

My Data Science Notes

Michael Foley

2020-02-20

Contents

Intro	5
1 Inference	7
2 Experiments	9
2.1 Example one	9
2.2 Example two	9
3 Regression	11
4 Generalized Linear Models	13
4.1 Logistic Regression	14
4.2 Poisson Regression	24
5 Classification	31
6 Classification	33
7 Decision Trees	35
7.1 Classification Tree	36
7.2 Regression Trees	61
7.3 Bagging	75
7.4 Random Forests	75
7.5 Gradient Boosting	89
7.6 Summary	152
7.7 Reference	154

8	Regularization	157
9	Non-linear Models	159
9.1	Splines	159
9.2	MARS	160
9.3	GAM	162
10	Support Vector Machines	165
10.1	Maximal Margin Classifier	165
10.2	Support Vector Classifier	166
10.3	Support Vector Machines	167
10.4	Example	168
10.5	Using Caret	178
11	Principal Components Analysis	181
12	Clustering	183
13	Text Mining	185
	Appendix	187
	Publishing to BookDown	187
	Shiny Apps	187
	Packages	187

Intro

These notes are pulled from various classes, tutorials, books, etc. and are intended for my own consumption. If you are finding this on the internet, I hope it is useful to you, but you should know that I am just a student and there's a good chance whatever you're reading here is mistaken. In fact, that should probably be your null hypothesis... or your prior. Whatever.

Chapter 1

Inference

Chapter 2

Experiments

Some *significant* applications are demonstrated in this chapter.

2.1 Example one

2.2 Example two

Chapter 3

Regression

Chapter 4

Generalized Linear Models

These notes are primarily from PSU Online course Analysis of Discrete Data which uses Alan Agresti's **Categorical Data Analysis** (which I have not yet purchased) (Agresti, 2013). I also reviewed:

Penn State University, STAT 501, "Lesson 15: Logistic, Poisson & Nonlinear Regression". <https://newonlinecourses.science.psu.edu/stat501/lesson/15>

"Generalized Linear Models in R". DataCamp. <https://www.datacamp.com/courses/generalized-linear-models-in-r>.

"Multiple and Logistic Regression". DataCamp. <https://www.datacamp.com/courses/multiple-and-logistic-regression>.

Molnar, Christoph. "Interpretable machine learning. A Guide for Making Black Box Models Explainable", 2019. <https://christophm.github.io/interpretable-ml-book/>.

In generalized linear models (GLMs), the modeled response is a *function* of the mean of y . There are three components to a GLM. The *random component* is the probability distribution of the response variable (normal, binomial, Poisson, etc.). The *systematic component* is the explanatory variables $X\beta$. The *link function* η specifies the link between random and systematic components. The link function converts the response value from a range $[0, 1]$ in logistic and probit and $[0, +\infty]$ for Poisson to a value ranging from $[-\infty, +\infty]$, and creates a linear relationship with the predictor variables.

For a standard linear regression, the link function is the identity function,

$$f(\mu_Y) = \mu_Y.$$

The standard linear regression is thus a special case of the GLM. For a logistic regression, the link function is

$$f(\mu_Y) = \ln\left(\frac{\pi}{1-\pi}\right)$$

where π is the event probability. For a probit regression, the link function is

$$f(\mu_Y) = \Phi^{-1}(\pi).$$

The difference between logistic and probit link function is theoretical - the practical significance is slight. Logistic regression has the advantage that it can be back-transformed from log odds to odds ratios. For a Poisson regression, the link function is

$$f(\mu_Y) = \ln(\lambda)$$

where λ is the expected event rate.

GLM uses maximum likelihood estimation (MLE) rather than ordinary least squares (OLS) to estimate the parameters, and thus relies on large-sample approximations.

In R, specify a GLM just like an linear model, but with the `glm()` function, specifying the distribution with the `family` parameter.

- `family = "gaussian"`: linear regression
- `family = "binomial"`: logistic regression
- `family = binomial(link = "probit")`: probit regression
- `family = "poisson"`: Poisson regression

4.1 Logistic Regression

Logistic regression estimates the probability of a particular level of a categorical response variable given a set of predictors. The response levels can be binary, nominal (multiple categories), or ordinal (multiple levels).

The **binary** logistic regression model is

$$y_i = \text{logit}(\pi_i) = \log\left(\frac{\pi_i}{1-\pi_i}\right) = X_i\beta$$

where π_i is the “success probability” that observation i is in a specified category of the binary y variable. You can solve for π to get

$$\pi = \frac{\exp(X\beta)}{1 + \exp(X\beta)}.$$

The model predicts the *log odds* of the response variable. The maximum likelihood estimator maximizes the the likelihood function

$$L(\beta; y, X) = \prod_{i=1}^n \pi_i^{y_i} (1 - \pi_i)^{(1-y_i)} = \prod_{i=1}^n \frac{\exp(y_i X_i \beta)}{1 + \exp(X_i \beta)}.$$

There is no closed-form solution, so GLM estimates coefficients with iteratively reweighted least squares.

Example

Dataset `leuk` contains response variable `REMISS` indicating whether leukemia remission occurred (1|0) and several explanatory variables.

```
data_dir <- "C:/Users/mpfol/OneDrive/Documents/Data Science/Data/"
leuk <- read_tsv(paste(data_dir, "leukemia_remission.txt", sep = "/"))
```

```
## Parsed with column specification:
## cols(
##   REMISS = col_double(),
##   CELL = col_double(),
##   SMEAR = col_double(),
##   INFIL = col_double(),
##   LI = col_double(),
##   BLAST = col_double(),
##   TEMP = col_double()
## )
```

```
str(leuk)
```

```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 27 obs. of 7 variables:
## $ REMISS: num 1 1 0 0 1 0 1 0 0 0 ...
## $ CELL : num 0.8 0.9 0.8 1 0.9 1 0.95 0.95 1 0.95 ...
## $ SMEAR : num 0.83 0.36 0.88 0.87 0.75 0.65 0.97 0.87 0.45 0.36 ...
## $ INFIL : num 0.66 0.32 0.7 0.87 0.68 0.65 0.92 0.83 0.45 0.34 ...
## $ LI : num 1.9 1.4 0.8 0.7 1.3 0.6 1 1.9 0.8 0.5 ...
## $ BLAST : num 1.1 0.74 0.18 1.05 0.52 0.52 1.23 1.35 0.32 0 ...
## $ TEMP : num 1 0.99 0.98 0.99 0.98 0.98 0.99 1.02 1 1.04 ...
## - attr(*, "spec")=
## .. cols(
## .. REMISS = col_double(),
## .. CELL = col_double(),
## .. SMEAR = col_double(),
```

```
## .. INFIL = col_double(),
## .. LI = col_double(),
## .. BLAST = col_double(),
## .. TEMP = col_double()
## .. )
```

Fit a logistic regression in R using `glm(formula, data, family = binomial)` where `family = binomial` specifies a binomial error distribution.

```
m1 <- glm(REMISS ~ ., family = binomial, data = leuk)
m1
```

```
##
## Call:  glm(formula = REMISS ~ ., family = binomial, data = leuk)
##
## Coefficients:
## (Intercept)          CELL          SMEAR          INFIL          LI          BLAST
##      64.25808      30.83006      24.68632     -24.97447       4.36045     -0.01153
##          TEMP
##     -100.17340
##
## Degrees of Freedom: 26 Total (i.e. Null);  20 Residual
## Null Deviance:      34.37
## Residual Deviance: 21.59      AIC: 35.59
```

```
summary(m1)
```

```
##
## Call:
## glm(formula = REMISS ~ ., family = binomial, data = leuk)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.95404  -0.66259  -0.02516   0.78184   1.57465
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   64.25808   74.96480   0.857    0.391
## CELL          30.83006   52.13520   0.591    0.554
## SMEAR         24.68632   61.52601   0.401    0.688
## INFIL        -24.97447   65.28088  -0.383    0.702
## LI            4.36045    2.65798   1.641    0.101
## BLAST        -0.01153    2.26634  -0.005    0.996
## TEMP        -100.17340   77.75289  -1.288    0.198
```



```
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 34.372  on 26  degrees of freedom
## Residual deviance: 21.594  on 20  degrees of freedom
## AIC: 35.594
##
## Number of Fisher Scoring iterations: 8
```

The *predicted* value \hat{y} is the estimated **log odds** of the response variable,

$$\hat{y} = X\hat{\beta} = \ln\left(\frac{\pi}{1-\pi}\right).$$

Suppose each predictor equals its mean value, then the log odds of REMISS is -2.684 .

```
pred <- predict(m1, newdata = data.frame(CELL = mean(leuk$CELL),
                                           SMEAR = mean(leuk$SMEAR),
                                           INFIL = mean(leuk$INFIL),
                                           LI = mean(leuk$LI),
                                           BLAST = mean(leuk$BLAST),
                                           TEMP = mean(leuk$TEMP)))
```

Log odds are not easy to interpret, but it is convenient for updating prior probabilities in Bayesian analyses. *See this article in Statistics How To*. Exponentiate the log odds to get the more intuitive **odds**.

$$\exp(\hat{y}) = \exp(X\hat{\beta}) = \frac{\pi}{1-\pi}.$$

The odds of having achieved remission when each predictor equals its mean value is $\exp(\hat{y}) = 0.068$.

```
exp(pred)
```

```
##      1
## 0.06826334
```

You might express that more commonly as $1 / 0.068 = 15:1$. So a person with average values of the predictors has an odds of “15 to 1” of having achieved remission.

```
1/exp(pred)
```

```
##          1
## 14.64915
```

Or, solve for π to get the **probability**.

$$\pi = \frac{\exp(X\beta)}{1 + \exp(X\beta)}$$

The probability of having achieved remission when each predictor equals its mean value is $\pi = 0.064$. The `predict()` function for a logistic model returns log-odds, but can also return π by specifying parameter `type = "response"`.

```
exp(pred) / (1 + exp(pred))
```

```
##          1
## 0.06390123
```

```
prob <- predict(m1, newdata = data.frame(CELL = mean(leuk$CELL),
                                           SMEAR = mean(leuk$SMEAR),
                                           INFIL = mean(leuk$INFIL),
                                           LI = mean(leuk$LI),
                                           BLAST = mean(leuk$BLAST),
                                           TEMP = mean(leuk$TEMP)),
               type = "response")
```

It is common to express the results in terms of the **odds ratio**. The *odds ratio* is the ratio of the odds before and after an increment to the predictors. It tells you how much the odds would be multiplied after a $X_1 - X_0$ unit increase in X .

$$\theta = \frac{\pi/(1-\pi)|_{X=X_1}}{\pi/(1-\pi)|_{X=X_0}} = \frac{\exp(X_1\hat{\beta})}{\exp(X_0\hat{\beta})} = \exp((X_1 - X_0)\hat{\beta}) = \exp(\delta\hat{\beta})$$

For example, increasing LI by .01 increases the odds of remission by a factor of $\exp(0.1 \cdot 4.36) = 1.547$ (from 15:1 to 23:1).

```
exp(.1 * m1$coefficients)
```

```
## (Intercept)          CELL          SMEAR          INFIL          LI          BLAST
## 6.175799e+02 2.182391e+01 1.180628e+01 8.229480e-02 1.546579e+00 9.988476e-01
##          TEMP
## 4.461948e-05
```

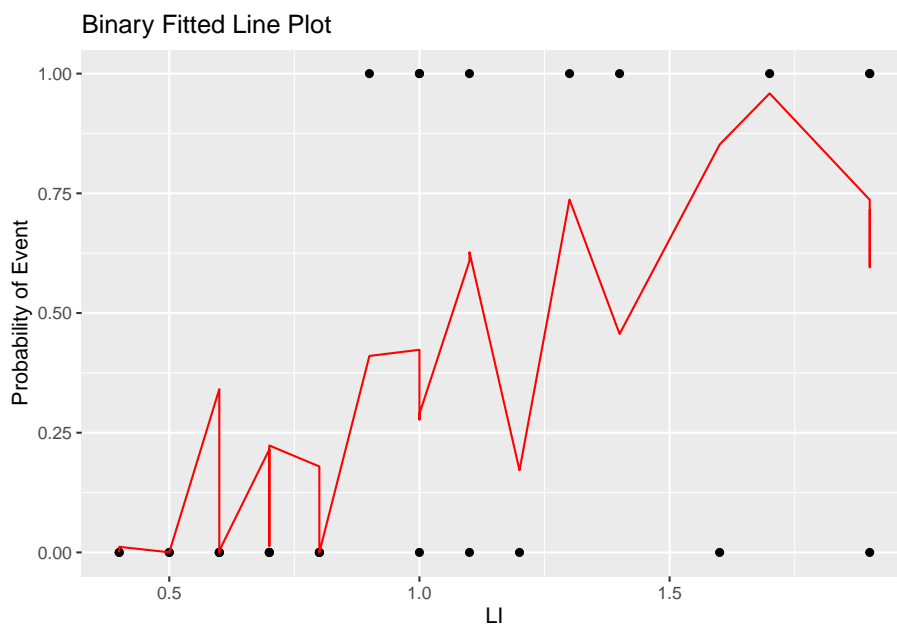
You can calculate an odds ratio using `oddsratio::or_glm()`.

```
library(oddsratio)
or_glm(data = leuk,
       model = m1,
       incr = list(CELL = 0.01,
                   SMEAR = 0.01,
                   INFIL = 5,
                   LI = 0.1,
                   BLAST = 1.0,
                   TEMP = 0.3))
```

```
## # A tibble: 6 x 5
##   predictor oddsratio `CI_low (2.5)` `CI_high (97.5)` increment
##   <chr>         <dbl>         <dbl>         <dbl> <chr>
## 1 CELL          1.36          0.747         4.64e  0 0.01
## 2 SMEAR         1.28          0.537         5.15e  0 0.01
## 3 INFIL          0            0            1.09e152 5
## 4 LI            1.55          1.04          2.99e  0 0.1
## 5 BLAST         0.989         0.009         9.10e  1 1
## 6 TEMP          0            0            1.10e  3 0.3
```

The predicted values can also be expressed as the probabilities π . This produces the familiar sigmoidal shape of the binary relationship.

```
augment(m1, type.predict = "response") %>%
  ggplot(aes(x = LI, y = REMISS)) +
  geom_point() +
  geom_line(aes(y = .fitted), color = "red") +
  labs(x = "LI",
       y = "Probability of Event",
       title = "Binary Fitted Line Plot")
```



Whereas in linear regression the coefficient p-values use the t test (t statistic), logistic regression coefficient p-values use the *Wald test* (** Z^* -statistic).

$$Z = \frac{\hat{\beta}_i}{SE(\hat{\beta}_i)}$$

```
round((z <- m1$coefficients / summary(m1)$coefficients[, "Std. Error"]), 3)
```

## (Intercept)	CELL	SMEAR	INFIL	LI	BLAST
## 0.857	0.591	0.401	-0.383	1.641	-0.005
## TEMP					
## -1.288					

```
round(pnorm(abs(z), lower.tail = FALSE) * 2, 3)
```

## (Intercept)	CELL	SMEAR	INFIL	LI	BLAST
## 0.391	0.554	0.688	0.702	0.101	0.996
## TEMP					
## 0.198					

Evaluate a logistic model fit with an analysis of deviance. Deviance is defined as -2 times the log-likelihood $-2l(\beta)$. The null deviance is the deviance of the null model and is analogous to SST in ANOVA. The residual deviance is analogous to SSE in ANOVA.

```
logLik(glm(REMISS ~ ., data = leuk, family = "binomial")) * (-2)
```

```
## 'log Lik.' 21.59385 (df=7)
```

```
anova(m1)
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: REMISS
##
## Terms added sequentially (first to last)
##
##
```

	Df	Deviance	Resid. Df	Resid. Dev
## NULL			26	34.372
## CELL 1	1	2.5800	25	31.792
## SMEAR 1	1	0.5188	24	31.273
## INFIL 1	1	0.2927	23	30.980
## LI 1	1	6.7818	22	24.199
## BLAST 1	1	0.3271	21	23.871
## TEMP 1	1	2.2775	20	21.594

```
m1
```

```
##
## Call: glm(formula = REMISS ~ ., family = binomial, data = leuk)
##
## Coefficients:
## (Intercept)          CELL          SMEAR          INFIL          LI          BLAST
##    64.25808    30.83006    24.68632   -24.97447     4.36045    -0.01153
##          TEMP
##   -100.17340
##
## Degrees of Freedom: 26 Total (i.e. Null);  20 Residual
## Null Deviance:      34.37
## Residual Deviance: 21.59    AIC: 35.59
```

```
summary(m1)
```

```
##
## Call:
```

```
## glm(formula = REMISS ~ ., family = binomial, data = leuk)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.95404  -0.66259  -0.02516   0.78184   1.57465
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  64.25808   74.96480   0.857   0.391
## CELL        30.83006   52.13520   0.591   0.554
## SMEAR       24.68632   61.52601   0.401   0.688
## INFIL      -24.97447   65.28088  -0.383   0.702
## LI          4.36045    2.65798   1.641   0.101
## BLAST       -0.01153    2.26634  -0.005   0.996
## TEMP       -100.17340   77.75289  -1.288   0.198
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 34.372  on 26  degrees of freedom
## Residual deviance: 21.594  on 20  degrees of freedom
## AIC: 35.594
##
## Number of Fisher Scoring iterations: 8
```

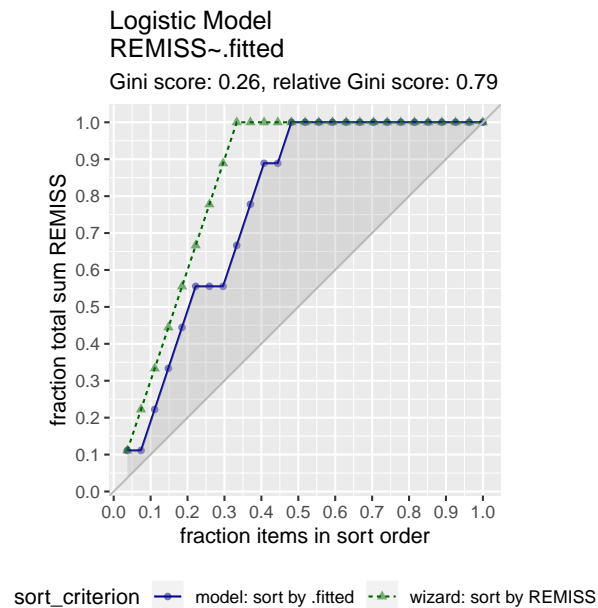
The deviance of the null model (no regressors) is 34.372. The deviance of the full model is 26.073.

```
glance(m1)
```

```
## # A tibble: 1 x 7
##   null.deviance df.null logLik   AIC   BIC deviance df.residual
##         <dbl>   <int>  <dbl> <dbl> <dbl>   <dbl>       <int>
## 1          34.4     26  -10.8  35.6  44.7    21.6         20
```

Use the `GainCurvePlot()` function to plot the gain curve (background on gain curve at Data Science Central from the model predictions. The x-axis is the fraction of items seen when sorted by the predicted value, and the y-axis is the cumulative summed true outcome. The “wizard” curve is the gain curve when the data is sorted by the true outcome. If the model’s gain curve is close to the wizard gain curve, then the model sorted the response variable well. The grey area is the gain over a random sorting.

```
augment(m1) %>% data.frame() %>%
  GainCurvePlot(xvar = ".fitted", truthVar = "REMISS", title = "Logistic Model")
```

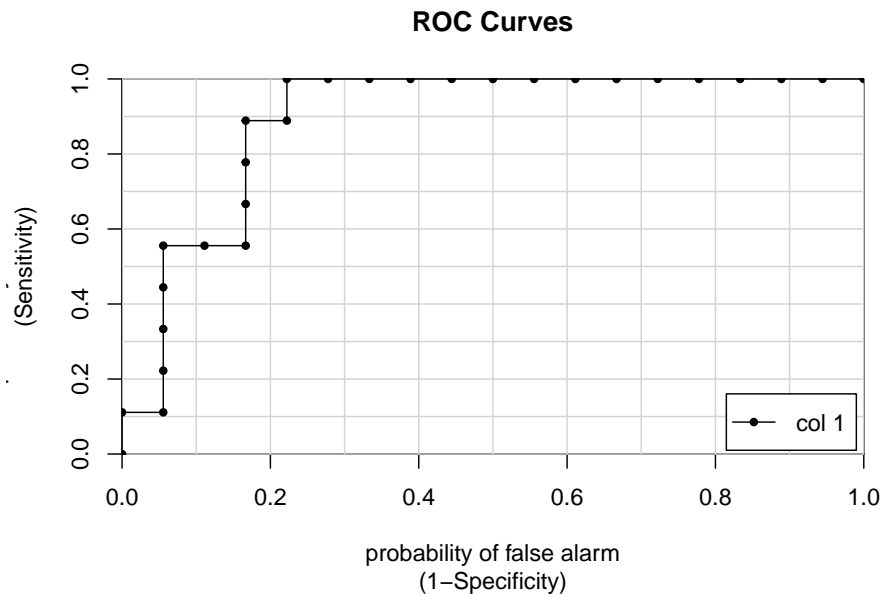


REMISS equals 1 in 9 of the 27 responses.

- The wizard curve shows that after sorting the responses it encounters all 9 1s (100%) after looking at 9 of the 27 response (33%).
- The bottom of the grey diagonal shows that after making random predictions and sorting the predictions, it encounters only 3 1s (33%) after looking at 9 of the 27 responses (33%). It has to look at all 27 responses (100%) to encounter all 9 1s (100%).
- The gain curve encounters 5 1s (55%) after looking at 9 of the 27 responses (33%). It has to look at 14 responses to encounter all 9 1s (100%).

Another way to evaluate the predictive model is the ROC curve. It evaluates all possible thresholds for splitting predicted probabilities into predicted classes. This is often a much more useful metric than simply ranking models by their accuracy at a set threshold, as different models might require different calibration steps (looking at a confusion matrix at each step) to find the optimal classification threshold for that model.

```
library(caTools)
colAUC(m1$fitted.values, m1$data$REMISS, plotROC = TRUE)
```



```
## [ ,1]
## 0 vs. 1 0.8950617
```

4.2 Poisson Regression

Poisson models count data, like “traffic tickets per day”, or “website hits per day”. The response is an expected *rate* or intensity. For count data, specify the generalized model, this time with `family = poisson` or `family = quasipoisson`.

Recall that the probability of achieving a count y when the expected rate is λ is distributed

$$P(Y = y|\lambda) = \frac{e^{-\lambda}\lambda^y}{y!}.$$

The poisson regression model is

$$\lambda = \exp(X\beta).$$

You can solve this for y to get

$$y = X\beta = \ln(\lambda).$$

That is, the model predicts the log of the response rate. For a sample of size n , the likelihood function is

$$L(\beta; y, X) = \prod_{i=1}^n \frac{e^{-\exp(X_i\beta)} \exp(X_i\beta)^{y_i}}{y_i!}.$$

The log-likelihood is

$$l(\beta) = \sum_{i=1}^n (y_i X_i \beta - \sum_{i=1}^n \exp(X_i \beta) - \sum_{i=1}^n \log(y_i!)).$$

Maximizing the log-likelihood has no closed-form solution, so the coefficient estimates are found through iteratively reweighted least squares.

Poisson processes assume the variance of the response variable equals its mean. “Equals” means the mean and variance are of a similar order of magnitude. If that assumption does not hold, use the quasi-poisson. Use Poisson regression for large datasets. If the predicted counts are much greater than zero (>30), the linear regression will work fine. Whereas RMSE is not useful for logistic models, it is a good metric in Poisson.

Example

Dataset `fire` contains response variable `injuries` counting the number of injuries during the month and one explanatory variable, the month `mo`.

```
fire <- read_csv(file = "C:/Users/mpfol/OneDrive/Documents/Data Science/Data/CivilInjury_0.csv")

## Parsed with column specification:
## cols(
##   ID = col_double(),
##   `Injury Date` = col_datetime(format = ""),
##   `Total Injuries` = col_double()
## )

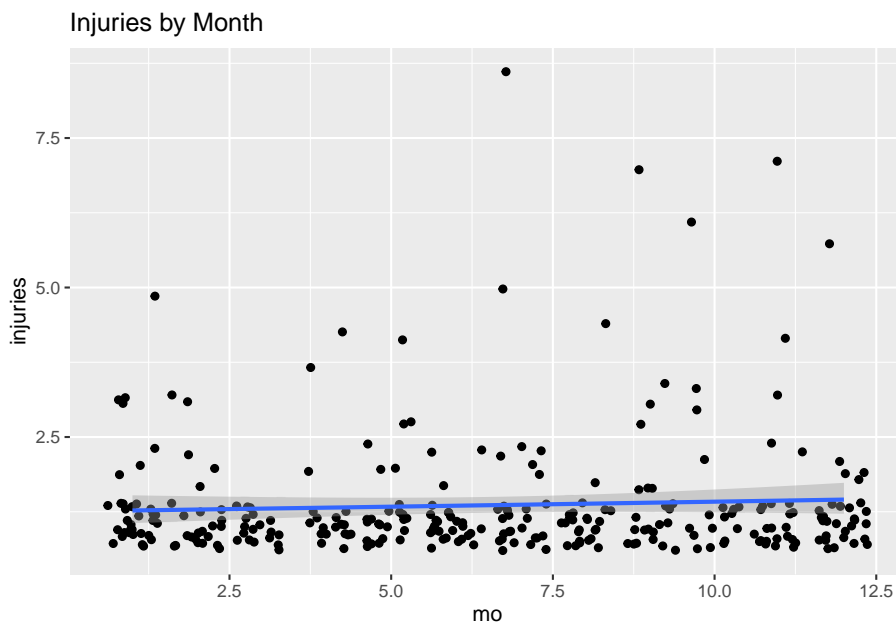
fire <- fire %>%
  mutate(mon = as.POSIXlt(`Injury Date`)$mon + 1) %>%
  rename(dt = `Injury Date`,
         injuries = `Total Injuries`)
str(fire)

## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 300 obs. of  4 variables:
## $ ID      : num  1 2 3 4 5 6 7 8 9 10 ...
```

```
## $ dt      : POSIXct, format: "2005-01-10" "2005-01-11" ...
## $ injuries: num  1 1 1 5 2 1 1 1 1 ...
## $ mo      : num  1 1 1 1 1 1 2 2 2 4 ...
```

In a situation like this where there the relationship is bivariate, start with a visualization.

```
ggplot(fire, aes(x = mo, y = injuries)) +
  geom_jitter() +
  geom_smooth(method = "glm", method.args = list(family = "poisson")) +
  labs(title = "Injuries by Month")
```



Fit a poisson regression in R using `glm(formula, data, family = poisson)`. But first, check whether the mean and variance of `injuries` are the same magnitude? If not, then use `family = quasipoisson`.

```
mean(fire$injuries)
```

```
## [1] 1.36
```

```
var(fire$injuries)
```

```
## [1] 1.020468
```

They are of the same magnitude, so fit the regression with `family = poisson`.

```
m2 <- glm(injuries ~ mo, family = poisson, data = fire)
summary(m2)

##
## Call:
## glm(formula = injuries ~ mo, family = poisson, data = fire)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -0.3987  -0.3473  -0.3034  -0.2502   4.3185
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.22805     0.10482   2.176  0.0296 *
## mo           0.01215     0.01397   0.870  0.3844
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##      Null deviance: 139.87  on 299  degrees of freedom
## Residual deviance: 139.11  on 298  degrees of freedom
## AIC: 792.08
##
## Number of Fisher Scoring iterations: 5
```

The *predicted* value \hat{y} is the estimated **log** of the response variable,

$$\hat{y} = X\hat{\beta} = \ln(\lambda).$$

Suppose `mo` is January (`mo = 1`), then the log of injuries' is $\hat{y} = 0.323787$. Or, more intuitively, the expected count of injuries is $\exp(0.323787) = 1.38$

```
predict(m2, newdata = data.frame(mo=1))

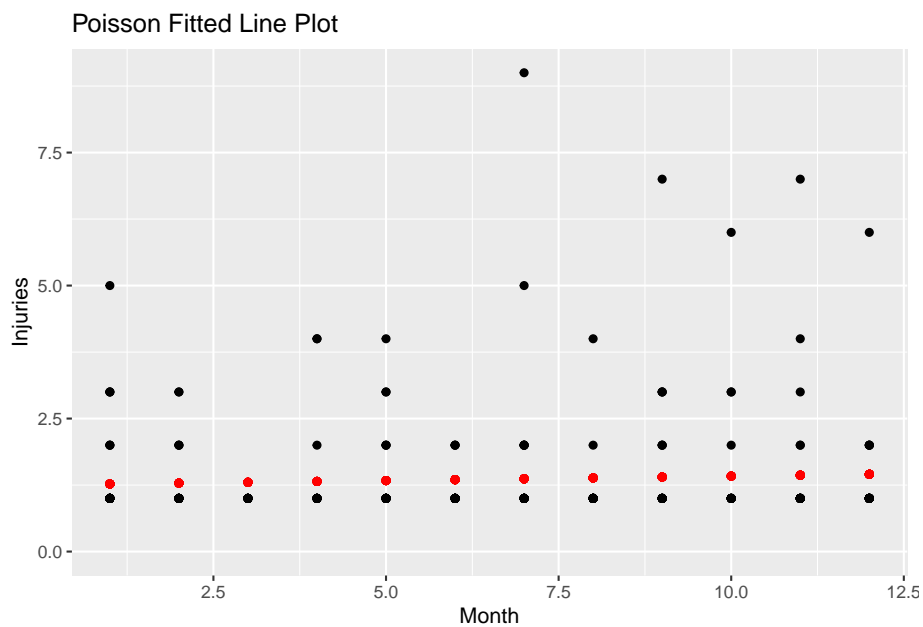
##      1
## 0.2401999

predict(m2, newdata = data.frame(mo=1), type = "response")

##      1
## 1.271503
```

Here is a plot of the predicted counts in red.

```
augment(m2, type.predict = "response") %>%
  ggplot(aes(x = mo, y = injuries)) +
  geom_point() +
  geom_point(aes(y = .fitted), color = "red") +
  scale_y_continuous(limits = c(0, NA)) +
  labs(x = "Month",
       y = "Injuries",
       title = "Poisson Fitted Line Plot")
```



Evaluate a logistic model fit with an analysis of deviance.

```
(perf <- glance(m2))
```

```
## # A tibble: 1 x 7
##   null.deviance df.null logLik   AIC   BIC deviance df.residual
##         <dbl>   <int> <dbl> <dbl> <dbl>   <dbl>       <int>
## 1         140.     299 -394.  792.  799.    139.        298
```

```
(pseudoR2 <- 1 - perf$deviance / perf$null.deviance)
```

```
## [1] 0.005413723
```

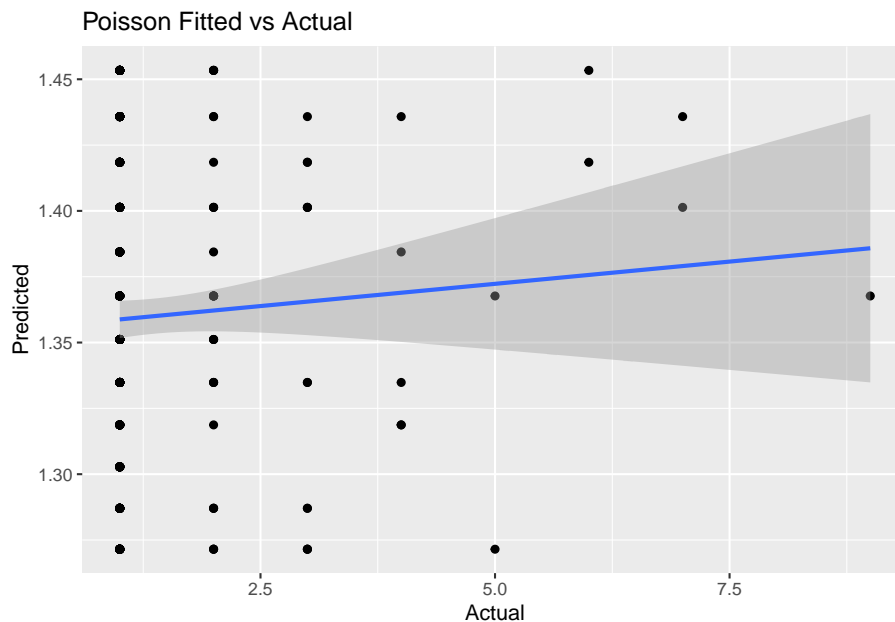
The deviance of the null model (no regressors) is 139.9. The deviance of the full model is 132.2. The psuedo-R2 is very low at .05. How about the RMSE?

```
RMSE(pred = predict(m2, type = "response"), obs = fire$injuries)
```

```
## [1] 1.006791
```

The average prediction error is about 0.99. That's almost as much as the variance of `injuries` - i.e., just predicting the mean of `injuries` would be almost as good! Use the `GainCurvePlot()` function to plot the gain curve.

```
augment(m2, type.predict = "response") %>%
  ggplot(aes(x = injuries, y = .fitted)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Actual",
       y = "Predicted",
       title = "Poisson Fitted vs Actual")
```



```
augment(m2) %>% data.frame() %>%
  GainCurvePlot(xvar = ".fitted", truthVar = "injuries", title = "Poisson Model")
```



It seems that `mo` was a poor predictor of `injuries`.

Chapter 5

Classification

Chapter 6

Classification

Chapter 7

Decision Trees

Decision trees, also known as classification and regression tree (CART) models, are tree-based methods for supervised machine learning. Simple *classification trees* and *regression trees* are easy to use and interpret, but are not competitive with the best machine learning methods. However, they form the foundation for **bagged trees**, **random forests**, and **boosted trees** models, which although less interpretable, are very accurate.

CART models segment the predictor space into K non-overlapping terminal nodes (leaves), A_1, A_2, \dots, A_K . Each node is described by a set of rules which can be used to predict new responses. The predicted value \hat{y} for each node is the mode (classification), or mean (regression).

CART models define the nodes through a *top-down greedy* process called *recursive binary splitting*. The process is *top-down* because it begins at the top of the tree with all observations in a single region and successively splits the predictor space. It is *greedy* because at each splitting step, the best split is made at that particular step without consideration to subsequent splits.

The best split is the predictor variable and cutpoint that minimizes a cost function. For a regression tree, the most common cost function is the sum of squared residuals,

$$RSS = \sum_{k=1}^K \sum_{i \in A_k} (y_i - \hat{y}_{A_k})^2.$$

For a classification tree, the most common cost functions are the Gini index,

$$G = \sum_{c=1}^C \hat{p}_{kc}(1 - \hat{p}_{kc}),$$

or the entropy

$$D = - \sum_{c=1}^C \hat{p}_{kc} \log \hat{p}_{kc}$$

where \hat{p}_{kc} is the proportion of training observations in node k node that are class c . A completely *pure* node in a binary tree will have $\hat{p} \in [0, 1]$ and $G = D = 0$. A completely impure node in a binary tree will have $\hat{p} = 0.5$ and $G = 0.5^2 \cdot 2 = 0.25$ and $D = -(0.5 \log(0.5)) \cdot 2 = 0.69$.

CART repeats the splitting process for each of the child nodes until a *stopping criterion* is satisfied, usually when no node size surpasses a predefined maximum, or continued splitting does not improve the model significantly. CART may also impose a minimum number of observations in each node.

The resulting tree likely over-fits the training data and therefore does not generalize well to test data, so CART *prunes* the tree, minimizing the cross-validated prediction error. Rather than cross-validating every possible subtree to find the one with minimum error, CART uses *cost-complexity pruning*. Cost-complexity is the tradeoff between error (cost) and tree size (complexity) where the tradeoff is quantified with cost-complexity parameter c_p . In the equation below, the cost complexity of the tree $R_{c_p}(T)$ is the sum of its risk (error) plus a “cost complexity” factor c_p multiple of the tree size $|T|$.

$$R_{c_p}(T) = R(T) + c_p|T|$$

c_p can take on any value from $[0, \infty]$, but it turns out there is an optimal tree for *ranges* of c_p values, so there are only a finite set of *interesting* values for c_p (James et al., 2013) (Therneau and Atkinson, 2019) (Kuhn and Johnson, 2016). A parametric algorithm identifies the interesting c_p values and their associated pruned trees, T_{c_p} .

CART uses cross-validation to determine which c_p is optimal.

7.1 Classification Tree

A simple classification tree is rarely performed on its own; the bagged, random forest, and gradient boosting methods build on this logic. However, it is good to start here to build understanding. I’ll learn by example. Using the `ISLR::OJ` data set, I will predict which brand of orange juice, Citrus Hill (CH) or Minute Maid = (MM), customers `Purchase` using from the 17 feature variables. Load the libraries and data.

```
library(ISLR) # OJ dataset
library(rpart) # classification and regression trees
library(caret) # modeling workflow
library(rpart.plot) # better formatted plots than the ones in rpart
library(plotROC) # ROC curves
library(ROCR)
library(tidyverse)
library(skimr) # neat alternative to glance & summary

oj_dat <- OJ
#skim_with(numeric = list(p0 = NULL, p25 = NULL, p50 = NULL, p75 = NULL,
#                          p100 = NULL, hist = NULL))
#skim(oj_dat)
```

I'll split `oj_dat` ($n = 1,070$) into `oj_train` (80%, $n = 857$) and `oj_test` (20%, $n = 213$). I'll fit a simple decision tree with `oj_train`, then later a bagged tree, a random forest, and a gradient boosting tree. I'll compare their predictive performance with `oj_test`.

```
set.seed(12345)
partition <- createDataPartition(y = oj_dat$Purchase, p = 0.8, list = FALSE)
oj_train <- oj_dat[partition, ]
oj_test <- oj_dat[-partition, ]
```

Function `rpart::rpart()` builds a full tree, minimizing the Gini index G by default (`parms = list(split = "gini")`), until the stopping criterion is satisfied. The default stopping criterion is

- only attempt a split if the current node has at least `minsplit = 20` observations,
- only accept a split if each of the two resulting nodes have at least `minbucket = round(minsplit/3)` observations, and
- only accept a split if the resulting overall fit improves by `cp = 0.01` (i.e., $\Delta G \leq 0.01$).

```
set.seed(123)
oj_model_1 <- rpart(
  formula = Purchase ~ .,
  data = oj_train,
  method = "class" # "class" for classification, "anova" for regression
)
print(oj_model_1)
```

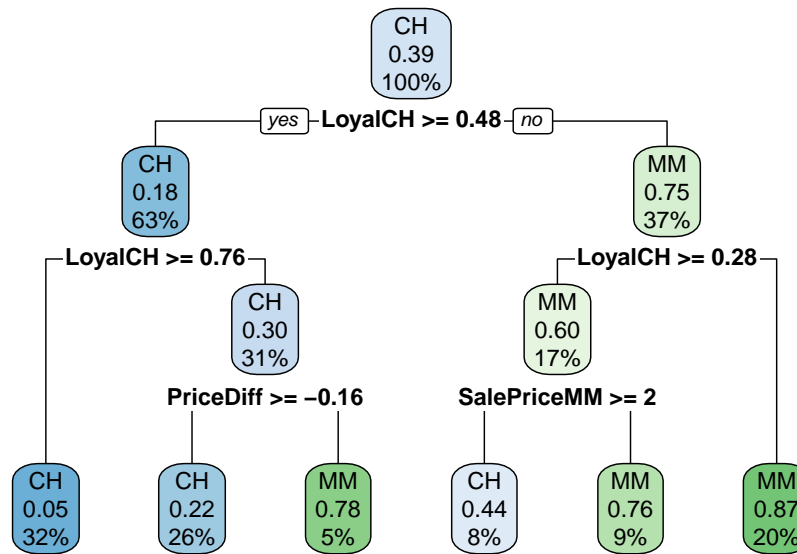
```
## n= 857
```

```
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
## 1) root 857 334 CH (0.61026838 0.38973162)
##    2) LoyalCH>=0.48285 537 94 CH (0.82495345 0.17504655)
##      4) LoyalCH>=0.7648795 271 13 CH (0.95202952 0.04797048) *
##      5) LoyalCH< 0.7648795 266 81 CH (0.69548872 0.30451128)
##        10) PriceDiff>=-0.165 226 50 CH (0.77876106 0.22123894) *
##        11) PriceDiff< -0.165 40 9 MM (0.22500000 0.77500000) *
##    3) LoyalCH< 0.48285 320 80 MM (0.25000000 0.75000000)
##      6) LoyalCH>=0.2761415 146 58 MM (0.39726027 0.60273973)
##        12) SalePriceMM>=2.04 71 31 CH (0.56338028 0.43661972) *
##        13) SalePriceMM< 2.04 75 18 MM (0.24000000 0.76000000) *
##      7) LoyalCH< 0.2761415 174 22 MM (0.12643678 0.87356322) *
```

The output starts with the root node. The predicted class at the root is **CH** and this prediction produces 334 errors on the 857 observations for a success rate of 0.61026838 and an error rate of 0.38973162. The child nodes of node “x” are labeled 2x) and 2x+1), so the child nodes of 1) are 2) and 3), and the child nodes of 2) are 4) and 5). Terminal nodes are labeled with an asterisk (*).

Surprisingly, only 3 of the 17 features were used the in full tree: **LoyalCH** (Customer brand loyalty for CH), **PriceDiff** (relative price of MM over CH), and **SalePriceMM** (absolute price of MM). The first split is at **LoyalCH** = 0.48285. Here is what the full (unpruned) tree looks like.

```
rpart.plot(oj_model_1, yesno = TRUE)
```



The boxes show the node classification (based on mode), the proportion of observations that are *not* CH, and the proportion of observations included in the node.

`rpart()` not only grew the full tree, it identified the set of cost complexity parameters, and measured the model performance of each corresponding tree using cross-validation. `printcp()` displays the candidate c_p values. You can use this table to decide how to prune the tree.

```
printcp(oj_model_1)
```

```
##
## Classification tree:
## rpart(formula = Purchase ~ ., data = oj_train, method = "class")
##
## Variables actually used in tree construction:
## [1] LoyalCH    PriceDiff  SalePriceMM
##
## Root node error: 334/857 = 0.38973
##
## n= 857
##
##      CP nsplit rel error  xerror   xstd
## 1 0.479042     0  1.00000 1.00000 0.042745
## 2 0.032934     1  0.52096 0.54192 0.035775
```

```
## 3 0.013473      3  0.45509 0.47006 0.033905
## 4 0.010000      5  0.42814 0.46407 0.033736
```

There are 4 c_p values in this model. The model with the smallest complexity parameter allows the most splits (`nsplit`). The highest complexity parameter corresponds to a tree with just a root node. `rel error` is the error rate relative to the root node. The root node absolute error is 0.38973162 (the proportion of MM), so its `rel error` is $0.38973162/0.38973162 = 1.0$. That means the absolute error of the full tree (at $CP = 0.01$) is $0.42814 * 0.38973162 = 0.1669$. You can verify that by calculating the error rate of the predicted values:

```
data.frame(pred = predict(oj_model_1, newdata = oj_train, type = "class")) %>%
  mutate(obs = oj_train$Purchase,
         err = if_else(pred != obs, 1, 0)) %>%
  summarize(mean_err = mean(err))
```

```
##      mean_err
## 1 0.1668611
```

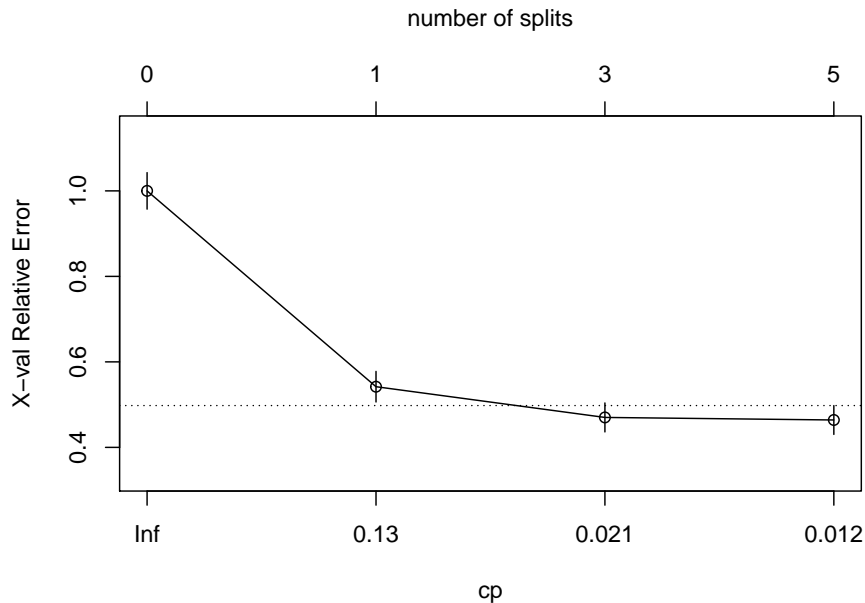
Finishing the CP table tour, `xerror` is the relative cross-validated error rate and `xstd` is its standard error. If you want the lowest possible error, then prune to the tree with the smallest relative CV error (`xerror`) ($c_p = 0.01$, CV error = 0.1809). If you want to balance predictive power with simplicity, prune to the smallest tree within 1 SE of the one with the smallest relative error. The CP table is not super-helpful for finding that tree. I'll add a column to find it.

```
oj_model_1$cptable %>%
  data.frame() %>%
  mutate(min_xerror_idx = which.min(oj_model_1$cptable[, "xerror"]),
         rownum = row_number(),
         xerror_cap = oj_model_1$cptable[min_xerror_idx, "xerror"] +
           oj_model_1$cptable[min_xerror_idx, "xstd"],
         eval = case_when(rownum == min_xerror_idx ~ "min xerror",
                          xerror < xerror_cap ~ "under cap",
                          TRUE ~ "")) %>%
  select(-rownum, -min_xerror_idx)
```

```
##      CP nsplit rel.error   xerror   xstd xerror_cap      eval
## 1 0.47904192    0 1.0000000 1.0000000 0.04274518  0.4978082
## 2 0.03293413    1 0.5209581 0.5419162 0.03577468  0.4978082
## 3 0.01347305    3 0.4550898 0.4700599 0.03390486  0.4978082 under cap
## 4 0.01000000    5 0.4281437 0.4640719 0.03373631  0.4978082 min xerror
```

The simplest tree using the 1-SE rule is $c_p = 0.01347305$, CV error = 0.1832). Fortunately, `plotcp()` presents a nice graphical representation of the relationship between `xerror` and `cp`.


```
plotcp(oj_model_1, upper = "splits")
```



The dashed line is set at the minimum `xerror` + `xstd`. The top axis shows the number of splits in the tree. I'm not sure why the CP values are not the same as in the table (they are close, but not the same). The figure suggests I should prune to 5 or 3 splits. I see this curve never really hits a minimum - it is still decreasing at 5 splits. The default tuning parameter value `cp = 0.01` may be too small, so I'll set it to `cp = 0.001` and start over.

```
set.seed(123)
oj_model_1b <- rpart(
  formula = Purchase ~ .,
  data = oj_train,
  method = "class",
  cp = 0.001
)
print(oj_model_1b)
```

```
## n= 857
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
##  1) root 857 334 CH (0.61026838 0.38973162)
```

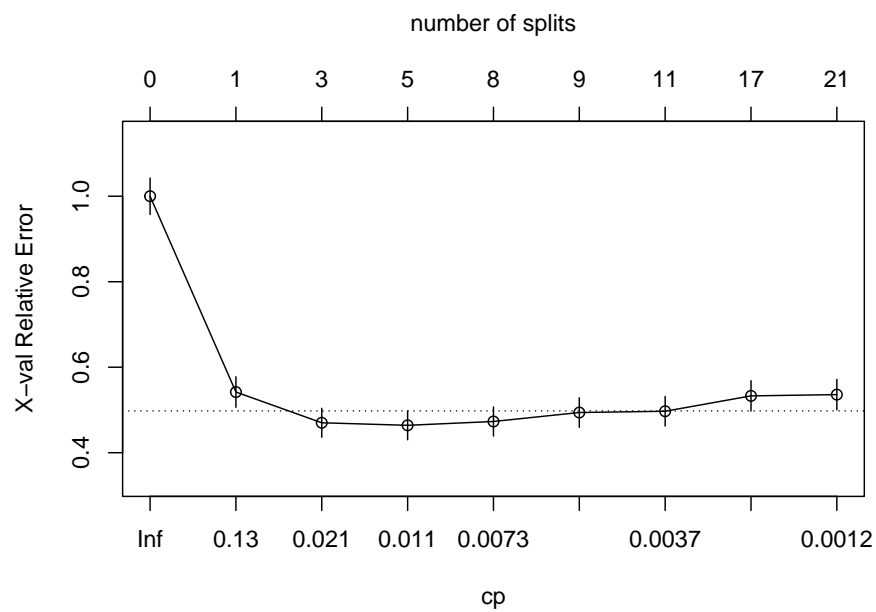
```

##      2) LoyalCH>=0.48285 537  94 CH (0.82495345 0.17504655)
##      4) LoyalCH>=0.7648795 271  13 CH (0.95202952 0.04797048) *
##      5) LoyalCH< 0.7648795 266  81 CH (0.69548872 0.30451128)
##      10) PriceDiff>=-0.165 226  50 CH (0.77876106 0.22123894)
##      20) ListPriceDiff>=0.255 115  11 CH (0.90434783 0.09565217) *
##      21) ListPriceDiff< 0.255 111  39 CH (0.64864865 0.35135135)
##      42) PriceMM>=2.155 19   2 CH (0.89473684 0.10526316) *
##      43) PriceMM< 2.155 92  37 CH (0.59782609 0.40217391)
##      86) DiscCH>=0.115 7    0 CH (1.00000000 0.00000000) *
##      87) DiscCH< 0.115 85  37 CH (0.56470588 0.43529412)
##      174) ListPriceDiff>=0.215 45  15 CH (0.66666667 0.33333333) *
##      175) ListPriceDiff< 0.215 40  18 MM (0.45000000 0.55000000)
##      350) LoyalCH>=0.527571 28  13 CH (0.53571429 0.46428571)
##      700) WeekofPurchase< 266.5 21   8 CH (0.61904762 0.38095238) *
##      701) WeekofPurchase>=266.5 7    2 MM (0.28571429 0.71428571) *
##      351) LoyalCH< 0.527571 12   3 MM (0.25000000 0.75000000) *
##      11) PriceDiff< -0.165 40   9 MM (0.22500000 0.77500000) *
##      3) LoyalCH< 0.48285 320  80 MM (0.25000000 0.75000000)
##      6) LoyalCH>=0.2761415 146  58 MM (0.39726027 0.60273973)
##      12) SalePriceMM>=2.04 71  31 CH (0.56338028 0.43661972)
##      24) LoyalCH< 0.303104 7    0 CH (1.00000000 0.00000000) *
##      25) LoyalCH>=0.303104 64  31 CH (0.51562500 0.48437500)
##      50) WeekofPurchase>=246.5 52  22 CH (0.57692308 0.42307692)
##      100) PriceCH< 1.94 35  11 CH (0.68571429 0.31428571)
##      200) StoreID< 1.5 9    1 CH (0.88888889 0.11111111) *
##      201) StoreID>=1.5 26  10 CH (0.61538462 0.38461538)
##      402) LoyalCH< 0.410969 17   4 CH (0.76470588 0.23529412) *
##      403) LoyalCH>=0.410969 9    3 MM (0.33333333 0.66666667) *
##      101) PriceCH>=1.94 17   6 MM (0.35294118 0.64705882) *
##      51) WeekofPurchase< 246.5 12   3 MM (0.25000000 0.75000000) *
##      13) SalePriceMM< 2.04 75  18 MM (0.24000000 0.76000000)
##      26) SpecialCH>=0.5 14   6 CH (0.57142857 0.42857143) *
##      27) SpecialCH< 0.5 61  10 MM (0.16393443 0.83606557) *
##      7) LoyalCH< 0.2761415 174  22 MM (0.12643678 0.87356322)
##      14) LoyalCH>=0.035047 117  21 MM (0.17948718 0.82051282)
##      28) WeekofPurchase< 273.5 104  21 MM (0.20192308 0.79807692)
##      56) PriceCH>=1.875 20   9 MM (0.45000000 0.55000000)
##      112) WeekofPurchase>=252.5 12   5 CH (0.58333333 0.41666667) *
##      113) WeekofPurchase< 252.5 8    2 MM (0.25000000 0.75000000) *
##      57) PriceCH< 1.875 84  12 MM (0.14285714 0.85714286) *
##      29) WeekofPurchase>=273.5 13   0 MM (0.00000000 1.00000000) *
##      15) LoyalCH< 0.035047 57   1 MM (0.01754386 0.98245614) *

```

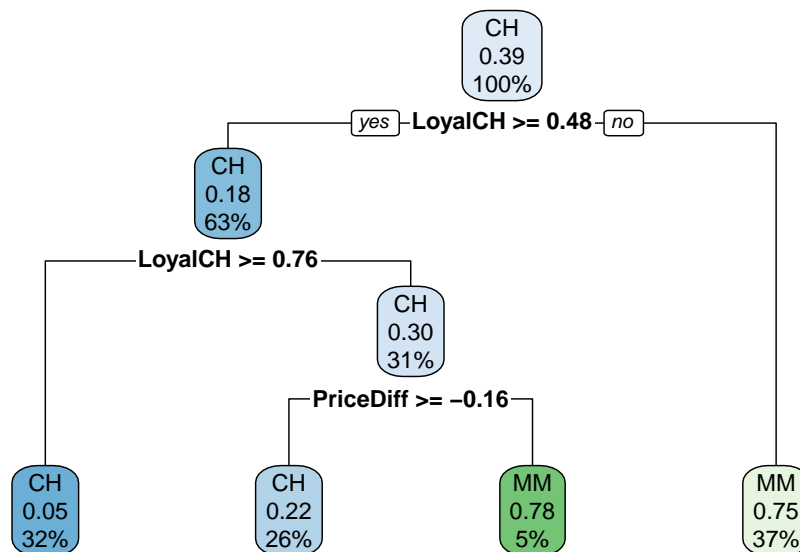
This is a much larger tree. Did I find a `cp` value that produces a local min?

```
plotcp(oj_model_1b, upper = "splits")
```



Yes, the min is at $CP = 0.011$ with 5 splits. The min + 1 SE is at $CP = 0.021$ with 3 splits. I'll prune the tree to 3 splits.

```
oj_model_1b_pruned <- prune(
  oj_model_1b,
  cp = oj_model_1b$cptable[oj_model_1b$cptable[, 2] == 3, "CP"]
)
rpart.plot(oj_model_1b_pruned, yesno = TRUE)
```



The most “important” indicator of `Purchase` appears to be `LoyalCH`. From the `rpart` vignette (page 12),

“An overall measure of variable importance is the sum of the goodness of split measures for each split for which it was the primary variable, plus goodness (adjusted agreement) for all splits in which it was a surrogate.”

Surrogates refer to alternative features for a node to handle missing data. For each split, CART evaluates a variety of alternative “surrogate” splits to use when the feature value for the primary split is NA. Surrogate splits are splits that produce results similar to the original split.

A variable’s importance is the sum of the improvement in the overall Gini (or RMSE) measure produced by the nodes in which it appears. Here is the variable importance for this model.

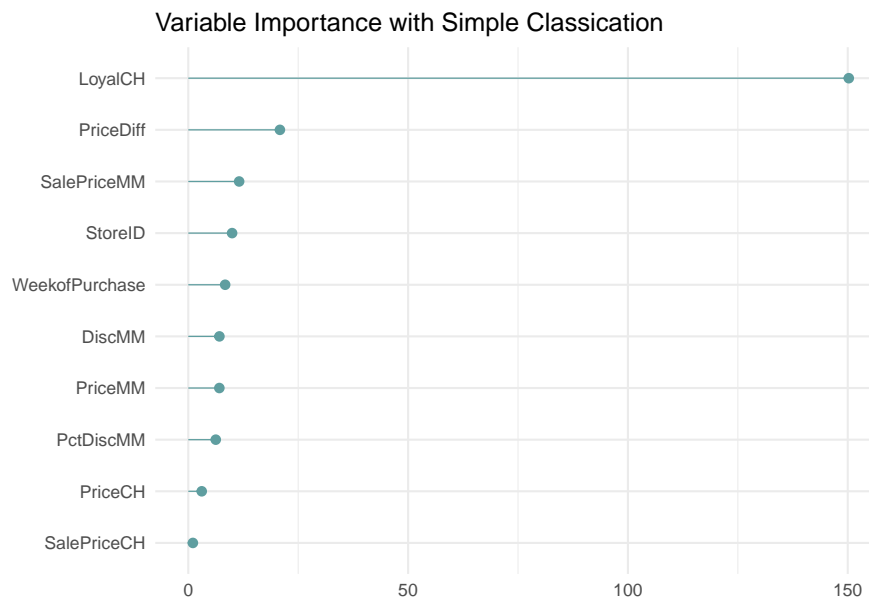
```
oj_model_1b_pruned$variable.importance
```

##	LoyalCH	PriceDiff	SalePriceMM	StoreID	WeekofPurchase
##	150.237336	20.843067	11.567443	9.965419	8.386282
##	DiscMM	PriceMM	PctDiscMM	PriceCH	SalePriceCH
##	7.081470	7.065493	6.252920	3.055594	1.042153

```

oj_model_1b_pruned$variable.importance %>%
  data.frame() %>%
  rownames_to_column(var = "Feature") %>%
  rename(Overall = '.') %>%
  ggplot(aes(x = fct_reorder(Feature, Overall), y = Overall)) +
  geom_pointrange(aes(ymin = 0, ymax = Overall), color = "cadetblue", size = .3) +
  theme_minimal() +
  coord_flip() +
  labs(x = "", y = "", title = "Variable Importance with Simple Classification")

```



LoyalCH is by far the most important variable, as expected from its position at the top of the tree, and one level down.

You can see how the surrogates appear in the model with the `summary()` function.

```
summary(oj_model_1b_pruned)
```

```

## Call:
## rpart(formula = Purchase ~ ., data = oj_train, method = "class",
##       cp = 0.001)
##   n= 857
##
##           CP nsplit rel error   xerror   xstd

```

```

## 1 0.47904192      0 1.0000000 1.0000000 0.04274518
## 2 0.03293413      1 0.5209581 0.5419162 0.03577468
## 3 0.01347305      3 0.4550898 0.4700599 0.03390486
##
## Variable importance
##      LoyalCH      PriceDiff      SalePriceMM      StoreID      WeekofPurchase
##           67           9           5           4           4
##      DiscMM      PriceMM      PctDiscMM      PriceCH
##           3           3           3           1
##
## Node number 1: 857 observations,      complexity param=0.4790419
##   predicted class=CH   expected loss=0.3897316   P(node) =1
##   class counts:      523      334
##   probabilities: 0.610 0.390
##   left son=2 (537 obs) right son=3 (320 obs)
##   Primary splits:
##     LoyalCH      < 0.48285   to the right, improve=132.56800, (0 missing)
##     StoreID      < 3.5       to the right, improve= 40.12097, (0 missing)
##     PriceDiff    < 0.015     to the right, improve= 24.26552, (0 missing)
##     ListPriceDiff < 0.255     to the right, improve= 22.79117, (0 missing)
##     SalePriceMM  < 1.84      to the right, improve= 20.16447, (0 missing)
##   Surrogate splits:
##     StoreID      < 3.5       to the right, agree=0.646, adj=0.053, (0 split)
##     PriceMM      < 1.89      to the right, agree=0.638, adj=0.031, (0 split)
##     WeekofPurchase < 229.5   to the right, agree=0.632, adj=0.016, (0 split)
##     DiscMM       < 0.77      to the left,  agree=0.629, adj=0.006, (0 split)
##     SalePriceMM  < 1.385     to the right, agree=0.629, adj=0.006, (0 split)
##
## Node number 2: 537 observations,      complexity param=0.03293413
##   predicted class=CH   expected loss=0.1750466   P(node) =0.6266044
##   class counts:      443      94
##   probabilities: 0.825 0.175
##   left son=4 (271 obs) right son=5 (266 obs)
##   Primary splits:
##     LoyalCH      < 0.7648795 to the right, improve=17.669310, (0 missing)
##     PriceDiff    < 0.015     to the right, improve=15.475200, (0 missing)
##     SalePriceMM  < 1.84      to the right, improve=13.951730, (0 missing)
##     ListPriceDiff < 0.255     to the right, improve=11.407560, (0 missing)
##     DiscMM       < 0.15      to the left,  improve= 7.795122, (0 missing)
##   Surrogate splits:
##     WeekofPurchase < 257.5   to the right, agree=0.594, adj=0.180, (0 split)
##     PriceCH       < 1.775     to the right, agree=0.590, adj=0.173, (0 split)
##     StoreID       < 3.5       to the right, agree=0.587, adj=0.165, (0 split)
##     PriceMM       < 2.04      to the right, agree=0.587, adj=0.165, (0 split)
##     SalePriceMM  < 2.04      to the right, agree=0.587, adj=0.165, (0 split)
##

```

```
## Node number 3: 320 observations
## predicted class=MM expected loss=0.25 P(node) =0.3733956
## class counts: 80 240
## probabilities: 0.250 0.750
##
## Node number 4: 271 observations
## predicted class=CH expected loss=0.04797048 P(node) =0.3162194
## class counts: 258 13
## probabilities: 0.952 0.048
##
## Node number 5: 266 observations, complexity param=0.03293413
## predicted class=CH expected loss=0.3045113 P(node) =0.3103851
## class counts: 185 81
## probabilities: 0.695 0.305
## left son=10 (226 obs) right son=11 (40 obs)
## Primary splits:
## PriceDiff < -0.165 to the right, improve=20.84307, (0 missing)
## ListPriceDiff < 0.235 to the right, improve=20.82404, (0 missing)
## SalePriceMM < 1.84 to the right, improve=16.80587, (0 missing)
## DiscMM < 0.15 to the left, improve=10.05120, (0 missing)
## PctDiscMM < 0.0729725 to the left, improve=10.05120, (0 missing)
## Surrogate splits:
## SalePriceMM < 1.585 to the right, agree=0.906, adj=0.375, (0 split)
## DiscMM < 0.57 to the left, agree=0.895, adj=0.300, (0 split)
## PctDiscMM < 0.264375 to the left, agree=0.895, adj=0.300, (0 split)
## WeekofPurchase < 274.5 to the left, agree=0.872, adj=0.150, (0 split)
## SalePriceCH < 2.075 to the left, agree=0.857, adj=0.050, (0 split)
##
## Node number 10: 226 observations
## predicted class=CH expected loss=0.2212389 P(node) =0.2637106
## class counts: 176 50
## probabilities: 0.779 0.221
##
## Node number 11: 40 observations
## predicted class=MM expected loss=0.225 P(node) =0.04667445
## class counts: 9 31
## probabilities: 0.225 0.775
```

The last step is to make predictions on the validation data set. For a classification tree, set argument `type = "class"`.

```
oj_model_1b_preds <- predict(oj_model_1b_pruned, oj_test, type = "class")
```

I'll evaluate the predictions and record the accuracy (correct classification percentage) for comparison to other models. Two ways to evaluate the model are the confusion matrix, and the ROC curve.

7.1.1 Confusion Matrix

Print the confusion matrix with `caret::confusionMatrix()` to see how well does this model performs against the test data set.

```
oj_model_1b_cm <- confusionMatrix(data = oj_model_1b_preds, reference = oj_test$Purchase,
oj_model_1b_cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##           CH 113  13
##           MM  17  70
##
##           Accuracy : 0.8592
##           95% CI : (0.8051, 0.9029)
##    No Information Rate : 0.6103
##    P-Value [Acc > NIR] : 1.265e-15
##
##           Kappa : 0.7064
##
##    McNemar's Test P-Value : 0.5839
##
##           Sensitivity : 0.8692
##           Specificity : 0.8434
##           Pos Pred Value : 0.8968
##           Neg Pred Value : 0.8046
##           Prevalence : 0.6103
##           Detection Rate : 0.5305
##           Detection Prevalence : 0.5915
##           Balanced Accuracy : 0.8563
##
##           'Positive' Class : CH
##
```

The confusion matrix is at the top. It also includes a lot of statistics. It's worth getting familiar with the stats. The model accuracy and 95% CI are calculated from the binomial test.

```
binom.test(x = 113 + 70, n = 213)
```

```
##
## Exact binomial test
```



```
##
## data: 113 + 70 and 213
## number of successes = 183, number of trials = 213, p-value < 2.2e-16
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
## 0.8050785 0.9029123
## sample estimates:
## probability of success
## 0.8591549
```

The “No Information Rate” (NIR) statistic is the class rate for the largest class. In this case CH is the largest class, so $NIR = 130/213 = 0.6103$. “P-Value [Acc > NIR]” is the binomial test that the model accuracy is significantly better than the NIR (i.e., significantly better than just always guessing CH).

```
binom.test(x = 113 + 70, n = 213, p = 130/213, alternative = "greater")
```

```
##
## Exact binomial test
##
## data: 113 + 70 and 213
## number of successes = 183, number of trials = 213, p-value = 1.265e-15
## alternative hypothesis: true probability of success is greater than 0.6103286
## 95 percent confidence interval:
## 0.8138446 1.0000000
## sample estimates:
## probability of success
## 0.8591549
```

The “Accuracy” statistic indicates the model predicts 0.8590 of the observations correctly. That’s good, but less impressive when you consider the prevalence of CH is 0.6103 - you could achieve 61% accuracy just by predicting CH every time. A measure that controls for the prevalence is Cohen’s kappa statistic. The kappa statistic is explained here. It compares the accuracy to the accuracy of a “random system”. It is defined as

$$\kappa = \frac{Acc - RA}{1 - RA}$$

where

$$RA = \frac{ActFalse \times PredFalse + ActTrue \times PredTrue}{Total \times Total}$$

is the hypotheical probability of a chance agreement. `ActFalse` will be the number of “MM” ($13 + 70 = 83$) and actual true will be the number of “CH” ($113 + 17 = 130$). The predicted counts are

```
table(oj_model_1b_preds)
```

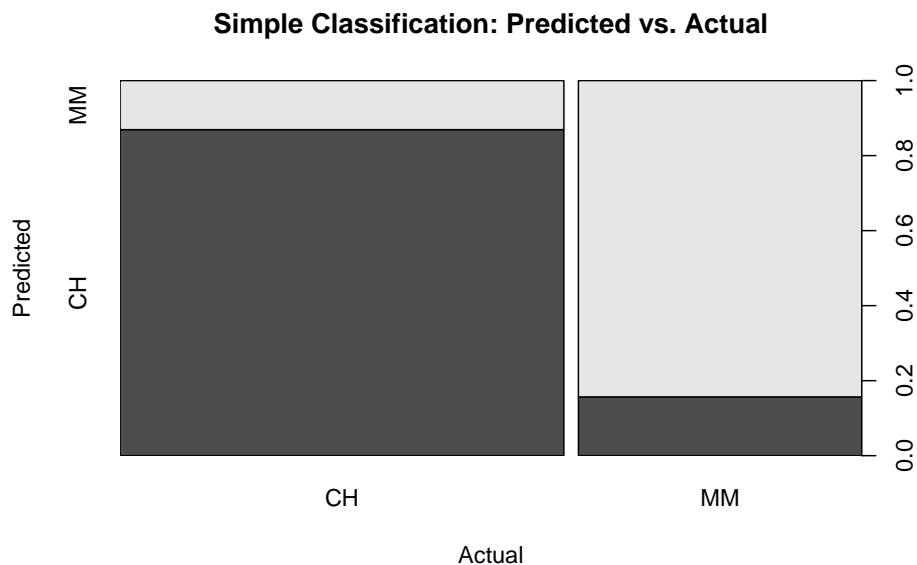
```
## oj_model_1b_preds
##  CH  MM
## 126  87
```

So, $RA = (83 * 87 + 130 * 126) / 213^2 = 0.5202$ and $\kappa = (0.8592 - 0.5202) / (1 - 0.5202) = 0.7064$. The kappa statistic varies from 0 to 1 where 0 means accurate predictions occur merely by chance, and 1 means the predictions are in perfect agreement with the observations. In this case, a kappa statistic of 0.7064 is “substantial”. See chart here.

The other measures from the `confusionMatrix()` output are various proportions and you can remind yourself of their definitions in the documentation with `?confusionMatrix`.

Visuals are almost always helpful. Here is a plot of the confusion matrix.

```
plot(oj_test$Purchase, oj_model_1b_preds,
     main = "Simple Classification: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
```



By the way, how does the validation set accuracy ()

```
oj_model_1b_train_preds <- predict(oj_model_1b_pruned, oj_train, type = "class")
oj_model_1b_train_cm <- confusionMatrix(data = oj_model_1b_train_preds, reference = oj_train$Purchase,
oj_model_1b_train_cm$overall
```

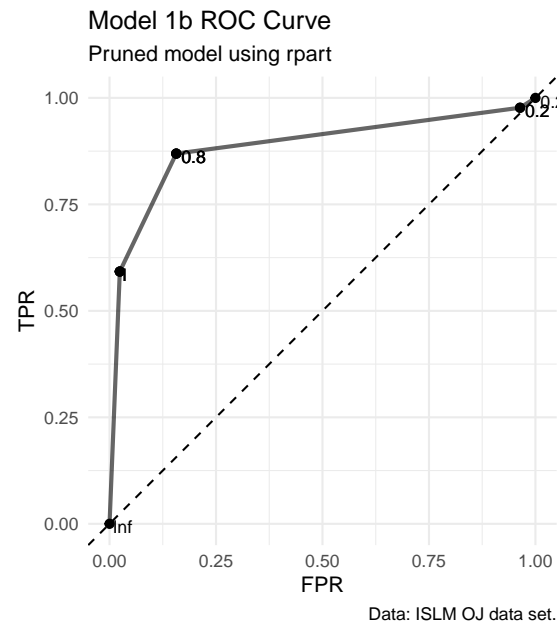
```
##      Accuracy      Kappa AccuracyLower AccuracyUpper AccuracyNull
## 8.226371e-01 6.323113e-01 7.953840e-01 8.476497e-01 6.102684e-01
## AccuracyPValue McNemarPValue
## 1.859617e-41 4.258396e-02
```

The accuracy on the training data set was a little lower than on the test data set. I thought it would be higher, not lower.

7.1.2 ROC Curve

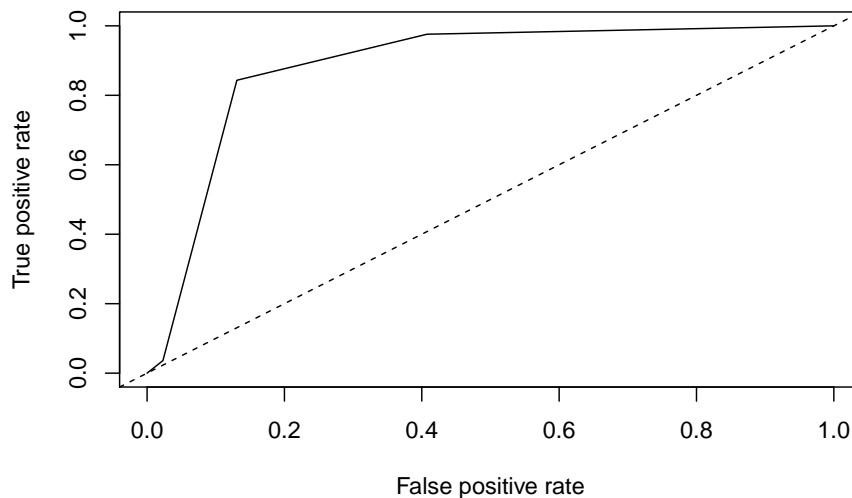
Another measure of accuracy is the ROC (receiver operating characteristics) curve (Fawcett, 2005). The ROC curve is a plot of the true positive rate (TPR, sensitivity) versus the false positive rate (FPR, 1 - specificity) for a set of thresholds. By default, the threshold for predicting the default classification is 0.50, but it could be any threshold. The ROC curves varies the thresholds. (I'll use the `geom_roc` geom from **plotROC**).

```
data.frame(M = predict(oj_model_1b_pruned, oj_test, "prob")[, 1],
           D = if_else(oj_test$Purchase == "CH", 1, 0)) %>%
  ggplot() +
  geom_roc(aes(m = M, d = D), hjust = -0.4, vjust = 1.5, linealpha = 0.6, labelsize = 3, n.cuts = 10) +
  geom_abline(intercept = 0, slope = 1, linetype = 2) +
  coord_equal() +
  theme_minimal() +
  labs(x = "FPR", y = "TPR",
       title = "Model 1b ROC Curve",
       subtitle = "Pruned model using rpart",
       caption = "Data: ISLM OJ data set.")
```



You can also use `prediction()` and `plot.prediction()` from the **ROCR** package.

```
pred <- prediction(predict(oj_model_1b_pruned, newdata = oj_test, type = "prob"), 2),
plot(performance(pred, "tpr", "fpr"))
abline(0, 1, lty = 2)
```



Hmm, not quite the same...

A few points on the ROC space are helpful for understanding how to use it.

- The lower left point (0, 0) is the result of *always* predicting “negative” or in this case “MM” if “CH” is taken as the default class. Sure, your false positive rate is zero, but since you never predict a positive, your true positive rate is also zero.
- The upper right point (1, 1) is the results of *always* predicting “positive” (or “CH” here). You catch all the positives, but you miss all the negatives.
- The upper left point (0, 1) is the result of perfect accuracy. You catch all the positives and all the negatives.
- The lower right point (1, 0) is the result of perfect imbecility. You made the exact wrong prediction every time.
- The 45 degree diagonal is the result of randomly guessing positive (CH) X percent of the time. If you guess positive 90% of the time and the prevalence is 50%, your TPR will be 90% and your FPR will also be 90%, etc.

From the last bullet, it is evident that any point below and to the right of the 45 degree diagonal represents an instance where the model would have been better off just predicting entirely one way or the other. The goal is for all nodes to bunch up in the upper left.

Points to the left of the diagonal with a low TPR can be thought of as “conservative” predictors - they only make positive (CH) predictions with strong

evidence. Points to the left of the diagonal with a high TPR can be thought of as “liberal” predictors - they make positive (CH) predictions with weak evidence.

7.1.3 Caret Approach

I can also fit the model with `caret::train()`. There are two ways to tune hyperparameters in `train()`:

- set the number of tuning parameter values to consider by setting `tuneLength`, or
- set particular values to consider for each parameter by defining a `tuneGrid`.

I’ll build the model using 10-fold cross-validation to optimize the hyperparameter CP. If you don’t have any idea what the tuning parameter ought to look like, use `tuneLength` to get close, then fine-tune with `tuneGrid`. That’s what I’ll do. I’ll create a training control object that I can re-use in other model builds.

```
oj_trControl = trainControl(
  method = "cv", # k-fold cross validation
  number = 10, # 10 folds
  savePredictions = "final", # save predictions for the optimal tuning parameter
  classProbs = TRUE # return class probabilities in addition to predicted values
# summaryFunction = twoClassSummary # computes sensitivity, specificity and the area
)
```

Now fit the model.

```
set.seed(1234)
oj_model_2 = train(
  Purchase ~ .,
  data = oj_train,
  method = "rpart",
  tuneLength = 5,
  metric = "Accuracy",
  trControl = oj_trControl
)
```

`caret` built a full tree using `rpart`’s default parameters: gini splitting index, at least 20 observations in a node in order to consider splitting it, and at least 6 observations in each node. Caret then calculated the accuracy for each candidate value of α . Here is the results.

```
print(oj_model_2)

## CART
##
## 857 samples
## 17 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 772, 772, 771, 770, 771, 771, ...
## Resampling results across tuning parameters:
##
##      cp          Accuracy   Kappa
## 0.005988024 0.8085999 0.5931149
## 0.008982036 0.8086267 0.5943277
## 0.013473054 0.8051657 0.5885521
## 0.032934132 0.7841798 0.5371171
## 0.479041916 0.6603904 0.1774773
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.008982036.
```

The second cp (0.008982036) produced the highest accuracy. I can drill into the best value of cp using a tuning grid. I'll try that now.

```
set.seed(1234)
oj_model_3 = train(
  Purchase ~ .,
  data = oj_train,
  method = "rpart",
  tuneGrid = expand.grid(cp = seq(from = 0.001, to = 0.010, length = 11)),
  metric = 'Accuracy',
  trControl = oj_trControl
)
print(oj_model_3)
```

```
## CART
##
## 857 samples
## 17 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
```

```
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 772, 772, 771, 770, 771, 771, ...
## Resampling results across tuning parameters:
##
##   cp      Accuracy   Kappa
##   0.0010  0.8004874  0.5753480
##   0.0019  0.8016502  0.5785232
##   0.0028  0.8039758  0.5845653
##   0.0037  0.8085999  0.5955198
##   0.0046  0.8039351  0.5851273
##   0.0055  0.8085863  0.5937949
##   0.0064  0.8085999  0.5931149
##   0.0073  0.8120883  0.6011446
##   0.0082  0.8120883  0.6011446
##   0.0091  0.8086267  0.5943277
##   0.0100  0.8086540  0.5953150
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.0082.
```

The beset model is at $cp = 0.009$. Here are the rules in the final model.

```
oj_model_3$finalModel
```

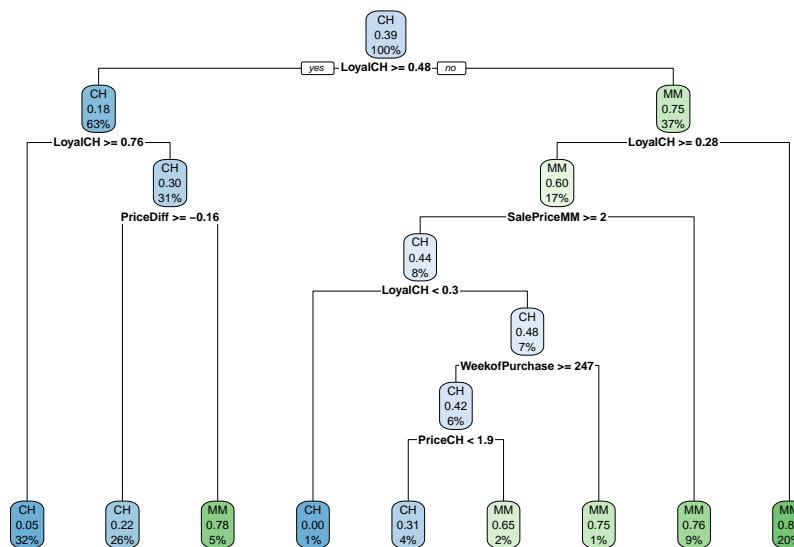
```
## n= 857
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
## 1) root 857 334 CH (0.61026838 0.38973162)
##    2) LoyalCH>=0.48285 537 94 CH (0.82495345 0.17504655)
##      4) LoyalCH>=0.7648795 271 13 CH (0.95202952 0.04797048) *
##      5) LoyalCH< 0.7648795 266 81 CH (0.69548872 0.30451128)
##        10) PriceDiff>=-0.165 226 50 CH (0.77876106 0.22123894) *
##        11) PriceDiff< -0.165 40 9 MM (0.22500000 0.77500000) *
##    3) LoyalCH< 0.48285 320 80 MM (0.25000000 0.75000000)
##      6) LoyalCH>=0.2761415 146 58 MM (0.39726027 0.60273973)
##        12) SalePriceMM>=2.04 71 31 CH (0.56338028 0.43661972)
##          24) LoyalCH< 0.303104 7 0 CH (1.00000000 0.00000000) *
##          25) LoyalCH>=0.303104 64 31 CH (0.51562500 0.48437500)
##            50) WeekofPurchase>=246.5 52 22 CH (0.57692308 0.42307692)
##              100) PriceCH< 1.94 35 11 CH (0.68571429 0.31428571) *
##              101) PriceCH>=1.94 17 6 MM (0.35294118 0.64705882) *
##            51) WeekofPurchase< 246.5 12 3 MM (0.25000000 0.75000000) *
##    13) SalePriceMM< 2.04 75 18 MM (0.24000000 0.76000000) *
```



```
##          7) LoyalCH< 0.2761415 174 22 MM (0.12643678 0.87356322) *
```

Here is the tree.

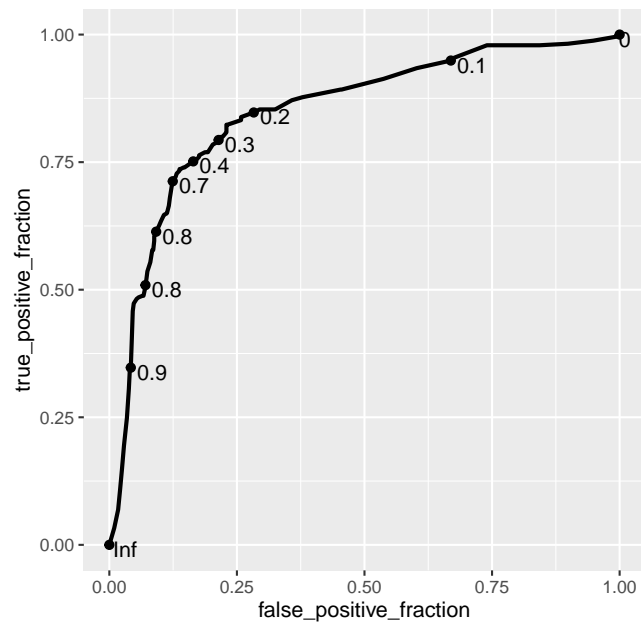
```
rpart.plot(oj_model_3$finalModel)
```



Here is the ROC curve.

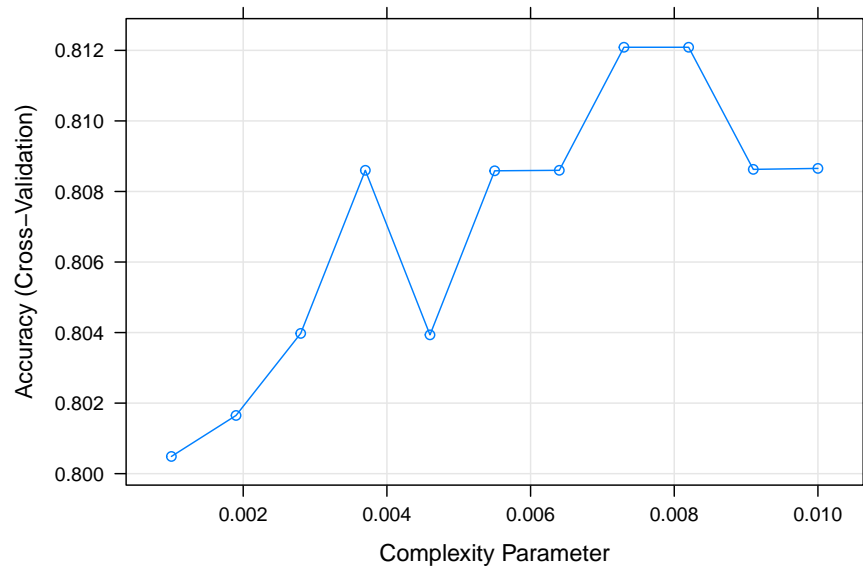
```
library(plotROC)
ggplot(oj_model_3$pred) +
  geom_roc(
    aes(
      m = MM,
      d = factor(obs, levels = c("CH", "MM"))
    ),
    hjust = -0.4, vjust = 1.5
  ) +
  coord_equal()
```

```
## Warning in verify_d(data$d): D not labeled 0/1, assuming CH = 0 and MM = 1!
```



Here are the cross-validated Accuracy for each candidate cp value.

```
plot(oj_model_3)
```



Evaluate the model by making predictions with the test data set.

```
oj_model_3_preds <- predict(oj_model_3, oj_test, type = "raw")
```

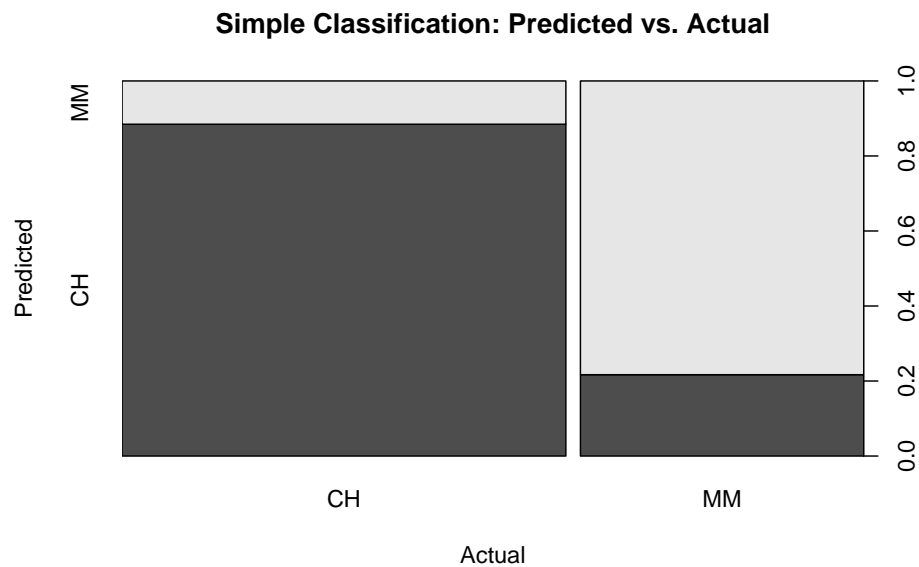
The confusion matrix shows the true positives and true negatives.

```
oj_model_3_cm <- confusionMatrix(
  data = oj_model_3_preds,
  reference = oj_test$Purchase
)
oj_model_3_cm

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##           CH 115  18
##           MM  15  65
##
##              Accuracy : 0.8451
##              95% CI : (0.7894, 0.8909)
##    No Information Rate : 0.6103
##    P-Value [Acc > NIR] : 6.311e-14
##
##              Kappa : 0.6721
##
##  Mcnemar's Test P-Value : 0.7277
##
##              Sensitivity : 0.8846
##              Specificity : 0.7831
##              Pos Pred Value : 0.8647
##              Neg Pred Value : 0.8125
##              Prevalence : 0.6103
##              Detection Rate : 0.5399
##    Detection Prevalence : 0.6244
##              Balanced Accuracy : 0.8339
##
##              'Positive' Class : CH
##
```

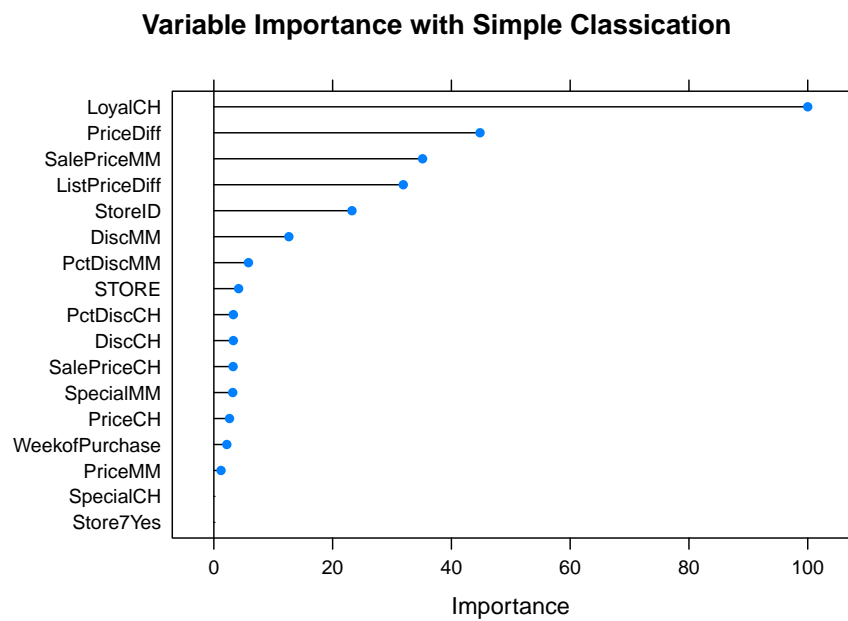
The accuracy metric is the slightly worse than in my previous model. Here is a graphical representation of the confusion matrix.

```
plot(oj_test$Purchase, oj_model_3_preds,
     main = "Simple Classification: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
```



Finally, here is the variable importance plot.

```
plot(varImp(oj_model_3), main="Variable Importance with Simple Classification")
```



Looks like the manual effort faired best. Here is a summary the accuracy rates of the three models.

```
rbind(data.frame(model = "Manual Class", Acc = round(oj_model_1b_cm$overall["Accuracy"], 5)),
      data.frame(model = "Caret w/tuneGrid", Acc = round(oj_model_3_cm$overall["Accuracy"], 5))
)
```

```
##              model      Acc
## Accuracy      Manual Class 0.85915
## Accuracy1 Caret w/tuneGrid 0.84507
```

7.2 Regression Trees

A simple regression tree is built in a manner similar to a simple classification tree, and like the simple classification tree, it is rarely invoked on its own; the bagged, random forest, and gradient boosting methods build on this logic. I'll learn by example again. Using the `ISLR::Carseats` data set, I will predict `Sales` using from the 10 feature variables. Load the data.

```
carseats_dat <- Carseats
#skim_with(numeric = list(p0 = NULL, p25 = NULL, p50 = NULL, p75 = NULL,
#                          p100 = NULL, hist = NULL))
#
#skim(carseats_dat)
```

I'll split `carseats_dat` ($n = 400$) into `carseats_train` (80%, $n = 321$) and `carseats_test` (20%, $n = 79$). I'll fit a simple decision tree with `carseats_train`, then later a bagged tree, a random forest, and a gradient boosting tree. I'll compare their predictive performance with `carseats_test`.

```
set.seed(12345)
partition <- createDataPartition(y = carseats_dat$Sales, p = 0.8, list = FALSE)
carseats_train <- carseats_dat[partition, ]
carseats_test <- carseats_dat[-partition, ]
```

The first step is to build a full tree, then perform k-fold cross-validation to help select the optimal cost complexity (cp). The only difference here is the `rpart()` parameter `method = "anova"` to produce a regression tree.

```
set.seed(1234)
carseats_model_1 <- rpart(
  formula = Sales ~ .,
  data = carseats_train,
  method = "anova",
```

```

xval = 10,
model = TRUE # to plot splits with factor variables.
)
print(carseats_model_1)

## n= 321
##
## node), split, n, deviance, yval
##      * denotes terminal node
##
## 1) root 321 2567.76800  7.535950
##    2) ShelfLoc=Bad,Medium 251 1474.14100  6.770359
##      4) Price>=105.5 168  719.70630  5.987024
##        8) ShelfLoc=Bad 50  165.70160  4.693600
##          16) Population< 201.5 20  48.35505  3.646500 *
##          17) Population>=201.5 30  80.79922  5.391667 *
##          9) ShelfLoc=Medium 118  434.91370  6.535085
##            18) Advertising< 11.5 88  290.05490  6.113068
##              36) CompPrice< 142 69  193.86340  5.769420
##                72) Price>=132.5 16  50.75440  4.455000 *
##                73) Price< 132.5 53  107.12060  6.166226 *
##              37) CompPrice>=142 19  58.45118  7.361053 *
##            19) Advertising>=11.5 30  83.21323  7.773000 *
##          5) Price< 105.5 83  442.68920  8.355904
##            10) Age>=63.5 32  153.42300  6.922500
##              20) Price>=85 25  66.89398  6.160800
##                40) ShelfLoc=Bad 9  18.39396  4.772222 *
##                41) ShelfLoc=Medium 16  21.38544  6.941875 *
##              21) Price< 85 7  20.22194  9.642857 *
##            11) Age< 63.5 51  182.26350  9.255294
##              22) Income< 57.5 12  28.03042  7.707500 *
##              23) Income>=57.5 39  116.63950  9.731538
##                46) Age>=50.5 14  21.32597  8.451429 *
##                47) Age< 50.5 25  59.52474  10.448400 *
##          3) ShelfLoc=Good 70  418.98290  10.281140
##            6) Price>=107.5 49  242.58730  9.441633
##              12) Advertising< 13.5 41  162.47820  8.926098
##                24) Age>=61 17  53.37051  7.757647 *
##                25) Age< 61 24  69.45776  9.753750 *
##              13) Advertising>=13.5 8  13.36599  12.083750 *
##            7) Price< 107.5 21  61.28200  12.240000 *

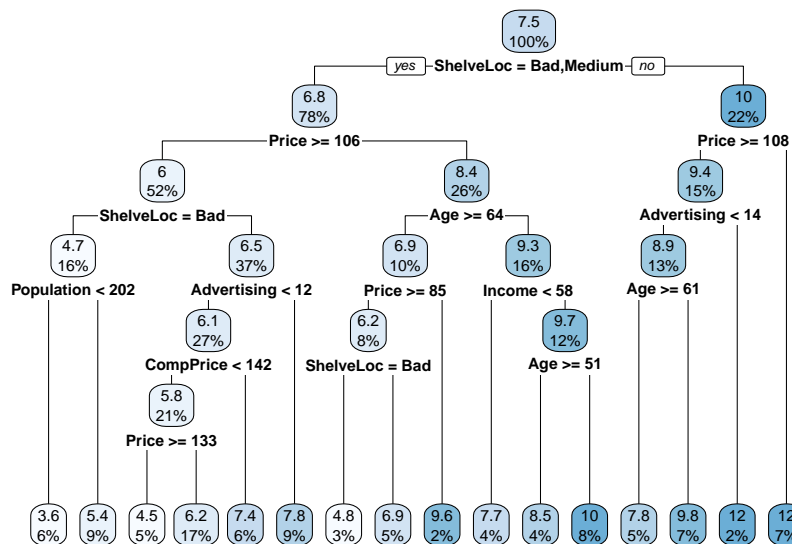
```

The output starts with the root node. The predicted **Sales** at the root is the mean **Sales** for the training data set, 7.535950 (values are \$000s). The deviance at the root is the SSE, 2567.768. The child nodes of node “x” are labeled 2x)

and $2x+1$), so the child nodes of 1) are 2) and 3), and the child nodes of 2) are 4) and 5). Terminal nodes are labeled with an asterisk (*).

The first split is at `ShelveLoc` = [Bad, Medium] vs Good. Here is what the full (unpruned) tree looks like.

```
rpart.plot(carseats_model_1, yesno = TRUE)
```



The boxes show the node predicted value (mean) and the proportion of observations that are in the node (or child nodes).

`rpart()` not only grew the full tree, it also used cross-validation to test the performance of the possible complexity hyperparameters. `printcp()` displays the candidate cp values. You can use this table to decide how to prune the tree.

```
printcp(carseats_model_1)
```

```
##
## Regression tree:
## rpart(formula = Sales ~ ., data = carseats_train, method = "anova",
##       model = TRUE, xval = 10)
##
## Variables actually used in tree construction:
## [1] Advertising Age      CompPrice  Income    Population Price
## [7] ShelveLoc
```

```
##
## Root node error: 2567.8/321 = 7.9993
##
## n= 321
##
##          CP nsplit rel error  xerror    xstd
## 1 0.262736      0  1.00000 1.00635 0.076664
## 2 0.121407      1  0.73726 0.74888 0.058981
## 3 0.046379      2  0.61586 0.65278 0.050839
## 4 0.044830      3  0.56948 0.67245 0.051638
## 5 0.041671      4  0.52465 0.66230 0.051065
## 6 0.025993      5  0.48298 0.62345 0.049368
## 7 0.025823      6  0.45698 0.61980 0.048026
## 8 0.024007      7  0.43116 0.62058 0.048213
## 9 0.015441      8  0.40715 0.58061 0.041738
## 10 0.014698     9  0.39171 0.56413 0.041368
## 11 0.014641    10  0.37701 0.56277 0.041271
## 12 0.014233    11  0.36237 0.56081 0.041097
## 13 0.014015    12  0.34814 0.55647 0.038308
## 14 0.013938    13  0.33413 0.55647 0.038308
## 15 0.010560    14  0.32019 0.57110 0.038872
## 16 0.010000    15  0.30963 0.56676 0.038090
```

There are 16 possible cp values in this model. The model with the smallest complexity parameter allows the most splits (`nsplit`). The highest complexity parameter corresponds to a tree with just a root node. `rel error` is the SSE relative to the root node. The root node SSE is 2567.76800, so its `rel error` is $2567.76800/2567.76800 = 1.0$. That means the absolute error of the full tree (at $CP = 0.01$) is $0.30963 * 2567.76800 = 795.058$. You can verify that by calculating the SSE of the model predicted values:

```
data.frame(pred = predict(carseats_model_1, newdata = carseats_train)) %>%
  mutate(obs = carseats_train$Sales,
         sq_err = (obs - pred)^2) %>%
  summarize(sse = sum(sq_err))
```

```
##          sse
## 1 795.0525
```

Finishing the CP table tour, `xerror` is the cross-validated SSE and `xstd` is its standard error. If you want the lowest possible error, then prune to the tree with the smallest relative SSE (`xerror`). If you want to balance predictive power with simplicity, prune to the smallest tree within 1 SE of the one with the smallest relative SSE. The CP table is not super-helpful for finding that tree. I'll add a column to find it.


```

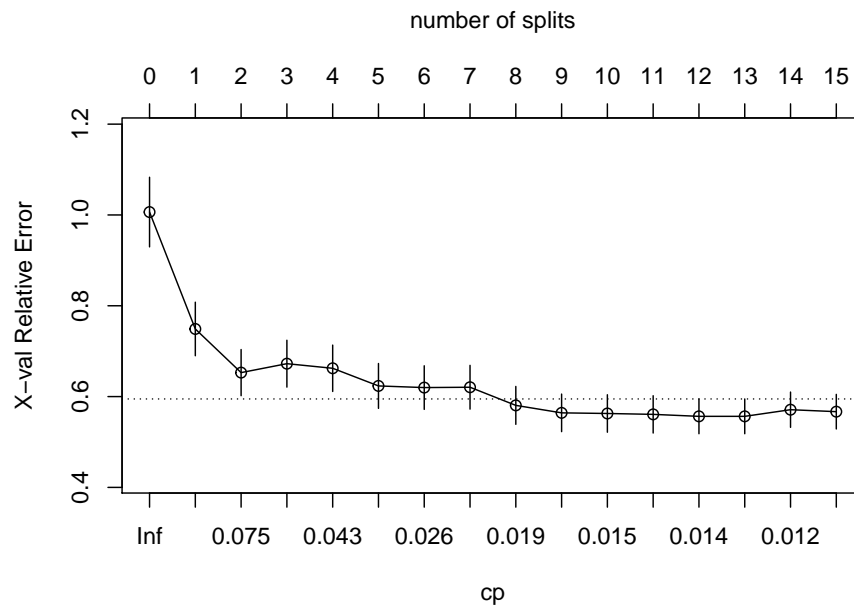
carseats_model_1$cptable %>%
  data.frame() %>%
  mutate(min_xerror_idx = which.min(carseats_model_1$cptable[, "xerror"]),
         rownum = row_number(),
         xerror_cap = carseats_model_1$cptable[min_xerror_idx, "xerror"] +
           carseats_model_1$cptable[min_xerror_idx, "xstd"],
         eval = case_when(rownum == min_xerror_idx ~ "min xerror",
                          xerror < xerror_cap ~ "under cap",
                          TRUE ~ "")) %>%
  select(-rownum, -min_xerror_idx)

```

##		CP	nsplit	rel.error	xerror	xstd	xerror_cap	eval
## 1	0.26273578	0	1.0000000	1.0063530	0.07666355	0.5947744		
## 2	0.12140705	1	0.7372642	0.7488767	0.05898146	0.5947744		
## 3	0.04637919	2	0.6158572	0.6527823	0.05083938	0.5947744		
## 4	0.04483023	3	0.5694780	0.6724529	0.05163819	0.5947744		
## 5	0.04167149	4	0.5246478	0.6623028	0.05106530	0.5947744		
## 6	0.02599265	5	0.4829763	0.6234457	0.04936799	0.5947744		
## 7	0.02582284	6	0.4569836	0.6198034	0.04802643	0.5947744		
## 8	0.02400748	7	0.4311608	0.6205756	0.04821332	0.5947744		
## 9	0.01544139	8	0.4071533	0.5806072	0.04173785	0.5947744	under cap	
## 10	0.01469771	9	0.3917119	0.5641331	0.04136793	0.5947744	under cap	
## 11	0.01464055	10	0.3770142	0.5627713	0.04127139	0.5947744	under cap	
## 12	0.01423309	11	0.3623736	0.5608073	0.04109662	0.5947744	under cap	
## 13	0.01401541	12	0.3481405	0.5564663	0.03830810	0.5947744	min xerror	
## 14	0.01393771	13	0.3341251	0.5564663	0.03830810	0.5947744	under cap	
## 15	0.01055959	14	0.3201874	0.5710951	0.03887227	0.5947744	under cap	
## 16	0.01000000	15	0.3096278	0.5667561	0.03808991	0.5947744	under cap	

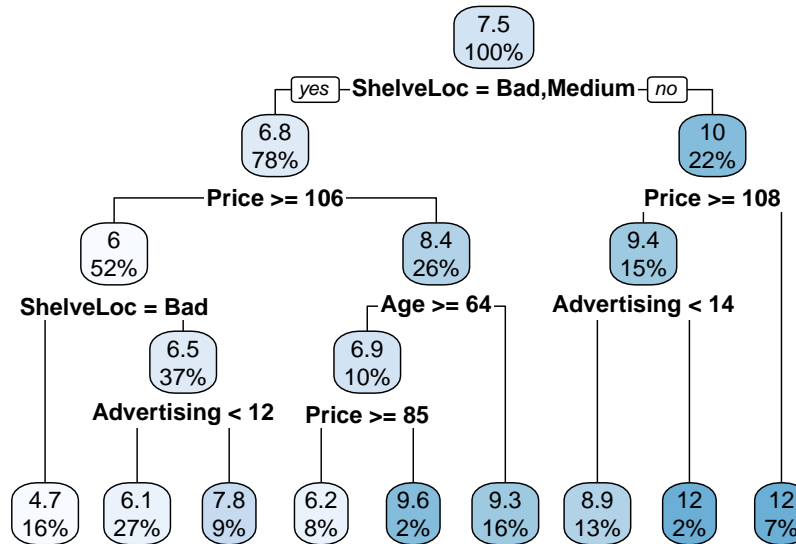
Okay, so the simplest tree is the one with CP = 0.01544139 (8 splits). Fortunately, `plotcp()` presents a nice graphical representation of the relationship between `xerror` and `cp`.

```
plotcp(carseats_model_1, upper = "splits")
```



The dashed line is set at the minimum `xerror` + `xstd`. The top axis shows the number of splits in the tree. I'm not sure why the CP values are not the same as in the table (they are close, but not the same). The smallest relative error is at 0.0140154, but the maximum CP below the dashed line (one standard deviation above the minimum error) is at CP = .019 (8 splits). Use the `prune()` function to prune the tree by specifying the associated cost-complexity `cp`.

```
carseats_model_1_pruned <- prune(
  carseats_model_1,
  cp = carseats_model_1$cptable[carseats_model_1$cptable[, 2] == 8, "CP"]
)
rpart.plot(carseats_model_1_pruned, yesno = TRUE)
```

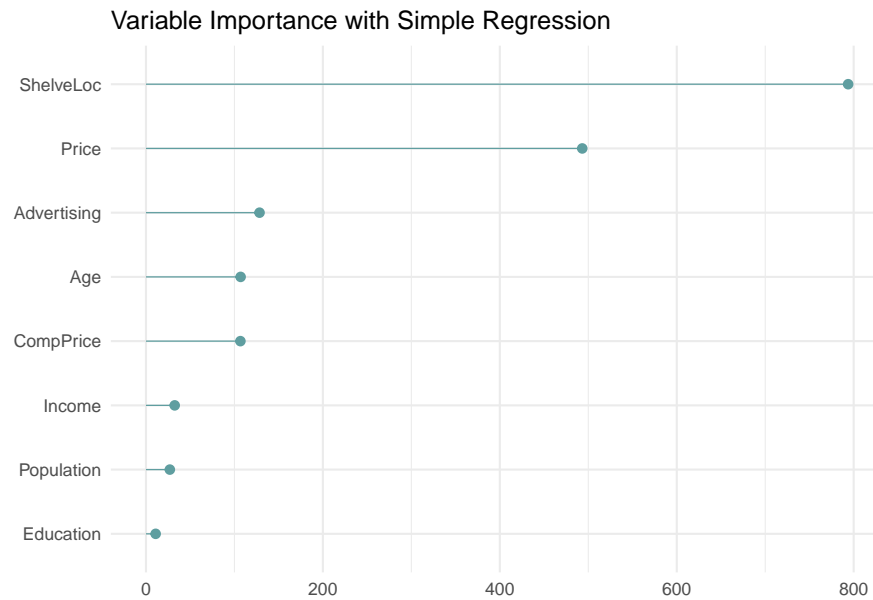


The most “important” indicator of Sales is `ShelveLoc`. Here are the importance values from the model.

```

carseats_model_1_pruned$variable.importance %>%
  data.frame() %>%
  rownames_to_column(var = "Feature") %>%
  rename(Overall = '.') %>%
  ggplot(aes(x = fct_reorder(Feature, Overall), y = Overall)) +
  geom_pointrange(aes(ymin = 0, ymax = Overall), color = "cadetblue", size = .3) +
  theme_minimal() +
  coord_flip() +
  labs(x = "", y = "", title = "Variable Importance with Simple Regression")

```



The most important indicator of **Sales** is **ShelveLoc**, then **Price**, then **Age**, all of which appear in the final model. **CompPrice** was also important.

The last step is to make predictions on the validation data set. The root mean squared error ($RMSE = \sqrt{(1/2) \sum (actual - pred)^2}$) and mean absolute error ($MAE = (1/n) \sum |actual - pred|$) are the two most common measures of predictive accuracy. The key difference is that RMSE punishes large errors more harshly. For a regression tree, set argument `type = "vector"` (or do not specify at all).

```
carseats_model_1_preds <- predict(
  carseats_model_1_pruned,
  carseats_test,
  type = "vector"
)

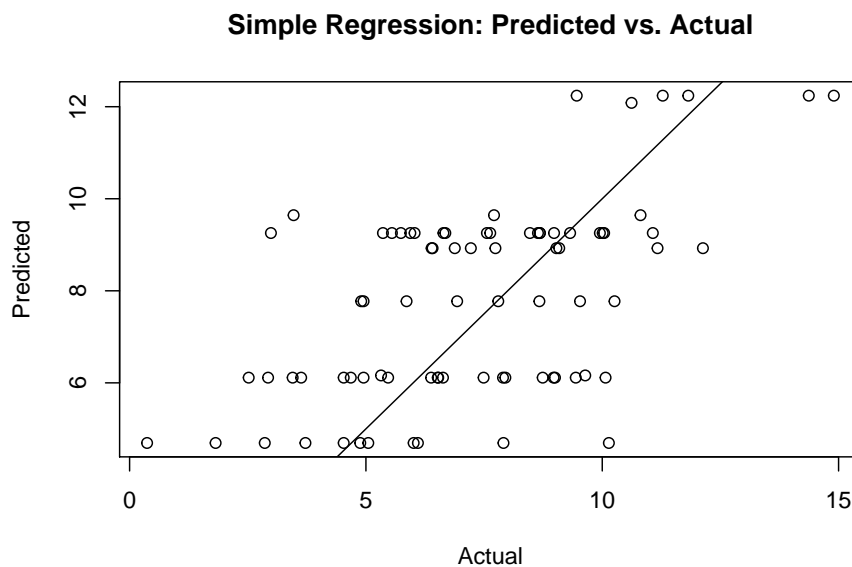
carseats_model_1_pruned_rmse <- RMSE(
  pred = carseats_model_1_preds,
  obs = carseats_test$Sales
)
carseats_model_1_pruned_rmse
```

```
## [1] 2.388059
```

The pruning process leads to an average prediction error of 2.388 in the test

data set. Not too bad considering the standard deviation of `Sales` is 2.801. Here is a predicted vs actual plot.

```
plot(carseats_test$Sales, carseats_model_1_preds,
     main = "Simple Regression: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
abline(0, 1)
```



The 6 possible predicted values do a decent job of binning the observations.

7.2.1 Caret Approach

I can also fit the model with `caret::train()`, specifying `method = "rpart"`.

I'll build the model using 10-fold cross-validation to optimize the hyperparameter CP.

```
carseats_trControl = trainControl(
  method = "cv", # k-fold cross validation
  number = 10, # 10 folds
  savePredictions = "final" # save predictions for the optimal tuning parameter
)
```

I'll let the model look for the best CP tuning parameter with `tuneLength` to get close, then fine-tune with `tuneGrid`.

```
set.seed(1234)
carseats_model_2 = train(
  Sales ~ .,
  data = carseats_train,
  method = "rpart", # for classification tree
  tuneLength = 5, # choose up to 5 combinations of tuning parameters (cp)
  metric = "RMSE", # evaluate hyperparameter combinations with RMSE
  trControl = carseats_trControl
)

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, :
## There were missing values in resampled performance measures.
```

```
print(carseats_model_2)
```

```
## CART
##
## 321 samples
## 10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 289, 289, 289, 289, ...
## Resampling results across tuning parameters:
##
##   cp          RMSE      Rsquared    MAE
## 0.04167149  2.209383  0.4065251  1.778797
## 0.04483023  2.243618  0.3849728  1.805027
## 0.04637919  2.275563  0.3684309  1.808814
## 0.12140705  2.400455  0.2942663  1.936927
## 0.26273578  2.692867  0.1898998  2.192774
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was cp = 0.04167149.
```

The first `cp` (0.04167149) produced the smallest RMSE. I can drill into the best value of `cp` using a tuning grid. I'll try that now.

```
myGrid <- expand.grid(cp = seq(from = 0, to = 0.1, by = 0.01))
carseats_model_3 = train(
  Sales ~ .,
```

```

data = carseats_train,
method = "rpart", # for classification tree
tuneGrid = myGrid, # choose up to 5 combinations of tuning parameters (cp)
metric = "RMSE", # evaluate hyperparameter combinations with RMSE
trControl = carseats_trControl
)
print(carseats_model_3)

```

```

## CART
##
## 321 samples
## 10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 289, 289, 288, 289, ...
## Resampling results across tuning parameters:
##
##   cp    RMSE      Rsquared    MAE
##   0.00  2.131814  0.4578761  1.725960
##   0.01  2.203111  0.4294647  1.790050
##   0.02  2.240209  0.3948080  1.834786
##   0.03  2.206168  0.4139717  1.762170
##   0.04  2.274313  0.3686176  1.795154
##   0.05  2.309746  0.3405228  1.830556
##   0.06  2.246757  0.3703977  1.780266
##   0.07  2.253725  0.3679986  1.794485
##   0.08  2.253725  0.3679986  1.794485
##   0.09  2.253725  0.3679986  1.794485
##   0.10  2.253725  0.3679986  1.794485
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was cp = 0.

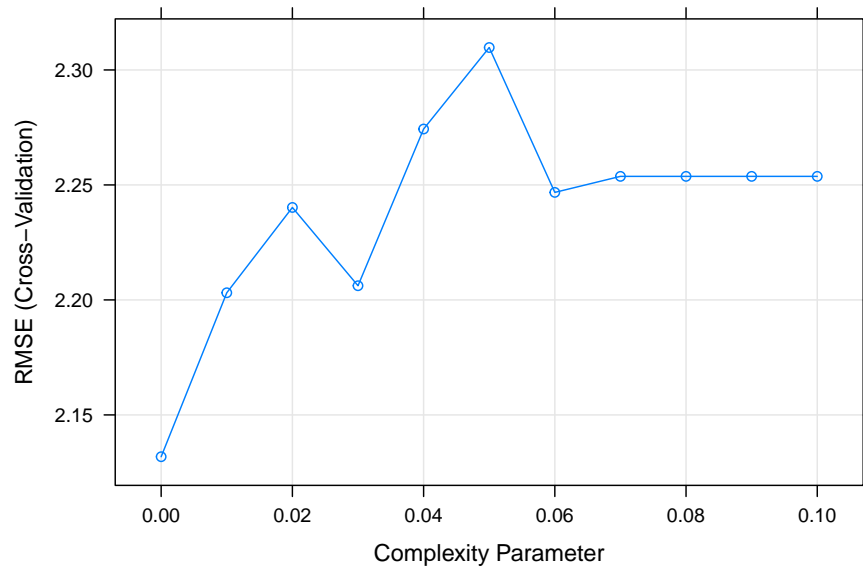
```

It looks like the best performing tree is the unpruned one.

```

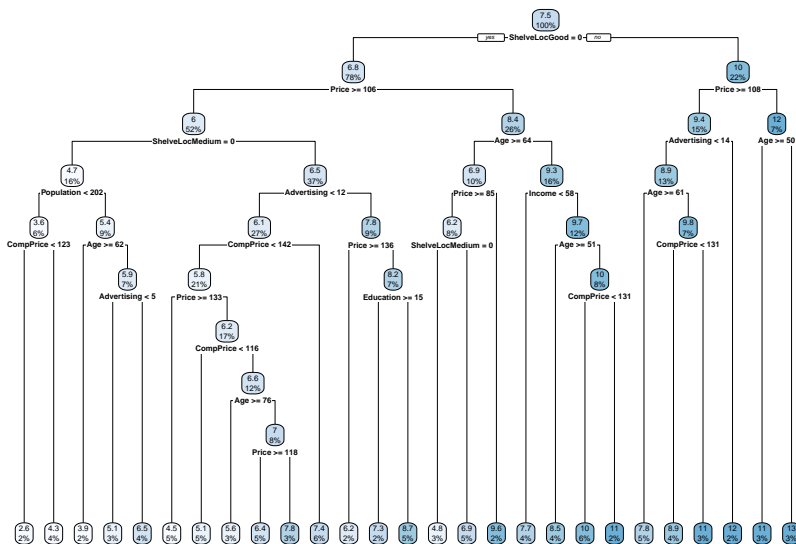
plot(carseats_model_3)

```



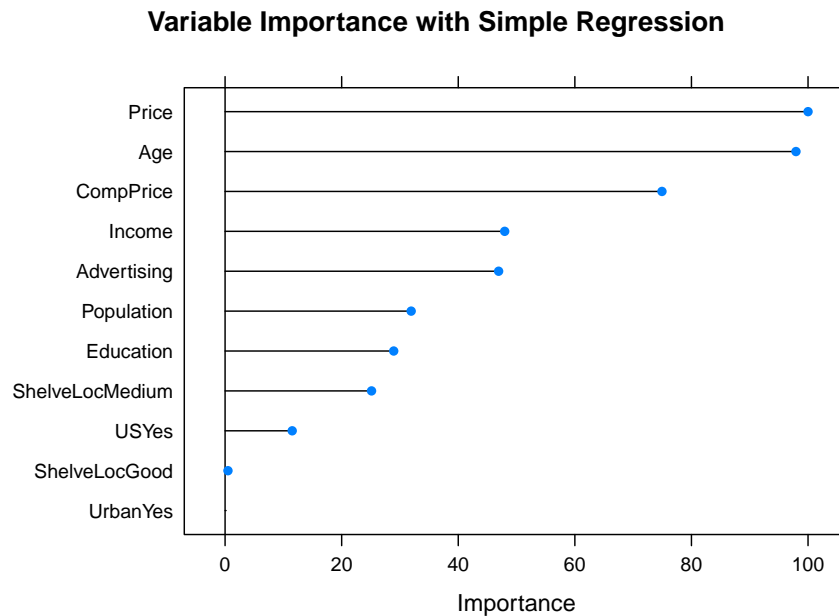
Lets's see the final model.

```
rpart.plot(carseats_model_3$finalModel)
```



What were the most important variables?

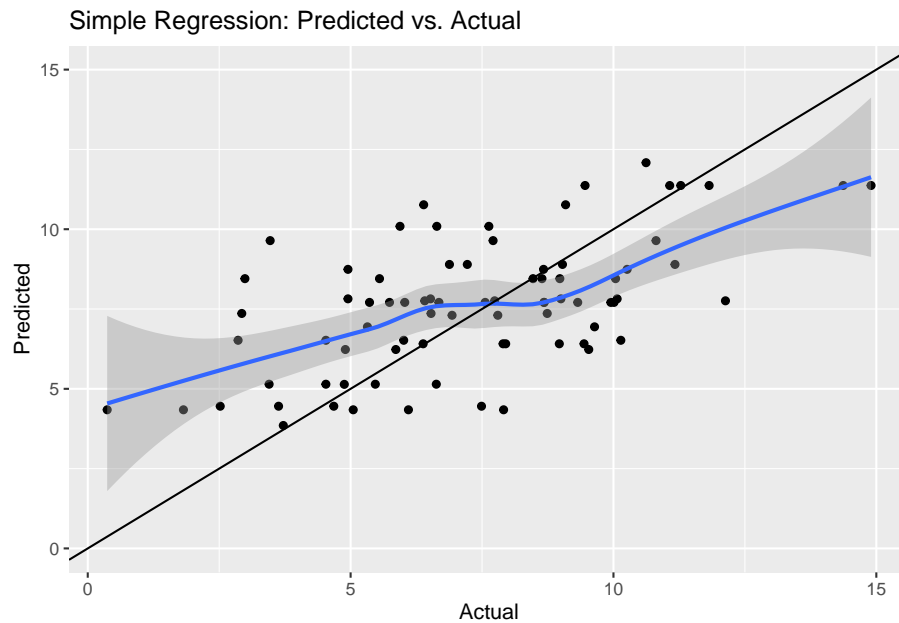

```
plot(varImp(carseats_model_3), main="Variable Importance with Simple Regression")
```



Evaluate the model by making predictions with the test data set.

```
carseats_model_3_preds <- predict(carseats_model_3, carseats_test, type = "raw")
data.frame(Actual = carseats_test$Sales, Predicted = carseats_model_3_preds) %>%
ggplot(aes(x = Actual, y = Predicted)) +
  geom_point() +
  geom_smooth() +
  geom_abline(slope = 1, intercept = 0) +
  scale_y_continuous(limits = c(0, 15)) +
  labs(title = "Simple Regression: Predicted vs. Actual")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Looks like the model over-estimates at the low end and underestimates at the high end. Calculate the test data set RMSE.

```
carseats_model_3_pruned_rmse <- RMSE(
  pred = carseats_model_3_preds,
  obs = carseats_test$Sales
)
carseats_model_3_pruned_rmse
```

```
## [1] 2.298331
```

Caret faired better in this model. Here is a summary the RMSE values of the two models.

```
rbind(data.frame(model = "Manual ANOVA",
  RMSE = round(carseats_model_1_pruned_rmse, 5)),
  data.frame(model = "Caret",
  RMSE = round(carseats_model_3_pruned_rmse, 5))
)
```

```
##          model    RMSE
## 1 Manual ANOVA 2.38806
## 2       Caret 2.29833
```

7.3 Bagging

Bootstrap aggregation, or *bagging*, is a general-purpose procedure for reducing the variance of a statistical learning method. The algorithm constructs B regression trees using B bootstrapped training sets, and averages the resulting predictions. These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. For classification trees, bagging takes the “majority vote” for the prediction. Use a value of B sufficiently large that the error has settled down.

To test the model accuracy, the out-of-bag observations are predicted from the models that do not use them. If $B/3$ of observations are in-bag, there are $B/3$ predictions per observation. These predictions are averaged for the test prediction. Again, for classification trees, a majority vote is taken.

The downside to bagging is that it improves accuracy at the expense of interpretability. There is no longer a single tree to interpret, so it is no longer clear which variables are more important than others.

Bagged trees are a special case of random forests, so see the next section for an example.

7.4 Random Forests

Random forests improve bagged trees by way of a small tweak that de-correlates the trees. As in bagging, the algorithm builds a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of $mtry$ predictors is chosen as split candidates from the full set of p predictors. A fresh sample of $mtry$ predictors is taken at each split. Typically $mtry \sim \sqrt{p}$. Bagged trees are thus a special case of random forests where $mtry = p$.

7.4.0.1 Bagging Classification Example

Again using the OJ data set to predict `Purchase`, this time I'll use the bagging method by specifying `method = "treebag"`. I'll use `tuneLength = 5` and not worry about `tuneGrid` anymore. Caret has no hyperparameters to tune with this model.

```
oj.bag = train(Purchase ~ .,
               data = oj_train,
               method = "treebag", # for bagging
               tuneLength = 5,    # choose up to 5 combinations of tuning parameters
               metric = "ROC",    # evaluate hyperparameter combinations with ROC
```

```

trControl = trainControl(
  method = "cv", # k-fold cross validation
  number = 10, # k=10 folds
  savePredictions = "final", # save predictions for the optimal t
  classProbs = TRUE, # return class probabilities in addition to p
  summaryFunction = twoClassSummary # for binary response variabl
)
)
oj.bag

```

```

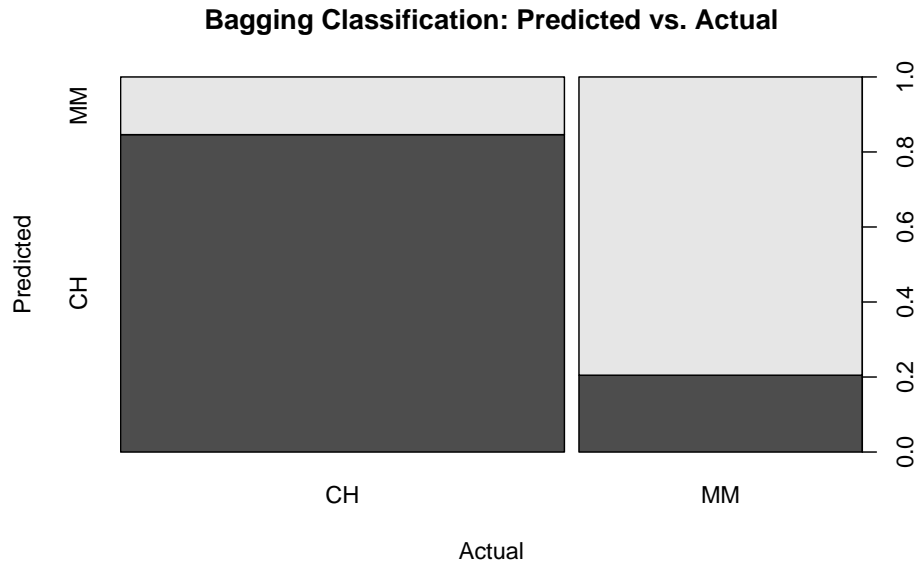
## Bagged CART
##
## 857 samples
## 17 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 771, 772, 771, 771, 771, 772, ...
## Resampling results:
##
## ROC      Sens      Spec
## 0.8524038 0.8165094 0.7217469

```

```

#plot(oj.bag$)
oj.pred <- predict(oj.bag, oj_test, type = "raw")
plot(oj_test$Purchase, oj.pred,
  main = "Bagging Classification: Predicted vs. Actual",
  xlab = "Actual",
  ylab = "Predicted")

```

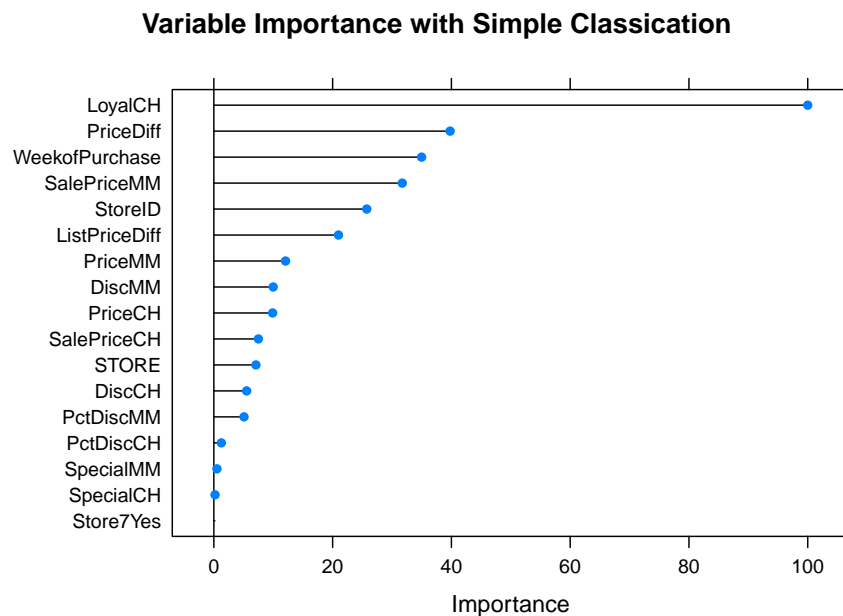


```
(oj.conf <- confusionMatrix(data = oj.pred,
                             reference = oj_test$Purchase))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##      CH 110  17
##      MM  20  66
##
##           Accuracy : 0.8263
##           95% CI : (0.7686, 0.8746)
##      No Information Rate : 0.6103
##      P-Value [Acc > NIR] : 7.121e-12
##
##           Kappa : 0.6372
##
##  McNemar's Test P-Value : 0.7423
##
##           Sensitivity : 0.8462
##           Specificity : 0.7952
##      Pos Pred Value : 0.8661
##      Neg Pred Value : 0.7674
##           Prevalence : 0.6103
```

```
##          Detection Rate : 0.5164
##    Detection Prevalence : 0.5962
##          Balanced Accuracy : 0.8207
##
##          'Positive' Class : CH
##
```

```
oj.bag.acc <- as.numeric(oj.conf$overall[1])
rm(oj.pred)
rm(oj.conf)
#plot(oj.bag$, oj.bag$finalModel$y)
plot(varImp(oj.bag), main="Variable Importance with Simple Classification")
```



7.4.0.2 Random Forest Classification Example

Now I'll try it with the random forest method by specifying `method = "ranger"`. I'll stick with `tuneLength = 5`. Caret tunes three hyperparameters:

- `mtry`: number of randomly selected predictors. Default is `sqrt(p)`.
- `splitrule`: splitting rule. For classification, options are "gini" (default) and "extratrees".
- `min.node.size`: minimal node size. Default is 1 for classification.

```

oj.frst = train(Purchase ~ .,
  data = oj_train,
  method = "ranger", # for random forest
  tuneLength = 5, # choose up to 5 combinations of tuning parameters
  metric = "ROC", # evaluate hyperparameter combinations with ROC
  trControl = trainControl(
    method = "cv", # k-fold cross validation
    number = 10, # 10 folds
    savePredictions = "final", # save predictions for the optimal tuning parameters
    classProbs = TRUE, # return class probabilities in addition to predicted values
    summaryFunction = twoClassSummary # for binary response variable
  )
)
oj.frst

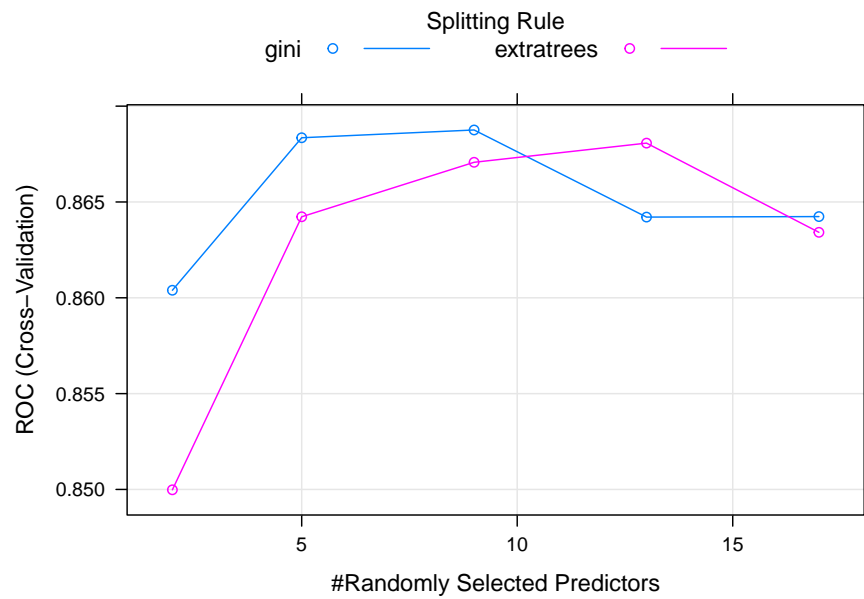
```

```

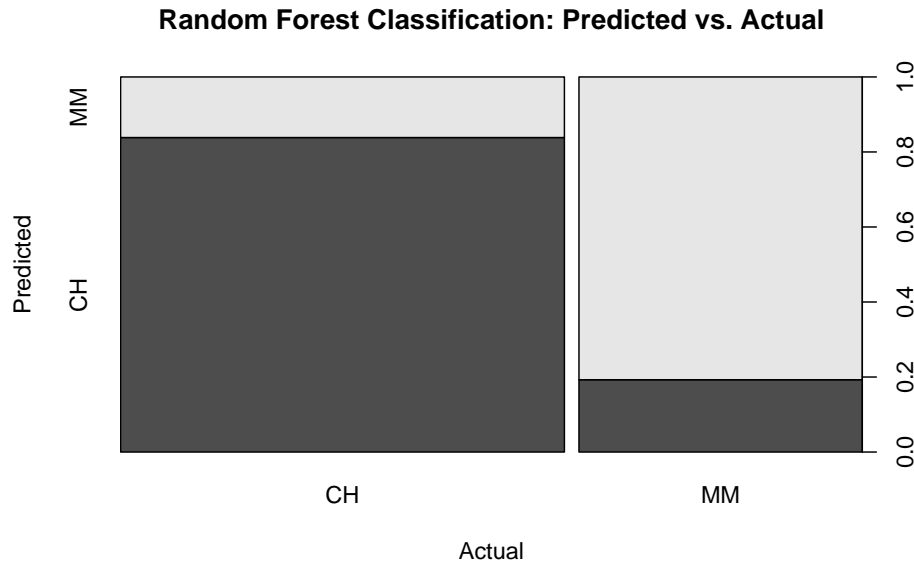
## Random Forest
##
## 857 samples
## 17 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 772, 771, 772, 770, 772, 772, ...
## Resampling results across tuning parameters:
##
##   mtry  splitrule  ROC          Sens          Spec
##   2     gini      0.8603930  0.8719158  0.6946524
##   2     extratrees 0.8499806  0.8814586  0.6287879
##   5     gini      0.8683505  0.8470247  0.7246881
##   5     extratrees 0.8642275  0.8584543  0.6886809
##   9     gini      0.8687568  0.8374456  0.7272727
##   9     extratrees 0.8670702  0.8451379  0.6858289
##   13    gini      0.8642114  0.8297896  0.7361854
##   13    extratrees 0.8680705  0.8298258  0.7064171
##   17    gini      0.8642378  0.8145501  0.7423351
##   17    extratrees 0.8634162  0.8260160  0.7093583
##
## Tuning parameter 'min.node.size' was held constant at a value of 1
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 9, splitrule = gini
## and min.node.size = 1.

```

```
plot(oj.frst)
```



```
oj.pred <- predict(oj.frst, oj_test, type = "raw")
plot(oj_test$Purchase, oj.pred,
     main = "Random Forest Classification: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
```

```
(oj.conf <- confusionMatrix(data = oj.pred,
                             reference = oj_test$Purchase))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##      CH 109  16
##      MM  21  67
##
##              Accuracy : 0.8263
##              95% CI : (0.7686, 0.8746)
##      No Information Rate : 0.6103
##      P-Value [Acc > NIR] : 7.121e-12
##
##              Kappa : 0.6387
##
##  McNemar's Test P-Value : 0.5108
##
##              Sensitivity : 0.8385
##              Specificity : 0.8072
##      Pos Pred Value : 0.8720
##      Neg Pred Value : 0.7614
##      Prevalence : 0.6103
```

```
##           Detection Rate : 0.5117
##      Detection Prevalence : 0.5869
##           Balanced Accuracy : 0.8228
##
##           'Positive' Class : CH
##
```

```
oj.first.acc <- as.numeric(oj.conf$overall[1])
rm(oj.pred)
rm(oj.conf)
#plot(oj.bag$, oj.bag$finalModel$y)
#plot(varImp(oj.frst), main="Variable Importance with Simple Classification")
```

The model algorithm explains “ROC was used to select the optimal model using the largest value. The final values used for the model were `mtry = 9`, `splitrule = extratrees` and `min.node.size = 1`.” You can see the results of tuning grid combinations in the associated plot of ROC AUC vs `mtry` grouped by splitting rule.

The bagging (accuracy = 0.80751) and random forest (accuracy = 0.81690) models fared pretty well, but the manual classification tree is still in first place. There’s still gradient boosting to investigate!

```
rbind(data.frame(model = "Manual Class", Accuracy = round(oj_model_1b_cm$overall["Accuracy"], 5)),
      data.frame(model = "Caret w.tuneGrid", Accuracy = round(oj_model_3_cm$overall["Accuracy"], 5)),
      data.frame(model = "Bagging", Accuracy = round(oj.bag.acc, 5)),
      data.frame(model = "Random Forest", Accuracy = round(oj.frst.acc, 5))
) %>% arrange(desc(Accuracy))
```

```
##           model Accuracy
## 1      Manual Class 0.85915
## 2 Caret w.tuneGrid 0.84507
## 3           Bagging 0.82629
## 4      Random Forest 0.82629
```

7.4.0.3 Bagging Regression Example

Again using the `Carseats` data set to predict `Sales`, this time I’ll use the bagging method by specifying `method = "treebag"`. I’ll use `tuneLength = 5` and not worry about `tuneGrid` anymore. `Caret` has no hyperparameters to tune with this model.

```
carseats.bag = train(Sales ~ .,
                     data = carseats_train,
```

```

        method = "treebag", # for bagging
        tuneLength = 5, # choose up to 5 combinations of tuning parameters
        metric = "RMSE", # evaluate hyperparameter combinations with RMSE
        trControl = trainControl(
            method = "cv", # k-fold cross validation
            number = 10, # 10 folds
            savePredictions = "final" # save predictions for the optimal tuning parameters
        )
carseats.bag

```

```

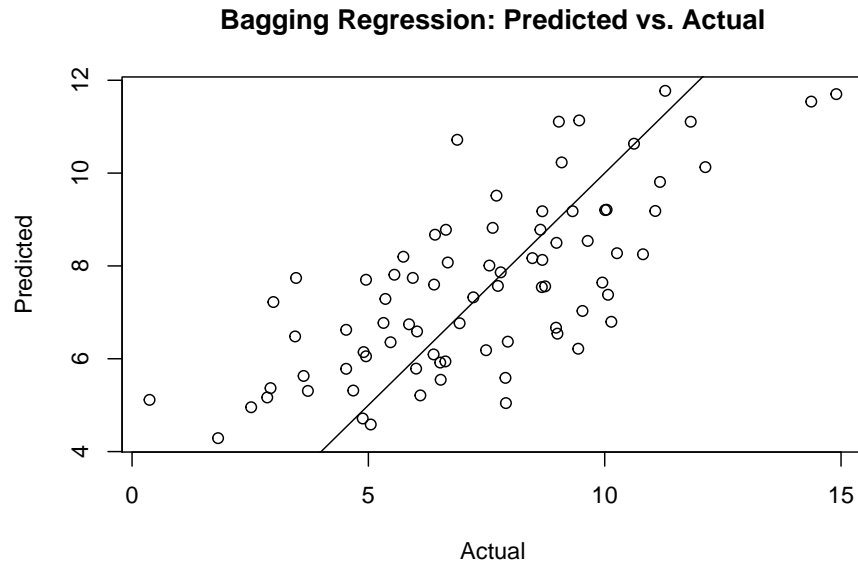
## Bagged CART
##
## 321 samples
## 10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 289, 288, 289, 289, ...
## Resampling results:
##
##   RMSE      Rsquared    MAE
## 1.709371 0.6532837 1.374155

```

```

#plot(carseats.bag$finalModel)
carseats.pred <- predict(carseats.bag, carseats_test, type = "raw")
plot(carseats_test$Sales, carseats.pred,
     main = "Bagging Regression: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
abline(0, 1)

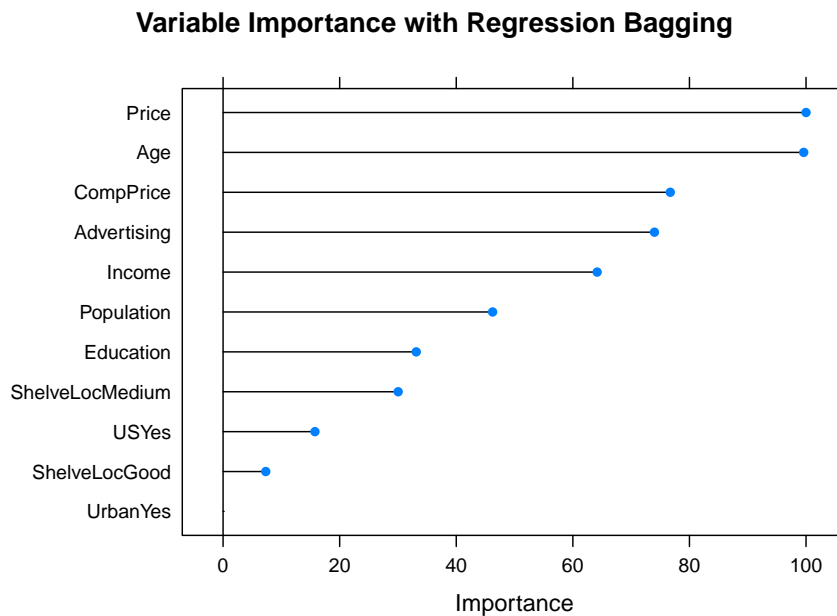
```



```
(carseats.bag.rmse <- RMSE(pred = carseats.pred,  
                           obs = carseats_test$Sales))
```

```
## [1] 1.932792
```

```
rm(carseats.pred)  
plot(varImp(carseats.bag), main="Variable Importance with Regression Bagging")
```



7.4.0.4 Random Forest Regression Example

Now I'll try it with the random forest method by specifying `method = "ranger"`. I'll stick with `tuneLength = 5`. Caret tunes three hyperparameters:

- `mtry`: number of randomly selected predictors
- `splitrule`: splitting rule. For regression, options are “variance” (default), “extratrees”, and “maxstat”.
- `min.node.size`: minimal node size

```

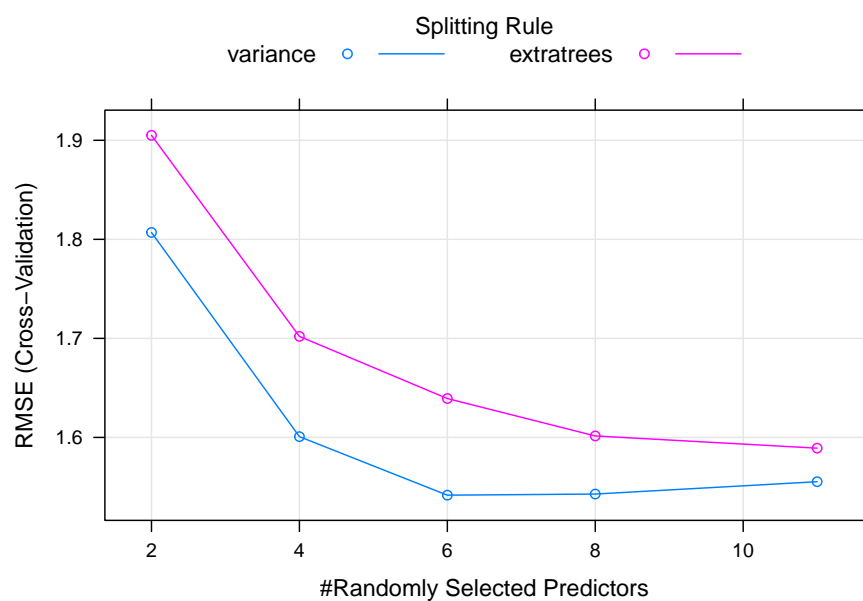
carseats.first = train(Sales ~ .,
  data = carseats_train,
  method = "ranger", # for random forest
  tuneLength = 5, # choose up to 5 combinations of tuning parameters
  metric = "RMSE", # evaluate hyperparameter combinations with RMSE
  trControl = trainControl(
    method = "cv", # k-fold cross validation
    number = 10, # 10 folds
    savePredictions = "final" # save predictions for the optimal tuning parameters
  )
)
carseats.first

## Random Forest

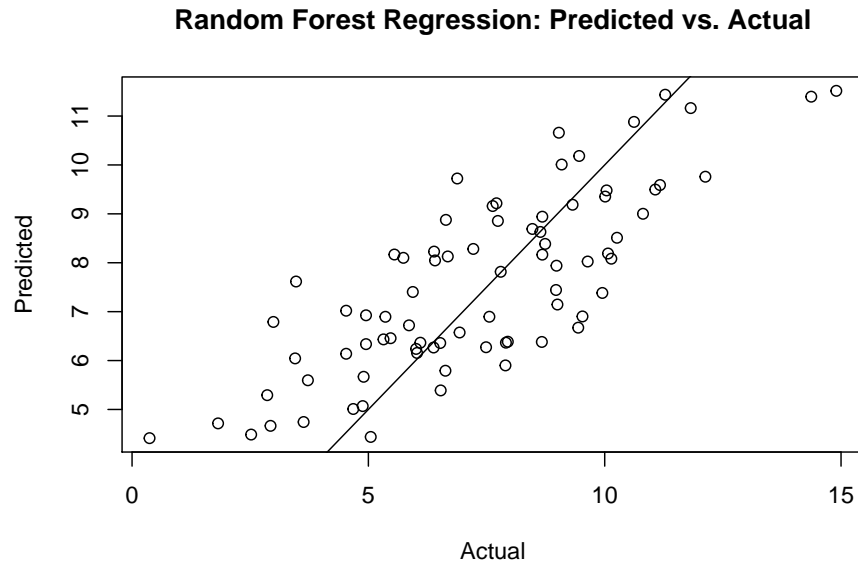
```

```
##
## 321 samples
## 10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 289, 289, 289, 288, ...
## Resampling results across tuning parameters:
##
##   mtry  splitrule  RMSE      Rsquared  MAE
##   2     variance  1.806943  0.6957452  1.446420
##   2     extratrees 1.905011  0.6466527  1.539096
##   4     variance  1.600763  0.7288625  1.266868
##   4     extratrees 1.702009  0.6862545  1.357981
##   6     variance  1.541675  0.7336448  1.217061
##   6     extratrees 1.639248  0.6966549  1.302159
##   8     variance  1.542806  0.7236085  1.221671
##   8     extratrees 1.601484  0.7053834  1.269484
##  11     variance  1.555271  0.7168108  1.230252
##  11     extratrees 1.589152  0.7058982  1.255090
##
## Tuning parameter 'min.node.size' was held constant at a value of 5
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were mtry = 6, splitrule = variance
## and min.node.size = 5.
```

```
plot(carseats.frst)
```



```
carseats.pred <- predict(carseats.first, carseats_test, type = "raw")
plot(carseats_test$Sales, carseats.pred,
     main = "Random Forest Regression: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
abline(0, 1)
```



```
(carseats.frst.rmse <- RMSE(pred = carseats.pred,
                             obs = carseats_test$Sales))
```

```
## [1] 1.758112
```

```
rm(carseats.pred)
```

```
#plot(varImp(carseats.frst), main="Variable Importance with Regression Random Forest")
```

The model algorithm explains “*RMSE was used to select the optimal model using the smallest value. The final values used for the model were mtry = 11, splitrule = variance and min.node.size = 5.*” You can see the results of tuning grid combinations in the associated plot of ROC AUC vs mtry grouped by splitting rule.

The bagging and random forest models faired very well - they took over the first and second place!

```
rbind(data.frame(model = "Manual ANOVA", RMSE = round(carseats_model_1_pruned_rmse, 5),
                 data.frame(model = "ANOVA w.tuneGrid", RMSE = round(carseats_model_3_pruned_rmse, 5),
                 data.frame(model = "Bagging", RMSE = round(carseats.bag.rmse, 5)),
                 data.frame(model = "Random Forest", RMSE = round(carseats.frst.rmse, 5))
) %>% arrange(RMSE)
```

```
##           model      RMSE
```



```
## 1    Random Forest 1.75811
## 2          Bagging 1.93279
## 3 ANOVA w.tuneGrid 2.29833
## 4      Manual ANOVA 2.38806
```

7.5 Gradient Boosting

Boosting is a method to improve (boost) the weak learners sequentially and increase the model accuracy with a combined model. There are several boosting algorithms. One of the earliest was AdaBoost (adaptive boost). A more recent innovation is gradient boosting.

Adaboost creates a single split tree (decision stump) then weights the observations by how well the initial tree performed, putting more weight on the difficult observations. It then creates a second tree using the weights so that it focuses on the difficult observations. Observations that are difficult to classify receive increasing larger weights until the algorithm identifies a model that correctly classifies them. The final model returns predictions that are a majority vote. (*I think Adaboost applies only to classification problems, not regressions*).

Gradient boosting generalizes the AdaBoost method, so that the object is to minimize a loss function. In the case of classification problems, the loss function is the log-loss; for regression problems, the loss function is mean squared error. The regression trees are additive, so that the successive models can be added together to correct the residuals in the earlier models. Gradient boosting constructs its trees in a “greedy” manner, meaning it chooses the best splits based on purity scores like Gini or minimizing the loss. It is common to constrain the weak learners by setting maximum tree size parameters. Gradient boosting continues until it reaches maximum number of trees or an acceptable error level. This can result in overfitting, so it is common to employ regularization methods that penalize aspects of the model.

Tree Constraints. In general the more constrained the tree, the more trees need to be grown. Parameters to optimize include number of trees, tree depth, number of nodes, minimum observations per split, and minimum improvement to loss.

Learning Rate. Each successive tree can be weighted to slow down the learning rate. Decreasing the learning rate increases the number of required trees. Common growth rates are 0.1 to 0.3.

The gradient boosting algorithm fits a shallow tree T_1 to the data, $M_1 = T_1$. Then it fits a tree T_2 to the residuals and adds a weighted sum of the tree to the original tree as $M_2 = M_1 + \gamma T_2$. For regularized boosting, include a learning rate factor $\eta \in (0..1)$, $M_2 = M_1 + \eta \gamma T_2$. A larger η produces faster learning, but risks overfitting. The process repeats until the residuals are small enough, or until it reaches the maximum iterations. Because overfitting is a risk, use

cross-validation to select the appropriate number of trees (the number of trees producing the lowest RMSE).

7.5.0.1 Gradient Boosting Classification Example

Again using the OJ data set to predict `Purchase`, this time I'll use the gradient boosting method by specifying `method = "gbm"`. I'll use `tuneLength = 5` and not worry about `tuneGrid` anymore. `Caret` tunes the following hyperparameters (see `modelLookup("gbm")`).

- `n.trees`: number of boosting iterations
- `interaction.depth`: maximum tree depth
- `shrinkage`: shrinkage
- `n.minobsinnode`: minimum terminal node size

```
oj.gbm <- train(Purchase ~ .,
  data = oj_train,
  method = "gbm", # for bagged tree
  tuneLength = 5, # choose up to 5 combinations of tuning parameters
  metric = "ROC", # evaluate hyperparameter combinations with ROC
  trControl = trainControl(
    method = "cv", # k-fold cross validation
    number = 10, # 10 folds
    savePredictions = "final", # save predictions for the optimal t
    classProbs = TRUE, # return class probabilities in addition to p
    summaryFunction = twoClassSummary # for binary response variabl
  )
)
```

## Iter	TrainDeviance	ValidDeviance	StepSize	Improve
## 1	1.2789	nan	0.1000	0.0273
## 2	1.2286	nan	0.1000	0.0245
## 3	1.1929	nan	0.1000	0.0175
## 4	1.1613	nan	0.1000	0.0148
## 5	1.1263	nan	0.1000	0.0146
## 6	1.0991	nan	0.1000	0.0105
## 7	1.0752	nan	0.1000	0.0102
## 8	1.0579	nan	0.1000	0.0087
## 9	1.0433	nan	0.1000	0.0047
## 10	1.0280	nan	0.1000	0.0082
## 20	0.9233	nan	0.1000	0.0026
## 40	0.8226	nan	0.1000	0.0010
## 60	0.7809	nan	0.1000	-0.0001
## 80	0.7595	nan	0.1000	-0.0002

```

##      100      0.7506      nan      0.1000     -0.0008
##      120      0.7407      nan      0.1000     -0.0005
##      140      0.7317      nan      0.1000     -0.0005
##      160      0.7277      nan      0.1000     -0.0009
##      180      0.7232      nan      0.1000     -0.0004
##      200      0.7181      nan      0.1000     -0.0007
##      220      0.7115      nan      0.1000     -0.0008
##      240      0.7096      nan      0.1000     -0.0010
##      250      0.7081      nan      0.1000     -0.0015
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           1.2695           nan      0.1000     0.0319
##      2           1.2150           nan      0.1000     0.0260
##      3           1.1702           nan      0.1000     0.0225
##      4           1.1260           nan      0.1000     0.0186
##      5           1.0913           nan      0.1000     0.0147
##      6           1.0586           nan      0.1000     0.0160
##      7           1.0276           nan      0.1000     0.0146
##      8           1.0045           nan      0.1000     0.0109
##      9           0.9836           nan      0.1000     0.0099
##     10           0.9624           nan      0.1000     0.0068
##     20           0.8337           nan      0.1000     0.0027
##     40           0.7525           nan      0.1000    -0.0005
##     60           0.7240           nan      0.1000    -0.0005
##     80           0.7063           nan      0.1000    -0.0006
##    100           0.6879           nan      0.1000    -0.0011
##    120           0.6751           nan      0.1000    -0.0018
##    140           0.6605           nan      0.1000    -0.0012
##    160           0.6477           nan      0.1000    -0.0013
##    180           0.6359           nan      0.1000    -0.0010
##    200           0.6274           nan      0.1000    -0.0018
##    220           0.6166           nan      0.1000    -0.0005
##    240           0.6078           nan      0.1000    -0.0011
##    250           0.6014           nan      0.1000    -0.0019
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           1.2548           nan      0.1000     0.0377
##      2           1.1905           nan      0.1000     0.0294
##      3           1.1343           nan      0.1000     0.0258
##      4           1.0935           nan      0.1000     0.0180
##      5           1.0529           nan      0.1000     0.0168
##      6           1.0172           nan      0.1000     0.0159
##      7           0.9824           nan      0.1000     0.0151
##      8           0.9534           nan      0.1000     0.0127
##      9           0.9277           nan      0.1000     0.0109
##     10           0.9066           nan      0.1000     0.0088

```

##	20	0.7870	nan	0.1000	0.0023
##	40	0.7150	nan	0.1000	-0.0008
##	60	0.6799	nan	0.1000	-0.0023
##	80	0.6520	nan	0.1000	-0.0012
##	100	0.6298	nan	0.1000	-0.0005
##	120	0.6117	nan	0.1000	-0.0024
##	140	0.5973	nan	0.1000	-0.0016
##	160	0.5849	nan	0.1000	-0.0023
##	180	0.5670	nan	0.1000	-0.0015
##	200	0.5548	nan	0.1000	-0.0006
##	220	0.5440	nan	0.1000	-0.0024
##	240	0.5290	nan	0.1000	-0.0020
##	250	0.5228	nan	0.1000	-0.0016

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2554	nan	0.1000	0.0399
##	2	1.1847	nan	0.1000	0.0346
##	3	1.1321	nan	0.1000	0.0199
##	4	1.0823	nan	0.1000	0.0224
##	5	1.0392	nan	0.1000	0.0208
##	6	1.0067	nan	0.1000	0.0145
##	7	0.9768	nan	0.1000	0.0139
##	8	0.9462	nan	0.1000	0.0123
##	9	0.9238	nan	0.1000	0.0095
##	10	0.8966	nan	0.1000	0.0090
##	20	0.7681	nan	0.1000	0.0007
##	40	0.6937	nan	0.1000	-0.0004
##	60	0.6552	nan	0.1000	-0.0017
##	80	0.6202	nan	0.1000	-0.0018
##	100	0.5887	nan	0.1000	-0.0027
##	120	0.5653	nan	0.1000	-0.0012
##	140	0.5434	nan	0.1000	-0.0017
##	160	0.5275	nan	0.1000	-0.0008
##	180	0.5068	nan	0.1000	-0.0012
##	200	0.4935	nan	0.1000	-0.0016
##	220	0.4801	nan	0.1000	-0.0018
##	240	0.4665	nan	0.1000	-0.0010
##	250	0.4603	nan	0.1000	-0.0012

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2513	nan	0.1000	0.0375
##	2	1.1795	nan	0.1000	0.0320
##	3	1.1264	nan	0.1000	0.0254
##	4	1.0742	nan	0.1000	0.0225
##	5	1.0282	nan	0.1000	0.0196
##	6	0.9888	nan	0.1000	0.0177

```

##      7      0.9547      nan      0.1000      0.0136
##      8      0.9303      nan      0.1000      0.0103
##      9      0.9008      nan      0.1000      0.0121
##     10      0.8803      nan      0.1000      0.0073
##     20      0.7563      nan      0.1000      0.0003
##     40      0.6715      nan      0.1000     -0.0012
##     60      0.6253      nan      0.1000     -0.0016
##     80      0.5868      nan      0.1000     -0.0021
##    100      0.5538      nan      0.1000     -0.0015
##    120      0.5285      nan      0.1000     -0.0034
##    140      0.5070      nan      0.1000     -0.0025
##    160      0.4872      nan      0.1000     -0.0012
##    180      0.4736      nan      0.1000     -0.0023
##    200      0.4566      nan      0.1000     -0.0015
##    220      0.4407      nan      0.1000     -0.0011
##    240      0.4262      nan      0.1000     -0.0013
##    250      0.4186      nan      0.1000     -0.0024
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1      1.2743      nan      0.1000      0.0307
##      2      1.2220      nan      0.1000      0.0240
##      3      1.1885      nan      0.1000      0.0165
##      4      1.1522      nan      0.1000      0.0177
##      5      1.1186      nan      0.1000      0.0136
##      6      1.0912      nan      0.1000      0.0111
##      7      1.0693      nan      0.1000      0.0106
##      8      1.0492      nan      0.1000      0.0089
##      9      1.0309      nan      0.1000      0.0093
##     10      1.0172      nan      0.1000      0.0069
##     20      0.9206      nan      0.1000      0.0030
##     40      0.8357      nan      0.1000     -0.0002
##     60      0.7936      nan      0.1000     -0.0000
##     80      0.7764      nan      0.1000     -0.0009
##    100      0.7682      nan      0.1000     -0.0004
##    120      0.7620      nan      0.1000     -0.0008
##    140      0.7582      nan      0.1000     -0.0011
##    160      0.7536      nan      0.1000     -0.0005
##    180      0.7501      nan      0.1000     -0.0006
##    200      0.7448      nan      0.1000     -0.0008
##    220      0.7409      nan      0.1000     -0.0006
##    240      0.7385      nan      0.1000     -0.0011
##    250      0.7368      nan      0.1000     -0.0007
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1      1.2697      nan      0.1000      0.0323
##      2      1.2121      nan      0.1000      0.0276

```

##	3	1.1636	nan	0.1000	0.0251
##	4	1.1220	nan	0.1000	0.0166
##	5	1.0826	nan	0.1000	0.0131
##	6	1.0537	nan	0.1000	0.0134
##	7	1.0269	nan	0.1000	0.0104
##	8	1.0061	nan	0.1000	0.0084
##	9	0.9858	nan	0.1000	0.0082
##	10	0.9678	nan	0.1000	0.0066
##	20	0.8429	nan	0.1000	0.0024
##	40	0.7685	nan	0.1000	-0.0010
##	60	0.7422	nan	0.1000	-0.0006
##	80	0.7228	nan	0.1000	-0.0009
##	100	0.7073	nan	0.1000	-0.0013
##	120	0.6937	nan	0.1000	-0.0024
##	140	0.6836	nan	0.1000	-0.0014
##	160	0.6703	nan	0.1000	-0.0022
##	180	0.6607	nan	0.1000	-0.0009
##	200	0.6529	nan	0.1000	-0.0011
##	220	0.6438	nan	0.1000	-0.0017
##	240	0.6370	nan	0.1000	-0.0015
##	250	0.6311	nan	0.1000	-0.0011

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2569	nan	0.1000	0.0361
##	2	1.1946	nan	0.1000	0.0301
##	3	1.1386	nan	0.1000	0.0266
##	4	1.0954	nan	0.1000	0.0205
##	5	1.0524	nan	0.1000	0.0204
##	6	1.0186	nan	0.1000	0.0149
##	7	0.9847	nan	0.1000	0.0126
##	8	0.9618	nan	0.1000	0.0086
##	9	0.9344	nan	0.1000	0.0114
##	10	0.9135	nan	0.1000	0.0095
##	20	0.8003	nan	0.1000	0.0027
##	40	0.7353	nan	0.1000	-0.0011
##	60	0.7042	nan	0.1000	-0.0027
##	80	0.6800	nan	0.1000	-0.0017
##	100	0.6602	nan	0.1000	-0.0008
##	120	0.6393	nan	0.1000	-0.0017
##	140	0.6231	nan	0.1000	-0.0016
##	160	0.6077	nan	0.1000	-0.0028
##	180	0.5977	nan	0.1000	-0.0012
##	200	0.5863	nan	0.1000	-0.0014
##	220	0.5749	nan	0.1000	-0.0013
##	240	0.5618	nan	0.1000	-0.0022
##	250	0.5577	nan	0.1000	-0.0022

```

##
## Iter    TrainDeviance    ValidDeviance    StepSize    Improve
##      1          1.2495          nan      0.1000      0.0419
##      2          1.1884          nan      0.1000      0.0265
##      3          1.1279          nan      0.1000      0.0248
##      4          1.0820          nan      0.1000      0.0208
##      5          1.0426          nan      0.1000      0.0166
##      6          1.0089          nan      0.1000      0.0146
##      7          0.9799          nan      0.1000      0.0126
##      8          0.9474          nan      0.1000      0.0140
##      9          0.9225          nan      0.1000      0.0079
##     10          0.9066          nan      0.1000      0.0048
##     20          0.7815          nan      0.1000      0.0014
##     40          0.7028          nan      0.1000     -0.0019
##     60          0.6661          nan      0.1000     -0.0011
##     80          0.6386          nan      0.1000     -0.0006
##    100          0.6075          nan      0.1000     -0.0005
##    120          0.5861          nan      0.1000     -0.0019
##    140          0.5674          nan      0.1000     -0.0020
##    160          0.5467          nan      0.1000     -0.0016
##    180          0.5318          nan      0.1000     -0.0020
##    200          0.5200          nan      0.1000     -0.0025
##    220          0.5050          nan      0.1000     -0.0009
##    240          0.4930          nan      0.1000     -0.0020
##    250          0.4883          nan      0.1000     -0.0016
##
## Iter    TrainDeviance    ValidDeviance    StepSize    Improve
##      1          1.2558          nan      0.1000      0.0336
##      2          1.1856          nan      0.1000      0.0326
##      3          1.1172          nan      0.1000      0.0305
##      4          1.0713          nan      0.1000      0.0222
##      5          1.0313          nan      0.1000      0.0171
##      6          0.9965          nan      0.1000      0.0164
##      7          0.9613          nan      0.1000      0.0156
##      8          0.9354          nan      0.1000      0.0103
##      9          0.9089          nan      0.1000      0.0111
##     10          0.8859          nan      0.1000      0.0059
##     20          0.7690          nan      0.1000      0.0006
##     40          0.6889          nan      0.1000     -0.0004
##     60          0.6452          nan      0.1000     -0.0021
##     80          0.6127          nan      0.1000     -0.0021
##    100          0.5811          nan      0.1000     -0.0032
##    120          0.5557          nan      0.1000     -0.0011
##    140          0.5332          nan      0.1000     -0.0012
##    160          0.5118          nan      0.1000     -0.0014
##    180          0.4879          nan      0.1000     -0.0012

```

##	200	0.4737	nan	0.1000	-0.0020
##	220	0.4591	nan	0.1000	-0.0024
##	240	0.4460	nan	0.1000	-0.0030
##	250	0.4386	nan	0.1000	-0.0013

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2717	nan	0.1000	0.0320
##	2	1.2209	nan	0.1000	0.0250
##	3	1.1751	nan	0.1000	0.0208
##	4	1.1404	nan	0.1000	0.0149
##	5	1.1056	nan	0.1000	0.0119
##	6	1.0782	nan	0.1000	0.0125
##	7	1.0569	nan	0.1000	0.0081
##	8	1.0356	nan	0.1000	0.0098
##	9	1.0176	nan	0.1000	0.0080
##	10	1.0042	nan	0.1000	0.0066
##	20	0.9023	nan	0.1000	0.0021
##	40	0.8156	nan	0.1000	0.0006
##	60	0.7793	nan	0.1000	0.0004
##	80	0.7609	nan	0.1000	-0.0014
##	100	0.7514	nan	0.1000	-0.0006
##	120	0.7448	nan	0.1000	-0.0005
##	140	0.7406	nan	0.1000	-0.0008
##	160	0.7353	nan	0.1000	-0.0007
##	180	0.7329	nan	0.1000	-0.0008
##	200	0.7281	nan	0.1000	-0.0014
##	220	0.7239	nan	0.1000	-0.0008
##	240	0.7211	nan	0.1000	-0.0011
##	250	0.7203	nan	0.1000	-0.0010

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2647	nan	0.1000	0.0368
##	2	1.2060	nan	0.1000	0.0275
##	3	1.1558	nan	0.1000	0.0232
##	4	1.1207	nan	0.1000	0.0178
##	5	1.0787	nan	0.1000	0.0172
##	6	1.0478	nan	0.1000	0.0141
##	7	1.0178	nan	0.1000	0.0117
##	8	0.9963	nan	0.1000	0.0099
##	9	0.9749	nan	0.1000	0.0107
##	10	0.9514	nan	0.1000	0.0095
##	20	0.8341	nan	0.1000	0.0029
##	40	0.7601	nan	0.1000	-0.0015
##	60	0.7335	nan	0.1000	-0.0010
##	80	0.7131	nan	0.1000	-0.0011
##	100	0.7006	nan	0.1000	-0.0018


```

##      120      0.6886      nan      0.1000     -0.0008
##      140      0.6740      nan      0.1000     -0.0014
##      160      0.6628      nan      0.1000     -0.0017
##      180      0.6522      nan      0.1000     -0.0010
##      200      0.6423      nan      0.1000     -0.0005
##      220      0.6353      nan      0.1000     -0.0020
##      240      0.6251      nan      0.1000     -0.0017
##      250      0.6211      nan      0.1000     -0.0013
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2533      nan      0.1000     0.0369
##      2      1.1914      nan      0.1000     0.0304
##      3      1.1304      nan      0.1000     0.0280
##      4      1.0808      nan      0.1000     0.0214
##      5      1.0411      nan      0.1000     0.0177
##      6      1.0095      nan      0.1000     0.0131
##      7      0.9791      nan      0.1000     0.0118
##      8      0.9487      nan      0.1000     0.0146
##      9      0.9234      nan      0.1000     0.0114
##     10      0.9050      nan      0.1000     0.0081
##     20      0.7868      nan      0.1000    -0.0018
##     40      0.7190      nan      0.1000    -0.0002
##     60      0.6929      nan      0.1000    -0.0013
##     80      0.6706      nan      0.1000    -0.0012
##    100      0.6511      nan      0.1000    -0.0008
##    120      0.6313      nan      0.1000    -0.0028
##    140      0.6161      nan      0.1000    -0.0010
##    160      0.6016      nan      0.1000    -0.0019
##    180      0.5875      nan      0.1000    -0.0013
##    200      0.5754      nan      0.1000    -0.0021
##    220      0.5613      nan      0.1000    -0.0011
##    240      0.5456      nan      0.1000    -0.0014
##    250      0.5423      nan      0.1000    -0.0016
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2526      nan      0.1000     0.0406
##      2      1.1845      nan      0.1000     0.0346
##      3      1.1309      nan      0.1000     0.0246
##      4      1.0809      nan      0.1000     0.0250
##      5      1.0380      nan      0.1000     0.0205
##      6      1.0017      nan      0.1000     0.0164
##      7      0.9661      nan      0.1000     0.0146
##      8      0.9372      nan      0.1000     0.0130
##      9      0.9115      nan      0.1000     0.0105
##     10      0.8921      nan      0.1000     0.0084
##     20      0.7698      nan      0.1000     0.0014

```

##	40	0.6902	nan	0.1000	-0.0016
##	60	0.6501	nan	0.1000	-0.0019
##	80	0.6200	nan	0.1000	-0.0006
##	100	0.5995	nan	0.1000	-0.0026
##	120	0.5766	nan	0.1000	-0.0019
##	140	0.5576	nan	0.1000	-0.0020
##	160	0.5428	nan	0.1000	-0.0027
##	180	0.5276	nan	0.1000	-0.0026
##	200	0.5091	nan	0.1000	-0.0011
##	220	0.4954	nan	0.1000	-0.0014
##	240	0.4801	nan	0.1000	-0.0028
##	250	0.4748	nan	0.1000	-0.0021

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2578	nan	0.1000	0.0379
##	2	1.1876	nan	0.1000	0.0342
##	3	1.1214	nan	0.1000	0.0297
##	4	1.0650	nan	0.1000	0.0254
##	5	1.0182	nan	0.1000	0.0188
##	6	0.9812	nan	0.1000	0.0172
##	7	0.9484	nan	0.1000	0.0116
##	8	0.9182	nan	0.1000	0.0116
##	9	0.8929	nan	0.1000	0.0068
##	10	0.8704	nan	0.1000	0.0085
##	20	0.7522	nan	0.1000	0.0002
##	40	0.6778	nan	0.1000	-0.0019
##	60	0.6318	nan	0.1000	-0.0015
##	80	0.5982	nan	0.1000	-0.0011
##	100	0.5669	nan	0.1000	-0.0032
##	120	0.5451	nan	0.1000	-0.0012
##	140	0.5224	nan	0.1000	-0.0002
##	160	0.5005	nan	0.1000	-0.0027
##	180	0.4834	nan	0.1000	-0.0015
##	200	0.4693	nan	0.1000	-0.0009
##	220	0.4530	nan	0.1000	-0.0016
##	240	0.4419	nan	0.1000	-0.0035
##	250	0.4324	nan	0.1000	-0.0016

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2761	nan	0.1000	0.0283
##	2	1.2243	nan	0.1000	0.0231
##	3	1.1816	nan	0.1000	0.0192
##	4	1.1477	nan	0.1000	0.0158
##	5	1.1174	nan	0.1000	0.0122
##	6	1.0930	nan	0.1000	0.0116
##	7	1.0704	nan	0.1000	0.0102

```

##      8      1.0499      nan      0.1000      0.0083
##      9      1.0340      nan      0.1000      0.0078
##     10      1.0167      nan      0.1000      0.0082
##     20      0.9152      nan      0.1000      0.0021
##     40      0.8226      nan      0.1000      0.0007
##     60      0.7895      nan      0.1000      0.0000
##     80      0.7690      nan      0.1000     -0.0008
##    100      0.7612      nan      0.1000     -0.0010
##    120      0.7541      nan      0.1000     -0.0004
##    140      0.7491      nan      0.1000     -0.0012
##    160      0.7443      nan      0.1000     -0.0006
##    180      0.7405      nan      0.1000     -0.0009
##    200      0.7369      nan      0.1000     -0.0009
##    220      0.7329      nan      0.1000     -0.0007
##    240      0.7287      nan      0.1000     -0.0014
##    250      0.7268      nan      0.1000     -0.0011
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2642      nan      0.1000      0.0359
##      2      1.2066      nan      0.1000      0.0279
##      3      1.1624      nan      0.1000      0.0237
##      4      1.1213      nan      0.1000      0.0202
##      5      1.0853      nan      0.1000      0.0162
##      6      1.0560      nan      0.1000      0.0116
##      7      1.0275      nan      0.1000      0.0125
##      8      1.0040      nan      0.1000      0.0109
##      9      0.9823      nan      0.1000      0.0077
##     10      0.9612      nan      0.1000      0.0105
##     20      0.8409      nan      0.1000      0.0026
##     40      0.7578      nan      0.1000     -0.0007
##     60      0.7294      nan      0.1000     -0.0007
##     80      0.7095      nan      0.1000     -0.0026
##    100      0.6965      nan      0.1000     -0.0012
##    120      0.6857      nan      0.1000     -0.0022
##    140      0.6751      nan      0.1000     -0.0004
##    160      0.6650      nan      0.1000     -0.0018
##    180      0.6581      nan      0.1000     -0.0017
##    200      0.6520      nan      0.1000     -0.0009
##    220      0.6436      nan      0.1000     -0.0011
##    240      0.6351      nan      0.1000     -0.0009
##    250      0.6312      nan      0.1000     -0.0016
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2577      nan      0.1000      0.0367
##      2      1.1908      nan      0.1000      0.0288
##      3      1.1314      nan      0.1000      0.0257

```

##	4	1.0832	nan	0.1000	0.0231
##	5	1.0473	nan	0.1000	0.0147
##	6	1.0104	nan	0.1000	0.0174
##	7	0.9743	nan	0.1000	0.0139
##	8	0.9460	nan	0.1000	0.0113
##	9	0.9236	nan	0.1000	0.0105
##	10	0.9026	nan	0.1000	0.0077
##	20	0.7942	nan	0.1000	0.0009
##	40	0.7292	nan	0.1000	-0.0018
##	60	0.6933	nan	0.1000	-0.0013
##	80	0.6650	nan	0.1000	-0.0008
##	100	0.6458	nan	0.1000	-0.0020
##	120	0.6252	nan	0.1000	-0.0018
##	140	0.6106	nan	0.1000	-0.0010
##	160	0.5953	nan	0.1000	-0.0009
##	180	0.5810	nan	0.1000	-0.0014
##	200	0.5683	nan	0.1000	-0.0016
##	220	0.5544	nan	0.1000	-0.0009
##	240	0.5425	nan	0.1000	-0.0010
##	250	0.5367	nan	0.1000	-0.0012
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2542	nan	0.1000	0.0373
##	2	1.1874	nan	0.1000	0.0310
##	3	1.1277	nan	0.1000	0.0279
##	4	1.0765	nan	0.1000	0.0231
##	5	1.0393	nan	0.1000	0.0179
##	6	1.0012	nan	0.1000	0.0141
##	7	0.9658	nan	0.1000	0.0139
##	8	0.9421	nan	0.1000	0.0101
##	9	0.9187	nan	0.1000	0.0088
##	10	0.8966	nan	0.1000	0.0095
##	20	0.7715	nan	0.1000	0.0019
##	40	0.6966	nan	0.1000	-0.0017
##	60	0.6516	nan	0.1000	-0.0010
##	80	0.6206	nan	0.1000	-0.0015
##	100	0.5991	nan	0.1000	-0.0028
##	120	0.5818	nan	0.1000	-0.0019
##	140	0.5644	nan	0.1000	-0.0018
##	160	0.5476	nan	0.1000	-0.0016
##	180	0.5329	nan	0.1000	-0.0016
##	200	0.5212	nan	0.1000	-0.0016
##	220	0.5055	nan	0.1000	-0.0032
##	240	0.4926	nan	0.1000	-0.0013
##	250	0.4850	nan	0.1000	-0.0006
##					

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2523	nan	0.1000	0.0397
##	2	1.1844	nan	0.1000	0.0301
##	3	1.1269	nan	0.1000	0.0263
##	4	1.0773	nan	0.1000	0.0216
##	5	1.0310	nan	0.1000	0.0211
##	6	0.9902	nan	0.1000	0.0176
##	7	0.9582	nan	0.1000	0.0119
##	8	0.9338	nan	0.1000	0.0106
##	9	0.9054	nan	0.1000	0.0120
##	10	0.8869	nan	0.1000	0.0083
##	20	0.7620	nan	0.1000	-0.0001
##	40	0.6734	nan	0.1000	-0.0010
##	60	0.6333	nan	0.1000	-0.0004
##	80	0.5982	nan	0.1000	-0.0023
##	100	0.5685	nan	0.1000	-0.0013
##	120	0.5436	nan	0.1000	-0.0024
##	140	0.5196	nan	0.1000	-0.0012
##	160	0.5010	nan	0.1000	-0.0031
##	180	0.4832	nan	0.1000	-0.0032
##	200	0.4677	nan	0.1000	-0.0014
##	220	0.4504	nan	0.1000	-0.0016
##	240	0.4333	nan	0.1000	-0.0013
##	250	0.4273	nan	0.1000	-0.0022
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2726	nan	0.1000	0.0321
##	2	1.2186	nan	0.1000	0.0259
##	3	1.1770	nan	0.1000	0.0190
##	4	1.1350	nan	0.1000	0.0186
##	5	1.1017	nan	0.1000	0.0147
##	6	1.0810	nan	0.1000	0.0096
##	7	1.0570	nan	0.1000	0.0105
##	8	1.0375	nan	0.1000	0.0098
##	9	1.0188	nan	0.1000	0.0081
##	10	1.0020	nan	0.1000	0.0072
##	20	0.9034	nan	0.1000	0.0023
##	40	0.8070	nan	0.1000	-0.0001
##	60	0.7671	nan	0.1000	-0.0003
##	80	0.7500	nan	0.1000	-0.0007
##	100	0.7415	nan	0.1000	-0.0008
##	120	0.7349	nan	0.1000	-0.0008
##	140	0.7282	nan	0.1000	-0.0006
##	160	0.7222	nan	0.1000	-0.0006
##	180	0.7190	nan	0.1000	-0.0003
##	200	0.7160	nan	0.1000	-0.0013

##	220	0.7128	nan	0.1000	-0.0008
##	240	0.7092	nan	0.1000	-0.0012
##	250	0.7066	nan	0.1000	-0.0005

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2597	nan	0.1000	0.0362
##	2	1.2019	nan	0.1000	0.0279
##	3	1.1504	nan	0.1000	0.0230
##	4	1.1084	nan	0.1000	0.0168
##	5	1.0702	nan	0.1000	0.0184
##	6	1.0378	nan	0.1000	0.0136
##	7	1.0090	nan	0.1000	0.0131
##	8	0.9848	nan	0.1000	0.0115
##	9	0.9632	nan	0.1000	0.0078
##	10	0.9410	nan	0.1000	0.0094
##	20	0.8215	nan	0.1000	0.0037
##	40	0.7478	nan	0.1000	-0.0001
##	60	0.7184	nan	0.1000	-0.0008
##	80	0.6985	nan	0.1000	-0.0007
##	100	0.6799	nan	0.1000	-0.0013
##	120	0.6660	nan	0.1000	0.0002
##	140	0.6534	nan	0.1000	-0.0009
##	160	0.6423	nan	0.1000	-0.0016
##	180	0.6325	nan	0.1000	-0.0016
##	200	0.6226	nan	0.1000	-0.0011
##	220	0.6139	nan	0.1000	-0.0010
##	240	0.6043	nan	0.1000	-0.0017
##	250	0.6016	nan	0.1000	-0.0012

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2549	nan	0.1000	0.0413
##	2	1.1843	nan	0.1000	0.0345
##	3	1.1274	nan	0.1000	0.0245
##	4	1.0809	nan	0.1000	0.0213
##	5	1.0418	nan	0.1000	0.0178
##	6	1.0056	nan	0.1000	0.0128
##	7	0.9797	nan	0.1000	0.0122
##	8	0.9562	nan	0.1000	0.0083
##	9	0.9288	nan	0.1000	0.0116
##	10	0.9070	nan	0.1000	0.0088
##	20	0.7819	nan	0.1000	0.0013
##	40	0.7119	nan	0.1000	-0.0008
##	60	0.6706	nan	0.1000	-0.0013
##	80	0.6482	nan	0.1000	-0.0018
##	100	0.6271	nan	0.1000	-0.0027
##	120	0.6065	nan	0.1000	-0.0013

```

##      140      0.5901      nan      0.1000     -0.0021
##      160      0.5709      nan      0.1000     -0.0009
##      180      0.5563      nan      0.1000     -0.0005
##      200      0.5427      nan      0.1000     -0.0025
##      220      0.5333      nan      0.1000     -0.0012
##      240      0.5239      nan      0.1000     -0.0015
##      250      0.5186      nan      0.1000     -0.0018
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2503      nan      0.1000     0.0405
##      2      1.1806      nan      0.1000     0.0343
##      3      1.1211      nan      0.1000     0.0276
##      4      1.0676      nan      0.1000     0.0242
##      5      1.0226      nan      0.1000     0.0193
##      6      0.9842      nan      0.1000     0.0164
##      7      0.9549      nan      0.1000     0.0120
##      8      0.9274      nan      0.1000     0.0125
##      9      0.8994      nan      0.1000     0.0100
##     10      0.8810      nan      0.1000     0.0068
##     20      0.7641      nan      0.1000     0.0015
##     40      0.6902      nan      0.1000    -0.0013
##     60      0.6497      nan      0.1000    -0.0026
##     80      0.6225      nan      0.1000    -0.0020
##    100      0.5898      nan      0.1000    -0.0009
##    120      0.5662      nan      0.1000    -0.0014
##    140      0.5460      nan      0.1000    -0.0015
##    160      0.5293      nan      0.1000    -0.0029
##    180      0.5142      nan      0.1000    -0.0018
##    200      0.5021      nan      0.1000    -0.0021
##    220      0.4850      nan      0.1000    -0.0025
##    240      0.4738      nan      0.1000    -0.0010
##    250      0.4697      nan      0.1000    -0.0025
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2510      nan      0.1000     0.0400
##      2      1.1788      nan      0.1000     0.0356
##      3      1.1185      nan      0.1000     0.0284
##      4      1.0652      nan      0.1000     0.0255
##      5      1.0228      nan      0.1000     0.0209
##      6      0.9819      nan      0.1000     0.0175
##      7      0.9467      nan      0.1000     0.0146
##      8      0.9149      nan      0.1000     0.0133
##      9      0.8872      nan      0.1000     0.0105
##     10      0.8681      nan      0.1000     0.0067
##     20      0.7435      nan      0.1000     0.0013
##     40      0.6620      nan      0.1000    -0.0012

```

##	60	0.6096	nan	0.1000	-0.0011
##	80	0.5775	nan	0.1000	-0.0017
##	100	0.5491	nan	0.1000	-0.0016
##	120	0.5248	nan	0.1000	-0.0030
##	140	0.5082	nan	0.1000	-0.0016
##	160	0.4873	nan	0.1000	-0.0022
##	180	0.4700	nan	0.1000	-0.0020
##	200	0.4543	nan	0.1000	-0.0013
##	220	0.4388	nan	0.1000	-0.0031
##	240	0.4269	nan	0.1000	-0.0013
##	250	0.4237	nan	0.1000	-0.0015

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2737	nan	0.1000	0.0306
##	2	1.2188	nan	0.1000	0.0241
##	3	1.1752	nan	0.1000	0.0184
##	4	1.1384	nan	0.1000	0.0168
##	5	1.1052	nan	0.1000	0.0134
##	6	1.0800	nan	0.1000	0.0103
##	7	1.0601	nan	0.1000	0.0106
##	8	1.0416	nan	0.1000	0.0097
##	9	1.0234	nan	0.1000	0.0070
##	10	1.0079	nan	0.1000	0.0055
##	20	0.9090	nan	0.1000	0.0034
##	40	0.8145	nan	0.1000	0.0004
##	60	0.7708	nan	0.1000	0.0003
##	80	0.7528	nan	0.1000	-0.0009
##	100	0.7433	nan	0.1000	-0.0010
##	120	0.7366	nan	0.1000	-0.0002
##	140	0.7317	nan	0.1000	-0.0007
##	160	0.7262	nan	0.1000	-0.0009
##	180	0.7217	nan	0.1000	-0.0005
##	200	0.7172	nan	0.1000	-0.0013
##	220	0.7141	nan	0.1000	-0.0012
##	240	0.7102	nan	0.1000	-0.0002
##	250	0.7086	nan	0.1000	-0.0012

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2660	nan	0.1000	0.0361
##	2	1.2076	nan	0.1000	0.0275
##	3	1.1575	nan	0.1000	0.0247
##	4	1.1197	nan	0.1000	0.0178
##	5	1.0845	nan	0.1000	0.0173
##	6	1.0529	nan	0.1000	0.0131
##	7	1.0259	nan	0.1000	0.0123
##	8	0.9969	nan	0.1000	0.0117


```

##      9      0.9748      nan      0.1000      0.0082
##     10      0.9535      nan      0.1000      0.0076
##     20      0.8305      nan      0.1000      0.0033
##     40      0.7475      nan      0.1000     -0.0002
##     60      0.7207      nan      0.1000     -0.0018
##     80      0.7037      nan      0.1000     -0.0017
##    100      0.6882      nan      0.1000     -0.0014
##    120      0.6802      nan      0.1000     -0.0010
##    140      0.6709      nan      0.1000     -0.0015
##    160      0.6613      nan      0.1000     -0.0014
##    180      0.6523      nan      0.1000     -0.0004
##    200      0.6449      nan      0.1000     -0.0011
##    220      0.6367      nan      0.1000     -0.0014
##    240      0.6286      nan      0.1000     -0.0013
##    250      0.6233      nan      0.1000     -0.0019
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##     1           1.2551           nan      0.1000    0.0346
##     2           1.1931           nan      0.1000    0.0315
##     3           1.1316           nan      0.1000    0.0272
##     4           1.0872           nan      0.1000    0.0204
##     5           1.0437           nan      0.1000    0.0187
##     6           1.0098           nan      0.1000    0.0139
##     7           0.9818           nan      0.1000    0.0107
##     8           0.9579           nan      0.1000    0.0111
##     9           0.9321           nan      0.1000    0.0129
##    10           0.9121           nan      0.1000    0.0066
##    20           0.7838           nan      0.1000    0.0022
##    40           0.7153           nan      0.1000   -0.0001
##    60           0.6832           nan      0.1000   -0.0011
##    80           0.6612           nan      0.1000   -0.0019
##   100           0.6415           nan      0.1000   -0.0020
##   120           0.6252           nan      0.1000   -0.0020
##   140           0.6084           nan      0.1000   -0.0017
##   160           0.5936           nan      0.1000   -0.0009
##   180           0.5834           nan      0.1000   -0.0021
##   200           0.5669           nan      0.1000   -0.0009
##   220           0.5568           nan      0.1000   -0.0011
##   240           0.5467           nan      0.1000   -0.0027
##   250           0.5422           nan      0.1000   -0.0015
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##     1           1.2574           nan      0.1000    0.0391
##     2           1.1858           nan      0.1000    0.0330
##     3           1.1308           nan      0.1000    0.0237
##     4           1.0768           nan      0.1000    0.0205

```

##	5	1.0311	nan	0.1000	0.0203
##	6	0.9934	nan	0.1000	0.0175
##	7	0.9643	nan	0.1000	0.0121
##	8	0.9342	nan	0.1000	0.0118
##	9	0.9061	nan	0.1000	0.0104
##	10	0.8869	nan	0.1000	0.0078
##	20	0.7608	nan	0.1000	0.0000
##	40	0.6861	nan	0.1000	-0.0012
##	60	0.6452	nan	0.1000	-0.0007
##	80	0.6147	nan	0.1000	-0.0020
##	100	0.5919	nan	0.1000	-0.0015
##	120	0.5685	nan	0.1000	-0.0015
##	140	0.5516	nan	0.1000	-0.0007
##	160	0.5322	nan	0.1000	-0.0024
##	180	0.5188	nan	0.1000	-0.0013
##	200	0.5045	nan	0.1000	-0.0011
##	220	0.4913	nan	0.1000	-0.0012
##	240	0.4791	nan	0.1000	-0.0021
##	250	0.4735	nan	0.1000	-0.0011

##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2526	nan	0.1000	0.0413
##	2	1.1759	nan	0.1000	0.0342
##	3	1.1130	nan	0.1000	0.0286
##	4	1.0692	nan	0.1000	0.0198
##	5	1.0244	nan	0.1000	0.0208
##	6	0.9858	nan	0.1000	0.0160
##	7	0.9512	nan	0.1000	0.0156
##	8	0.9230	nan	0.1000	0.0086
##	9	0.8945	nan	0.1000	0.0128
##	10	0.8741	nan	0.1000	0.0071
##	20	0.7489	nan	0.1000	0.0012
##	40	0.6645	nan	0.1000	-0.0017
##	60	0.6174	nan	0.1000	-0.0018
##	80	0.5875	nan	0.1000	-0.0022
##	100	0.5620	nan	0.1000	-0.0009
##	120	0.5374	nan	0.1000	-0.0026
##	140	0.5202	nan	0.1000	-0.0017
##	160	0.5003	nan	0.1000	-0.0014
##	180	0.4829	nan	0.1000	-0.0021
##	200	0.4660	nan	0.1000	-0.0024
##	220	0.4560	nan	0.1000	-0.0024
##	240	0.4425	nan	0.1000	-0.0020
##	250	0.4349	nan	0.1000	-0.0021

##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve

```

##      1      1.2722      nan      0.1000      0.0322
##      2      1.2256      nan      0.1000      0.0203
##      3      1.1806      nan      0.1000      0.0216
##      4      1.1444      nan      0.1000      0.0179
##      5      1.1141      nan      0.1000      0.0147
##      6      1.0890      nan      0.1000      0.0118
##      7      1.0707      nan      0.1000      0.0085
##      8      1.0493      nan      0.1000      0.0108
##      9      1.0332      nan      0.1000      0.0069
##     10      1.0157      nan      0.1000      0.0086
##     20      0.9131      nan      0.1000      0.0039
##     40      0.8200      nan      0.1000      0.0010
##     60      0.7820      nan      0.1000      0.0004
##     80      0.7654      nan      0.1000     -0.0014
##    100      0.7561      nan      0.1000     -0.0009
##    120      0.7476      nan      0.1000     -0.0011
##    140      0.7412      nan      0.1000     -0.0003
##    160      0.7365      nan      0.1000     -0.0004
##    180      0.7340      nan      0.1000     -0.0005
##    200      0.7299      nan      0.1000     -0.0007
##    220      0.7266      nan      0.1000     -0.0008
##    240      0.7237      nan      0.1000     -0.0008
##    250      0.7234      nan      0.1000     -0.0005
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1      1.2651      nan      0.1000      0.0334
##      2      1.2070      nan      0.1000      0.0269
##      3      1.1557      nan      0.1000      0.0231
##      4      1.1153      nan      0.1000      0.0171
##      5      1.0815      nan      0.1000      0.0155
##      6      1.0482      nan      0.1000      0.0152
##      7      1.0172      nan      0.1000      0.0130
##      8      0.9922      nan      0.1000      0.0112
##      9      0.9735      nan      0.1000      0.0078
##     10      0.9545      nan      0.1000      0.0083
##     20      0.8345      nan      0.1000      0.0033
##     40      0.7573      nan      0.1000     -0.0001
##     60      0.7333      nan      0.1000     -0.0011
##     80      0.7105      nan      0.1000     -0.0006
##    100      0.6966      nan      0.1000     -0.0011
##    120      0.6815      nan      0.1000     -0.0012
##    140      0.6641      nan      0.1000     -0.0008
##    160      0.6482      nan      0.1000     -0.0016
##    180      0.6413      nan      0.1000     -0.0012
##    200      0.6329      nan      0.1000     -0.0008
##    220      0.6240      nan      0.1000     -0.0015

```

```

##      240      0.6160      nan      0.1000     -0.0011
##      250      0.6111      nan      0.1000     -0.0017
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           1.2637           nan      0.1000     0.0352
##      2           1.1913           nan      0.1000     0.0330
##      3           1.1333           nan      0.1000     0.0254
##      4           1.0844           nan      0.1000     0.0231
##      5           1.0391           nan      0.1000     0.0195
##      6           1.0021           nan      0.1000     0.0152
##      7           0.9729           nan      0.1000     0.0118
##      8           0.9513           nan      0.1000     0.0086
##      9           0.9272           nan      0.1000     0.0096
##     10           0.9064           nan      0.1000     0.0085
##     20           0.7931           nan      0.1000     0.0041
##     40           0.7237           nan      0.1000    -0.0004
##     60           0.6981           nan      0.1000    -0.0012
##     80           0.6718           nan      0.1000    -0.0026
##    100           0.6529           nan      0.1000    -0.0016
##    120           0.6396           nan      0.1000    -0.0010
##    140           0.6233           nan      0.1000    -0.0013
##    160           0.6101           nan      0.1000    -0.0011
##    180           0.5932           nan      0.1000    -0.0015
##    200           0.5794           nan      0.1000    -0.0010
##    220           0.5645           nan      0.1000    -0.0011
##    240           0.5498           nan      0.1000    -0.0018
##    250           0.5448           nan      0.1000    -0.0012
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           1.2583           nan      0.1000     0.0380
##      2           1.1835           nan      0.1000     0.0345
##      3           1.1231           nan      0.1000     0.0268
##      4           1.0730           nan      0.1000     0.0218
##      5           1.0329           nan      0.1000     0.0173
##      6           0.9952           nan      0.1000     0.0161
##      7           0.9630           nan      0.1000     0.0130
##      8           0.9348           nan      0.1000     0.0126
##      9           0.9109           nan      0.1000     0.0108
##     10           0.8940           nan      0.1000     0.0058
##     20           0.7714           nan      0.1000     0.0009
##     40           0.6915           nan      0.1000    -0.0022
##     60           0.6527           nan      0.1000    -0.0006
##     80           0.6232           nan      0.1000    -0.0011
##    100           0.6008           nan      0.1000    -0.0018
##    120           0.5783           nan      0.1000    -0.0017
##    140           0.5562           nan      0.1000    -0.0023

```

```

##      160      0.5375      nan      0.1000     -0.0014
##      180      0.5165      nan      0.1000     -0.0019
##      200      0.5043      nan      0.1000     -0.0021
##      220      0.4907      nan      0.1000     -0.0022
##      240      0.4783      nan      0.1000     -0.0019
##      250      0.4728      nan      0.1000     -0.0022
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2435      nan      0.1000     0.0421
##      2      1.1748      nan      0.1000     0.0315
##      3      1.1147      nan      0.1000     0.0294
##      4      1.0672      nan      0.1000     0.0236
##      5      1.0265      nan      0.1000     0.0188
##      6      0.9873      nan      0.1000     0.0165
##      7      0.9555      nan      0.1000     0.0151
##      8      0.9289      nan      0.1000     0.0100
##      9      0.9030      nan      0.1000     0.0085
##     10      0.8836      nan      0.1000     0.0077
##     20      0.7615      nan      0.1000     0.0016
##     40      0.6861      nan      0.1000    -0.0016
##     60      0.6393      nan      0.1000    -0.0017
##     80      0.6022      nan      0.1000    -0.0021
##    100      0.5787      nan      0.1000    -0.0023
##    120      0.5532      nan      0.1000    -0.0024
##    140      0.5303      nan      0.1000    -0.0017
##    160      0.5079      nan      0.1000    -0.0015
##    180      0.4894      nan      0.1000    -0.0017
##    200      0.4716      nan      0.1000    -0.0019
##    220      0.4537      nan      0.1000    -0.0017
##    240      0.4389      nan      0.1000    -0.0007
##    250      0.4327      nan      0.1000    -0.0013
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2786      nan      0.1000     0.0297
##      2      1.2244      nan      0.1000     0.0249
##      3      1.1873      nan      0.1000     0.0203
##      4      1.1524      nan      0.1000     0.0140
##      5      1.1216      nan      0.1000     0.0141
##      6      1.0961      nan      0.1000     0.0125
##      7      1.0730      nan      0.1000     0.0103
##      8      1.0536      nan      0.1000     0.0086
##      9      1.0376      nan      0.1000     0.0067
##     10      1.0208      nan      0.1000     0.0083
##     20      0.9191      nan      0.1000     0.0025
##     40      0.8304      nan      0.1000     0.0006
##     60      0.7967      nan      0.1000    -0.0009

```

##	80	0.7809	nan	0.1000	-0.0007
##	100	0.7724	nan	0.1000	-0.0012
##	120	0.7661	nan	0.1000	-0.0004
##	140	0.7613	nan	0.1000	-0.0012
##	160	0.7558	nan	0.1000	-0.0004
##	180	0.7511	nan	0.1000	-0.0004
##	200	0.7460	nan	0.1000	-0.0006
##	220	0.7421	nan	0.1000	-0.0013
##	240	0.7388	nan	0.1000	-0.0007
##	250	0.7365	nan	0.1000	-0.0009

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2662	nan	0.1000	0.0324
##	2	1.2096	nan	0.1000	0.0223
##	3	1.1565	nan	0.1000	0.0234
##	4	1.1199	nan	0.1000	0.0184
##	5	1.0844	nan	0.1000	0.0144
##	6	1.0520	nan	0.1000	0.0153
##	7	1.0254	nan	0.1000	0.0130
##	8	1.0023	nan	0.1000	0.0091
##	9	0.9789	nan	0.1000	0.0100
##	10	0.9608	nan	0.1000	0.0074
##	20	0.8450	nan	0.1000	0.0015
##	40	0.7728	nan	0.1000	-0.0010
##	60	0.7478	nan	0.1000	-0.0011
##	80	0.7312	nan	0.1000	-0.0005
##	100	0.7193	nan	0.1000	-0.0008
##	120	0.7053	nan	0.1000	-0.0015
##	140	0.6952	nan	0.1000	-0.0006
##	160	0.6855	nan	0.1000	-0.0007
##	180	0.6758	nan	0.1000	-0.0013
##	200	0.6663	nan	0.1000	-0.0011
##	220	0.6599	nan	0.1000	-0.0008
##	240	0.6511	nan	0.1000	-0.0015
##	250	0.6471	nan	0.1000	-0.0012

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2605	nan	0.1000	0.0359
##	2	1.1990	nan	0.1000	0.0295
##	3	1.1393	nan	0.1000	0.0250
##	4	1.0933	nan	0.1000	0.0198
##	5	1.0564	nan	0.1000	0.0142
##	6	1.0189	nan	0.1000	0.0153
##	7	0.9922	nan	0.1000	0.0114
##	8	0.9677	nan	0.1000	0.0095
##	9	0.9451	nan	0.1000	0.0105

```

##      10      0.9247      nan      0.1000      0.0070
##      20      0.8103      nan      0.1000      0.0028
##      40      0.7386      nan      0.1000     -0.0008
##      60      0.7043      nan      0.1000     -0.0029
##      80      0.6768      nan      0.1000     -0.0006
##     100      0.6509      nan      0.1000     -0.0012
##     120      0.6340      nan      0.1000     -0.0015
##     140      0.6159      nan      0.1000     -0.0020
##     160      0.6042      nan      0.1000     -0.0015
##     180      0.5892      nan      0.1000     -0.0009
##     200      0.5800      nan      0.1000     -0.0015
##     220      0.5693      nan      0.1000     -0.0015
##     240      0.5583      nan      0.1000     -0.0014
##     250      0.5545      nan      0.1000     -0.0009
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1          1.2535          nan      0.1000   0.0373
##      2          1.1854          nan      0.1000   0.0315
##      3          1.1321          nan      0.1000   0.0243
##      4          1.0809          nan      0.1000   0.0232
##      5          1.0425          nan      0.1000   0.0193
##      6          1.0039          nan      0.1000   0.0155
##      7          0.9741          nan      0.1000   0.0124
##      8          0.9470          nan      0.1000   0.0120
##      9          0.9198          nan      0.1000   0.0119
##     10          0.9026          nan      0.1000   0.0068
##     20          0.7913          nan      0.1000   0.0012
##     40          0.7186          nan      0.1000  -0.0003
##     60          0.6837          nan      0.1000  -0.0016
##     80          0.6483          nan      0.1000  -0.0009
##    100          0.6236          nan      0.1000  -0.0014
##    120          0.5982          nan      0.1000  -0.0024
##    140          0.5790          nan      0.1000  -0.0015
##    160          0.5596          nan      0.1000  -0.0035
##    180          0.5411          nan      0.1000  -0.0007
##    200          0.5267          nan      0.1000  -0.0020
##    220          0.5139          nan      0.1000  -0.0012
##    240          0.4973          nan      0.1000  -0.0021
##    250          0.4919          nan      0.1000  -0.0009
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1          1.2535          nan      0.1000   0.0388
##      2          1.1840          nan      0.1000   0.0318
##      3          1.1232          nan      0.1000   0.0262
##      4          1.0698          nan      0.1000   0.0244
##      5          1.0285          nan      0.1000   0.0185

```

##	6	0.9955	nan	0.1000	0.0134
##	7	0.9661	nan	0.1000	0.0146
##	8	0.9362	nan	0.1000	0.0139
##	9	0.9138	nan	0.1000	0.0090
##	10	0.8958	nan	0.1000	0.0069
##	20	0.7770	nan	0.1000	0.0004
##	40	0.7015	nan	0.1000	-0.0011
##	60	0.6610	nan	0.1000	-0.0024
##	80	0.6300	nan	0.1000	-0.0035
##	100	0.5995	nan	0.1000	-0.0016
##	120	0.5673	nan	0.1000	-0.0024
##	140	0.5459	nan	0.1000	-0.0038
##	160	0.5245	nan	0.1000	-0.0029
##	180	0.5053	nan	0.1000	-0.0016
##	200	0.4902	nan	0.1000	-0.0012
##	220	0.4748	nan	0.1000	-0.0026
##	240	0.4577	nan	0.1000	-0.0022
##	250	0.4513	nan	0.1000	-0.0027

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2725	nan	0.1000	0.0315
##	2	1.2219	nan	0.1000	0.0246
##	3	1.1809	nan	0.1000	0.0195
##	4	1.1501	nan	0.1000	0.0136
##	5	1.1172	nan	0.1000	0.0151
##	6	1.0930	nan	0.1000	0.0122
##	7	1.0669	nan	0.1000	0.0116
##	8	1.0449	nan	0.1000	0.0093
##	9	1.0285	nan	0.1000	0.0066
##	10	1.0180	nan	0.1000	0.0038
##	20	0.9148	nan	0.1000	0.0024
##	40	0.8255	nan	0.1000	-0.0002
##	60	0.7874	nan	0.1000	0.0005
##	80	0.7699	nan	0.1000	-0.0002
##	100	0.7627	nan	0.1000	-0.0013
##	120	0.7552	nan	0.1000	-0.0003
##	140	0.7490	nan	0.1000	-0.0009
##	160	0.7430	nan	0.1000	-0.0010
##	180	0.7382	nan	0.1000	-0.0008
##	200	0.7349	nan	0.1000	-0.0005
##	220	0.7314	nan	0.1000	-0.0013
##	240	0.7268	nan	0.1000	-0.0007
##	250	0.7253	nan	0.1000	-0.0006

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2623	nan	0.1000	0.0345


```

##      2      1.2026      nan    0.1000    0.0294
##      3      1.1548      nan    0.1000    0.0218
##      4      1.1129      nan    0.1000    0.0177
##      5      1.0812      nan    0.1000    0.0157
##      6      1.0509      nan    0.1000    0.0147
##      7      1.0280      nan    0.1000    0.0099
##      8      1.0037      nan    0.1000    0.0104
##      9      0.9810      nan    0.1000    0.0097
##     10      0.9609      nan    0.1000    0.0089
##     20      0.8459      nan    0.1000    0.0019
##     40      0.7693      nan    0.1000   -0.0002
##     60      0.7363      nan    0.1000   -0.0008
##     80      0.7162      nan    0.1000   -0.0023
##    100      0.7052      nan    0.1000   -0.0015
##    120      0.6917      nan    0.1000   -0.0028
##    140      0.6756      nan    0.1000   -0.0020
##    160      0.6652      nan    0.1000   -0.0003
##    180      0.6557      nan    0.1000   -0.0014
##    200      0.6468      nan    0.1000   -0.0006
##    220      0.6353      nan    0.1000   -0.0006
##    240      0.6277      nan    0.1000   -0.0007
##    250      0.6244      nan    0.1000   -0.0011
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1      1.2566      nan    0.1000    0.0386
##      2      1.1937      nan    0.1000    0.0297
##      3      1.1355      nan    0.1000    0.0258
##      4      1.0909      nan    0.1000    0.0201
##      5      1.0509      nan    0.1000    0.0179
##      6      1.0145      nan    0.1000    0.0162
##      7      0.9840      nan    0.1000    0.0123
##      8      0.9551      nan    0.1000    0.0122
##      9      0.9369      nan    0.1000    0.0088
##     10      0.9164      nan    0.1000    0.0072
##     20      0.8017      nan    0.1000    0.0007
##     40      0.7255      nan    0.1000   -0.0002
##     60      0.6940      nan    0.1000   -0.0009
##     80      0.6684      nan    0.1000   -0.0008
##    100      0.6504      nan    0.1000   -0.0011
##    120      0.6325      nan    0.1000   -0.0013
##    140      0.6140      nan    0.1000   -0.0013
##    160      0.5974      nan    0.1000   -0.0002
##    180      0.5833      nan    0.1000   -0.0010
##    200      0.5666      nan    0.1000   -0.0011
##    220      0.5550      nan    0.1000   -0.0020
##    240      0.5451      nan    0.1000   -0.0023

```

```

##      250      0.5372      nan      0.1000     -0.0015
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2498      nan      0.1000     0.0417
##      2      1.1832      nan      0.1000     0.0281
##      3      1.1271      nan      0.1000     0.0244
##      4      1.0777      nan      0.1000     0.0218
##      5      1.0407      nan      0.1000     0.0151
##      6      1.0058      nan      0.1000     0.0141
##      7      0.9753      nan      0.1000     0.0148
##      8      0.9445      nan      0.1000     0.0133
##      9      0.9192      nan      0.1000     0.0115
##     10      0.9036      nan      0.1000     0.0060
##     20      0.7800      nan      0.1000     0.0017
##     40      0.7027      nan      0.1000    -0.0018
##     60      0.6571      nan      0.1000    -0.0015
##     80      0.6285      nan      0.1000    -0.0028
##    100      0.6088      nan      0.1000    -0.0028
##    120      0.5816      nan      0.1000    -0.0033
##    140      0.5649      nan      0.1000    -0.0020
##    160      0.5430      nan      0.1000    -0.0012
##    180      0.5294      nan      0.1000    -0.0016
##    200      0.5109      nan      0.1000    -0.0021
##    220      0.4943      nan      0.1000    -0.0027
##    240      0.4793      nan      0.1000    -0.0010
##    250      0.4732      nan      0.1000    -0.0016
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2514      nan      0.1000     0.0374
##      2      1.1825      nan      0.1000     0.0323
##      3      1.1249      nan      0.1000     0.0244
##      4      1.0758      nan      0.1000     0.0248
##      5      1.0288      nan      0.1000     0.0205
##      6      0.9950      nan      0.1000     0.0151
##      7      0.9607      nan      0.1000     0.0120
##      8      0.9307      nan      0.1000     0.0132
##      9      0.9071      nan      0.1000     0.0100
##     10      0.8828      nan      0.1000     0.0094
##     20      0.7598      nan      0.1000     0.0013
##     40      0.6744      nan      0.1000    -0.0017
##     60      0.6343      nan      0.1000    -0.0018
##     80      0.5947      nan      0.1000    -0.0013
##    100      0.5667      nan      0.1000    -0.0019
##    120      0.5457      nan      0.1000    -0.0029
##    140      0.5235      nan      0.1000    -0.0024
##    160      0.5036      nan      0.1000    -0.0019

```

```

##      180      0.4856      nan      0.1000     -0.0030
##      200      0.4678      nan      0.1000     -0.0020
##      220      0.4534      nan      0.1000     -0.0023
##      240      0.4412      nan      0.1000     -0.0018
##      250      0.4336      nan      0.1000     -0.0019
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2803      nan      0.1000     0.0300
##      2      1.2282      nan      0.1000     0.0252
##      3      1.1859      nan      0.1000     0.0212
##      4      1.1515      nan      0.1000     0.0150
##      5      1.1195      nan      0.1000     0.0146
##      6      1.0928      nan      0.1000     0.0127
##      7      1.0679      nan      0.1000     0.0107
##      8      1.0480      nan      0.1000     0.0092
##      9      1.0320      nan      0.1000     0.0078
##     10      1.0185      nan      0.1000     0.0060
##     20      0.9144      nan      0.1000     0.0035
##     40      0.8227      nan      0.1000     0.0009
##     60      0.7776      nan      0.1000    -0.0003
##     80      0.7595      nan      0.1000    -0.0005
##    100      0.7498      nan      0.1000    -0.0002
##    120      0.7441      nan      0.1000    -0.0002
##    140      0.7392      nan      0.1000    -0.0006
##    160      0.7347      nan      0.1000    -0.0004
##    180      0.7323      nan      0.1000    -0.0013
##    200      0.7259      nan      0.1000    -0.0003
##    220      0.7226      nan      0.1000    -0.0003
##    240      0.7204      nan      0.1000    -0.0004
##    250      0.7180      nan      0.1000    -0.0010
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1      1.2639      nan      0.1000     0.0365
##      2      1.2041      nan      0.1000     0.0277
##      3      1.1555      nan      0.1000     0.0224
##      4      1.1135      nan      0.1000     0.0190
##      5      1.0741      nan      0.1000     0.0174
##      6      1.0444      nan      0.1000     0.0128
##      7      1.0172      nan      0.1000     0.0120
##      8      0.9928      nan      0.1000     0.0114
##      9      0.9698      nan      0.1000     0.0105
##     10      0.9523      nan      0.1000     0.0071
##     20      0.8290      nan      0.1000     0.0023
##     40      0.7563      nan      0.1000    -0.0012
##     60      0.7274      nan      0.1000    -0.0011
##     80      0.7078      nan      0.1000    -0.0014

```

##	100	0.6940	nan	0.1000	-0.0008
##	120	0.6795	nan	0.1000	-0.0015
##	140	0.6698	nan	0.1000	-0.0008
##	160	0.6585	nan	0.1000	-0.0011
##	180	0.6453	nan	0.1000	-0.0002
##	200	0.6358	nan	0.1000	-0.0007
##	220	0.6313	nan	0.1000	-0.0019
##	240	0.6263	nan	0.1000	-0.0020
##	250	0.6222	nan	0.1000	-0.0013

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2566	nan	0.1000	0.0352
##	2	1.1939	nan	0.1000	0.0309
##	3	1.1369	nan	0.1000	0.0252
##	4	1.0871	nan	0.1000	0.0217
##	5	1.0427	nan	0.1000	0.0188
##	6	1.0083	nan	0.1000	0.0132
##	7	0.9772	nan	0.1000	0.0138
##	8	0.9496	nan	0.1000	0.0146
##	9	0.9264	nan	0.1000	0.0103
##	10	0.9027	nan	0.1000	0.0097
##	20	0.7804	nan	0.1000	0.0026
##	40	0.7150	nan	0.1000	-0.0014
##	60	0.6814	nan	0.1000	-0.0023
##	80	0.6575	nan	0.1000	-0.0021
##	100	0.6395	nan	0.1000	-0.0021
##	120	0.6198	nan	0.1000	-0.0014
##	140	0.6030	nan	0.1000	-0.0016
##	160	0.5852	nan	0.1000	-0.0013
##	180	0.5747	nan	0.1000	-0.0017
##	200	0.5616	nan	0.1000	-0.0014
##	220	0.5461	nan	0.1000	-0.0019
##	240	0.5350	nan	0.1000	-0.0025
##	250	0.5289	nan	0.1000	-0.0008

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	1.2508	nan	0.1000	0.0399
##	2	1.1799	nan	0.1000	0.0327
##	3	1.1202	nan	0.1000	0.0267
##	4	1.0681	nan	0.1000	0.0230
##	5	1.0267	nan	0.1000	0.0188
##	6	0.9903	nan	0.1000	0.0144
##	7	0.9579	nan	0.1000	0.0146
##	8	0.9311	nan	0.1000	0.0122
##	9	0.9093	nan	0.1000	0.0100
##	10	0.8920	nan	0.1000	0.0071

```

##      20      0.7669      nan      0.1000      0.0027
##      40      0.6912      nan      0.1000     -0.0021
##      60      0.6458      nan      0.1000     -0.0026
##      80      0.6124      nan      0.1000     -0.0004
##     100      0.5870      nan      0.1000     -0.0014
##     120      0.5656      nan      0.1000     -0.0013
##     140      0.5518      nan      0.1000     -0.0023
##     160      0.5341      nan      0.1000     -0.0011
##     180      0.5194      nan      0.1000     -0.0010
##     200      0.5076      nan      0.1000     -0.0026
##     220      0.4978      nan      0.1000     -0.0018
##     240      0.4803      nan      0.1000     -0.0012
##     250      0.4720      nan      0.1000     -0.0022
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1          1.2479           nan      0.1000    0.0415
##      2          1.1762           nan      0.1000    0.0296
##      3          1.1123           nan      0.1000    0.0268
##      4          1.0657           nan      0.1000    0.0197
##      5          1.0210           nan      0.1000    0.0199
##      6          0.9815           nan      0.1000    0.0159
##      7          0.9479           nan      0.1000    0.0105
##      8          0.9197           nan      0.1000    0.0126
##      9          0.8955           nan      0.1000    0.0088
##     10          0.8742           nan      0.1000    0.0090
##     20          0.7586           nan      0.1000    0.0012
##     40          0.6843           nan      0.1000   -0.0018
##     60          0.6355           nan      0.1000   -0.0027
##     80          0.5939           nan      0.1000   -0.0014
##    100          0.5648           nan      0.1000   -0.0014
##    120          0.5434           nan      0.1000   -0.0016
##    140          0.5223           nan      0.1000   -0.0021
##    160          0.5026           nan      0.1000   -0.0021
##    180          0.4889           nan      0.1000   -0.0021
##    200          0.4758           nan      0.1000   -0.0012
##    220          0.4520           nan      0.1000   -0.0008
##    240          0.4417           nan      0.1000   -0.0027
##    250          0.4358           nan      0.1000   -0.0023
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1          1.2643           nan      0.1000    0.0332
##      2          1.2100           nan      0.1000    0.0272
##      3          1.1593           nan      0.1000    0.0240
##      4          1.1180           nan      0.1000    0.0196
##      5          1.0825           nan      0.1000    0.0170
##      6          1.0508           nan      0.1000    0.0135

```

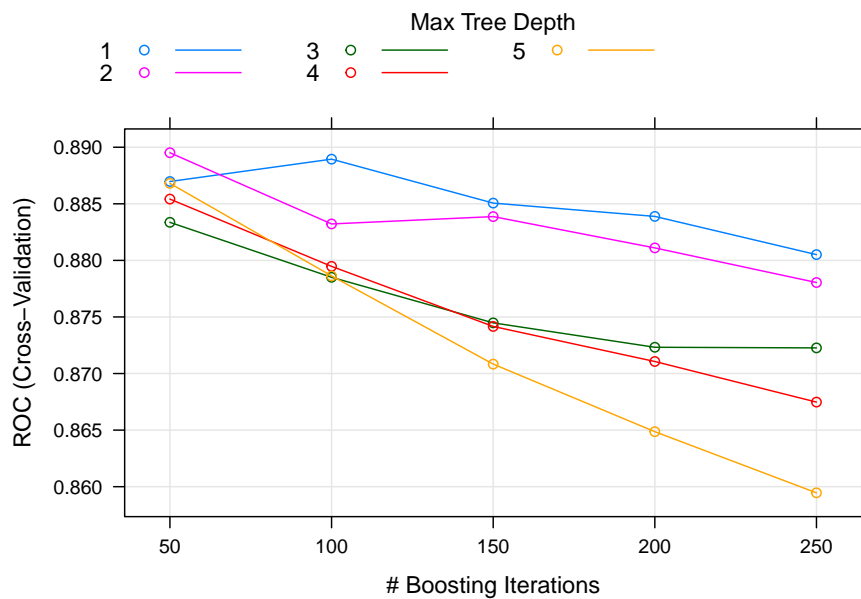
##	7	1.0219	nan	0.1000	0.0112
##	8	0.9984	nan	0.1000	0.0118
##	9	0.9778	nan	0.1000	0.0072
##	10	0.9565	nan	0.1000	0.0096
##	20	0.8389	nan	0.1000	0.0025
##	40	0.7624	nan	0.1000	-0.0009
##	50	0.7453	nan	0.1000	-0.0006

oj.gbm

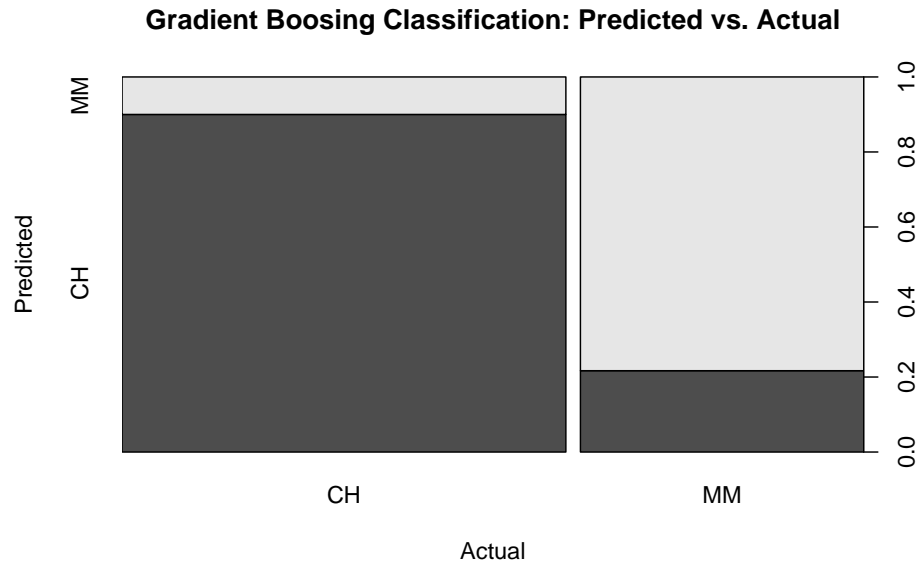
```
## Stochastic Gradient Boosting
##
## 857 samples
## 17 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 772, 770, 771, 771, 772, 771, ...
## Resampling results across tuning parameters:
##
## interaction.depth n.trees ROC Sens Spec
## 1 50 0.8869715 0.8720247 0.7245098
## 1 100 0.8889466 0.8738389 0.7302139
## 1 150 0.8850612 0.8680697 0.7303030
## 1 200 0.8838868 0.8738026 0.7213012
## 1 250 0.8805090 0.8622642 0.7275401
## 2 50 0.8895139 0.8719521 0.7451872
## 2 100 0.8832208 0.8623367 0.7334225
## 2 150 0.8838687 0.8679971 0.7394831
## 2 200 0.8811028 0.8642598 0.7245989
## 2 250 0.8780432 0.8489840 0.7336898
## 3 50 0.8833620 0.8700290 0.7245989
## 3 100 0.8785063 0.8604862 0.7454545
## 3 150 0.8744782 0.8470972 0.7156863
## 3 200 0.8723217 0.8414731 0.7246881
## 3 250 0.8722653 0.8337446 0.7217469
## 4 50 0.8854180 0.8604862 0.7605169
## 4 100 0.8794721 0.8413280 0.7336898
## 4 150 0.8741609 0.8432511 0.7308378
## 4 200 0.8710638 0.8470972 0.7249554
## 4 250 0.8674771 0.8490203 0.7218360
## 5 50 0.8868076 0.8585994 0.7574866
## 5 100 0.8786285 0.8471698 0.7486631
## 5 150 0.8708328 0.8395138 0.7458111
```

```
##      5              200      0.8648622 0.8452467 0.7455437
##      5              250      0.8594578 0.8375544 0.7336007
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 50, interaction.depth =
## 2, shrinkage = 0.1 and n.minobsinnode = 10.
```

```
plot(oj.gbm)
```



```
oj.pred <- predict(oj.gbm, oj_test, type = "raw")
plot(oj_test$Purchase, oj.pred,
     main = "Gradient Boosting Classification: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
```



```
(oj.conf <- confusionMatrix(data = oj.pred,
                             reference = oj_test$Purchase))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##      CH 117  18
##      MM  13  65
##
##           Accuracy : 0.8545
##           95% CI : (0.7998, 0.8989)
##      No Information Rate : 0.6103
##      P-Value [Acc > NIR] : 4.83e-15
##
##           Kappa : 0.6907
##
##  McNemar's Test P-Value : 0.4725
##
##           Sensitivity : 0.9000
##           Specificity : 0.7831
##      Pos Pred Value : 0.8667
##      Neg Pred Value : 0.8333
##           Prevalence : 0.6103
```



```
##          Detection Rate : 0.5493
##    Detection Prevalence : 0.6338
##          Balanced Accuracy : 0.8416
##
##          'Positive' Class : CH
##

oj.gbm.acc <- as.numeric(oj.conf$overall[1])
rm(oj.pred)
rm(oj.conf)
#plot(oj.bag$, oj.bag$finalModel$y)
#plot(varImp(oj.gbm), main="Variable Importance with Gradient Boosting")
```

7.5.0.2 Gradient Boosting Regression Example

Again using the `Carseats` data set to predict `Sales`, this time I'll use the gradient boosting method by specifying `method = "gbm"`. I'll use `tuneLength = 5` and not worry about `tuneGrid` anymore. `Caret` tunes the following hyperparameters.

- `n.trees`: number of boosting iterations (increasing `n.trees` reduces the error on training set, but may lead to over-fitting)
- `interaction.depth`: maximum tree depth (the default six - node tree appears to do an excellent job)
- `shrinkage`: learning rate (reduces the impact of each additional fitted base-learner (tree) by reducing the size of incremental steps and thus penalizes the importance of each consecutive iteration. The intuition is that it is better to improve a model by taking many small steps than by taking fewer large steps. If one of the boosting iterations turns out to be erroneous, its negative impact can be easily corrected in subsequent steps.)
- `n.minobsinnode`: minimum terminal node size

```
carseats.gbm <- train(Sales ~ .,
  data = carseats_train,
  method = "gbm", # for bagged tree
  tuneLength = 5, # choose up to 5 combinations of tuning parameters
  metric = "RMSE", # evaluate hyperparameter combinations with ROC
  trControl = trainControl(
    method = "cv", # k-fold cross validation
    number = 10, # 10 folds
    savePredictions = "final", # save predictions for the optimal tuning
    verboseIter = FALSE,
    returnData = FALSE
  )
)
```

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.6155	nan	0.1000	0.3126
##	2	7.3677	nan	0.1000	0.2471
##	3	7.1383	nan	0.1000	0.1559
##	4	6.8796	nan	0.1000	0.3000
##	5	6.6084	nan	0.1000	0.2696
##	6	6.3846	nan	0.1000	0.1575
##	7	6.1551	nan	0.1000	0.2016
##	8	5.9837	nan	0.1000	0.1171
##	9	5.7969	nan	0.1000	0.1558
##	10	5.6503	nan	0.1000	0.1243
##	20	4.5758	nan	0.1000	0.0472
##	40	3.3276	nan	0.1000	0.0043
##	60	2.6161	nan	0.1000	0.0154
##	80	2.1215	nan	0.1000	-0.0029
##	100	1.7822	nan	0.1000	-0.0166
##	120	1.5354	nan	0.1000	-0.0016
##	140	1.3313	nan	0.1000	0.0067
##	160	1.2074	nan	0.1000	-0.0030
##	180	1.0966	nan	0.1000	0.0005
##	200	1.0083	nan	0.1000	-0.0019
##	220	0.9572	nan	0.1000	-0.0008
##	240	0.9048	nan	0.1000	-0.0052
##	250	0.8922	nan	0.1000	-0.0051
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.4939	nan	0.1000	0.5303
##	2	6.9609	nan	0.1000	0.4503
##	3	6.5646	nan	0.1000	0.3312
##	4	6.2135	nan	0.1000	0.2836
##	5	5.9347	nan	0.1000	0.2472
##	6	5.6654	nan	0.1000	0.2045
##	7	5.3757	nan	0.1000	0.2134
##	8	5.1883	nan	0.1000	0.1810
##	9	5.0431	nan	0.1000	0.1205
##	10	4.8440	nan	0.1000	0.0749
##	20	3.4421	nan	0.1000	0.1291
##	40	2.0285	nan	0.1000	0.0246
##	60	1.4136	nan	0.1000	0.0003
##	80	1.0839	nan	0.1000	0.0003
##	100	0.9011	nan	0.1000	0.0052
##	120	0.8195	nan	0.1000	-0.0075
##	140	0.7664	nan	0.1000	-0.0036
##	160	0.7243	nan	0.1000	-0.0010
##	180	0.6839	nan	0.1000	-0.0028
##	200	0.6448	nan	0.1000	-0.0048

```

##      220      0.6123      nan      0.1000     -0.0035
##      240      0.5897      nan      0.1000     -0.0038
##      250      0.5767      nan      0.1000     -0.0050
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.3122           nan      0.1000     0.5956
##      2           6.7737           nan      0.1000     0.4295
##      3           6.3033           nan      0.1000     0.4215
##      4           5.9081           nan      0.1000     0.2966
##      5           5.5342           nan      0.1000     0.3320
##      6           5.1666           nan      0.1000     0.2748
##      7           4.9365           nan      0.1000     0.1688
##      8           4.6557           nan      0.1000     0.2144
##      9           4.4412           nan      0.1000     0.1072
##     10           4.2075           nan      0.1000     0.1861
##     20           2.7722           nan      0.1000     0.0107
##     40           1.4659           nan      0.1000     0.0203
##     60           1.0163           nan      0.1000    -0.0055
##     80           0.8430           nan      0.1000    -0.0074
##    100           0.7427           nan      0.1000    -0.0031
##    120           0.6731           nan      0.1000    -0.0078
##    140           0.6151           nan      0.1000    -0.0080
##    160           0.5814           nan      0.1000    -0.0074
##    180           0.5452           nan      0.1000    -0.0054
##    200           0.5023           nan      0.1000    -0.0071
##    220           0.4697           nan      0.1000    -0.0058
##    240           0.4340           nan      0.1000    -0.0022
##    250           0.4207           nan      0.1000    -0.0088
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.3508           nan      0.1000     0.7012
##      2           6.7011           nan      0.1000     0.5533
##      3           6.1585           nan      0.1000     0.5138
##      4           5.7274           nan      0.1000     0.3542
##      5           5.2438           nan      0.1000     0.3597
##      6           4.9357           nan      0.1000     0.2545
##      7           4.6359           nan      0.1000     0.2195
##      8           4.3960           nan      0.1000     0.2294
##      9           4.1786           nan      0.1000     0.1539
##     10           3.9850           nan      0.1000     0.1414
##     20           2.4927           nan      0.1000     0.0532
##     40           1.2882           nan      0.1000     0.0112
##     60           0.8551           nan      0.1000    -0.0074
##     80           0.6752           nan      0.1000    -0.0044
##    100           0.5795           nan      0.1000    -0.0097
##    120           0.4983           nan      0.1000    -0.0038

```

##	140	0.4440	nan	0.1000	-0.0032
##	160	0.4044	nan	0.1000	-0.0056
##	180	0.3666	nan	0.1000	-0.0076
##	200	0.3337	nan	0.1000	-0.0065
##	220	0.3066	nan	0.1000	-0.0022
##	240	0.2736	nan	0.1000	-0.0039
##	250	0.2608	nan	0.1000	-0.0038

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.2719	nan	0.1000	0.7029
##	2	6.5623	nan	0.1000	0.5180
##	3	6.0426	nan	0.1000	0.4144
##	4	5.5613	nan	0.1000	0.4419
##	5	5.1070	nan	0.1000	0.2878
##	6	4.7641	nan	0.1000	0.2383
##	7	4.4677	nan	0.1000	0.2141
##	8	4.2399	nan	0.1000	0.0706
##	9	3.9751	nan	0.1000	0.1927
##	10	3.7100	nan	0.1000	0.1701
##	20	2.2565	nan	0.1000	0.0859
##	40	1.1000	nan	0.1000	-0.0080
##	60	0.7464	nan	0.1000	-0.0020
##	80	0.5790	nan	0.1000	-0.0024
##	100	0.4941	nan	0.1000	-0.0054
##	120	0.4290	nan	0.1000	-0.0030
##	140	0.3723	nan	0.1000	-0.0052
##	160	0.3330	nan	0.1000	-0.0042
##	180	0.2951	nan	0.1000	-0.0056
##	200	0.2627	nan	0.1000	-0.0038
##	220	0.2336	nan	0.1000	-0.0018
##	240	0.2089	nan	0.1000	-0.0017
##	250	0.1964	nan	0.1000	-0.0029

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.8758	nan	0.1000	0.1611
##	2	7.4638	nan	0.1000	0.4117
##	3	7.1509	nan	0.1000	0.2868
##	4	6.8550	nan	0.1000	0.2921
##	5	6.6144	nan	0.1000	0.2482
##	6	6.4519	nan	0.1000	0.1654
##	7	6.3108	nan	0.1000	0.0792
##	8	6.1631	nan	0.1000	0.1260
##	9	6.0160	nan	0.1000	0.1119
##	10	5.8043	nan	0.1000	0.1567
##	20	4.6275	nan	0.1000	0.0715
##	40	3.4435	nan	0.1000	-0.0080

```

##      60      2.6905      nan      0.1000      0.0048
##      80      2.1544      nan      0.1000      0.0127
##     100      1.7772      nan      0.1000     -0.0062
##     120      1.4927      nan      0.1000     -0.0058
##     140      1.3013      nan      0.1000      0.0010
##     160      1.1609      nan      0.1000      0.0023
##     180      1.0670      nan      0.1000     -0.0058
##     200      0.9890      nan      0.1000     -0.0088
##     220      0.9407      nan      0.1000      0.0000
##     240      0.9016      nan      0.1000     -0.0042
##     250      0.8853      nan      0.1000     -0.0037
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.6412           nan      0.1000   0.5298
##      2           7.0819           nan      0.1000   0.4917
##      3           6.5521           nan      0.1000   0.3297
##      4           6.1791           nan      0.1000   0.2744
##      5           5.9412           nan      0.1000   0.1842
##      6           5.6434           nan      0.1000   0.2504
##      7           5.3703           nan      0.1000   0.2167
##      8           5.1224           nan      0.1000   0.1739
##      9           4.9715           nan      0.1000   0.1184
##     10           4.7654           nan      0.1000   0.1615
##     20           3.3795           nan      0.1000   0.0636
##     40           2.0395           nan      0.1000   0.0080
##     60           1.4605           nan      0.1000  -0.0029
##     80           1.1344           nan      0.1000  -0.0025
##    100           0.9495           nan      0.1000  -0.0087
##    120           0.8562           nan      0.1000  -0.0037
##    140           0.7855           nan      0.1000  -0.0043
##    160           0.7298           nan      0.1000  -0.0057
##    180           0.6813           nan      0.1000  -0.0010
##    200           0.6441           nan      0.1000  -0.0027
##    220           0.6118           nan      0.1000  -0.0040
##    240           0.5838           nan      0.1000  -0.0071
##    250           0.5734           nan      0.1000  -0.0007
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.4922           nan      0.1000   0.6088
##      2           6.9029           nan      0.1000   0.5644
##      3           6.4669           nan      0.1000   0.4526
##      4           5.9980           nan      0.1000   0.4160
##      5           5.5505           nan      0.1000   0.2603
##      6           5.2643           nan      0.1000   0.2640
##      7           5.0169           nan      0.1000   0.1944
##      8           4.8024           nan      0.1000   0.1732

```

```

##      9      4.5720      nan      0.1000      0.1124
##     10      4.3508      nan      0.1000      0.1700
##     20      2.8015      nan      0.1000      0.0839
##     40      1.5058      nan      0.1000      0.0041
##     60      1.0433      nan      0.1000     -0.0028
##     80      0.8409      nan      0.1000     -0.0038
##    100      0.7262      nan      0.1000     -0.0091
##    120      0.6504      nan      0.1000     -0.0011
##    140      0.5944      nan      0.1000     -0.0082
##    160      0.5390      nan      0.1000     -0.0053
##    180      0.5004      nan      0.1000     -0.0058
##    200      0.4655      nan      0.1000     -0.0034
##    220      0.4306      nan      0.1000     -0.0051
##    240      0.4014      nan      0.1000     -0.0052
##    250      0.3922      nan      0.1000     -0.0045
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1          7.5384           nan      0.1000      0.6216
##      2          6.8492           nan      0.1000      0.5668
##      3          6.2425           nan      0.1000      0.4236
##      4          5.7761           nan      0.1000      0.4459
##      5          5.3374           nan      0.1000      0.3815
##      6          5.0229           nan      0.1000      0.2852
##      7          4.7392           nan      0.1000      0.2818
##      8          4.4323           nan      0.1000      0.1682
##      9          4.2010           nan      0.1000      0.2173
##     10          3.9190           nan      0.1000      0.1539
##     20          2.3973           nan      0.1000      0.0126
##     40          1.2163           nan      0.1000      0.0078
##     60          0.8634           nan      0.1000     -0.0088
##     80          0.7010           nan      0.1000     -0.0174
##    100          0.6125           nan      0.1000     -0.0047
##    120          0.5409           nan      0.1000     -0.0043
##    140          0.4880           nan      0.1000     -0.0087
##    160          0.4416           nan      0.1000     -0.0077
##    180          0.4022           nan      0.1000     -0.0087
##    200          0.3700           nan      0.1000     -0.0066
##    220          0.3326           nan      0.1000     -0.0048
##    240          0.3010           nan      0.1000     -0.0020
##    250          0.2897           nan      0.1000     -0.0057
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1          7.4641           nan      0.1000      0.6228
##      2          6.7842           nan      0.1000      0.6612
##      3          6.2574           nan      0.1000      0.5100
##      4          5.6693           nan      0.1000      0.5417

```

```

##      5      5.2506      nan      0.1000      0.3664
##      6      4.8195      nan      0.1000      0.3499
##      7      4.4803      nan      0.1000      0.2959
##      8      4.1807      nan      0.1000      0.1964
##      9      3.9058      nan      0.1000      0.1460
##     10      3.6831      nan      0.1000      0.1246
##     20      2.1373      nan      0.1000      0.0659
##     40      1.0923      nan      0.1000      0.0073
##     60      0.7575      nan      0.1000     -0.0129
##     80      0.6127      nan      0.1000     -0.0132
##    100      0.5130      nan      0.1000     -0.0123
##    120      0.4313      nan      0.1000     -0.0060
##    140      0.3732      nan      0.1000     -0.0102
##    160      0.3229      nan      0.1000     -0.0040
##    180      0.2862      nan      0.1000     -0.0015
##    200      0.2556      nan      0.1000     -0.0035
##    220      0.2289      nan      0.1000     -0.0049
##    240      0.2066      nan      0.1000     -0.0024
##    250      0.1957      nan      0.1000     -0.0030
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1      8.0133      nan      0.1000      0.2136
##      2      7.6170      nan      0.1000      0.3813
##      3      7.2757      nan      0.1000      0.3566
##      4      6.9741      nan      0.1000      0.2780
##      5      6.7490      nan      0.1000      0.2272
##      6      6.6030      nan      0.1000      0.0955
##      7      6.3914      nan      0.1000      0.1949
##      8      6.2217      nan      0.1000      0.1345
##      9      6.0156      nan      0.1000      0.1629
##     10      5.8483      nan      0.1000      0.1217
##     20      4.7672      nan      0.1000      0.0084
##     40      3.5325      nan      0.1000      0.0072
##     60      2.7373      nan      0.1000      0.0204
##     80      2.2393      nan      0.1000      0.0177
##    100      1.8533      nan      0.1000     -0.0056
##    120      1.5588      nan      0.1000     -0.0006
##    140      1.3684      nan      0.1000     -0.0059
##    160      1.2137      nan      0.1000     -0.0029
##    180      1.0929      nan      0.1000      0.0053
##    200      1.0225      nan      0.1000     -0.0018
##    220      0.9612      nan      0.1000     -0.0014
##    240      0.9268      nan      0.1000     -0.0089
##    250      0.9073      nan      0.1000     -0.0005
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve

```

##	1	7.6769	nan	0.1000	0.4910
##	2	7.1336	nan	0.1000	0.4624
##	3	6.7130	nan	0.1000	0.3977
##	4	6.3914	nan	0.1000	0.2512
##	5	6.0813	nan	0.1000	0.3267
##	6	5.8205	nan	0.1000	0.1912
##	7	5.5575	nan	0.1000	0.1760
##	8	5.3300	nan	0.1000	0.1566
##	9	5.1406	nan	0.1000	0.0895
##	10	4.9999	nan	0.1000	0.0993
##	20	3.4990	nan	0.1000	0.1416
##	40	2.1301	nan	0.1000	0.0205
##	60	1.4553	nan	0.1000	-0.0010
##	80	1.1386	nan	0.1000	-0.0129
##	100	0.9532	nan	0.1000	-0.0062
##	120	0.8461	nan	0.1000	-0.0020
##	140	0.7941	nan	0.1000	-0.0103
##	160	0.7479	nan	0.1000	-0.0069
##	180	0.7076	nan	0.1000	-0.0046
##	200	0.6764	nan	0.1000	-0.0034
##	220	0.6375	nan	0.1000	-0.0039
##	240	0.6118	nan	0.1000	-0.0084
##	250	0.5967	nan	0.1000	-0.0038

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.5573	nan	0.1000	0.6889
##	2	6.9522	nan	0.1000	0.5641
##	3	6.4789	nan	0.1000	0.3647
##	4	6.1178	nan	0.1000	0.1990
##	5	5.7289	nan	0.1000	0.3137
##	6	5.4295	nan	0.1000	0.1935
##	7	5.0791	nan	0.1000	0.2922
##	8	4.8212	nan	0.1000	0.2147
##	9	4.5435	nan	0.1000	0.1970
##	10	4.3324	nan	0.1000	0.1800
##	20	2.8598	nan	0.1000	0.0779
##	40	1.5947	nan	0.1000	0.0099
##	60	1.0802	nan	0.1000	-0.0061
##	80	0.8896	nan	0.1000	-0.0082
##	100	0.7621	nan	0.1000	-0.0069
##	120	0.6690	nan	0.1000	-0.0080
##	140	0.6158	nan	0.1000	-0.0050
##	160	0.5707	nan	0.1000	-0.0019
##	180	0.5336	nan	0.1000	-0.0046
##	200	0.4984	nan	0.1000	-0.0062
##	220	0.4607	nan	0.1000	-0.0045


```

##      240      0.4323      nan      0.1000     -0.0052
##      250      0.4196      nan      0.1000     -0.0041
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.4906           nan      0.1000     0.6789
##      2           6.9506           nan      0.1000     0.3624
##      3           6.3953           nan      0.1000     0.4020
##      4           5.9823           nan      0.1000     0.3891
##      5           5.5966           nan      0.1000     0.2061
##      6           5.1879           nan      0.1000     0.2865
##      7           4.8319           nan      0.1000     0.2808
##      8           4.5432           nan      0.1000     0.2153
##      9           4.2635           nan      0.1000     0.2355
##     10           3.9960           nan      0.1000     0.1091
##     20           2.4108           nan      0.1000     0.1056
##     40           1.2650           nan      0.1000     0.0002
##     60           0.8721           nan      0.1000     0.0013
##     80           0.6979           nan      0.1000    -0.0035
##    100           0.5984           nan      0.1000    -0.0062
##    120           0.5296           nan      0.1000    -0.0059
##    140           0.4772           nan      0.1000    -0.0068
##    160           0.4300           nan      0.1000    -0.0125
##    180           0.3866           nan      0.1000    -0.0057
##    200           0.3463           nan      0.1000    -0.0064
##    220           0.3136           nan      0.1000    -0.0027
##    240           0.2853           nan      0.1000    -0.0041
##    250           0.2719           nan      0.1000    -0.0052
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.4823           nan      0.1000     0.6726
##      2           6.8225           nan      0.1000     0.6054
##      3           6.2303           nan      0.1000     0.4431
##      4           5.7262           nan      0.1000     0.3136
##      5           5.2917           nan      0.1000     0.3020
##      6           4.9481           nan      0.1000     0.2905
##      7           4.5620           nan      0.1000     0.2910
##      8           4.3225           nan      0.1000     0.1791
##      9           4.0699           nan      0.1000     0.2349
##     10           3.8179           nan      0.1000     0.1849
##     20           2.3059           nan      0.1000     0.0418
##     40           1.1688           nan      0.1000    -0.0035
##     60           0.7851           nan      0.1000     0.0072
##     80           0.6170           nan      0.1000    -0.0038
##    100           0.5177           nan      0.1000    -0.0026
##    120           0.4381           nan      0.1000    -0.0056
##    140           0.3801           nan      0.1000    -0.0035

```

##	160	0.3340	nan	0.1000	-0.0031
##	180	0.2933	nan	0.1000	-0.0080
##	200	0.2558	nan	0.1000	-0.0058
##	220	0.2289	nan	0.1000	-0.0028
##	240	0.2021	nan	0.1000	-0.0050
##	250	0.1882	nan	0.1000	-0.0018

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.4102	nan	0.1000	0.3839
##	2	7.0494	nan	0.1000	0.3040
##	3	6.8614	nan	0.1000	0.1707
##	4	6.6267	nan	0.1000	0.2365
##	5	6.4159	nan	0.1000	0.2192
##	6	6.2072	nan	0.1000	0.1818
##	7	5.9995	nan	0.1000	0.1441
##	8	5.8238	nan	0.1000	0.1470
##	9	5.6784	nan	0.1000	0.0881
##	10	5.5111	nan	0.1000	0.1191
##	20	4.4453	nan	0.1000	0.0629
##	40	3.2797	nan	0.1000	0.0303
##	60	2.5893	nan	0.1000	0.0080
##	80	2.1025	nan	0.1000	-0.0054
##	100	1.7328	nan	0.1000	0.0016
##	120	1.5056	nan	0.1000	0.0039
##	140	1.3304	nan	0.1000	-0.0021
##	160	1.2081	nan	0.1000	-0.0085
##	180	1.1043	nan	0.1000	0.0002
##	200	1.0182	nan	0.1000	0.0011
##	220	0.9519	nan	0.1000	-0.0138
##	240	0.9161	nan	0.1000	-0.0036
##	250	0.9019	nan	0.1000	-0.0080

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.3296	nan	0.1000	0.5238
##	2	6.8236	nan	0.1000	0.4935
##	3	6.4024	nan	0.1000	0.3814
##	4	6.0460	nan	0.1000	0.3352
##	5	5.7799	nan	0.1000	0.2079
##	6	5.5073	nan	0.1000	0.2249
##	7	5.2509	nan	0.1000	0.1626
##	8	5.0841	nan	0.1000	0.0927
##	9	4.8799	nan	0.1000	0.1436
##	10	4.7477	nan	0.1000	0.0329
##	20	3.3287	nan	0.1000	0.0705
##	40	2.0283	nan	0.1000	0.0048
##	60	1.3816	nan	0.1000	0.0072

```

##      80      1.0728      nan      0.1000     -0.0016
##     100      0.9094      nan      0.1000      0.0076
##     120      0.8114      nan      0.1000     -0.0058
##     140      0.7446      nan      0.1000     -0.0069
##     160      0.7042      nan      0.1000     -0.0008
##     180      0.6679      nan      0.1000     -0.0038
##     200      0.6315      nan      0.1000     -0.0026
##     220      0.6063      nan      0.1000     -0.0076
##     240      0.5797      nan      0.1000     -0.0034
##     250      0.5662      nan      0.1000     -0.0046
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.2648           nan      0.1000     0.6118
##      2           6.7163           nan      0.1000     0.5338
##      3           6.2474           nan      0.1000     0.4288
##      4           5.8703           nan      0.1000     0.2640
##      5           5.5532           nan      0.1000     0.3193
##      6           5.2463           nan      0.1000     0.3198
##      7           4.9416           nan      0.1000     0.1943
##      8           4.6990           nan      0.1000     0.1722
##      9           4.4949           nan      0.1000     0.1469
##     10           4.2995           nan      0.1000     0.1804
##     20           2.7721           nan      0.1000     0.0861
##     40           1.4803           nan      0.1000     0.0123
##     60           1.0423           nan      0.1000    -0.0095
##     80           0.8424           nan      0.1000     0.0072
##    100           0.7137           nan      0.1000    -0.0079
##    120           0.6440           nan      0.1000    -0.0074
##    140           0.5827           nan      0.1000    -0.0044
##    160           0.5432           nan      0.1000    -0.0097
##    180           0.5079           nan      0.1000    -0.0044
##    200           0.4749           nan      0.1000    -0.0006
##    220           0.4448           nan      0.1000    -0.0068
##    240           0.4197           nan      0.1000    -0.0041
##    250           0.4073           nan      0.1000    -0.0040
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.1301           nan      0.1000     0.6518
##      2           6.4611           nan      0.1000     0.5540
##      3           5.8979           nan      0.1000     0.4264
##      4           5.4531           nan      0.1000     0.3997
##      5           5.0649           nan      0.1000     0.2551
##      6           4.7495           nan      0.1000     0.2099
##      7           4.4799           nan      0.1000     0.2317
##      8           4.1899           nan      0.1000     0.2269
##      9           4.0084           nan      0.1000     0.0679

```

```

##      10      3.8183      nan      0.1000      0.0965
##      20      2.3118      nan      0.1000      0.0409
##      40      1.2467      nan      0.1000      0.0054
##      60      0.8880      nan      0.1000     -0.0007
##      80      0.7226      nan      0.1000     -0.0048
##     100      0.6209      nan      0.1000     -0.0051
##     120      0.5444      nan      0.1000     -0.0087
##     140      0.4935      nan      0.1000     -0.0049
##     160      0.4445      nan      0.1000     -0.0053
##     180      0.3989      nan      0.1000     -0.0063
##     200      0.3638      nan      0.1000     -0.0079
##     220      0.3337      nan      0.1000     -0.0052
##     240      0.3055      nan      0.1000     -0.0048
##     250      0.2926      nan      0.1000     -0.0020
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.1126           nan      0.1000     0.7800
##      2           6.4349           nan      0.1000     0.5736
##      3           5.8711           nan      0.1000     0.5557
##      4           5.4355           nan      0.1000     0.3727
##      5           5.0058           nan      0.1000     0.2490
##      6           4.5813           nan      0.1000     0.2892
##      7           4.2200           nan      0.1000     0.3013
##      8           3.8990           nan      0.1000     0.2893
##      9           3.6677           nan      0.1000     0.1964
##     10           3.4290           nan      0.1000     0.1762
##     20           2.1135           nan      0.1000     0.0418
##     40           1.0761           nan      0.1000    -0.0015
##     60           0.7465           nan      0.1000    -0.0119
##     80           0.6002           nan      0.1000    -0.0082
##    100           0.4915           nan      0.1000    -0.0042
##    120           0.4217           nan      0.1000    -0.0055
##    140           0.3558           nan      0.1000    -0.0027
##    160           0.3103           nan      0.1000    -0.0039
##    180           0.2649           nan      0.1000    -0.0052
##    200           0.2323           nan      0.1000    -0.0017
##    220           0.2076           nan      0.1000    -0.0018
##    240           0.1858           nan      0.1000    -0.0025
##    250           0.1740           nan      0.1000    -0.0033
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.3456           nan      0.1000     0.2909
##      2           7.1114           nan      0.1000     0.1263
##      3           6.7570           nan      0.1000     0.2751
##      4           6.5482           nan      0.1000     0.1337
##      5           6.3257           nan      0.1000     0.2345

```

```

##      6      6.1206      nan    0.1000    0.1383
##      7      5.9301      nan    0.1000    0.1914
##      8      5.7797      nan    0.1000    0.1295
##      9      5.6019      nan    0.1000    0.1135
##     10      5.4685      nan    0.1000    0.0847
##     20      4.4722      nan    0.1000    0.0460
##     40      3.3338      nan    0.1000    0.0101
##     60      2.6477      nan    0.1000    0.0048
##     80      2.1648      nan    0.1000    0.0153
##    100      1.7916      nan    0.1000   -0.0011
##    120      1.5267      nan    0.1000    0.0018
##    140      1.3281      nan    0.1000   -0.0029
##    160      1.1995      nan    0.1000   -0.0011
##    180      1.1018      nan    0.1000    0.0001
##    200      1.0288      nan    0.1000   -0.0059
##    220      0.9667      nan    0.1000   -0.0033
##    240      0.9174      nan    0.1000   -0.0045
##    250      0.8974      nan    0.1000   -0.0025
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1          7.1892          nan    0.1000    0.5473
##      2          6.7512          nan    0.1000    0.4586
##      3          6.4426          nan    0.1000    0.2760
##      4          6.1358          nan    0.1000    0.2140
##      5          5.8079          nan    0.1000    0.2639
##      6          5.5512          nan    0.1000    0.2127
##      7          5.3613          nan    0.1000    0.1317
##      8          5.0590          nan    0.1000    0.2354
##      9          4.9117          nan    0.1000    0.1361
##     10          4.7130          nan    0.1000    0.1626
##     20          3.4020          nan    0.1000    0.0257
##     40          2.0751          nan    0.1000    0.0224
##     60          1.4101          nan    0.1000   -0.0005
##     80          1.1014          nan    0.1000    0.0065
##    100          0.9405          nan    0.1000   -0.0067
##    120          0.8391          nan    0.1000   -0.0066
##    140          0.7718          nan    0.1000   -0.0054
##    160          0.7291          nan    0.1000   -0.0095
##    180          0.6810          nan    0.1000   -0.0036
##    200          0.6457          nan    0.1000   -0.0071
##    220          0.6189          nan    0.1000   -0.0034
##    240          0.5895          nan    0.1000   -0.0023
##    250          0.5794          nan    0.1000   -0.0084
##
## Iter  TrainDeviance  ValidDeviance  StepSize  Improve
##      1          7.1359          nan    0.1000    0.6309

```

##	2	6.5967	nan	0.1000	0.4591
##	3	6.1698	nan	0.1000	0.4132
##	4	5.7599	nan	0.1000	0.3928
##	5	5.4519	nan	0.1000	0.3237
##	6	5.1401	nan	0.1000	0.2872
##	7	4.9050	nan	0.1000	0.1392
##	8	4.6196	nan	0.1000	0.2846
##	9	4.3738	nan	0.1000	0.1828
##	10	4.1835	nan	0.1000	0.1700
##	20	2.8099	nan	0.1000	0.0580
##	40	1.6151	nan	0.1000	-0.0065
##	60	1.1301	nan	0.1000	0.0002
##	80	0.8944	nan	0.1000	-0.0077
##	100	0.7517	nan	0.1000	-0.0139
##	120	0.6730	nan	0.1000	-0.0085
##	140	0.6057	nan	0.1000	-0.0038
##	160	0.5547	nan	0.1000	-0.0097
##	180	0.5094	nan	0.1000	-0.0103
##	200	0.4766	nan	0.1000	-0.0066
##	220	0.4450	nan	0.1000	-0.0040
##	240	0.4151	nan	0.1000	-0.0027
##	250	0.4013	nan	0.1000	-0.0047

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.0294	nan	0.1000	0.7373
##	2	6.4316	nan	0.1000	0.6149
##	3	5.9113	nan	0.1000	0.3359
##	4	5.5546	nan	0.1000	0.3535
##	5	5.1917	nan	0.1000	0.2802
##	6	4.8371	nan	0.1000	0.3563
##	7	4.5468	nan	0.1000	0.2089
##	8	4.3248	nan	0.1000	0.1893
##	9	4.0651	nan	0.1000	0.1052
##	10	3.8138	nan	0.1000	0.1666
##	20	2.3320	nan	0.1000	0.0422
##	40	1.2332	nan	0.1000	0.0086
##	60	0.8464	nan	0.1000	0.0058
##	80	0.6737	nan	0.1000	-0.0128
##	100	0.5934	nan	0.1000	-0.0097
##	120	0.5242	nan	0.1000	-0.0050
##	140	0.4705	nan	0.1000	-0.0050
##	160	0.4233	nan	0.1000	-0.0078
##	180	0.3867	nan	0.1000	-0.0076
##	200	0.3531	nan	0.1000	-0.0040
##	220	0.3242	nan	0.1000	-0.0039
##	240	0.3013	nan	0.1000	-0.0036

```

##      250      0.2843      nan      0.1000     -0.0029
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.0098           nan      0.1000     0.5845
##      2           6.3343           nan      0.1000     0.4158
##      3           5.8514           nan      0.1000     0.3832
##      4           5.3437           nan      0.1000     0.4617
##      5           4.9793           nan      0.1000     0.3545
##      6           4.6111           nan      0.1000     0.3660
##      7           4.2957           nan      0.1000     0.2518
##      8           3.9570           nan      0.1000     0.2273
##      9           3.7320           nan      0.1000     0.1854
##     10           3.5140           nan      0.1000     0.1503
##     20           2.1057           nan      0.1000     0.0764
##     40           1.0592           nan      0.1000     0.0046
##     60           0.7323           nan      0.1000    -0.0010
##     80           0.5881           nan      0.1000    -0.0058
##    100           0.4956           nan      0.1000    -0.0051
##    120           0.4248           nan      0.1000    -0.0025
##    140           0.3718           nan      0.1000    -0.0073
##    160           0.3294           nan      0.1000    -0.0062
##    180           0.2819           nan      0.1000    -0.0026
##    200           0.2497           nan      0.1000    -0.0034
##    220           0.2233           nan      0.1000    -0.0058
##    240           0.1990           nan      0.1000    -0.0018
##    250           0.1871           nan      0.1000    -0.0021
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.5868           nan      0.1000     0.4115
##      2           7.2833           nan      0.1000     0.2331
##      3           7.0345           nan      0.1000     0.2570
##      4           6.8101           nan      0.1000     0.1071
##      5           6.5893           nan      0.1000     0.1563
##      6           6.3092           nan      0.1000     0.1664
##      7           6.1025           nan      0.1000     0.1476
##      8           5.9663           nan      0.1000     0.0749
##      9           5.7474           nan      0.1000     0.1445
##     10           5.5960           nan      0.1000     0.0876
##     20           4.4901           nan      0.1000     0.0532
##     40           3.2925           nan      0.1000     0.0364
##     60           2.6190           nan      0.1000     0.0090
##     80           2.1208           nan      0.1000     0.0132
##    100           1.7732           nan      0.1000    -0.0007
##    120           1.5132           nan      0.1000     0.0046
##    140           1.3283           nan      0.1000     0.0031
##    160           1.1925           nan      0.1000    -0.0002

```

##	180	1.0847	nan	0.1000	-0.0013
##	200	0.9981	nan	0.1000	-0.0003
##	220	0.9475	nan	0.1000	-0.0035
##	240	0.9021	nan	0.1000	-0.0047
##	250	0.8843	nan	0.1000	-0.0029

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.3282	nan	0.1000	0.5172
##	2	6.9512	nan	0.1000	0.3872
##	3	6.5318	nan	0.1000	0.3539
##	4	6.1978	nan	0.1000	0.2757
##	5	5.9066	nan	0.1000	0.2460
##	6	5.6366	nan	0.1000	0.2517
##	7	5.3731	nan	0.1000	0.1878
##	8	5.1982	nan	0.1000	0.1405
##	9	5.0115	nan	0.1000	0.1659
##	10	4.8495	nan	0.1000	0.1095
##	20	3.3891	nan	0.1000	0.1033
##	40	2.0749	nan	0.1000	0.0310
##	60	1.4400	nan	0.1000	0.0118
##	80	1.1141	nan	0.1000	0.0030
##	100	0.9360	nan	0.1000	0.0034
##	120	0.8234	nan	0.1000	-0.0051
##	140	0.7648	nan	0.1000	-0.0025
##	160	0.7108	nan	0.1000	-0.0075
##	180	0.6698	nan	0.1000	-0.0052
##	200	0.6366	nan	0.1000	-0.0069
##	220	0.6020	nan	0.1000	-0.0060
##	240	0.5770	nan	0.1000	-0.0049
##	250	0.5640	nan	0.1000	-0.0035

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.2616	nan	0.1000	0.6773
##	2	6.8094	nan	0.1000	0.2167
##	3	6.3219	nan	0.1000	0.4713
##	4	5.9148	nan	0.1000	0.3566
##	5	5.5959	nan	0.1000	0.2337
##	6	5.2724	nan	0.1000	0.3200
##	7	4.9638	nan	0.1000	0.1900
##	8	4.7283	nan	0.1000	0.1678
##	9	4.5198	nan	0.1000	0.1260
##	10	4.3024	nan	0.1000	0.1451
##	20	2.7404	nan	0.1000	0.0379
##	40	1.5392	nan	0.1000	-0.0055
##	60	1.0399	nan	0.1000	0.0123
##	80	0.8192	nan	0.1000	0.0004


```

##      100      0.7016      nan      0.1000     -0.0083
##      120      0.6325      nan      0.1000     -0.0080
##      140      0.5744      nan      0.1000     -0.0063
##      160      0.5259      nan      0.1000     -0.0054
##      180      0.4819      nan      0.1000     -0.0111
##      200      0.4502      nan      0.1000     -0.0055
##      220      0.4164      nan      0.1000     -0.0031
##      240      0.3874      nan      0.1000     -0.0056
##      250      0.3749      nan      0.1000     -0.0012
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.4180           nan      0.1000     0.3827
##      2           6.7805           nan      0.1000     0.4994
##      3           6.2576           nan      0.1000     0.4550
##      4           5.8389           nan      0.1000     0.2288
##      5           5.4627           nan      0.1000     0.3687
##      6           5.1119           nan      0.1000     0.3303
##      7           4.7963           nan      0.1000     0.3248
##      8           4.5143           nan      0.1000     0.2401
##      9           4.2403           nan      0.1000     0.1986
##     10           4.0630           nan      0.1000     0.1325
##     20           2.4827           nan      0.1000     0.0652
##     40           1.2766           nan      0.1000     0.0153
##     60           0.8585           nan      0.1000    -0.0103
##     80           0.6931           nan      0.1000    -0.0009
##    100           0.5966           nan      0.1000    -0.0085
##    120           0.5283           nan      0.1000    -0.0111
##    140           0.4718           nan      0.1000    -0.0106
##    160           0.4263           nan      0.1000    -0.0058
##    180           0.3870           nan      0.1000    -0.0051
##    200           0.3533           nan      0.1000    -0.0027
##    220           0.3194           nan      0.1000    -0.0032
##    240           0.2922           nan      0.1000    -0.0019
##    250           0.2785           nan      0.1000    -0.0040
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.1462           nan      0.1000     0.6854
##      2           6.5338           nan      0.1000     0.3961
##      3           6.0020           nan      0.1000     0.4159
##      4           5.5196           nan      0.1000     0.3879
##      5           5.0940           nan      0.1000     0.3601
##      6           4.7927           nan      0.1000     0.2963
##      7           4.4249           nan      0.1000     0.2827
##      8           4.1084           nan      0.1000     0.2133
##      9           3.8253           nan      0.1000     0.1881
##     10           3.6343           nan      0.1000     0.1233

```

##	20	2.1522	nan	0.1000	0.0354
##	40	1.0580	nan	0.1000	0.0161
##	60	0.7191	nan	0.1000	0.0018
##	80	0.5670	nan	0.1000	-0.0057
##	100	0.4832	nan	0.1000	-0.0050
##	120	0.4124	nan	0.1000	-0.0054
##	140	0.3602	nan	0.1000	-0.0078
##	160	0.3179	nan	0.1000	-0.0036
##	180	0.2778	nan	0.1000	-0.0047
##	200	0.2450	nan	0.1000	-0.0059
##	220	0.2183	nan	0.1000	-0.0040
##	240	0.1929	nan	0.1000	-0.0048
##	250	0.1839	nan	0.1000	-0.0040

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.4540	nan	0.1000	0.3641
##	2	7.1373	nan	0.1000	0.3291
##	3	6.8416	nan	0.1000	0.2730
##	4	6.5864	nan	0.1000	0.1830
##	5	6.3867	nan	0.1000	0.1603
##	6	6.1413	nan	0.1000	0.2019
##	7	5.9610	nan	0.1000	0.1395
##	8	5.8194	nan	0.1000	0.1027
##	9	5.6546	nan	0.1000	0.0827
##	10	5.4631	nan	0.1000	0.0971
##	20	4.3922	nan	0.1000	0.0188
##	40	3.2426	nan	0.1000	0.0345
##	60	2.5668	nan	0.1000	0.0223
##	80	2.0713	nan	0.1000	0.0088
##	100	1.7438	nan	0.1000	-0.0065
##	120	1.4921	nan	0.1000	0.0043
##	140	1.3219	nan	0.1000	0.0085
##	160	1.1951	nan	0.1000	0.0011
##	180	1.1011	nan	0.1000	-0.0045
##	200	1.0331	nan	0.1000	-0.0013
##	220	0.9765	nan	0.1000	-0.0020
##	240	0.9362	nan	0.1000	-0.0053
##	250	0.9130	nan	0.1000	0.0009

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.2752	nan	0.1000	0.5651
##	2	6.7895	nan	0.1000	0.4457
##	3	6.4533	nan	0.1000	0.2484
##	4	6.1292	nan	0.1000	0.3299
##	5	5.8643	nan	0.1000	0.2757
##	6	5.6456	nan	0.1000	0.1455

```

##      7      5.3780      nan      0.1000      0.2465
##      8      5.1363      nan      0.1000      0.1833
##      9      4.9400      nan      0.1000      0.1011
##     10      4.7344      nan      0.1000      0.1801
##     20      3.2834      nan      0.1000      0.0459
##     40      2.0454      nan      0.1000      0.0186
##     60      1.4428      nan      0.1000      0.0008
##     80      1.0931      nan      0.1000      0.0050
##    100      0.9245      nan      0.1000     -0.0066
##    120      0.8152      nan      0.1000     -0.0040
##    140      0.7456      nan      0.1000     -0.0028
##    160      0.6979      nan      0.1000     -0.0052
##    180      0.6573      nan      0.1000     -0.0035
##    200      0.6239      nan      0.1000     -0.0037
##    220      0.5949      nan      0.1000     -0.0042
##    240      0.5695      nan      0.1000     -0.0039
##    250      0.5578      nan      0.1000     -0.0039
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.2004           nan      0.1000    0.5575
##      2           6.6875           nan      0.1000    0.5080
##      3           6.1988           nan      0.1000    0.4364
##      4           5.7727           nan      0.1000    0.3191
##      5           5.4082           nan      0.1000    0.3497
##      6           5.1722           nan      0.1000    0.1589
##      7           4.8592           nan      0.1000    0.1847
##      8           4.6133           nan      0.1000    0.2159
##      9           4.3824           nan      0.1000    0.2002
##     10           4.1895           nan      0.1000    0.1182
##     20           2.7390           nan      0.1000    0.0733
##     40           1.5162           nan      0.1000   -0.0008
##     60           1.0207           nan      0.1000    0.0026
##     80           0.8292           nan      0.1000   -0.0016
##    100           0.7057           nan      0.1000   -0.0089
##    120           0.6346           nan      0.1000   -0.0085
##    140           0.5675           nan      0.1000   -0.0056
##    160           0.5176           nan      0.1000   -0.0074
##    180           0.4752           nan      0.1000   -0.0050
##    200           0.4333           nan      0.1000   -0.0054
##    220           0.4032           nan      0.1000   -0.0070
##    240           0.3743           nan      0.1000   -0.0022
##    250           0.3642           nan      0.1000   -0.0062
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.1706           nan      0.1000    0.6334
##      2           6.5472           nan      0.1000    0.5452

```

##	3	5.9610	nan	0.1000	0.4332
##	4	5.5223	nan	0.1000	0.3950
##	5	5.1475	nan	0.1000	0.2716
##	6	4.8185	nan	0.1000	0.2069
##	7	4.5333	nan	0.1000	0.2125
##	8	4.2886	nan	0.1000	0.2125
##	9	4.0286	nan	0.1000	0.1507
##	10	3.8317	nan	0.1000	0.1457
##	20	2.3895	nan	0.1000	0.0665
##	40	1.2552	nan	0.1000	0.0223
##	60	0.8795	nan	0.1000	0.0039
##	80	0.7069	nan	0.1000	-0.0107
##	100	0.6120	nan	0.1000	-0.0067
##	120	0.5331	nan	0.1000	-0.0065
##	140	0.4731	nan	0.1000	-0.0091
##	160	0.4256	nan	0.1000	-0.0038
##	180	0.3789	nan	0.1000	-0.0053
##	200	0.3447	nan	0.1000	-0.0045
##	220	0.3123	nan	0.1000	-0.0067
##	240	0.2807	nan	0.1000	-0.0018
##	250	0.2696	nan	0.1000	-0.0060
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.1181	nan	0.1000	0.6333
##	2	6.4637	nan	0.1000	0.5209
##	3	5.8982	nan	0.1000	0.5202
##	4	5.4003	nan	0.1000	0.3026
##	5	4.9882	nan	0.1000	0.2926
##	6	4.6393	nan	0.1000	0.2601
##	7	4.3294	nan	0.1000	0.2207
##	8	4.0488	nan	0.1000	0.2276
##	9	3.7542	nan	0.1000	0.2489
##	10	3.5442	nan	0.1000	0.1313
##	20	2.0640	nan	0.1000	0.0561
##	40	1.0503	nan	0.1000	0.0116
##	60	0.7338	nan	0.1000	-0.0011
##	80	0.5913	nan	0.1000	-0.0046
##	100	0.4891	nan	0.1000	-0.0047
##	120	0.4050	nan	0.1000	-0.0044
##	140	0.3428	nan	0.1000	-0.0083
##	160	0.2935	nan	0.1000	-0.0027
##	180	0.2561	nan	0.1000	0.0005
##	200	0.2280	nan	0.1000	-0.0051
##	220	0.1969	nan	0.1000	-0.0034
##	240	0.1764	nan	0.1000	-0.0047
##	250	0.1670	nan	0.1000	-0.0035

```

##
## Iter    TrainDeviance    ValidDeviance    StepSize    Improve
##      1          7.9913          nan      0.1000      0.3761
##      2          7.7852          nan      0.1000      0.1808
##      3          7.4353          nan      0.1000      0.3342
##      4          7.2002          nan      0.1000      0.1743
##      5          6.8764          nan      0.1000      0.3014
##      6          6.7249          nan      0.1000      0.1377
##      7          6.5608          nan      0.1000      0.0969
##      8          6.3857          nan      0.1000      0.1119
##      9          6.2081          nan      0.1000      0.2099
##     10          6.0676          nan      0.1000      0.0350
##     20          4.8481          nan      0.1000      0.0326
##     40          3.5311          nan      0.1000      0.0243
##     60          2.7637          nan      0.1000     -0.0042
##     80          2.2298          nan      0.1000      0.0035
##    100          1.8541          nan      0.1000      0.0040
##    120          1.5720          nan      0.1000     -0.0142
##    140          1.3723          nan      0.1000      0.0089
##    160          1.2296          nan      0.1000     -0.0073
##    180          1.1206          nan      0.1000     -0.0095
##    200          1.0353          nan      0.1000     -0.0009
##    220          0.9629          nan      0.1000     -0.0014
##    240          0.9290          nan      0.1000     -0.0011
##    250          0.9156          nan      0.1000     -0.0036
##
## Iter    TrainDeviance    ValidDeviance    StepSize    Improve
##      1          7.7709          nan      0.1000      0.4607
##      2          7.2930          nan      0.1000      0.4519
##      3          6.7154          nan      0.1000      0.2983
##      4          6.4052          nan      0.1000      0.2385
##      5          6.0937          nan      0.1000      0.2545
##      6          5.7996          nan      0.1000      0.2344
##      7          5.6159          nan      0.1000      0.1053
##      8          5.4356          nan      0.1000      0.1415
##      9          5.2146          nan      0.1000      0.1220
##     10          5.0671          nan      0.1000      0.1222
##     20          3.5111          nan      0.1000      0.0306
##     40          2.0860          nan      0.1000      0.0249
##     60          1.4474          nan      0.1000      0.0088
##     80          1.1308          nan      0.1000     -0.0062
##    100          0.9648          nan      0.1000      0.0020
##    120          0.8736          nan      0.1000     -0.0089
##    140          0.8043          nan      0.1000     -0.0064
##    160          0.7498          nan      0.1000     -0.0083
##    180          0.7018          nan      0.1000      0.0001

```

##	200	0.6598	nan	0.1000	-0.0037
##	220	0.6329	nan	0.1000	-0.0035
##	240	0.6049	nan	0.1000	-0.0068
##	250	0.5905	nan	0.1000	-0.0046

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.5947	nan	0.1000	0.5713
##	2	6.9919	nan	0.1000	0.5903
##	3	6.5417	nan	0.1000	0.4225
##	4	6.0968	nan	0.1000	0.3213
##	5	5.7124	nan	0.1000	0.3371
##	6	5.3694	nan	0.1000	0.2772
##	7	5.0464	nan	0.1000	0.2062
##	8	4.7692	nan	0.1000	0.2608
##	9	4.6153	nan	0.1000	0.0811
##	10	4.4195	nan	0.1000	0.0977
##	20	2.9502	nan	0.1000	0.1132
##	40	1.5511	nan	0.1000	0.0204
##	60	1.0570	nan	0.1000	0.0142
##	80	0.8503	nan	0.1000	-0.0047
##	100	0.7300	nan	0.1000	-0.0177
##	120	0.6452	nan	0.1000	-0.0077
##	140	0.5915	nan	0.1000	-0.0082
##	160	0.5400	nan	0.1000	-0.0058
##	180	0.5009	nan	0.1000	-0.0040
##	200	0.4669	nan	0.1000	-0.0059
##	220	0.4400	nan	0.1000	-0.0040
##	240	0.4131	nan	0.1000	-0.0079
##	250	0.4039	nan	0.1000	-0.0059

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.6580	nan	0.1000	0.6541
##	2	7.0177	nan	0.1000	0.5653
##	3	6.4657	nan	0.1000	0.5278
##	4	5.9869	nan	0.1000	0.4523
##	5	5.5715	nan	0.1000	0.3559
##	6	5.1874	nan	0.1000	0.2871
##	7	4.8558	nan	0.1000	0.2317
##	8	4.5402	nan	0.1000	0.2236
##	9	4.3011	nan	0.1000	0.1610
##	10	4.0022	nan	0.1000	0.2235
##	20	2.4352	nan	0.1000	0.0364
##	40	1.2432	nan	0.1000	0.0032
##	60	0.8629	nan	0.1000	-0.0036
##	80	0.7137	nan	0.1000	-0.0020
##	100	0.6171	nan	0.1000	-0.0102

```

##      120      0.5389      nan      0.1000     -0.0066
##      140      0.4772      nan      0.1000     -0.0067
##      160      0.4290      nan      0.1000     -0.0091
##      180      0.3873      nan      0.1000     -0.0116
##      200      0.3411      nan      0.1000     -0.0052
##      220      0.3122      nan      0.1000     -0.0030
##      240      0.2868      nan      0.1000     -0.0020
##      250      0.2726      nan      0.1000     -0.0027
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.4799           nan      0.1000     0.8543
##      2           6.8255           nan      0.1000     0.5990
##      3           6.2454           nan      0.1000     0.4987
##      4           5.7558           nan      0.1000     0.3832
##      5           5.3790           nan      0.1000     0.3350
##      6           5.0578           nan      0.1000     0.2406
##      7           4.7324           nan      0.1000     0.1941
##      8           4.3901           nan      0.1000     0.2642
##      9           4.0528           nan      0.1000     0.2389
##     10           3.7874           nan      0.1000     0.2426
##     20           2.1952           nan      0.1000     0.0768
##     40           1.1025           nan      0.1000    -0.0081
##     60           0.7689           nan      0.1000    -0.0002
##     80           0.6122           nan      0.1000    -0.0061
##    100           0.5200           nan      0.1000    -0.0109
##    120           0.4400           nan      0.1000    -0.0069
##    140           0.3837           nan      0.1000    -0.0055
##    160           0.3328           nan      0.1000    -0.0034
##    180           0.2920           nan      0.1000    -0.0077
##    200           0.2544           nan      0.1000    -0.0035
##    220           0.2241           nan      0.1000    -0.0036
##    240           0.1988           nan      0.1000    -0.0047
##    250           0.1865           nan      0.1000    -0.0028
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.2748           nan      0.1000     0.3849
##      2           6.9549           nan      0.1000     0.3086
##      3           6.6188           nan      0.1000     0.2131
##      4           6.3662           nan      0.1000     0.2414
##      5           6.1655           nan      0.1000     0.1744
##      6           5.9621           nan      0.1000     0.1800
##      7           5.7756           nan      0.1000     0.1729
##      8           5.6347           nan      0.1000     0.1348
##      9           5.5429           nan      0.1000     0.0634
##     10           5.3761           nan      0.1000     0.0928
##     20           4.3688           nan      0.1000     0.0591

```

##	40	3.2068	nan	0.1000	0.0402
##	60	2.5475	nan	0.1000	0.0114
##	80	2.0354	nan	0.1000	0.0023
##	100	1.7011	nan	0.1000	-0.0072
##	120	1.4362	nan	0.1000	-0.0015
##	140	1.2570	nan	0.1000	-0.0034
##	160	1.1369	nan	0.1000	0.0021
##	180	1.0223	nan	0.1000	-0.0062
##	200	0.9563	nan	0.1000	-0.0038
##	220	0.8991	nan	0.1000	-0.0058
##	240	0.8598	nan	0.1000	-0.0059
##	250	0.8460	nan	0.1000	-0.0086

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.1432	nan	0.1000	0.5851
##	2	6.7241	nan	0.1000	0.4090
##	3	6.2802	nan	0.1000	0.3640
##	4	5.9965	nan	0.1000	0.1559
##	5	5.7218	nan	0.1000	0.2864
##	6	5.4669	nan	0.1000	0.2326
##	7	5.1666	nan	0.1000	0.2190
##	8	4.9344	nan	0.1000	0.2265
##	9	4.7538	nan	0.1000	0.1466
##	10	4.5570	nan	0.1000	0.1311
##	20	3.2524	nan	0.1000	0.1132
##	40	1.9609	nan	0.1000	0.0491
##	60	1.3733	nan	0.1000	-0.0028
##	80	1.0764	nan	0.1000	-0.0083
##	100	0.8912	nan	0.1000	-0.0043
##	120	0.7917	nan	0.1000	-0.0084
##	140	0.7298	nan	0.1000	-0.0027
##	160	0.6894	nan	0.1000	-0.0061
##	180	0.6543	nan	0.1000	-0.0077
##	200	0.6175	nan	0.1000	-0.0009
##	220	0.5885	nan	0.1000	-0.0089
##	240	0.5540	nan	0.1000	-0.0036
##	250	0.5386	nan	0.1000	-0.0019

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.0286	nan	0.1000	0.5874
##	2	6.5422	nan	0.1000	0.4614
##	3	6.0807	nan	0.1000	0.4788
##	4	5.6220	nan	0.1000	0.3139
##	5	5.3339	nan	0.1000	0.1849
##	6	4.9990	nan	0.1000	0.2584
##	7	4.7023	nan	0.1000	0.2324


```

##      8      4.4521      nan      0.1000      0.2002
##      9      4.2254      nan      0.1000      0.1721
##     10      4.0158      nan      0.1000      0.1061
##     20      2.5603      nan      0.1000      0.0649
##     40      1.4633      nan      0.1000      0.0221
##     60      1.0126      nan      0.1000      0.0040
##     80      0.8089      nan      0.1000     -0.0011
##    100      0.6913      nan      0.1000     -0.0016
##    120      0.6212      nan      0.1000     -0.0039
##    140      0.5555      nan      0.1000      0.0013
##    160      0.5178      nan      0.1000     -0.0070
##    180      0.4779      nan      0.1000     -0.0062
##    200      0.4357      nan      0.1000     -0.0056
##    220      0.4076      nan      0.1000     -0.0034
##    240      0.3838      nan      0.1000     -0.0051
##    250      0.3724      nan      0.1000     -0.0045
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.0052           nan      0.1000    0.7357
##      2           6.3136           nan      0.1000    0.6391
##      3           5.8579           nan      0.1000    0.3395
##      4           5.4261           nan      0.1000    0.4118
##      5           5.0380           nan      0.1000    0.2541
##      6           4.7502           nan      0.1000    0.2364
##      7           4.5107           nan      0.1000    0.1393
##      8           4.2559           nan      0.1000    0.2325
##      9           4.0139           nan      0.1000    0.1431
##     10           3.7788           nan      0.1000    0.1690
##     20           2.2976           nan      0.1000    0.0670
##     40           1.1641           nan      0.1000    0.0258
##     60           0.7945           nan      0.1000   -0.0042
##     80           0.6385           nan      0.1000   -0.0010
##    100           0.5468           nan      0.1000   -0.0065
##    120           0.4711           nan      0.1000   -0.0070
##    140           0.4186           nan      0.1000   -0.0051
##    160           0.3792           nan      0.1000   -0.0062
##    180           0.3426           nan      0.1000   -0.0072
##    200           0.3149           nan      0.1000   -0.0066
##    220           0.2897           nan      0.1000   -0.0081
##    240           0.2642           nan      0.1000   -0.0020
##    250           0.2526           nan      0.1000   -0.0020
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.0074           nan      0.1000    0.6779
##      2           6.3049           nan      0.1000    0.6277
##      3           5.7428           nan      0.1000    0.5037

```

##	4	5.3843	nan	0.1000	0.2738
##	5	4.9412	nan	0.1000	0.4013
##	6	4.5370	nan	0.1000	0.2175
##	7	4.2296	nan	0.1000	0.2194
##	8	3.9306	nan	0.1000	0.2566
##	9	3.6976	nan	0.1000	0.1642
##	10	3.4742	nan	0.1000	0.1155
##	20	2.0717	nan	0.1000	0.0554
##	40	1.0961	nan	0.1000	0.0158
##	60	0.7387	nan	0.1000	-0.0071
##	80	0.5947	nan	0.1000	-0.0086
##	100	0.4878	nan	0.1000	-0.0035
##	120	0.4191	nan	0.1000	-0.0069
##	140	0.3558	nan	0.1000	-0.0047
##	160	0.3193	nan	0.1000	-0.0029
##	180	0.2757	nan	0.1000	-0.0060
##	200	0.2412	nan	0.1000	-0.0011
##	220	0.2154	nan	0.1000	-0.0022
##	240	0.1911	nan	0.1000	-0.0019
##	250	0.1798	nan	0.1000	-0.0025
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.5373	nan	0.1000	0.3255
##	2	7.2302	nan	0.1000	0.3081
##	3	7.0287	nan	0.1000	0.0792
##	4	6.8235	nan	0.1000	0.1648
##	5	6.5846	nan	0.1000	0.2528
##	6	6.3572	nan	0.1000	0.2142
##	7	6.2170	nan	0.1000	0.0731
##	8	6.0241	nan	0.1000	0.1693
##	9	5.8911	nan	0.1000	0.1379
##	10	5.7890	nan	0.1000	0.0396
##	20	4.6672	nan	0.1000	0.0610
##	40	3.3823	nan	0.1000	0.0241
##	60	2.6081	nan	0.1000	0.0076
##	80	2.1083	nan	0.1000	0.0027
##	100	1.7823	nan	0.1000	-0.0038
##	120	1.5329	nan	0.1000	0.0083
##	140	1.3371	nan	0.1000	-0.0004
##	160	1.1999	nan	0.1000	-0.0035
##	180	1.0969	nan	0.1000	-0.0001
##	200	1.0178	nan	0.1000	-0.0020
##	220	0.9601	nan	0.1000	-0.0065
##	240	0.9259	nan	0.1000	-0.0059
##	250	0.9093	nan	0.1000	-0.0037
##					

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.3367	nan	0.1000	0.5068
##	2	6.8403	nan	0.1000	0.4133
##	3	6.4575	nan	0.1000	0.3430
##	4	6.1040	nan	0.1000	0.2802
##	5	5.9071	nan	0.1000	0.1333
##	6	5.6534	nan	0.1000	0.2078
##	7	5.4050	nan	0.1000	0.2252
##	8	5.2113	nan	0.1000	0.1294
##	9	4.9828	nan	0.1000	0.1934
##	10	4.8171	nan	0.1000	0.1146
##	20	3.3732	nan	0.1000	0.0232
##	40	2.0529	nan	0.1000	0.0254
##	60	1.4630	nan	0.1000	0.0107
##	80	1.1301	nan	0.1000	-0.0184
##	100	0.9347	nan	0.1000	-0.0020
##	120	0.8346	nan	0.1000	-0.0053
##	140	0.7561	nan	0.1000	-0.0116
##	160	0.7038	nan	0.1000	-0.0083
##	180	0.6608	nan	0.1000	-0.0050
##	200	0.6231	nan	0.1000	-0.0062
##	220	0.5885	nan	0.1000	-0.0053
##	240	0.5666	nan	0.1000	-0.0073
##	250	0.5537	nan	0.1000	-0.0016
##					
##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.3914	nan	0.1000	0.5561
##	2	6.7379	nan	0.1000	0.5924
##	3	6.1931	nan	0.1000	0.3638
##	4	5.7353	nan	0.1000	0.2706
##	5	5.4272	nan	0.1000	0.2604
##	6	5.1269	nan	0.1000	0.1462
##	7	4.9406	nan	0.1000	0.0649
##	8	4.7122	nan	0.1000	0.1369
##	9	4.5181	nan	0.1000	0.1104
##	10	4.3030	nan	0.1000	0.1313
##	20	2.8485	nan	0.1000	0.0934
##	40	1.6130	nan	0.1000	-0.0081
##	60	1.1086	nan	0.1000	-0.0165
##	80	0.8744	nan	0.1000	-0.0020
##	100	0.7519	nan	0.1000	-0.0066
##	120	0.6638	nan	0.1000	-0.0039
##	140	0.5997	nan	0.1000	-0.0044
##	160	0.5553	nan	0.1000	-0.0082
##	180	0.5230	nan	0.1000	-0.0054
##	200	0.4907	nan	0.1000	-0.0056

##	220	0.4566	nan	0.1000	-0.0051
##	240	0.4234	nan	0.1000	-0.0047
##	250	0.4103	nan	0.1000	-0.0062

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.3959	nan	0.1000	0.5344
##	2	6.7840	nan	0.1000	0.6032
##	3	6.2042	nan	0.1000	0.5214
##	4	5.7561	nan	0.1000	0.3567
##	5	5.3542	nan	0.1000	0.3154
##	6	5.0748	nan	0.1000	0.2697
##	7	4.6856	nan	0.1000	0.3381
##	8	4.4209	nan	0.1000	0.2125
##	9	4.1834	nan	0.1000	0.1959
##	10	3.9558	nan	0.1000	0.1757
##	20	2.5107	nan	0.1000	0.0757
##	40	1.3385	nan	0.1000	0.0197
##	60	0.9193	nan	0.1000	0.0067
##	80	0.7247	nan	0.1000	-0.0105
##	100	0.6080	nan	0.1000	-0.0060
##	120	0.5370	nan	0.1000	-0.0086
##	140	0.4808	nan	0.1000	-0.0094
##	160	0.4331	nan	0.1000	-0.0018
##	180	0.3896	nan	0.1000	-0.0076
##	200	0.3487	nan	0.1000	-0.0060
##	220	0.3156	nan	0.1000	-0.0044
##	240	0.2857	nan	0.1000	-0.0023
##	250	0.2735	nan	0.1000	-0.0045

##

##	Iter	TrainDeviance	ValidDeviance	StepSize	Improve
##	1	7.2002	nan	0.1000	0.7128
##	2	6.5508	nan	0.1000	0.5996
##	3	5.9803	nan	0.1000	0.5293
##	4	5.5269	nan	0.1000	0.4154
##	5	5.1301	nan	0.1000	0.3923
##	6	4.7897	nan	0.1000	0.2007
##	7	4.4224	nan	0.1000	0.2482
##	8	4.1528	nan	0.1000	0.1840
##	9	3.8740	nan	0.1000	0.1938
##	10	3.6367	nan	0.1000	0.1894
##	20	2.1725	nan	0.1000	0.0933
##	40	1.0934	nan	0.1000	0.0063
##	60	0.7734	nan	0.1000	-0.0003
##	80	0.6267	nan	0.1000	-0.0113
##	100	0.5248	nan	0.1000	-0.0115
##	120	0.4509	nan	0.1000	-0.0086

```
##      140      0.3909      nan      0.1000     -0.0042
##      160      0.3417      nan      0.1000     -0.0095
##      180      0.2979      nan      0.1000     -0.0010
##      200      0.2641      nan      0.1000     -0.0051
##      220      0.2324      nan      0.1000     -0.0048
##      240      0.2065      nan      0.1000     -0.0029
##      250      0.1944      nan      0.1000     -0.0033
##
## Iter   TrainDeviance   ValidDeviance   StepSize   Improve
##      1           7.5853           nan      0.1000     0.4622
##      2           7.2268           nan      0.1000     0.2276
##      3           6.9725           nan      0.1000     0.2317
##      4           6.7255           nan      0.1000     0.1945
##      5           6.5310           nan      0.1000     0.1967
##      6           6.3548           nan      0.1000     0.1452
##      7           6.1331           nan      0.1000     0.1877
##      8           6.0205           nan      0.1000     0.0816
##      9           5.8651           nan      0.1000     0.1151
##     10           5.7272           nan      0.1000     0.0983
##     20           4.5797           nan      0.1000     0.0457
##     40           3.3805           nan      0.1000     0.0281
##     60           2.6821           nan      0.1000    -0.0024
##     80           2.1887           nan      0.1000     0.0083
##    100           1.8328           nan      0.1000     0.0108
##    120           1.5572           nan      0.1000    -0.0039
##    140           1.3454           nan      0.1000     0.0011
##    160           1.2090           nan      0.1000    -0.0017
##    180           1.0998           nan      0.1000     0.0009
##    200           1.0155           nan      0.1000    -0.0061
##    220           0.9510           nan      0.1000    -0.0040
##    240           0.9159           nan      0.1000    -0.0058
##    250           0.8952           nan      0.1000    -0.0036
```

```
carseats.gbm
```

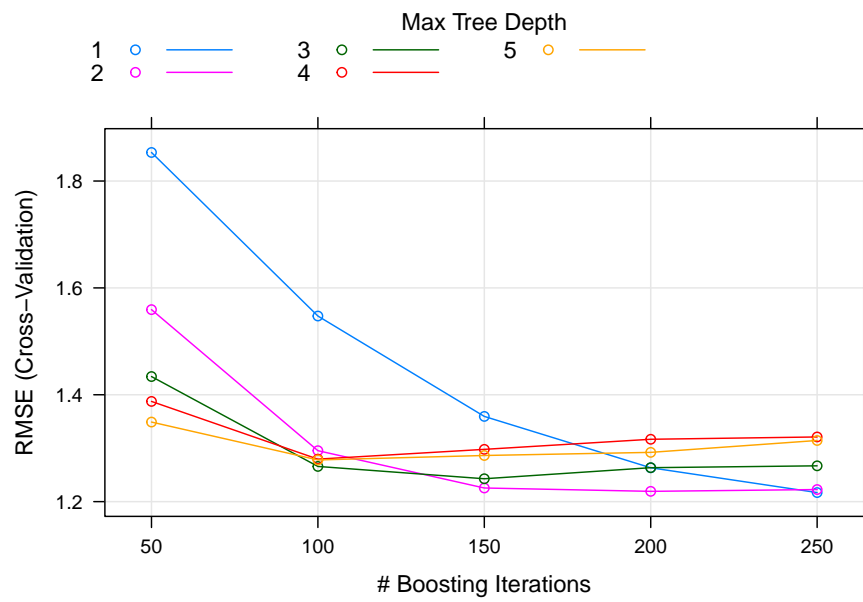
```
## Stochastic Gradient Boosting
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 289, 289, 289, 289, 289, 289, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE      Rsquared  MAE
##   1                  50      1.853439  0.6550573  1.4985804
##   1                  100     1.547356  0.7484921  1.2635775
```

```

##      1      150      1.359464  0.7941783  1.1116649
##      1      200      1.263360  0.8152576  1.0273078
##      1      250      1.216972  0.8238354  0.9842987
##      2       50      1.559349  0.7549692  1.2703898
##      2      100      1.295349  0.8121515  1.0550231
##      2      150      1.225522  0.8259530  0.9949898
##      2      200      1.219263  0.8282347  0.9847772
##      2      250      1.222610  0.8269006  0.9882843
##      3       50      1.434103  0.7828979  1.1618345
##      3      100      1.265869  0.8154077  1.0207212
##      3      150      1.242808  0.8187860  0.9946575
##      3      200      1.263512  0.8139170  1.0132197
##      3      250      1.266998  0.8123660  1.0169904
##      4       50      1.387541  0.7897254  1.1230060
##      4      100      1.279528  0.8095927  1.0299009
##      4      150      1.297827  0.8045587  1.0399179
##      4      200      1.316687  0.7979882  1.0556691
##      4      250      1.321038  0.7972775  1.0621347
##      5       50      1.348922  0.7980728  1.0974910
##      5      100      1.277990  0.8108521  1.0389748
##      5      150      1.286355  0.8066147  1.0393349
##      5      200      1.292184  0.8042104  1.0372671
##      5      250      1.314433  0.7973539  1.0572041
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 250, interaction.depth =
## 1, shrinkage = 0.1 and n.minobsinnode = 10.

```

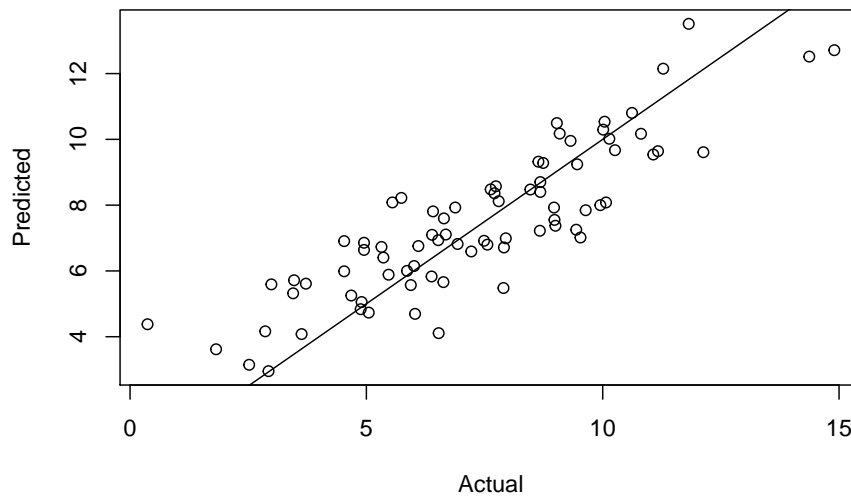
```
plot(carseats.gbm)
```



```

carseats.pred <- predict(carseats.gbm, carseats_test, type = "raw")
plot(carseats_test$Sales, carseats.pred,
     main = "Gradient Boosting Regression: Predicted vs. Actual",
     xlab = "Actual",
     ylab = "Predicted")
abline(0,1)

```

Gradient Boosing Regression: Predicted vs. Actual

```
(carseats.gbm.rmse <- RMSE(pred = carseats.pred,
                           obs = carseats_test$Sales))
```

```
## [1] 1.402428
```

```
rm(carseats.pred)
```

```
#plot(varImp(carseats.gbm), main="Variable Importance with Gradient Boosting")
```

7.6 Summary

Okay, I'm going to tally up the results! For the classification division, the winner is the manual classification tree! Gradient boosting made a valiant run at it, but came up just a little short.

```
rbind(data.frame(model = "Manual Class", Acc = round(oj_model_1b_cm$overall["Accuracy"], 5),
                 data.frame(model = "Class w.tuneGrid", Acc = round(oj_model_3_cm$overall["Accuracy"], 5)),
                 data.frame(model = "Bagging", Acc = round(oj.bag.acc, 5)),
                 data.frame(model = "Random Forest", Acc = round(oj.frst.acc, 5)),
                 data.frame(model = "Gradient Boosting", Acc = round(oj.gbm.acc, 5))
) %>% arrange(desc(Acc))
```



```
##           model      Acc
## 1      Manual Class 0.85915
## 2 Gradient Boosting 0.85446
## 3 Class w.tuneGrid 0.84507
## 4           Bagging 0.82629
## 5      Random Forest 0.82629
```

And now for the regression division, the winner is... gradient boosting!

```
rbind(data.frame(model = "Manual ANOVA", RMSE = round(carseats_model_1_pruned_rmse, 5)),
      data.frame(model = "ANOVA w.tuneGrid", RMSE = round(carseats_model_3_pruned_rmse, 5)),
      data.frame(model = "Bagging", RMSE = round(carseats.bag.rmse, 5)),
      data.frame(model = "Random Forest", RMSE = round(carseats.frst.rmse, 5)),
      data.frame(model = "Gradient Boosting", RMSE = round(carseats.gbm.rmse, 5))
) %>% arrange(RMSE)
```

```
##           model      RMSE
## 1 Gradient Boosting 1.40243
## 2      Random Forest 1.75811
## 3           Bagging 1.93279
## 4 ANOVA w.tuneGrid 2.29833
## 5      Manual ANOVA 2.38806
```

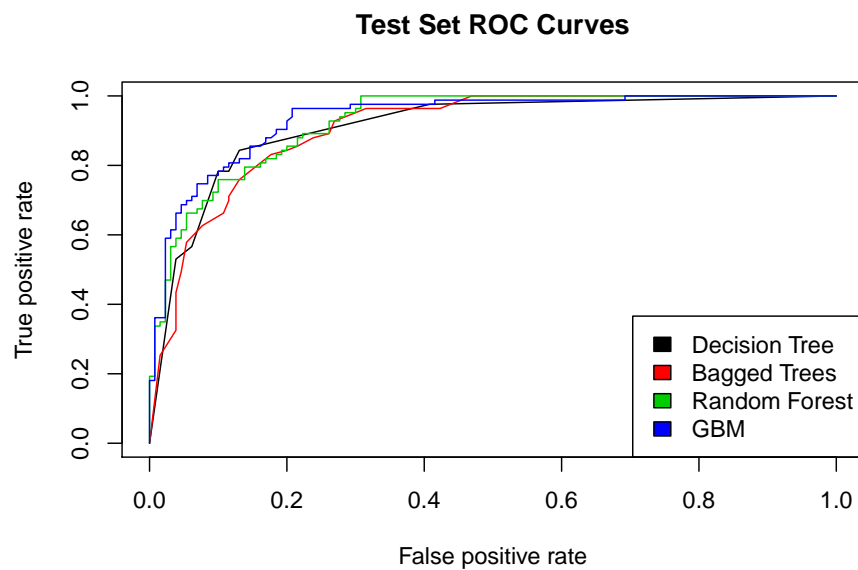
Here are plots of the ROC curves for all the models (one from each chapter) on the same graph. The `ROCR` package provides the `prediction()` and `performance()` functions which generate the data required for plotting the ROC curve, given a set of predictions and actual (true) values. The more “up and to the left” the ROC curve of a model is, the better the model. The AUC performance metric is literally the “Area Under the ROC Curve”, so the greater the area under this curve, the higher the AUC, and the better-performing the model is.

```
library(ROCR)
# List of predictions
oj.class.pred <- predict(oj_model_3, oj_test, type = "prob")[,2]
oj.bag.pred <- predict(oj.bag, oj_test, type = "prob")[,2]
oj.frst.pred <- predict(oj.frst, oj_test, type = "prob")[,2]
oj.gbm.pred <- predict(oj.gbm, oj_test, type = "prob")[,2]

preds_list <- list(oj.class.pred, oj.bag.pred, oj.frst.pred, oj.gbm.pred)
#preds_list <- list(oj.class.pred)

# List of actual values (same for all)
m <- length(preds_list)
actuals_list <- rep(list(oj_test$Purchase), m)
```

```
# Plot the ROC curves
pred <- prediction(preds_list, actuals_list)
#pred <- prediction(oj.class.pred[,2], oj_test$Purchase)
rocs <- performance(pred, "tpr", "fpr")
plot(rocs, col = as.list(1:m), main = "Test Set ROC Curves")
legend(x = "bottomright",
      legend = c("Decision Tree", "Bagged Trees", "Random Forest", "GBM"),
      fill = 1:m)
```



7.7 Reference

Penn State University, STAT 508: Applied Data Mining and Statistical Learning, “Lesson 11: Tree-based Methods”. <https://newonlinecourses.science.psu.edu/stat508/lesson/11>.

Brownlee, Jason. “Classification And Regression Trees for Machine Learning”, Machine Learning Mastery. <https://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/>.

Brownlee, Jason. “A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning”, Machine Learning Mastery. <https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/>.

DataCamp: Machine Learning with Tree-Based Models in R

An Introduction to Statistical Learning by Gareth James, et al.

SAS Documentation

StatMethods: Tree-Based Models

Machine Learning Plus

GBM (Boosted Models) Tuning Parameters from Listen Data

Harry Southworth on GitHub

Gradient Boosting Classification with GBM in R in DataTechNotes

Molnar, Christoph. “Interpretable machine learning. A Guide for Making Black Box Models Explainable”, 2019. <https://christophm.github.io/interpretable-ml-book/>.

Chapter 8

Regularization

Chapter 9

Non-linear Models

Linear methods can model nonlinear relationships by including polynomial terms, interaction effects, and variable transformations. However, it is often difficult to identify how to formulate the model. Nonlinear models may be preferable because you do not need to know the the exact form of the nonlinearity prior to model training.

9.1 Splines

A regression spline fits a piecewise polynomial to the range of X partitioned by *knots* (K knots produce $K + 1$ piecewise polynomials) **James et al** (James et al., 2013). The polynomials can be of any degree d , but are usually in the range $[0, 3]$, most commonly 3 (a cubic spline). To avoid discontinuities in the fit, a degree- d spline is constrained to have continuity in derivatives up to degree $d-1$ at each knot.

A cubic spline fit to a data set with K knots, performs least squares regression with an intercept and $3 + K$ predictors, of the form

$$y_i = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 h(X, \xi_1) + \beta_5 h(X, \xi_2) + \dots + \beta_{K+3} h(X, \xi_K)$$

where ξ_1, \dots, ξ_K are the knots are truncated power basis functions $h(X, \xi) = (X - \xi)^3$ if $X > \xi$, else 0.

Splines can have high variance at the outer range of the predictors. A **natural spline** is a regression spline additionally constrained to be linear at the boundaries.

How many knots should there be, and Where should the knots be placed? It is common to place knots in a uniform fashion, with equal numbers of points

between each knot. The number of knots is typically chosen by trial and error using cross-validation to minimize the RSS. The number of knots is usually expressed in terms of degrees of freedom. A cubic spline will have $K + 3 + 1$ degrees of freedom. A natural spline has $K + 3 + 1 - 5$ degrees of freedom due to the constraints at the endpoints.

A further constraint can be added to reduce overfitting by enforcing smoothness in the spline. Instead of minimizing the loss function $\sum (y - g(x))^2$ where $g(x)$ is a natural spline, minimize a loss function with an additional penalty for variability:

$$L = \sum (y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt.$$

The function $g(x)$ that minimizes the loss function is a *natural cubic spline* with knots at each x_1, \dots, x_n . This is called a **smoothing spline**. The larger λ is, the greater the penalty on variation in the spline. In a smoothing spline, you do not optimize the number or location of the knots – there is a knot at each training observation. Instead, you optimize λ . One way to optimize λ is cross-validation to minimize RSS. Leave-one-out cross-validation (LOOCV) can be computed efficiently for smoothing splines.

9.2 MARS

Multivariate adaptive regression splines (MARS) is a non-parametric algorithm that creates a piecewise linear model to capture nonlinearities and interactions effects. The resulting model is a weighted sum of *basis* functions $B_i(X)$:

$$\hat{y} = \sum_{i=1}^k w_i B_i(x)$$

The basis functions are either a constant (for the intercept), a *hinge* function of the form $\max(0, x - x_0)$ or $\max(0, x_0 - x)$ (a more concise representation is $[\pm(x - x_0)]_+$), or products of two or more hinge functions (for interactions). MARS automatically selects which predictors to use and what predictor values to serve as the *knots* of the hinge functions.

MARS builds a model in two phases: the forward pass and the backward pass, similar to growing and pruning of tree models. MARS starts with a model consisting of just the intercept term equaling the mean of the response values. It then assesses every predictor to find a basis function pair consisting of opposing sides of a mirrored hinge function which produces the maximum improvement in the model error. MARS repeats the process until either it reaches a predefined limit of terms or the error improvement reaches a predefined limit.

MARS generalizes the model by removing terms according to the generalized cross validation (GCV) criterion. GCV is a form of regularization: it trades off goodness-of-fit against model complexity.

The `earth::earth()` function (documentation) performs the MARS algorithm (*the term “MARS” is trademarked, so open-source implementations use “Earth” instead*). The caret implementation tunes two parameters: `nprune` and `degree`. `nprune` is the maximum number of terms in the pruned model. `degree` is the maximum degree of interaction (default is 1 (no interactions)). However, there are other hyperparameters in the model that may improve performance, including `minspan` which regulates the number of knots in the predictors.

Here is an example using the Ames housing data set (following this tutorial).

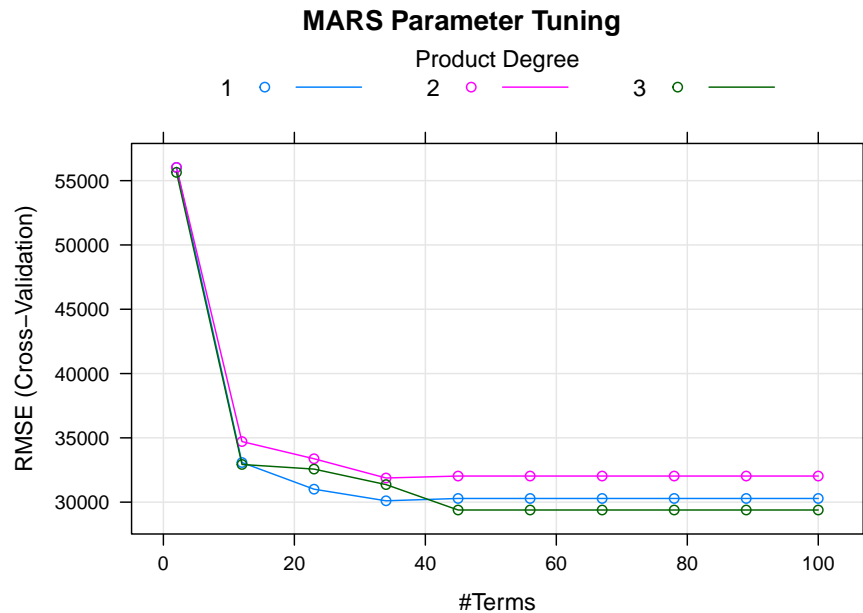
```
library(tidyverse)
library(earth)
library(caret)

# set up
ames <- AmesHousing::make_ames()
set.seed(12345)
idx <- createDataPartition(ames$Sale_Price, p = 0.80, list = FALSE)
ames_train <- ames[idx, ] %>% as.data.frame()
ames_test  <- ames[-idx, ]

m <- train(
  x = subset(ames_train, select = -Sale_Price),
  y = ames_train$Sale_Price,
  method = "earth",
  metric = "RMSE",
  minspan = -15,
  trControl = trainControl(method = "cv", number = 10),
  tuneGrid = expand.grid(
    degree = 1:3,
    nprune = seq(2, 100, length.out = 10) %>% floor()
  )
)
```

The model plot shows the best tuning parameter combination.

```
plot(m, main = "MARS Parameter Tuning")
```



```
m$bestTune
```

```
##      nprune degree
## 25      45      3
```

How does this model perform against the holdout data?

```
caret::postResample(
  pred = log(predict(m, newdata = ames_test)),
  obs = log(ames_test$Sale_Price)
)
```

```
##      RMSE  Rsquared      MAE
## 0.16515620 0.85470300 0.09319503
```

9.3 GAM

Generalized additive models (GAM) allow for non-linear relationships between each feature and the response by replacing each linear component $\beta_j x_{ij}$ with a nonlinear function $f_j(x_{ij})$. The GAM model is of the form

$$y_i = \beta_0 + \sum f_j(x_{ij}) + \epsilon_i.$$

It is called an additive model because we calculate a separate f_j for each X_j , and then add together all of their contributions.

The advantage of GAMs is that they automatically model non-linear relationships so you do not need to manually try out many different transformations on each variable individually. And because the model is additive, you can still examine the effect of each X_j on Y individually while holding all of the other variables fixed. The main limitation of GAMs is that the model is restricted to be additive, so important interactions can be missed unless you explicitly add them.

Chapter 10

Support Vector Machines

These notes rely on (James et al., 2013), (Hastie et al., 2017), and (Kuhn and Johnson, 2016). I also reviewed the material in PSU’s Applied Data Mining and Statistical Learning (STAT 508), and the *e1071* Support Vector Machines vignette.

The Support Vector Machines (SVM) algorithm finds the optimal separating hyperplane between members of two classes using an appropriate nonlinear mapping to a sufficiently high dimension. The hyperplane is defined by the observations that lie within a margin optimized by a cost hyperparameter. These observations are called the *support vectors*.

SVM is an extension of the *support vector classifier* which in turn is a generalization of the simple and intuitive *maximal margin classifier*.

10.1 Maximal Margin Classifier

The maximal margin classifier is the optimal hyperplane defined in the (rare) case where two classes are *linearly separable*. Given an $n \times p$ data matrix X with binary response variable defined as $y \in [-1, 1]$ it *may* be possible to define a p -dimensional hyperplane $h(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \cdots + \beta_p X_p = x_i^T \beta + \beta_0 = 0$ such that all observations of each class fall on opposite sides of the hyperplane. This “separating hyperplane” has the property that if β is constrained to be a unit vector, $\|\beta\| = \sum \beta^2 = 1$, then the product of the hyperplane and response variables are positive perpendicular distances from the hyperplane, the smallest of which may be termed the hyperplane *margin*, M ,

$$y_i(x_i' \beta + \beta_0) \geq M.$$

The maximal margin classifier is the hyperplane with the maximum margin. That is, $\max\{M\}$ subject to $\|\beta\| = 1$. A separating hyperplane rarely exists. In fact, even if a separating hyperplane does exist, its resulting margin is probably undesirably narrow.

10.2 Support Vector Classifier

The maximal margin classifier can be generalized to non-separable cases using a so-called “soft margin”. The generalization is called the *support vector classifier*. The soft margin allows some misclassification in the interest of greater robustness to individual observations. The support vector classifier optimizes

$$y_i(x_i' \beta + \beta_0) \geq M(1 - \xi_i)$$

where the ξ_i are positive *slack variables* whose sum is bounded by some constant tuning parameter $\sum \xi_i \leq \text{constant}$. The slack variable values indicate where the observation lies: $\xi_i = 0$ observations lie on the correct side of the margin; $\xi_i > 0$ observations lie on the wrong side of the margin; $\xi_i > 1$ observations lie on the wrong side of the hyperplane. The constant sets the tolerance for margin violation. If $\text{constant} = 0$, then all observations must reside on the correct side of the margin, as in the maximal margin classifier. The *constant* controls the bias-variance trade-off. As the *constant* increases, the margin widens and allows more violations. The classifier bias increases but its variance decreases.

The support vector classifier is usually defined by dropping the $\|\beta\| = 1$ constraint, and defining $M = 1/\|\beta\|$. The optimization problem then becomes

$$\min \|\beta\| \quad \text{s.t.} \quad \begin{cases} y_i(x_i^T \beta + \beta_0) \geq 1 - \xi_i, & \forall i \\ \xi_i \geq 0, & \sum \xi_i \leq \text{constant}. \end{cases}$$

This is a quadratic equation with linear inequality constraints, so it is a convex optimization problem which can be solved using Lagrange multipliers. Re-express the optimization problem as

$$\min_{\beta_0, \beta} \frac{1}{2} \|\beta\|^2 = C \sum_{i=1}^N \xi_i \quad \text{s.t.} \quad \xi_i \geq 0, \quad y_i(x_i^T \beta + \beta_0) \geq 1 - \xi_i, \quad \forall i$$

where the “cost” parameter C replaces the constant and penalizes large residuals. This optimization problem is equivalent to *another* optimization problem, the familiar *loss + penalty* formulation:

$$\min_{\beta_0, \beta} \sum_{i=1}^N [1 - y_i f(x_i)]_+ + \frac{\lambda}{2} \|\beta\|^2$$

where $\lambda = 1/C$ and $[1 - y_i f(x_i)]_+$ is a “hinge” loss function with $f(x_i) = \text{sign}[Pr(Y = +1|x) - 1/2]$.

The parameter estimates can be written as functions of a set of unknown parameters (α_i) and data points. The solution to the optimization problem requires only the inner products of the observations, represented as $\langle x_i, x_j \rangle$,

$$f(x) = \beta_0 + \sum_{i=1}^n \alpha_i \langle x, x_i \rangle$$

The solution has the interesting property that only observations on or within the margin affect the hyperplane. These observations are known as support vectors. As the constant increases, the number of violating observations increase, and thus the number of support vectors increases. This property makes the algorithm robust to the extreme observations far away from the hyperplane.

The parameter estimators for α_i are nonzero only for the support vectors in the solution—that is, if a training observation is not a support vector, then its α_i equals zero.

The only shortcoming with the algorithm is that it presumes a linear decision boundary.

10.3 Support Vector Machines

Enlarging the feature space of the support vector classifier accommodates non-linear relationships. Support vector machines do this in a specific way, using *kernels*. The kernel is a generalization of the inner product with form $K(x_i, x'_i)$. So the linear kernel is simply

$$K(x_i, x'_i) = \langle x, x_i \rangle$$

and the solution is

$$f(x) = \beta_0 + \sum_{i=1}^n \alpha_i K(x_i, x'_i)$$

K can take another form instead, such as polynomial

$$K(x, x') = (\gamma \langle x, x' \rangle + c_0)^d$$

or radial

$$K(x, x') = \exp\{-\gamma \|x - x'\|^2\}.$$

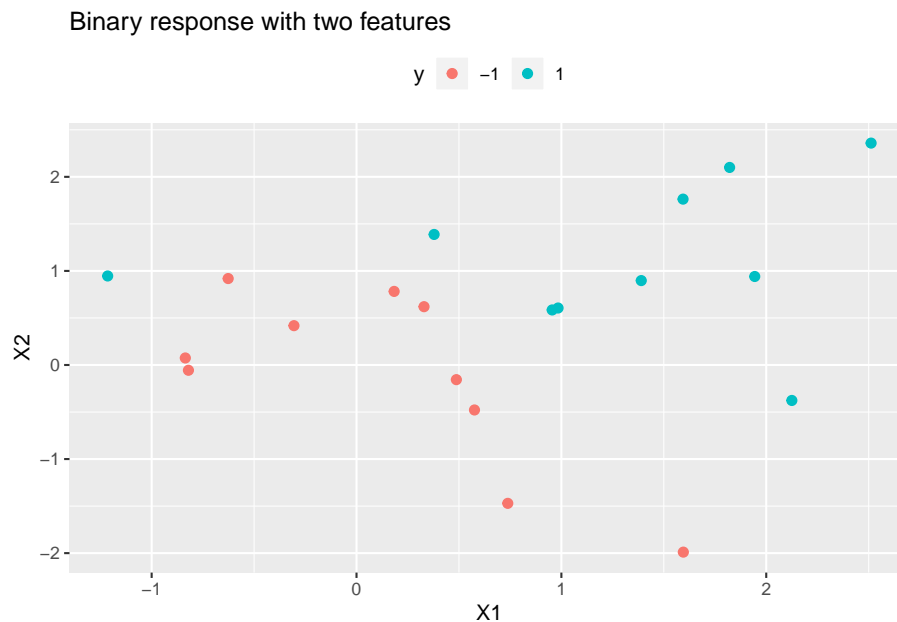
10.4 Example

Here is a data set of two classes $y \in [-1, 1]$ described by two features $X1$ and $X2$.

```
library(tidyverse)
set.seed(1)
x <- matrix(rnorm(20*2), ncol=2)
y <- c(rep(-1, 10), rep(1, 10))
x[y==1, ] <- x[y==1, ] + 1
train_data <- data.frame(x, y)
train_data$y <- as.factor(y)
```

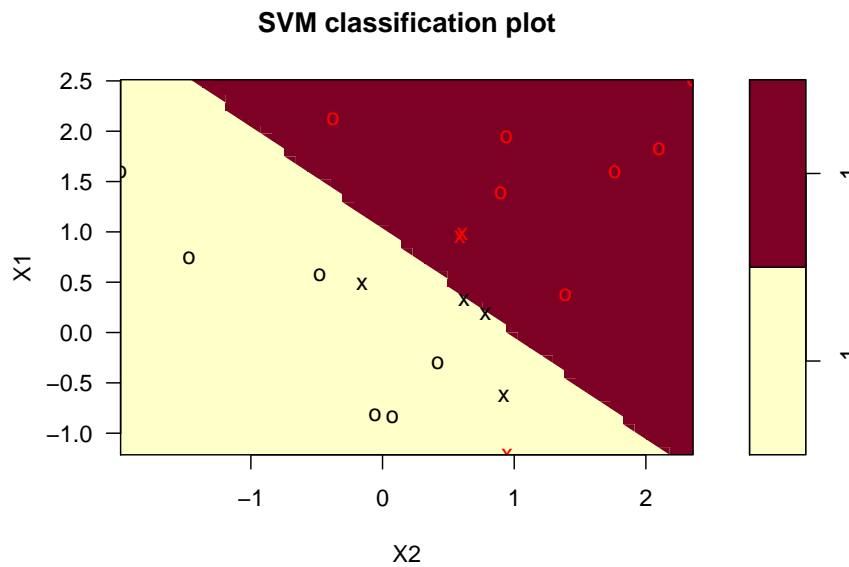
A scatter plot reveals whether the classes are linearly separable.

```
ggplot(train_data, aes(x = X1, y = X2, color = y)) +
  geom_point(size = 2) +
  labs(title = "Binary response with two features") +
  theme(legend.position = "top")
```



No, they are not linearly separable. Now fit a support vector machine. The **e1071** library implements the SVM algorithm. `svm(..., kernel="linear")` fits a support vector classifier. Change the kernel to `c("polynomial", "radial")` for SVM. Try a cost of 10.


```
library(e1071)
m <- svm(
  y ~ .,
  data = train_data,
  kernel = "linear",
  type = "C-classification", # (default) for classification
  cost = 10, # default is 1
  scale = FALSE # do not standardize features
)
plot(m, train_data)
```



The support vectors are plotted as “x’s”. There are seven of them.

```
m$index
```

```
## [1] 1 2 5 7 14 16 17
```

The summary shows adds additional information, including the distribution of the support vector classes.

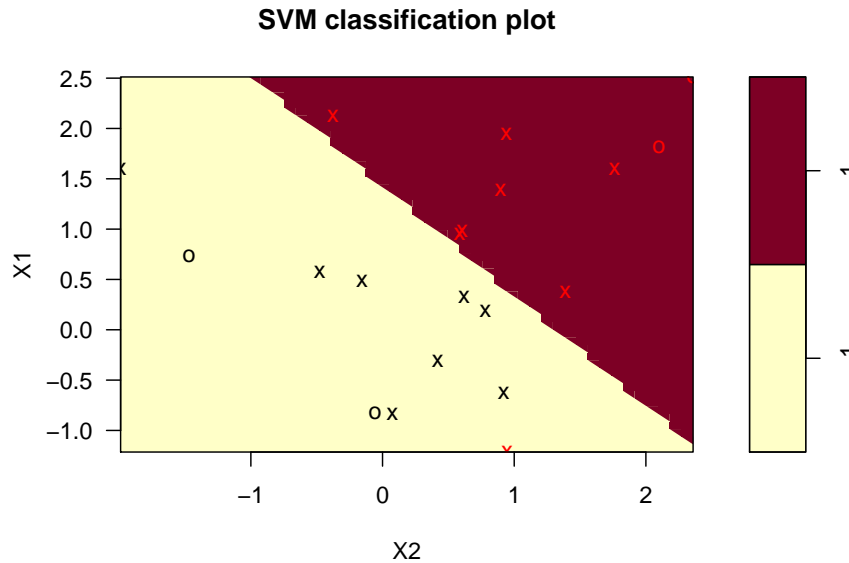
```
summary(m)
```

```
##
```

```
## Call:
## svm(formula = y ~ ., data = train_data, kernel = "linear", type = "C-classification",
##      cost = 10, scale = FALSE)
##
##
## Parameters:
##      SVM-Type:  C-classification
##      SVM-Kernel: linear
##              cost: 10
##
## Number of Support Vectors:  7
##
##   ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##   -1 1
```

The seven support vectors are comprised of four in one class, three in the other. What if we lower the cost of margin violations? This will increase bias and lower variance.

```
m <- svm(
  y ~ .,
  data = train_data,
  kernel = "linear",
  type = "C-classification",
  cost = 0.1,
  scale = FALSE
)
plot(m, train_data)
```



There are many more support vectors now. *(In case you hoped to see the linear decision boundary formulation, or at least a graphical representation of the margins, keep hoping. The model is generalized beyond two features, so it evidently does not worry too much about supporting sanitized two-feature demos.)*

Which cost level yields the *best* predictive performance on holdout data? Use cross validation to find out. SVM defaults to 10-fold CV. I'll try seven candidate values for `cost`.

```
set.seed(1)
m_tune <- tune(
  svm,
  y ~ .,
  data = train_data,
  kernel = "linear",
  ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100))
)
summary(m_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
```

```
## cost
## 0.1
##
## - best performance: 0.05
##
## - Detailed performance results:
##      cost error dispersion
## 1 1e-03 0.55 0.4377975
## 2 1e-02 0.55 0.4377975
## 3 1e-01 0.05 0.1581139
## 4 1e+00 0.15 0.2415229
## 5 5e+00 0.15 0.2415229
## 6 1e+01 0.15 0.2415229
## 7 1e+02 0.15 0.2415229
```

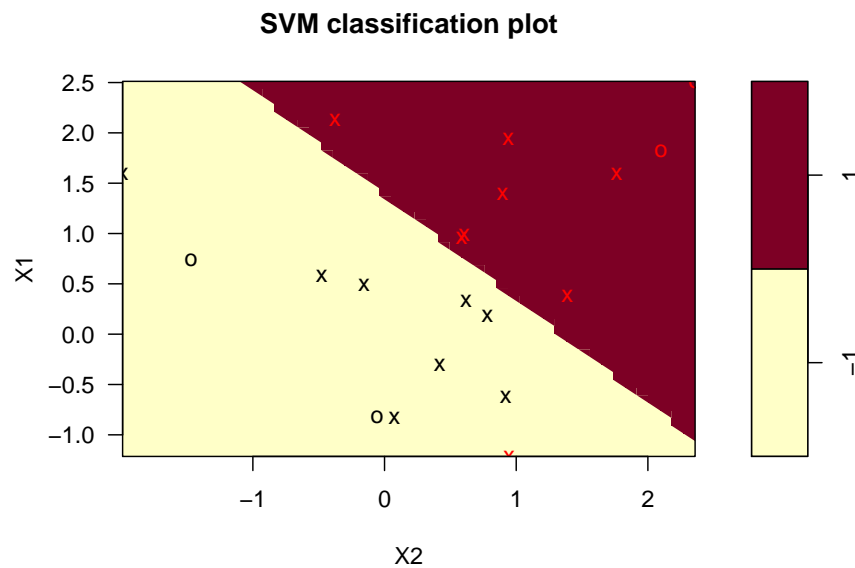
The lowest cross-validation error rate is 0.10 with `cost = 0.1`. `tune()` saves the best tuning parameter value.

```
m_best <- m_tune$best.model
summary(m_best)
```

```
##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = train_data, ranges = list(cost = c(
##      0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##
##
## Parameters:
##      SVM-Type:  C-classification
##      SVM-Kernel: linear
##              cost: 0.1
##
## Number of Support Vectors: 16
##
##      ( 8 8 )
##
##
## Number of Classes: 2
##
## Levels:
##      -1 1
```

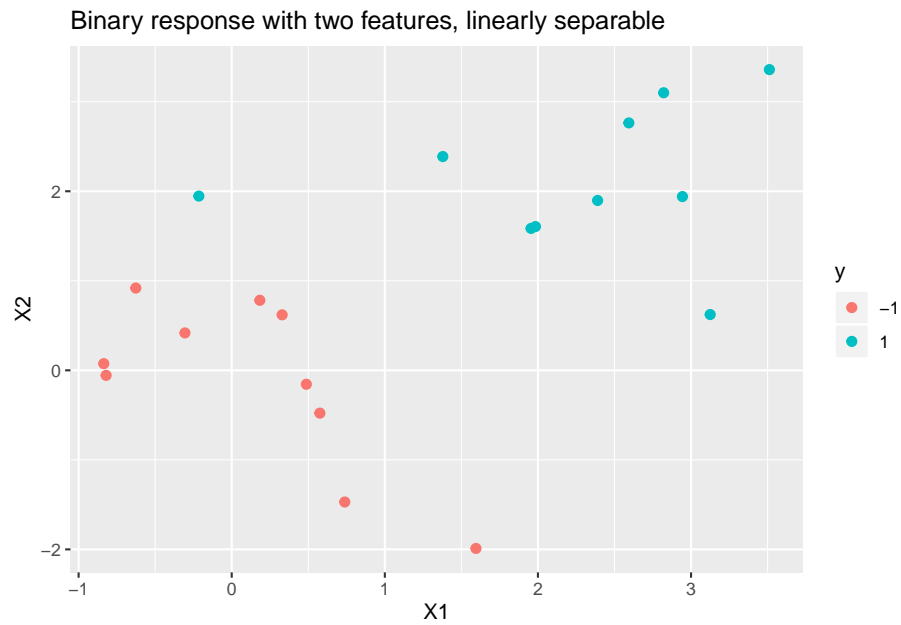
There are 16 support vectors, 8 in each class. This is a pretty wide margin.

```
plot(m_best, train_data)
```



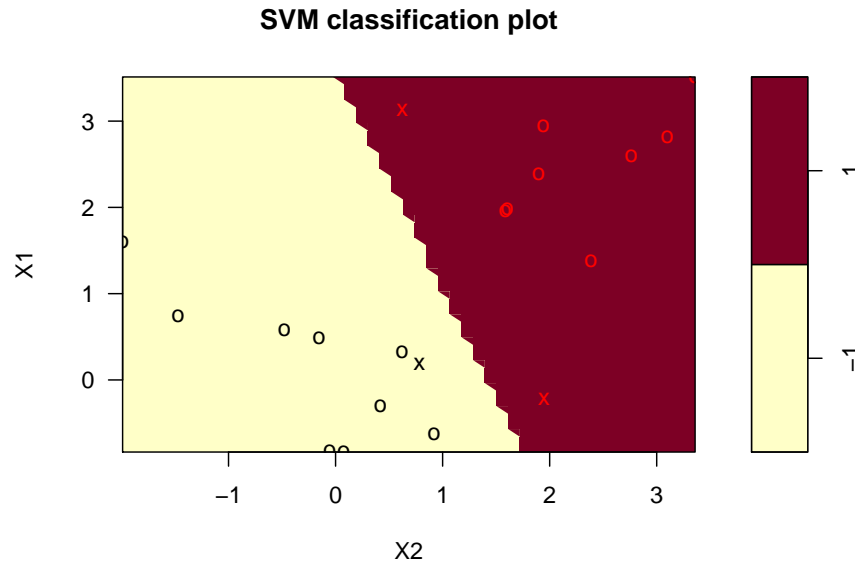
What if the classes had been linearly separable? Then we could create a maximal margin classifier.

```
train_data_2 <- train_data %>%
  mutate(
    X1 = X1 + ifelse(y==1, 1.0, 0),
    X2 = X2 + ifelse(y==1, 1.0, 0)
  )
ggplot(train_data_2, aes(x = X1, y = X2, color = y)) +
  geom_point(size = 2) +
  labs(title = "Binary response with two features, linearly separable")
```



Specify a huge cost = $1e5$ so that no support vectors violate the margin.

```
m2 <- svm(  
  y ~ .,  
  data = train_data_2,  
  kernel = "linear",  
  cost = 1e5,  
  scale = FALSE # do not standardize features  
)  
plot(m2, train_data_2)
```



```
summary(m2)
```

```
##
## Call:
## svm(formula = y ~ ., data = train_data_2, kernel = "linear", cost = 1e+05,
##      scale = FALSE)
##
##
## Parameters:
##   SVM-Type:  C-classification
##   SVM-Kernel: linear
##         cost: 1e+05
##
## Number of Support Vectors:  3
##
##   ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##   -1 1
```

This model will have very low bias, but very high variance. To fit an SVM, use a different kernel. You can use `kernel = c("polynomial", "radial",`

"sigmoid"). For a polynomial model, also specify the polynomial degree. For a radial model, include the gamma value.

```
set.seed(1)
m3_tune <- tune(
  svm,
  y ~ .,
  data = train_data,
  kernel = "polynomial",
  ranges = list(
    cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100),
    degree = c(1, 2, 3)
  )
)
summary(m3_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost degree
##     1      1
##
## - best performance: 0.1
##
## - Detailed performance results:
##   cost degree error dispersion
## 1 1e-03      1 0.55 0.4377975
## 2 1e-02      1 0.55 0.4377975
## 3 1e-01      1 0.30 0.2581989
## 4 1e+00      1 0.10 0.2108185
## 5 5e+00      1 0.10 0.2108185
## 6 1e+01      1 0.15 0.2415229
## 7 1e+02      1 0.15 0.2415229
## 8 1e-03      2 0.70 0.4216370
## 9 1e-02      2 0.70 0.4216370
## 10 1e-01     2 0.70 0.4216370
## 11 1e+00     2 0.65 0.2415229
## 12 5e+00     2 0.50 0.3333333
## 13 1e+01     2 0.50 0.3333333
## 14 1e+02     2 0.50 0.3333333
## 15 1e-03     3 0.65 0.3374743
## 16 1e-02     3 0.65 0.3374743
```



```
## 17 1e-01      3  0.50  0.3333333
## 18 1e+00      3  0.40  0.3162278
## 19 5e+00      3  0.35  0.3374743
## 20 1e+01      3  0.35  0.3374743
## 21 1e+02      3  0.35  0.3374743
```

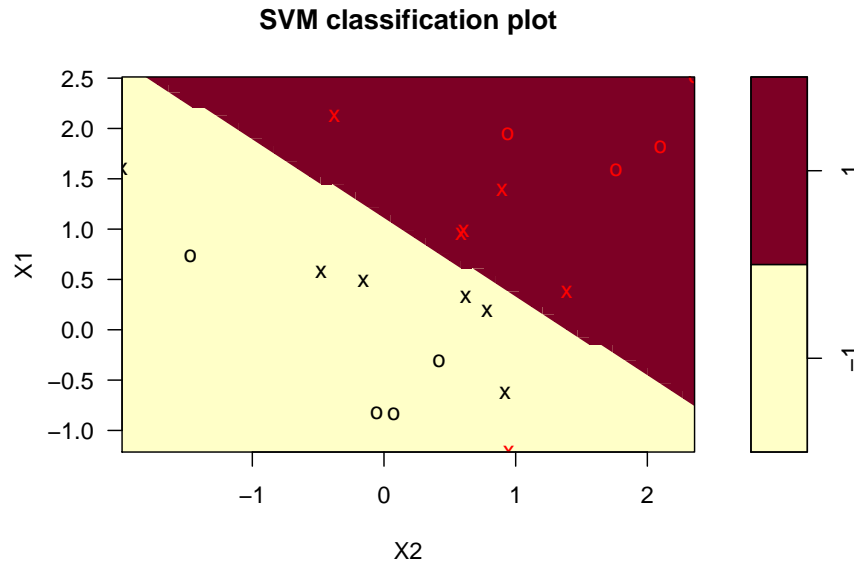
The lowest cross-validation error rate is 0.10 with cost = 1, polynomial degree 1.

```
m3_best <- m3_tune$best.model
summary(m3_best)
```

```
##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = train_data, ranges = list(cost = c(0.001,
##      0.01, 0.1, 1, 5, 10, 100), degree = c(1, 2, 3)), kernel = "polynomial")
##
##
## Parameters:
##   SVM-Type:  C-classification
## SVM-Kernel:  polynomial
##      cost:   1
##    degree:   1
##   coef.0:   0
##
## Number of Support Vectors:  12
##
## ( 6 6 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

There are 12 support vectors, 6 in each class. This is a pretty wide margin.

```
plot(m3_best, train_data)
```



10.5 Using Caret

The model can also be fit using **caret**. I'll used LOOCV since the data set is so small. Normalize the variables to make their scale comparable.

```
library(caret)
library(kernlab)

train_data_3 <- train_data %>%
  mutate(y = factor(y, labels = c("A", "B")))

m4 <- train(
  y ~ .,
  data = train_data_3,
  method = "svmPoly",
  preProcess = c("center", "scale"),
  trControl = trainControl(
    method = "cv",
    number = 5,
    summaryFunction = twoClassSummary, # Use AUC to pick the best model
    classProbs=TRUE
  )
)
```

```
m4$bestTune
```

```
## degree scale C  
## 8      1    0.1 0.5
```

```
#plot(m4)
```


Chapter 11

Principal Components Analysis

Chapter 12

Clustering

Chapter 13

Text Mining

Appendix

Here are miscellaneous skills, knowledge, and technologies I should know.

Publishing to BookDown

The **bookdown** package, written by Yihui Xie, is built on top of R Markdown and the **knitr** package. Use it to publish a book or long manuscript where each chapter is a separate file. There are instructions for how to author a book in his bookdown book (Xie, 2019). The main advantage of **bookdown** over R Markdown is that you can produce multi-page HTML output with numbered headers, equations, figures, etc., just like in a book. I'm using **bookdown** to create a compendium of all my data science notes.

The first step to using **bookdown** is installing the `**bookdown*` package with `install.packages("bookdown")`.

Next, create an account at bookdown.org, and connect the account to RStudio. Follow the instructions at <https://bookdown.org/home/about/>.

Finally, create a project in R Studio by creating a new project of type *Book Project using Bookdown*.

After creating all of your Markdown pages, knit the book or click the **Build Book** button in the Build panel.

Shiny Apps

Packages

R Packages (Wickham, 2015) by Hadley Wickham is a good manual on packages, but it does not include a full tutorial. The Developing R Packages Data Camp course is also helpful. I will set up my own exercise and present it here.

I will create a package for my pretend organization, “MF”. The package will include the following:

- R Markdown template. My template will integrate code, output, and commentary in a single R Markdown. The template will produce a familiar work product containing standard content (summary, data management, exploratory analysis, methods, results, conclusions), and a standard style (colors, typeface, size, logo).
- Functions. Common I/O functions for database retrieval, writing to Excel. Common graphing functions for ggplot styling.

I am mostly copying the logic and code from the `ggthemes` `economist.R` script.

Create a package

1. In the RStudio IDE, click `File > New Project`. Select “New Directory”. Select “R Package”. You can also use `devtools::create("mfstylr")`. This will create the minimum items for an R package.

The screenshot shows the 'New Project' dialog box in RStudio, with the 'Create R Package' tab selected. The dialog includes the following elements:

- Back** button: A button with a left-pointing arrow.
- Create R Package** title: The title of the current tab.
- Type:** A dropdown menu set to 'Package'.
- Package name:** A text input field containing 'mfstylr'.
- Create package based on source files:** A list box with an 'Add...' button and a 'Remove' button.
- Create project as subdirectory of:** A text input field containing 'C:/Users/mpfol/OneDrive/Documents/GitHub' and a 'Browse...' button.
- Options:** Two checkboxes: 'Create a git repository' (checked) and 'Use packrat with this project' (unchecked).
- Open in new session:** A checkbox at the bottom left.
- Create Project** and **Cancel** buttons: Buttons at the bottom right.

- + R directory: R scripts with function definitions.
- + man directory: documentation
- + NAMESPACE file: information about imported functions and functions made available (m
- + DESCRIPTION file: metadata about the package

2. Write functions in R scripts in R directory. Document with tags readable by *roxygen2* package.
3. Select XYZ > Install and Restart.

13.0.1 Document Functions with roxygen

Add roxygen documentation with `#'` characters. The first three lines are always the title, Description, and Details. They don't need any tags, but you need to separate them with blank lines.

Create Data

Add an RData file to your package with `use_data()`

Create Vignette

Add a directory and template vignette with `use_vignette(name, title)`.

```
use_vignette("Creating-Plots-with-mfstylr", "Creating Plots with mfstylr")
```

Step 2: Create an R Markdown template

I relied on this blog at *free range statistics* for a lot what follows. There is also good information about R Markdown and templates in Yihui Xie's **R Markdown: The Definitive Guide** (Xie et al., 2019).

Use `usethis::use_rmarkdown_template()` to create an Rmd template. I will create a "Kaggle Report" template. In the Console (or a script), enter

```
usethis::use_rmarkdown_template(
  template_name = "Kaggle Report",
  template_dir = "kaggle_report",
  template_description = "Template for creating Kaggle reports in RMarkdown.",
  template_create_dir = FALSE
)
```

Since my project directory is `C:\Users\mpfol\OneDrive\Documents\GitHub\mfstylr`, `use_rmarkdown_template()` creates subdirectories `.\inst\rmarkdown\templates\kaggle_report\skeleton` with three files

- `.\inst\rmarkdown\templates\kaggle_report\template.yaml`

- `.\inst\rmarkdown\templates\kaggle_report\skeleton\skeleton.Rmd`

My kaggle report template will include a logo. Looks like there are two ways to embed an image in your document. One is a direct image loading reference `!()`, but I don't think you can control the attributes this way. A second way is adding html tags.

```


# or for more control

```

Bibliography

- Agresti, A. (2013). *Categorical Data Analysis*. Wiley, 3rd edition. ISBN 978-0470463635.
- Fawcett, T. (2005). *An introduction to ROC analysis*. ELSEVIER.
- Hastie, T., Tibshirani, R., and Friedman, J. (2017). *The Elements of Statistical Learning*. Springer, New York, NY, 2nd edition. ISBN 978-0387848570978-0387848570.
- James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer, New York, NY, 1st edition. ISBN 978-1461471370.
- Kuhn, M. and Johnson, K. (2016). *Applied Predictive Modeling*. Springer, New York, NY, 1st edition. ISBN 978-1-4614-6848-6.
- Therneau, T. and Atkinson, E. (2019). *An Introduction to Recursive Partitioning Using the RPART Routines*. Chapman and Hall/CRC, Boca Raton, Florida.
- Wickham, H. (2015). *R Packages*. O'Reilly Media, Inc., Sebastopol, California.
- Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 978113870010.
- Xie, Y., Allaire, J. J., and Grolemund, G. (2019). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 9781138359338.