Stat 495 - Chapter 8 Examples

A.S. Wagaman

You've already seen logistic regression (refer to our separate notes there). These examples are designed to show you Poisson regression and regression trees. The code is re-used from previous courses, so there may be better / more modern packages, which you can feel free to use if you know them.

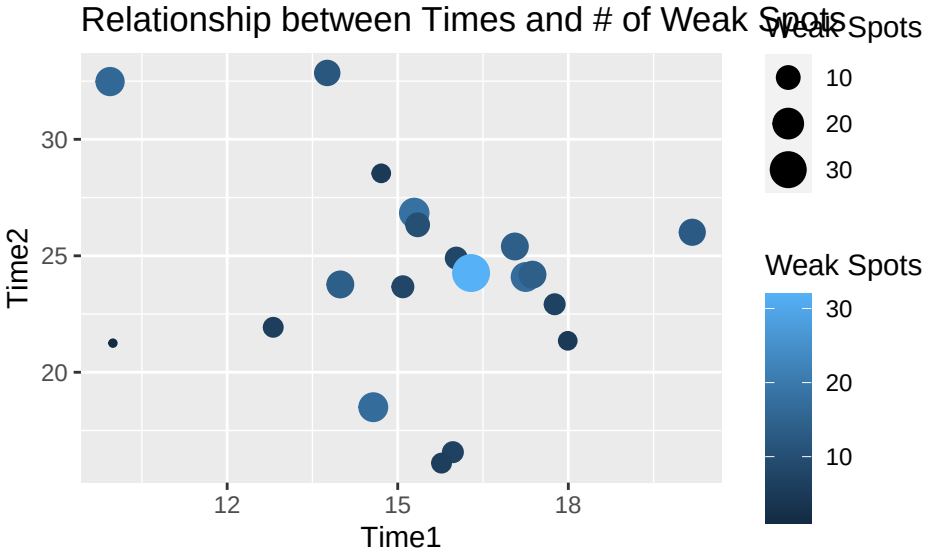
Poisson Regression

In Poisson regression, the response variable is a count. In this data set, we have data on fabric being mass-produced. In tests for quality control, it is subjected to two different processes for different amounts of time (seconds), and the number of weak spots for each sheet of fabric are counted. We want to assess if there is a relationship between the time it spends in each process and the number of weak spots, including whether there is an interaction effect.

```
fabric <- read.csv("https://awagaman.people.amherst.edu/stat495/fabricdata.csv", header = T)
```

We can visualize the data since there are only three variables fairly easily.

```
ggplot(data = fabric, aes(x = Time1, y = Time2)) +
  geom_point()  + aes(colour = WeakSpots, size = WeakSpots) +
  theme(legend.position = "right") +
  labs(title = "Relationship between Times and # of Weak Spots",
       color = "Weak Spots",
       size = "Weak Spots")
```



To fit the model, we use our *glm* function.

```
fabricmod <- glm(WeakSpots ~ Time1 * Time2, data = fabric, family = poisson (link = log))
msummary(fabricmod)
```

```
## Coefficients:
```

```
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.264250   2.814786  -0.804    0.421
## Time1        0.247466   0.191614   1.291    0.197
## Time2        0.148321   0.104076   1.425    0.154
## Time1:Time2 -0.007317   0.007241  -1.011    0.312
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 74.040  on 19  degrees of freedom
## Residual deviance: 63.637  on 16  degrees of freedom
## AIC: 153.81
##
## Number of Fisher Scoring iterations: 5
```

```
lrtest(fabricmod)
```

```
## Likelihood ratio test
##
## Model 1: WeakSpots ~ Time1 * Time2
## Model 2: WeakSpots ~ 1
##   #Df  LogLik Df  Chisq Pr(>Chisq)
## 1   4 -72.905
## 2   1 -78.106 -3 10.402    0.01544 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that the default link for the poisson is the log link, so you can actually just state family = poisson and it will use that link.

The resulting test output suggests that there is at least one significant predictor. However, we don't see any in the output. We may not need that interaction term, so let's refit and see what happens.

```
fabricmod2 <- glm(WeakSpots ~ Time1 + Time2, data = fabric, family = poisson (link = log))
msummary(fabricmod2)
```

```
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.46825    0.68294   0.686  0.49295
## Time1        0.05714    0.02819   2.027  0.04270 *
## Time2        0.04462    0.01668   2.675  0.00748 **
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 74.040  on 19  degrees of freedom
## Residual deviance: 64.649  on 17  degrees of freedom
## AIC: 152.82
##
## Number of Fisher Scoring iterations: 5
```

```
lrtest(fabricmod2)
```

```
## Likelihood ratio test
##
## Model 1: WeakSpots ~ Time1 + Time2
## Model 2: WeakSpots ~ 1
##   #Df  LogLik Df  Chisq Pr(>Chisq)
## 1   3 -73.410
```

```
## 2   1 -78.106 -2 9.3912   0.009135 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Aha! So, both predictors are significant in the presence of each other but the interaction is not significant. When looking at the coefficients, the "link'' function is the log, so you are looking at expected log count changes in the response.

Overdispersion can be a problem here, and there are ways to estimate robust coefficients and standard errors. Just like there was a quasibinomial option for the family in R to handle overdispersion, there is a quasipoisson you can use for the same purpose. There's a lot more you can learn about Poisson regression if you want to apply it to a problem.

Trees - Short Example from an Activity by Prof. Horton - Classification

"Regression trees (chapter 8 of CASI) are an important nonparametric modeling approach (see https://en.wikipedia.org/wiki/Decision_tree_learning). Here we consider an example using the HELP (Health Evaluation and Linkage to Primary Care) study where we are interested in predictors of being homeless (defined as one or more nights on the street or in a shelter in the past 90 days).''

This is a classification example, not typical regression, since it has a binary response (note the method = "class"), like logistic regression. This is similar to the book's spam example.

```
homeless.rpart <- rpart(homeless ~ female + i1 + substance + sexrisk + mcs +
   pcs, method="class", data = HELPrct)
printcp(homeless.rpart)
```

```
##
## Classification tree:
## rpart(formula = homeless ~ female + i1 + substance + sexrisk +
##     mcs + pcs, data = HELPrct, method = "class")
##
## Variables actually used in tree construction:
## [1] female  i1      mcs     pcs     sexrisk
##
## Root node error: 209/453 = 0.46137
##
## n= 453
##
##         CP nsplit rel error  xerror     xstd
## 1 0.095694      0   1.00000 1.00000 0.050766
## 2 0.049442      1   0.90431 1.05742 0.050903
## 3 0.033493      4   0.75598 1.02871 0.050853
## 4 0.021531      5   0.72249 1.00000 0.050766
## 5 0.014354      7   0.67943 0.94258 0.050484
## 6 0.010000      9   0.65072 0.90431 0.050215
```

```
pdf("party.pdf", width = 14, height = 14)
plot(as.party(homeless.rpart))
dev.off()
```

```
## pdf
##   2
```

The tree here is output to a .pdf called "party.pdf". Open it to view the tree. Be sure to include the dev.off() line here.

How would this compare to the results from a logistic regression model?

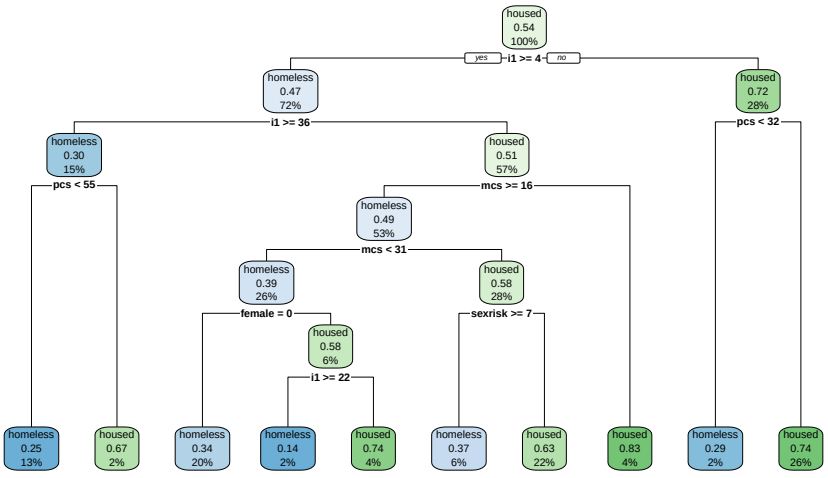
```
mod <- glm(homeless == "homeless" ~ i1 + female + sexrisk + pcs, family = binomial,
  data = HELPrct)
msummary(mod)
```

```
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.093964   0.516699  -0.182   0.8557
## i1           0.024671   0.005797   4.255 2.09e-05 ***
## female      -0.522167   0.242724  -2.151   0.0315 *
## sexrisk      0.072232   0.035896   2.012   0.0442 *
## pcs         -0.014776   0.009456  -1.563   0.1181
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 625.28  on 452  degrees of freedom
## Residual deviance: 588.95  on 448  degrees of freedom
## AIC: 598.95
##
## Number of Fisher Scoring iterations: 4
```

"We see that the results are similar: more drinking (higher scores for `i1`) are associated with higher odds of being homeless. The same is true for PCS (see the comparison of the first two nodes or the comparison of the last two nodes)." '

There are several options for packages in R to help display trees. Rpart.plot is one of these. This could be written to a file as well (see above).

```
rpart.plot(homeless.rpart)
```



Trees can also predict quantitative responses, in that case being a regression tree, as described in the text, and shown in the next example.

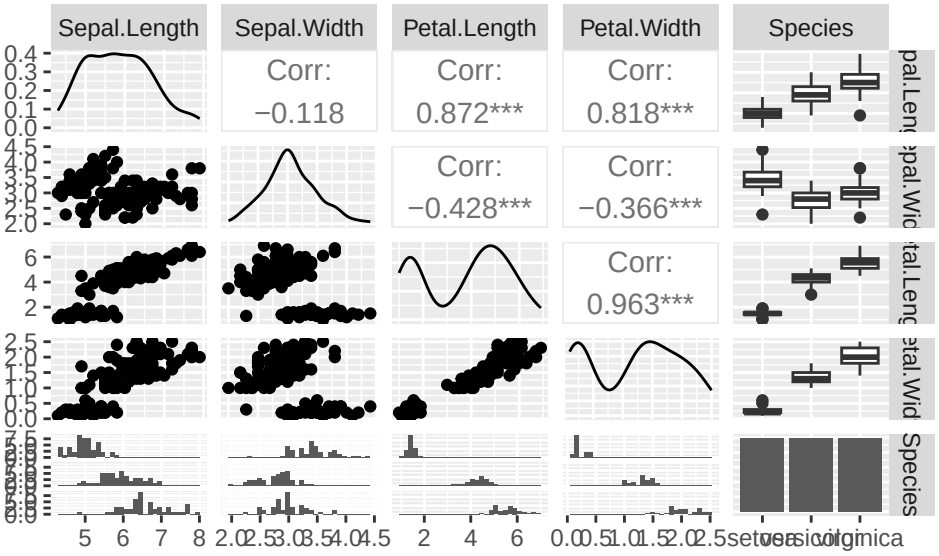
Trees - Iris - Regression

Now we want to use a quantitative response in our regression tree.

```
data(iris)
ggpairs(iris)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

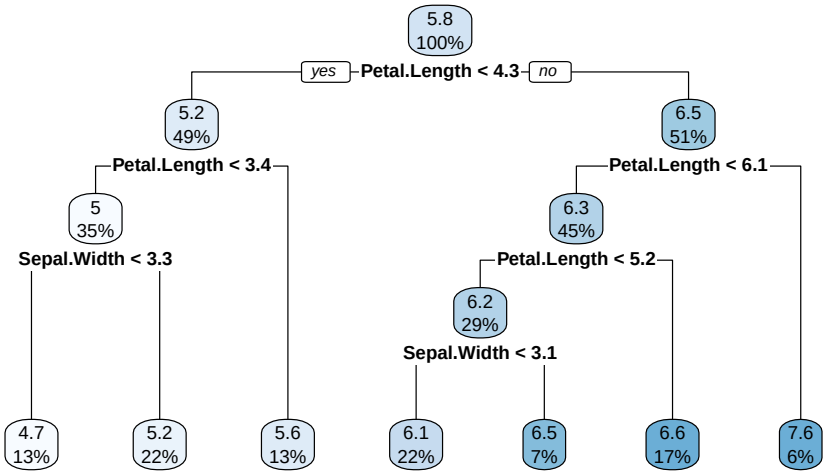


Suppose we want to predict Sepal.Length using the other variables as predictors.

```
iris.rpart <- rpart(Sepal.Length ~ . , data = iris, method = "anova") #method = "anova" for regression
printcp(iris.rpart)
```

```
##
## Regression tree:
## rpart(formula = Sepal.Length ~ ., data = iris, method = "anova")
##
## Variables actually used in tree construction:
## [1] Petal.Length Sepal.Width
##
## Root node error: 102.17/150 = 0.68112
##
## n= 150
##
##         CP nsplit rel error  xerror     xstd
## 1 0.613462      0   1.00000 1.00682 0.098504
## 2 0.121807      1   0.38654 0.40588 0.048880
## 3 0.057189      2   0.26473 0.30069 0.032871
## 4 0.029805      3   0.20754 0.25279 0.031628
## 5 0.023032      4   0.17774 0.26419 0.031665
## 6 0.016980      5   0.15471 0.24367 0.029987
## 7 0.010000      6   0.13773 0.23153 0.028687
```

```
#one tree
rpart.plot(iris.rpart)
```




```
# example tree written to pdf using other "pretty" function for tree
pdf("iris.pdf", width = 14, height = 14)
plot(as.party(iris.rpart)) # could sub in rpart.plot line here instead
dev.off()
```

```
## pdf
##   2
```

We print the tree to this pdf with one command and use a second to output the same tree to iris.pdf. They will look different based on the package doing the graphic. The one sent to pdf shows boxplots for the nodes in terms of the response values, but not the predicted value. If we want to look at the predictions, we can get those via:

```
iris2 <- mutate(iris, fittedtree = predict(iris.rpart))
```

We could then calculate the MSE. Note that I added the fitted values to a new data set for convenience here.

```
mean((iris2$fittedtree - iris2$Sepal.Length)^2)
```

```
## [1] 0.0938078
```

Note that this is the MSE on the model fit to the entire data set (it would be better to have a training/test split).

There are many options that can be supplied to rpart to control properties of the tree (how big it is, what criteria is used for splitting, etc.).

For example, the control option below forces the tree to stop way sooner (much too soon in this case), by making it so that when a split is made, at least 30 observations have to end up in either node from the split. Given that there are only 150 observations, this is not a good choice for this example.

```
iris.rpart2 <- rpart(Sepal.Length ~ . , data = iris, control = rpart.control(minbucket = 30))
printcp(iris.rpart2)
```

```
##
## Regression tree:
## rpart(formula = Sepal.Length ~ ., data = iris, control = rpart.control(minbucket = 30))
##
## Variables actually used in tree construction:
## [1] Petal.Length
##
## Root node error: 102.17/150 = 0.68112
##
```

```
## n= 150
##
##         CP nsplit rel error xerror     xstd
## 1 0.61346      0   1.00000 1.0143 0.099037
## 2 0.01000      1   0.38654 0.3974 0.049486
```

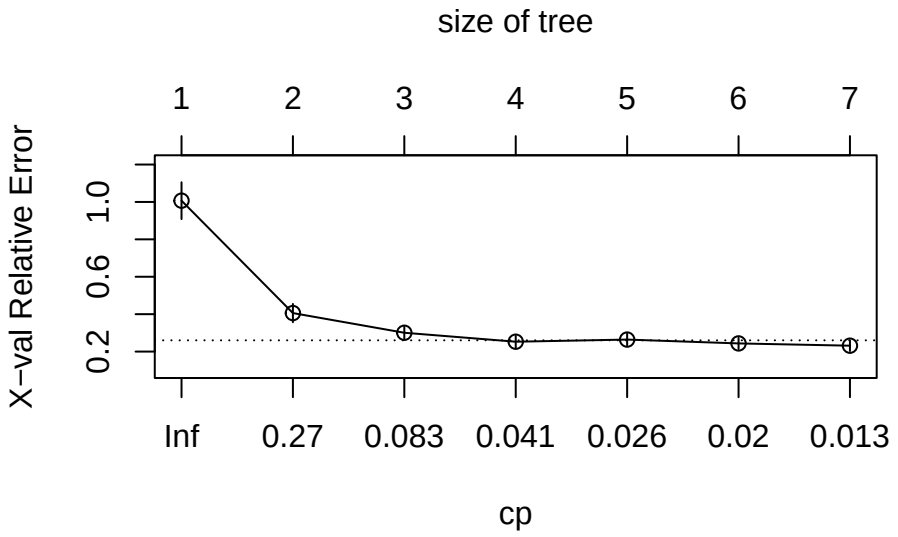
You can explore other options in rpart.control via the help menu. There is a default value of a complexity parameter, cp, of 0.01. It tends to override the other control parameters unless they supersede it. If you want to let a tree really grow, governed by minsplit and minbucket, you should set cp to 0 (or something very very small), as in this example:

```
iris.rpart3 <- rpart(Sepal.Length ~ . , data = iris, control = rpart.control(minbucket = 4, cp = 0, mins
printcp(iris.rpart3)
```

```
##
## Regression tree:
## rpart(formula = Sepal.Length ~ ., data = iris, control = rpart.control(minbucket = 4,
##     cp = 0, minsplit = 10))
##
## Variables actually used in tree construction:
## [1] Petal.Length Petal.Width  Sepal.Width  Species
##
## Root node error: 102.17/150 = 0.68112
##
## n= 150
##
##             CP nsplit rel error  xerror     xstd
## 1  0.61346237      0  1.000000 1.01395 0.099389
## 2  0.12180701      1  0.386538 0.40614 0.049836
## 3  0.05718872      2  0.264731 0.30355 0.033276
## 4  0.02980452      3  0.207542 0.24835 0.031258
## 5  0.02303165      4  0.177737 0.26226 0.029249
## 6  0.01698037      5  0.154706 0.23668 0.028118
## 7  0.00835880      6  0.137725 0.22125 0.027662
## 8  0.00692256      7  0.129367 0.22652 0.028110
## 9  0.00568332      8  0.122444 0.21628 0.027834
## 10 0.00475686     10  0.111077 0.19901 0.026154
## 11 0.00431641     11  0.106321 0.18609 0.021754
## 12 0.00326501     12  0.102004 0.18607 0.021757
## 13 0.00303502     13  0.098739 0.18409 0.021710
## 14 0.00282867     14  0.095704 0.18634 0.021855
## 15 0.00263512     15  0.092875 0.18607 0.021890
## 16 0.00186505     17  0.087605 0.18574 0.021997
## 17 0.00157909     18  0.085740 0.18212 0.021799
## 18 0.00062642     19  0.084161 0.18204 0.021793
## 19 0.00060061     20  0.083535 0.18415 0.021620
## 20 0.00040782     21  0.082934 0.18528 0.021625
## 21 0.00035358     22  0.082526 0.18550 0.021616
## 22 0.00015293     23  0.082173 0.18647 0.021680
## 23 0.00000000     24  0.082020 0.18631 0.021687
```

You can get an estimate of performance with an option of using cross-validation in the creation of the tree. The default is 10-fold CV. This is governed by the xval control parameter. Here, we see some output related to the cross-validation process, which may help you determine where you can prune the tree.

```
plotcp(iris.rpart)
```



If you want to start from your existing tree and prune it, you can do that using the *prune* function. You need to specify a complexity parameter to use to stop the prune. Note that these are output in the printout so you can see how many levels you will end up with (roughly).

```
iris.prune <- prune(iris.rpart, cp = 0.02)
printcp(iris.prune)
```

```
##
## Regression tree:
## rpart(formula = Sepal.Length ~ ., data = iris, method = "anova")
##
## Variables actually used in tree construction:
## [1] Petal.Length Sepal.Width
##
## Root node error: 102.17/150 = 0.68112
##
## n= 150
##
##         CP nsplit rel error  xerror     xstd
## 1 0.613462      0   1.00000 1.00682 0.098504
## 2 0.121807      1   0.38654 0.40588 0.048880
## 3 0.057189      2   0.26473 0.30069 0.032871
## 4 0.029805      3   0.20754 0.25279 0.031628
## 5 0.023032      4   0.17774 0.26419 0.031665
## 6 0.020000      5   0.15471 0.24367 0.029987
```

```
pdf("irisprune.pdf", width = 14, height = 14)
plot(as.party(iris.prune))
dev.off()
```

```
## pdf
##   2
```

The various listed error rates in the printout do have meaning but I'm used to interpreting them in the context of classification problems. You can probably find the details for regression trees if you want.

There is also another library available to create trees called *tree* which was developed to support particular

textbooks. rpart has the same functionality.