in most statistical packages, including R
    - Results in maximum loss of information

2. Mean/median/multivariate imputation: replace missing data with the observed mean/median (or draws from the multivatiate distribution of the variable)
    - Understates the variability in the imputed variable, leading to an underestimation of the variance and standard errors
    - Makes no (or a limited) attempt to recover associations between the variables
    - Does not introduce bias and improves efficiency (but as stated above, it may underestimate standard errors)

3. Regression-based imputation: predicts missing values based on linear/logistic regression for all other available variables.
    - Not a bad method
    - Note: it is okay to use the dependent variable here to predict the missing value.
    - Understates the variability of the imputed variable, as the quality of the estimate for the missing value is not included when using it to estimate the dependent variable.
        - I.e., the imputed value is an uncertain estimate, but this uncertainty is not used when making further estimates in future models.
    - This will improve efficiency but will likely lead to an underestimation of standard errors in future models.

4. Interpolation of panel data: for time-series/panel data, replaces missing observations using either the last observation or the mean of the other panels.
    - Not terrible - frequently used in clinical trials for drop-outs/missed follow-ups
    - There is some evidence that this technique introduces bias and overconfidence of estimates
        - There is no way to know *a priori* if this will improve or reduce the quality of your estimate
    - Only works for panel data

5. KNN: Impute the missing entry using one (or an average) of the values from the k-nearest neighbours.

6. Bagged Trees: Uses bagging + decision trees to predict the value of the predictor instead of the response.

7. Multiple imputation: See below.

9.3.3. Multiple Imputation

Multiple imputation uses multiple models (ex., regression) to predict missing observations from non-missing observations. Each model has slight variations, so the variation in the predictions generated by the different models provides a measure of uncertainty in our estimate.

A general work flow for a regression model is as follows:

1. The variable *x* can be given as a function of noise and other variables: $x \sim \alpha_0 + \alpha_1 z + \alpha_2 y$. Each α follows a distribution (recall: for a linear regression model to apply this should be a normal distribution)
    - Note in this context,the α_1 is similar to β_1 if we were predicting *y*, but we're predicting *x*.
2. We draw *m* number of each of α_0, α_1, and α_2 (from the data) for each missing entry and create *m* number of models with these draws to predict *m* number of values of *x*.
    - This produces *m* number of data sets that differ in their predicted values of *x* (similar to the bootstrap).
3. We plug each of the *m* predictions to create *m* estimates of β_0, β_1, and β_2 for *y*.
4. Finally, we pool each estimate of β_0, β_1, and β_2, which is done simply by taking the mean and variance across all the estimates.

$$Var_{\beta} = W + \left(1 + \frac{1}{m}\right) B$$

W is an estimate of the within-imputation variation (i.e., the mean of the within-imputation variances).

B is an estimate of the between-imputation variation (i.e., the mean of the variance between different estimates of β.

$$W = \sum_{m=1}^{M} \frac{s_m^2}{m} \quad ; \quad B = \sum_{m=1}^{M} \frac{(\hat{\beta}_m - \hat{\beta})^2}{m-1}$$

The $\left(\mathbf{1} + \frac{\mathbf{1}}{\boldsymbol{m}}\right) \boldsymbol{B}$ term is the inflation in the standard errors of $\hat{\beta}$ that we use to correct for imputation (i.e., the additional noise that we introduce by imputing estimates for *x*). From this term we see that increasing the number of models (*m*) increases the confidence of our estimate by decreasing the amount of noise that we introduce.

In practice, you'll never have to do this by hand. There are several statistical packages that will do the computations for you.

9.3.4. MICE: Multiple imputation through chained equations

MICE uses an iterative bootstrapping approach to create multiple imputations.

MICE algorithm:

1. Discard all observations for which all variables are missing.
2. For all missing entries, fill in the missing data with random draws from the observed values.
3. Move through the columns of variables and perform single-variable imputation using the method of choice, replacing the original (quasi-random) replacements with the fitted replacements.
    - The first variable will be imputed using the quasi-random entries drawn in step 2.
    - As you progress through the variables, missing entries will be imputed using the imputed values of the preceding variables. This improves the estimate as the algorithm iterates.
4. Repeat step 3 for *o* number of iterations.
5. Repeat steps 1-4 *m* number of times to create *m* number of imputed data sets.
6. Make estimates using each of the *m* data sets. Pool these estimates together (as described above).

The critical steps for MICE are 3 and 4.

- There are many ways to impute the variables:
    - Regression (linear, logistic, or multinomial)
        - Pick the maximum likelihood $\hat{y}$ OR sample from a distribution of *y*'s.
    - Predictive mean matching (PMM) - the default for MICE for continuous variables in R.
        - Creates a predicted value for missing entries from a regression model. It then picks *q* number of *real entries* that have the closest predicted value (i.e., closest Euclidean distance) and randomly chooses one of those impute.
        - I.e., the missing entry takes on the value of a real entry that is close to value predicted by regression, but there is an element of randomness involved.

- You must choose how many data sets (*m*) to use (5 a good minimum) and how many iterations (*o*) of imputation should be performed for each *m*. For PMM, you must choose how many neighbours (*q*) the prediction should be drawn from (default = 3).

9.3.4.1. MICE and validation

Note: In order for results of multiple imputation to be generalizable[1]:

- Use supervised multiple imputation *inside* cross-validation or OOB bootstrap validation
    - Increasing the number of bootstraps tends to improve accuracy of estimates more than the iterations of imputation.
- OR Use *unsupervised* multiple imputation *outside* of cross-validation/OOB.
    - This is less robust (generally more pessimistic) than the first method, but computationally much easier.

- Using *supervised* imputaiton outside of CV/OOB yields optimistic bias.

9.3.4.2. Multiple imputation on data that are NMAR

Contrary to popular belief, `mice` is not restricted to data that are MAR. While it is true that imputation techniques commonly assume MAR, the theory of multiple imputation is generalizeable to NMAR. It is true that, when data are NMAR, the model fitted to complete cases will be incorrect for the incomplete cases; however, multiple imputation methods are still better than other alternatives[2].

9.3.5. `mice()` for multiple imputation through chained equations

The basic function for imputing missing values is `mice()`. Other functions are included to assess the effect/appropriateness of the imputation.

9.3.5.1. Imputing data

`mice()` - Basic function for imputing missing values. Creates a new data frame containing `m` number of data sets.

- `data` - The dataframe. Contains the missing entries and all variables that will be used to estimation and prediction.
- `method` - Character vector. Method to use for imputatoin. "`pmm`", "`logreg`", "`polyreg`", "`cart`"
    - `mice()` supports many other methods. See help file.
- `m` - Numeric. Number of data sets to create. Default = 5
- `maxit` - Numeric. Maximum number of iterations of imputation for each `m`. Default = 5

Usage notes:

- Generally when using this function, you'll want to store it in a new object.
- For reproducibility, set seed prior to imputation.
- This function is computationally demanding, based on `m` and `maxit`.

9.3.5.2. Visual diagnostics for imputations

```
pwplot(mice.data)

densityplot(mice.data)

xyplot(mice.data, x + y + z ~ .imp)
```

9.3.5.3. Creating and pooling models

* In general, you will want to create a model using your pooled MICE data and a model without imputation (i.e., list-wise deletion) so that you can compare them.

`with()` - Use the way you normally use the `with()` function on your MICE data frame; this allows you to create *m* number of models with a single `lm()` command and store them in a single object.

`lm()` - Model-fitting function (could be `glm()` or some other). Use the way you normally would. Combine with `with()` to create *m* number of models with a single command and store them in a single object.

`pool()` - Pool together the different models from a MICE model object into

9.3.5.4. Dealing with data that are not missing at random

The post argument of mice() allows you to adjust your argument.

1. Wahl S, Boulesteix A-L, Zierer A, Thorand B, van de Wiel MA. Assessment of predictive performance in incomplete data by combining internal validation and multiple imputation. *BMC Med Res Methodol*. 2016;16(1):144. doi:10.1186/s12874-016-0239-7.
2. Graham JW. Missing Data Analysis: Making It Work in the Real World. *Annu Rev Psychol*. 2009;60:549-576. doi:10.1146/annurev.psych.58.110405.085530.