4. Logistic Regression

4.1 Overview

Despite its name, when we use logistic regression, we are performing **classification**, not regression. Whereas in linear regression, our target variable was a *quantitative* variable, in logistic regression, our target variable is *qualitative*: we want to know what class an observation is in. In the most common classification scenario, the target variable is a binary output so that we classify into one class or the other. As we will see later in the chapter, there are techniques that allow classification into more than two classes.

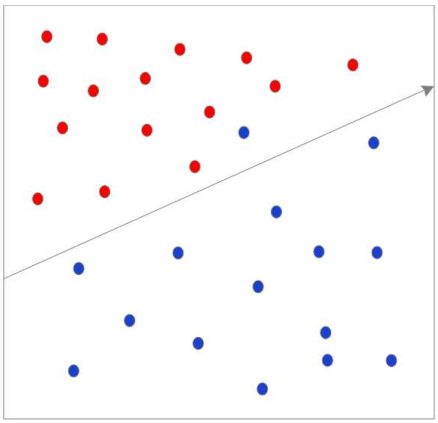

Figure 4.1: Decision Boundary for Binary Classification

Linear models for classification create decision boundaries to separate the observations into regions in which most observations are of the same classs. The decision boundary is a linear combination of the X parameters. As we see in Figure 4.1 the two classes are almost perfectly separated by the decision boundary. There is one misclassified observation.

4.2 Logistic Regression in R

Let's take a look at the plasma data set in package HSAUR. This is a blood screening data set for 32 patients that gives measures of two plasma proteins, fibrinogen and globulin, and a binary indicator associated with the two protein levels. The fibrinogen and globulin variables are quantitative. The qualitative variable ESR>20 indicates whether the erythrocyte sedimentation rate, the rate at which red blood cells settle in blood plasma, is over 20 or not. In logistic regression our target needs to be a qualitative variable. In this data set, ESR>20 is our target. The ESR>20 variable has been coded as a factor in R so that ESR>20 = 2 means that ESR is over 20 and 1 means otherwise. We want to learn to predict whether ESR>20 or not, based on the levels of the plasma proteins fibrinogen and globulin. Values >20 indicate some possible associations with various health conditions.

4.2.1 Plotting factor data

Code 4.2.1 shows how the plots in Figure 4.2 were generated. First, we specify a 1x2 layout for the plots, then use the plot(x, y) command. The parameter `varwidth=TRUE` makes the boxplot widths proportional to the square root of the samples sizes. This easily lets us see that ESR<20 is more common than ESR>20. More importantly, the box plots show that ESR>20 observations are associated with slightly higher levels of globulin and significantly higher levels of fibronogen.

Code 4.2.1 — Plotting Factors. Boxplots.

```
par(mfrow=c(1,2))
plot(ESR, fibrinogen, data=plasma, main="Fibrinogen", varwidth=TRUE)
plot(ESR, globulin, data=plasma, main="Globulin", varwidth=TRUE)
```

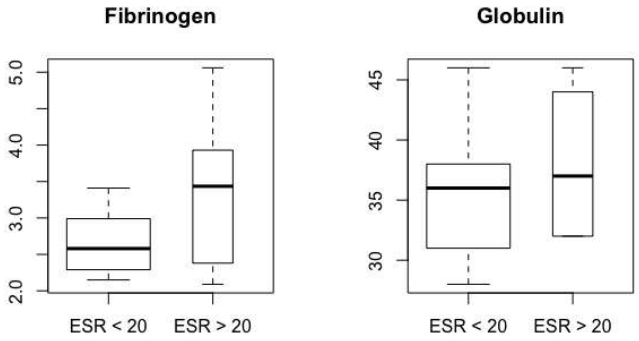

Figure 4.2: Box Plots

Caution 4.1 — Plotting Qualitative Data. If you use the R plot() function with command:

```
plot(globulin, ESR, data=plasma)
```

you will see a row of points at y=1 and another row of points at y=0 on the y axis. This should be your first clue that you need to rethink your plot. The plot function expects the data in (x, y) order. In Code 4.2.1 above we have globulin as x and ESR as y so we get the box plot as shown in figure 4.2. ■

In Code 4.2.2 we make two conditional density (CD) plots, shown in Figure 4.3. We can make the same observations as we did when looking at the box plots. Here they are just visualized differently. The total probability space is the rectangle, with the lighter grey indicating ESR>20.

Code 4.2.2 — Plotting Factors. CD Plots.

```
par(mfrow=c(1,2))
cdplot(ESR~fibrinogen)
cdplot(ESR~globulin)
```

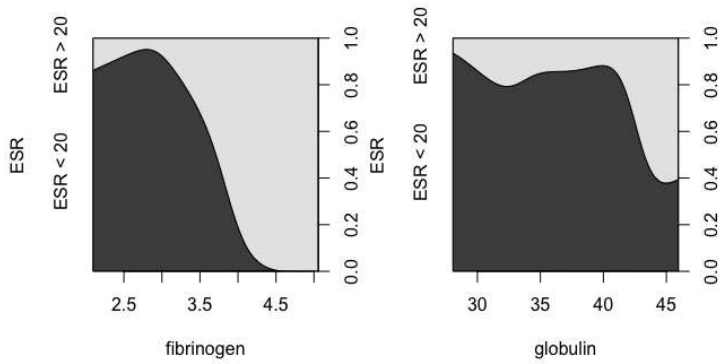

Figure 4.3: Conditional Density Plots

4.2.2 Train and Test on the Plasma Data

Even though this data set is very small, only 32 observations, we divided it into train and test sets, then created a logistic regression model using fibrinogen to predict ESR>20. The full notebook is available online.

Code 4.2.3 — Logistic Regression. Using glm().

```
set.seed(1234)
i <- sample(1:nrow(plasma), 0.75*nrow(plasma), replace=FALSE)
train <- plasma[i,]
test <- plasma[-i,]
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
summary(glm1)
```

For logistic regression we use the glm() *generalized* linear function instead of lm() that we used for linear regression. Also we need the parameter `family=binomial` for logistic regression. Otherwise the glm() function call looks similar to what we have done previously for lm(). The first argument is the formula, which seeks to learn the target ESR with one predictor, fibrinogen.

You will notice that the output of the summary() function is very similar to the output we saw for linear regression, with 4 sections:

- the glm() call
- the residual distribution
- the coefficients with statistical significance metrics
- metrics for the model

Here is the summary output:

```
Call:
glm(formula = ESR ~ fibrinogen, family = binomial, data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.0112  -0.4648  -0.3517  -0.2692   2.6543

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -8.383      3.627  -2.311   0.0208 *
fibrinogen     2.340      1.151   2.033   0.0421 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 24.564  on 23  degrees of freedom
Residual deviance: 17.107  on 22  degrees of freedom
AIC: 21.107

Number of Fisher Scoring iterations: 5
```

4.2.3 Interpreting summary() in Logistical Regression

For the logistic regression output, it is important to note that the residuals are *deviance residuals*. What does that mean? The deviance residual is a mathematical transformation of the loss function (discussed in a later section) and quantifies a given point's contribution to the overall likelihood. These deviance residuals can then be used to form RSS-like statistics.

At the bottom of the output, the statistics section is quite different. We have null deviance and residual deviance for metrics. The null deviance measures the lack of fit of the model, considering only the intercept. The residual deviance measures the lack of fit of the entire model. The AIC is most useful in comparing models. AIC stands for Akaike Information Criterion and is based on deviance. AIC penalizes more complex models. The Fisher scoring algorithm is a modified form of Newton's method of solving a maximum likelihood problem.

The most significant difference is the interpretation of the coefficient estimates. Whereas the coefficient of a linear regression predictor quantifies the difference in the target variable as the predictor changes. This is not true for logistic regression. Instead of quantifying the difference in the target variable, it quantifies the difference in the log odds of the target variable. We will discuss odds, log odds, and probability in the Metrics section.

4.2.4 Evaluation on the Test Data

Next, we look at the output of predict() for logistic regression created in Code 4.2.4. Notice the parameter `type="response"`. This is important to get probabilities out of the model. The model outputs log-odds but by requesting "response" we get these numbers converted to probabilities for us. The probabilities for the first few test observations are:

```
         7          8          9         10         11         18 . . .
0.29967653 0.03871679 0.26646707 0.09116524 0.04631732 0.10790621 . . .
```

The ifelse() function is needed to convert these probabilities to 1 or 2, the internal coding for our target variable. Once we have these predictions in variable pred, we can compare them to the actual values in the test observations to get accuracy, the percentage of observations that were classified correctly. We also output a table of predictions and actual values. All the predictions were of class 1 and all but 7 of the 8 actual values were of class 1.

Code 4.2.4 — Logistic Regression. Evaluating the output.

```
probs <- predict(glm1, newdata=test, type="response")
pred <- ifelse(probs>0.5, 2, 1)
acc1 <- mean(pred==as.integer(test$ESR))
print(paste("glm1 accuracy = ", acc1))
table(pred, as.integer(test$ESR))
```

```
[1] "glm1 accuracy =  0.875"

pred 1 2
   1 7 1
```

4.3 Learning (or Not) From Data

Note one odd thing about the table above: the model always predicted class 1 (ESR<20). Even though 87.5% accuracy sounds good, we cannot be impressed. In our test sample, 7 were of one class and only 1 of the other, therefore if the model just guesses the majority class, as it did in this case, it will be right 87.5% of the time.

There are two problems here: (1) a small amount of data, (2) an unbalanced data set.

4.3.1 Not Enough Data

The first problem is that we have a very small data set. In order to create a stronger model, much more data would have to be collected. Data collection can be expensive and require domain expert assistance. In the case of this data set, more blood samples would have to be drawn from a random population, analyzed by technicians, and supervised by a hematologist or other clinician.

When additional data cannot be obtained, another but less preferable option is to use sampling techniques. Consider our example above, with 8 test cases randomly sampled from the data. What if we randomly sampled the data multiple times? Of course we would have to remove the set.seed() function. By randomly sampling the data multiple times, each time we would have a different test set. We could average our test accuracy over all these samples and get a better idea of the accuracy of our model. One such sampling technique is called cross-validation which we will explore in a later chapter.

4.3.2 Unbalanced Data

The second problem with the data set is that it is unbalanced. Of the 32 observations, only 6 have ESR>20. This is a 81% ESR<20 to 19% ESR>20 split. Unbalanced data sets also pose problems for classification. Again, more data would be helpful. If that additional data is still unbalanced, sampling techniques may be of help. The idea is to oversample from the minority class and undersample from the majority class to come up with a data set that is more balanced. We will explore techniques for doing this in future chapters. For now, we will take our model as created, but take it with a grain of salt.

4.4 Metrics

Classification can be evaluated by many measures. In this section we will look at accuracy, sensitivity and specificity, Kappa, AUC, and ROC curves.

4.4.1 Accuracy, sensitivity and specificity

The most common metric to evaluate results in classification is accuracy:

$$acc = \frac{C}{N} \tag{4.1}$$

where C is the number of correct predictions, and N is the total number of test observations. The output of the table() command above shows more details than simple accuracy. It is also called a confusion matrix. The diagonal upper left to lower right are the correctly classified instances. In that case it is a diagonal of 1: 7 observations. The flip side of accuracy is the error rate, calculated by subtracting accuracy from 1. We can break down each component of the confusion matrix as follows:

- TP - true positive: these items are true and were classified as true
- FP - false positive: these items are false and were classified as true
- TN - true negative: these items are false and were classified as false
- FN - false negative: these items are true and were classified as false

$$accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{4.2}$$

$$error\ \ rate = \frac{FP+FN}{TP+TN+FP+FN} = 1-accuracy \tag{4.3}$$

The **sensitivity** measures the true positive rate:

$$sensitivity = \frac{TP}{TP+FN} \tag{4.4}$$

The **specificity** measures the true negative rate:

$$specificity = \frac{TN}{TN+FP} \tag{4.5}$$

Sensitivity and specificity range from 0 to 1, just as accuracy does, with values closer to 1 being better. They help to quantify the extent to which a given class was misclassified.

4.4.2 Kappa

Cohen's Kappa is a statistic that attempts to adjust accuracy by accounting for the possibility of a correct prediction by chance alone. Kappa is often used to quantify agreement between two annotators of data. Here we are quantifying agreement between our predictions and the actual values. Kappa is computed as follows:

$$\kappa = \frac{Pr(a)-Pr(e)}{1-Pr(e)} \tag{4.6}$$

where Pr(a) is the actual agreement and Pr(e) is the expected agreement based on the distribution of the classes. The following interpretation of Kappa is often used but there is not universal agreement on this. Kappa scores:

- <2.0 poor agreement
- 0.2 to 0.4 fair agreement
- 0.4 to 0.6 moderate agreement
- 0.6 to 0.8 good agreement
- 0.8 to 1.0 very good agreement

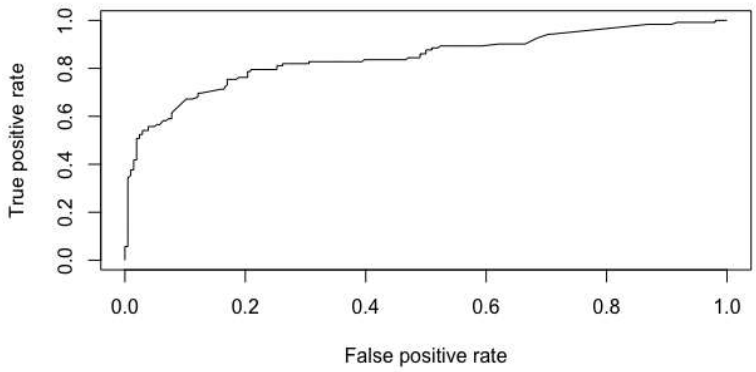

Figure 4.4: ROC Curve

4.4.3 ROC Curves and AUC

The ROC curve is a visualization of the performance of machine learning algorithms. The name ROC stands for Receiver Operating Characteristics which reflects its origin in communication technology in WWII in detecting between true signals and false alarms. The ROC curve shows the tradeoff between prediction true positives while avoiding false positives.

Figure 4.4 shows an ROC curve. The y axis is the true positive rate while the x axis is the false positive rate. The predictions are first sorted according to the estimation probability of the positive class. A perfect classifier would shoot straight up from the origin since it classified all correctly. We want to see the classifier shoot up and leave little space at the top left. Starting at the orgin, each predictions impact on the curve is vertical for correct predictions and horizontal for incorrect ones. If we see a diagonal line from the lower left to the upper right, then our classifier had no predictive value.

A related metric is AUC, the area under the curve. AUC values range from 0.5 for a classifier with no predictive value to 1.0 for a perfect classifier.

4.4.4 Probability, odds, and log odds

What is the difference between odds and probability? Let's look at this using a sports outcome example. Imagine we played 10 games with a friend and won 7. That means we lost 3 of course. Assuming we will do as well next time, our odds are 7 to 3:

$$odds = \frac{number\ \ of\ \ wins}{number\ \ of\ \ losses} = \frac{7}{3} \tag{4.7}$$

If we want to express the same data as a probability:

$$probability = \frac{number\ \ of\ \ wins}{number\ \ of\ \ games} = \frac{7}{10} \tag{4.8}$$

Notice that probability will always range from 0 to 1, whereas odds will not. Recall that the glm() algorithm coefficients represent a change in log odds. Just as it sounds, log odds are the log of the odds: log(odds). So to find the odds, we use the inverse of log, the exp() function. And to convert odds to probability, we use the following:

$$probability = \frac{odds}{1 + odds} \tag{4.9}$$

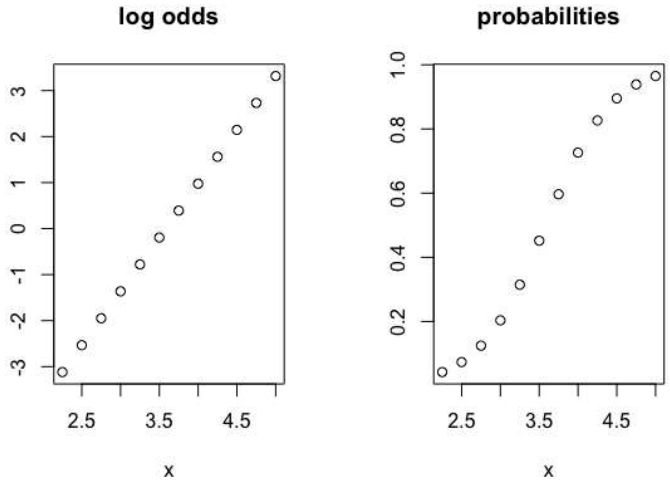

Figure 4.5: Log Odds versus Probability

Let's see what this means in terms of the plasma data logistic regression model above. The predictor is fibrinogen, which ranges from around 2.09 to 5.06 in this data set. Let's compare the effect of fibrinogen across values from 2.25 to 5.0. Figure 4.5 plots the log odds across this range of values on the left, and on the right, the associated probabilities across the same range of values. Observe that the results of the logistic regression model (the log odd) are linear in the parameters (w, b) but that the associated probabilities are not linear.

The coefficient for fibrinogen in the linear regression model was 2.34. For every one-unit increase in x, the probability of ESR>20 changes by exp(2.34)/[1+exp(2.34)]. Let's look at some sample values in Table 4.1. The X column is fibrinogen for a range of values. The log odds is found by plugging in the value of x for the logistic regression formula given by: 2.34 * x - 8.383.

X	Log Odds	Probability
2.5	-2.53	0.07
3.0	-1.36	0.20
3.5	-0.19	0.45
4.0	0.977	0.73
4.5	2.147	0.89

Table 4.1: Log Odds and Probability for Plasma Data

As you see in Table 4.1 and more clearly in Figure 4.5, the log odds are linear in the parameters but the probability is not linear, we can discern a subtle S-shape in the probabilities.

Caution 4.2 — Logistic Regression Coefficients. In linear regression we interpret a predictor coefficient as the amount of change in y for a 1-unit change in x. We cannot make this intepretation in logistic regression. The predictor coefficient in a logistic regression model specifies the change in log-odds for a 1-unit change in x.

4.5 The Algorithm

The linear regression output was a quantitative value that could range over all the real numbers. What we need for logistic regression classification is a function that will output probabilities in the range [0, 1]. The sigmoid, or logistic, function is used for this purpose and of course is where the algorithm gets its name. When real numbers are input to the logistic function, the output is squashed into the range [0,1] as seen in Figure 4.6. The logistic function is:

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.10}$$

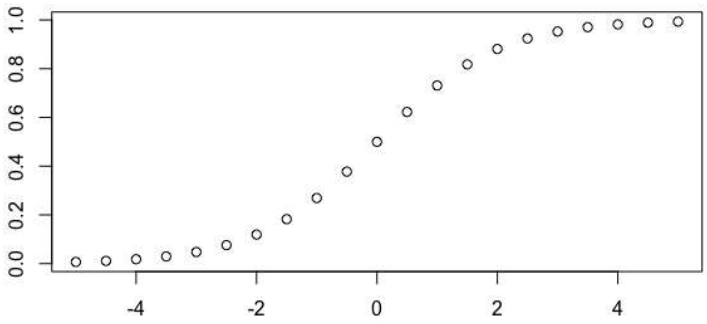

Figure 4.6: The Logistic Function

The logistic regression algorithm computes the log odds from the estimated parameters. The log odds is just log(odds). The odds are the probability of the positive class, p(x), over the probability of the negative class (1 - p(x)). If we have a single predictor w_1 and an intercept w_0, the log odds are:

$$log\frac{p(x)}{1-p(x)} = w_o + w_1 x \tag{4.11}$$

Solving for p gives us the logistic function:

$$p(x) = \frac{e^{-(w_0+w_1 x)}}{1+e^{-(w_0+w_1 x)}} = \frac{1}{1+e^{-(w_0+w_1 x)}} \tag{4.12}$$

We can see in Equation 4.11 why logistic regression is considered a linear classifier. It creates a linear boundary between classes in which the distance from the boundary determines the probability. When we use the logistic function for classification, the cut-off point is usually 0.5. Notice in Figure 4.6 that this is the inflection point of the S-curve. Probabilities greater than 0.5 are classified as the positive class and probabilities less than 0.5 are classified in the other class.

4.6 Mathematical Foundations

How are the parameters, **w** found for logistic regression? First an appropriate loss function is established, then an optimization technique such as gradient descent is used to find optimal parameters.

Recall the loss function for linear regression:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{4.13}$$

If we plug in the logistic function for f(x), it will not be a convex function. That is a problem because gradient descent works only for convex functions. A suitable loss function for logistic regression can be found by starting with the likelihood function, L:

$$L(w_0, w_1) = \prod_{i=1}^{n} f(x_i)^{y_i}\ (1 - f(x_i))^{1-y_i} \tag{4.14}$$

Notice in the likelihood equation that one of the terms in the product will always reduce to 1 because the y values are either 0 or 1. To simplify the likelihood equation we will take the log of it to find the log-likelihood, ℓ:

$$\ell = \sum_{i=1}^{n} y_i\ log f(x_i) + (1 - y_i)\ log(1 - f(x_i)) \tag{4.15}$$

The log-likelihood equation for each instance:

$$\ell = y\ log f(x) + (1 - y)\ log(1 - f(x)) \tag{4.16}$$

In training the classifier, we want to penalize it for wrong classifications. This is our loss function. The penalties follow directly from Equation 4.16. Our Loss is:

$$\mathcal{L} = -log(f(x))\ if\ y = 1 \qquad \mathcal{L} = -log(1 - f(x))\ if\ y = 0 \tag{4.17}$$

These two loss functions are visualized in Figure 4.7. In the leftmost graph which represents -log(fx), the closer the function gets to 1, the smaller the penalty but the closer it gets to 0, the higher. We want to penalize -log(f(x)) severely if it classifies as 0 when the true target is 1. The reverse is true when the true target is 0. This is shown in the rightmost graph. Here we penalize -log(1-f(x)) severely as it moves toward 1 because the true target is 0.

We can put the two loss functions into one equation as shown below, where they are summed over all observations. Notice that one of the terms will always be zero because y will be either 0 or 1.

$$\mathcal{L} = -\Big[\sum_{i=i}^{N} y_i log(f(x_i)) \quad + \quad (1 - y_i) log(1 - f(x_i))\Big] \tag{4.18}$$

where f(x) =

$$f(x) = \frac{1}{1 + e^{-(w^T x)}} \tag{4.19}$$

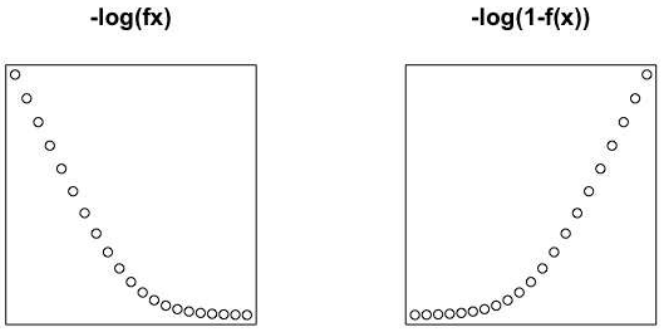

Figure 4.7: Loss Function for Logistic Regression

The parameters w are then determined using gradient descent. You can see from Figure 4.7 that when you put together the two loss functions you will have a convex function suitable for gradient descent. Once the parameters are known, classifying new instances is done by plugging in the instance into the logistic function shown in Equation 4.19. This returns a probability in the range [0, 1]. Typically we establish a threshold such as 0.5. Values over that threshold are classified as 1, values below are classified as 0.

4.7 Logistic Regression with Multiple Predictors

The associated website has a logistic regression example on a Titanic data set. We will just cover a few points here. The first is that the data had a lot of NA values. NA values can cause a lot of problems for many classifiers and it is a good idea to remove them. There are a couple of approaches to take. One is to simple remove rows with NAs which is a good choice if you have lots of remaining data. We saw an example of that using R's complete.cases() function in the linear regression chapter. Another approach is to replace NA values with either the mean or median of the variable for quantitative variables, or the most common factor for qualitative variables. The code below shows the detection of NAs with the sapply() function.

Code 4.7.1 — Detecing NAs. Using sapply().

```
sapply(df, function(x) sum(is.na(x)==TRUE))
  pclass survived      sex      age
       1        1        0      264
```

Since there are single NAs for pclass and survived, we can just remove those rows, for age, we will replace the NA with the median value. That is shown in the code below.

Code 4.7.2 — Removing NAs. By deleting rows or replacing with the median.

```
df <- df[!is.na(df$pclass),]
df <- df[!is.na(df$survived),]
df$age[is.na(df$age)] <- median(df$age,na.rm=T)
```

In the online notebook you will see that we divided the data into train and test sets and build a logistic regression model which achieved 79% accuracy. The code also gives an example of using the `caret` package to output the confusion matrix and other statistics including the Kappa which was 0.5563. The ROC curve for our predictions is the ROC curve shown in the Metrics section above. The AUC for that curve was 0.84. The following is the key output of summary() for the model predicting survival with all predictors.

```
Call:
glm(formula = survived ~ ., family = "binomial", data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.5650  -0.6931  -0.4586   0.6847   2.3496

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.313570   0.342936   9.662  < 2e-16 ***
pclass2     -1.234362   0.244633  -5.046 4.52e-07 ***
pclass3     -2.253769   0.225046 -10.015  < 2e-16 ***
sexmale     -2.391903   0.169846 -14.083  < 2e-16 ***
age         -0.030974   0.006996  -4.427 9.54e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

    Null deviance: 1307.9  on 980  degrees of freedom
Residual deviance:  932.5  on 976  degrees of freedom
AIC: 942.5
```

4.8 Advanced Topic: Optimization Methods

Earlier we specified our log-loss function for logistic regression as follows:

$$\ell = \sum_{i=1}^{n} y_i \, log f(x_i) + (1 - y_i) \, log(1 - f(x_i)) \tag{4.20}$$

where f(x) =

$$f(x) = \frac{1}{1 + e^{-(w^T x)}} \tag{4.21}$$

Next we find the gradient of the log likelihood. The gradient, g, is the partial derivative with respect to the parameters **w**. The gradient is the slope which is going to tell the algorithm which direction to move to find the minimum. The gradient equation is given below. Notice that it is really the X matrix multiplied by how wrong f(x) is at this point in predicting the true y.

$$g = \frac{\partial \ell}{\partial \mathbf{w}} = \sum_i (f(x_i) - y_i)\mathbf{x}_i = \mathbf{X}^T(f(x) - \mathbf{y}) \tag{4.22}$$

All we need for gradient descent is the first derivative, the gradient. For other optimization methods we will also need the second derivative, the Hessian, H. The Hessian is a square matrix of

second partial derivatives that gives the local curvature of a function. Note that in the following equation for the Hessian, the S represents $S \triangleq diag(p_i(1-p_i))$

$$H = \frac{\partial}{\partial \mathbf{w}} g(\mathbf{w})^T = \sum_i (\nabla_w f(x_i)) \mathbf{x}_i^T = \sum_i f(x_i)\,(1 - f(x_i)\,\mathbf{x}_i\,\mathbf{x}_i^T = \mathbf{X}^T \mathbf{S} \mathbf{X} \tag{4.23}$$

There are many algorithms to find the optimal parameters, $\mathbf{w}$. The first one we examine is gradient descent.

4.8.1 Gradient Descent

Gradient descent is an iterative approach where at each step the estminated parameters, θ get closer to the optimal values. We can express this as follows:

$$\theta_{k+1} \;=\; \theta_k - \eta_k\; g_k \tag{4.24}$$

where η is the learning rate, which specifies how big of a step to take at each iteration. If the eta (step size) is too slow the algorithm will take a long time to converge. If eta is too large, the true minimum can be stepped over and then the algorithm will not be able to converge.

4.8.2 Stochastic Gradient Descent

For large data sets, gradient descent can bog down. An alternative is *stochastic gradient descent* which processing the data either one at a time or in small batches instead of all at once. If the data is processed one at a time it means that the gradient has to be computed at each step. It turns out that the gradient will reflect the underlying function better if it is computed from a small batch. This also improves computation time.

4.8.3 Newton's Method

Gradient descent finds the optimal parameters using the first derivative. Newton's method is another approach; it uses the second derivative, the Hessian defined above. Newton's method (also called Newton-Raphson) is also an iterative method. At each step either the full Hessian is recalculated, or it is updated in which we call the method quasi-Newton. In a well-behaved convex function, Newton's method will converge faster.

A key insight in Newton's method is that if it is computationally difficult to compute a minimum for a given function, then come up with a function that shares important properties with the original function but is easier to minimize. At each iteration, Newton's method constructs a quadratic approximation to the objective function in which the first and second derivatives are the same. The approximate function is minimized instead of the original.

How is this approximate function found? A Taylor series about the point is used, but ignores derivatives past the second. A Taylor series converts a function into a power function and the first few terms can be used to get an approximate value for a function.

4.8.4 Optimization from Scratch

We are going to take a closer look at gradient descent by finding our optimal coefficients in the plasma data set from scratch. The R Notebook for this is available online. First we recreate the logistic regression model from earlier in the chapter. Our coefficients were $w_0 = -8.38$ and $w_1 = 2.34$.

The first thing we need to do is define our sigmoid/logistic function that will take an input matrix and return a vector of sigmoid values for each observation. We initialize w_0 and w_1 to 1. We

make a data matrix where column 1 is all 1s that will be multiplied by the intercept, and column 2 is fibrinogen which will be multiplied by w_1. We will also need the labels but since they were coded as 1-2 instead of 0-1 we subtract 1.

Code 4.8.1 — Logistic Regression from Scratch. Set up code.

```
sigmoid <- function(z){
  1.0 / (1+exp(-z))
}
# set up weight vector, label vector, and data matrix
weights <- c(1, 1)
data_matrix <- cbind(rep(1, nrow(train)), train$fibrinogen)
labels <- as.integer(train$ESR) - 1
```

Now we are ready to iterate. The code below iterates 500,000 times. In each iteration it does the following:

1. multiplies the data by the weights to get the log likelihood, then runs these values through the sigmoid() function to get a vector of probabilities
2. computes the error: the true values (0 or 1) minus the probabilities
3. updates the weights by weights plus the learning rate times the gradient; Recall that the gradient is the X values times the errors as shown in Equation 4.22; Notice also the operator for matrix multiplication is an asterisk surrounded by percent signs.

Code 4.8.2 — Gradient Descent from Scratch. Three steps per iteration.

```
learning_rate <- 0.001
for (i in 1:500000){
  prob_vector <- sigmoid(data_matrix %*% weights)
  error <- labels - prob_vector
  weights <- weights + learning_rate * t(data_matrix) %*% error
}
weights
```

Try running this code several times, changing the number of iterations. The following table shows the weights af various numbers of iterations.

No. Iterations	(w_0)	(w_0)
50	0.45	-0.36
500	-0.25	-0.27
5000	-4.15	1.00
50000	-8.26	2.30
500000	-8.38	2.34

Table 4.2: Optimized Weights by Number of Iterations

The 500,000 iterations gives use the same coefficients as R's glm(). However, it was slow compared to R's optimized code. R functions are heavily optimized and will reliably give good performance.

Finally, we create a plot that confirms Equation 4.11. The linear combination of the weights we calculated in the code above and X values give us the log odds. In the graph the ESR>20 are color coded.

Code 4.8.3 — Log Odds. Linear combination of $w_0 + w_1 x$

```
plasma_log_odds <- cbind(rep(1, 32), plasma$fibrinogen) %*% weights
plot(plasma$fibrinogen, plasma_log_odds, col=plasma$ESR)
abline(weights[1], weights[2])
```

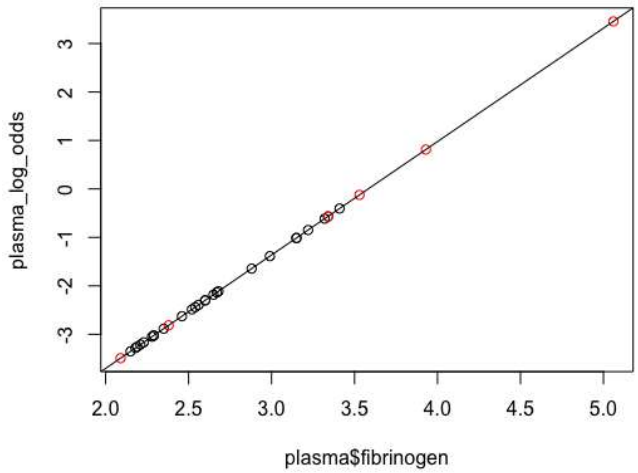

Figure 4.8: LogOdds is a Linear Combination of the Parameters

4.9 Multiclass Classification

The classification examples we have seen so far have been binary, classifying into one of two possible classes. What if we want to classify in a scenario where there are more than two classes? One technique that can be used is **one-versus-all** classification in which we perform multiple binary classifications.

Let's look at the iris data set as an example. The notebook for this is in github as usual. The iris data set contains 150 observations of flower measurements. There are 50 observations each of 3 species: virginica, setosa, and versicolor. First we look at a couple of graphs for data exploration. Figure 4.9 shows the pairs() output for the predictor columns with the color of each observation representing one of the 3 classes. Figure 4.10 plots petal width and length, color coded as well. The code for these plots is given below. The as.integer() function was used to make Species an integer 1, 2, or 3, which in turn is used to match colors red, yellow and blue.

Code 4.9.1 — Code to Generate Plots. Using as.integer()

```
pairs(iris[1:4], pch = 21,
     bg = c("red", "yellow", "blue")[as.integer(Species)])

plot(Petal.Length, Petal.Width, pch=21,
    bg=c("red","yellow","blue")as.integer(Species)])
```

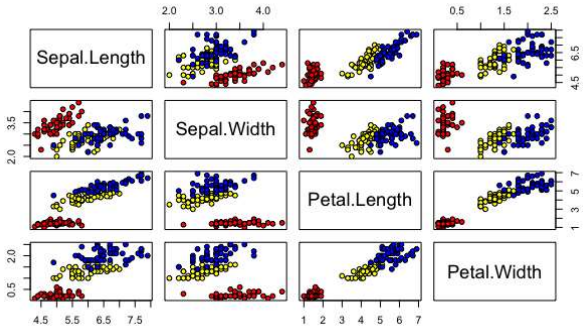

Figure 4.9: Iris Pairs

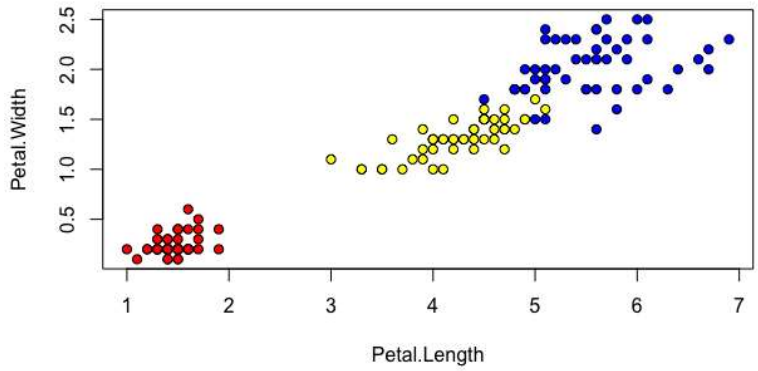

Figure 4.10: Iris Petal Length and Width

To perform one-versus-all classification for 3 classes, we have to create 3 data sets, one for each class as shown next. Each time we copy the original data set, then set the Species column to be a binary factor for that species or not.

Code 4.9.2 — Make a data set for each class. With a binary target.

```
# reclassify as virginica or not
iris_virginica <- iris
iris_virginica$Species <-
    as.factor(ifelse (iris_virginica$Species=="virginica",1,0))
# reclassify as setosa or not
iris_setosa <- iris
iris_setosa$Species <-
    as.factor(ifelse (iris_setosa$Species=="setosa",1,0))
# reclassify as versicolor or not
iris_versicolor <- iris
iris_versicolor$Species <-
    as.factor(ifelse (iris_versicolor$Species=="versicolor",1,0))
```

We next write a function to enable us to run the same code 3 times.

Code 4.9.3 — Function Definition. For repeated code.

```
fun <- function(df, i){
    train <- df[i,]
    test <- df[-i,]
    glm1 <- glm(Species~., data=train, family="binomial")
    probs <- predict(glm1, newdata=test)
    pred <- ifelse(probs>0.5, 1, 0)
    acc <- mean(pred==test$Species)
    print(paste("accuracy = ", acc))
    table(pred, test$Species)
}
```

Next we make one set of indices to divide the train and test sets, and run the function on each data set.

Code 4.9.4 — Run the function. On each data set.

```
set.seed(1234)
i <- sample(1:150, 100, replace=FALSE)
fun(iris_virginica, i)
fun(iris_setosa, i)
fun(iris_versicolor, i)
```

The accuracies for the 3 runs were: 0.98, 1,0, and 0.62. The average gives 87% overall accuracy. From the figures above it is clear that separating setosa (yellow) and versicolor (blue) is challenging and it looks like the versicolor classifier is the weakest one.

Caution 4.3 — Warnings in glm(). For logistic regression, if your training data is perfectly or nearly perfectly linearly separable, R will throw out several warning messages. This is due to the inability to maximize the likelihood which already has separated the data perfectly. For example, for the iris data which is too easy to classify, the sample notebook on github shows the error messages:

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Since there are just warnings, they do not stop the algorithm from producing a model. ■

4.10 Generalized Linear Models

Logistic regression is part of the GLM (generalized linear model) family because its response is determined by a predictor in which the parameters are linear. In R's glm() function we used the family parameter to specify the family and link function.

```
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
```

For binomial, the link function is "logit". Other available families include gaussian, Gamma, poisson and more, as indicated in the R documentation.

Why do we need a link function? In linear regression the algorithm assumes that the target variable is normally distributed over the real numbers. This is not the case for logistic regression

and other generalized linear models. The link function links the mean of the target $\mu_i = E(Y_i)$ to the linear term $x_i^T w$. The link function "links" the linear predictors to the response. The canonical link for mapping real numbers to [0,1] is the logit.

4.11 Summary

Logistic regression is something of a misnomer because we use it for classification, not regression. It is considered a linear model because it is linear in the parameters. The sigmoid function shapes the output to be in range [0, 1] for probabilities. Here are the stengths and weaknesses of logistic regression:

Strengths:

- Separates classes well if they are linearly separable
- Computationally inexpensive
- Nice probabilistic output

Weaknesses:

- Prone to underfitting; Not flexible enough to capture complex non-linear decision boundaries

In this chapter we showed how to perform logistic regression with one predictor or multiple predictors. In addtion, we discussed how to use the one-versus-all technique for multi-class classification.

4.11.1 New Terminology in this Chapter

Refer back to the chapter or look at the glossary if you are unsure of the meaning of these terms.

New metrics:

- accuracy
- error rate
- sensitivity
- specificity
- Kappa
- ROC Curves
- AUC

Mathematical terms:

- probability
- odds
- log odds

4.11.2 Quick Reference

Reference 4.11.1 Build and Examine a Logistic Regression Model

```
glmName <- glm(formula, data=train, family=binomial)
summary(glmName)
```

Reference 4.11.2 Predict on Test Data

```
probs <- predict(glmName, newdata=test, type="response")
pred <- ifelse(probs>0.5, 1, 0)
table(pred, test$target)
acc <- mean(pred==test$target)
```

Reference 4.11.3 Counting NAs with sapply(). The first argument to sapply is a list, and a data frame can be considered a list of vectors. The second argument in our example is an anonymous function that sums the number of NAs. The sapply function applies this anonymous function to every column in the data frame.

```
sapply(df, function(x) sum(is.na(x)==TRUE)
```

Reference 4.11.4 Fixing NAs with lapply(). In the code below we first define a function to replace NAs in a vector with the average of the vector. Notice we have to use the parameter `na.rm=TRUE` to get the mean value. As a reminder, in R the last statement evaluated is returned. There is no need for a return() statement in this simple function. The last statement below uses lapply() to run the fix_NA function on every vector. Use lapply instead of sapply when you want a vector back instead of a list.

```
fix_NA <- function(x){
  ifelse(!is.na(x),x,mean(x, na.rm=TRUE))
}

df[] <- lapply(df, fix_NA)
```

4.11.3 Going Further

A free online lesson on Logistic Regression is available here: `https://onlinecourses.science.psu.edu/stat504/node/149`