Appendix R3: Basic Models

WLS, GLM, Logistic and Trees

J. Alberto Espinosa

2/21/2023

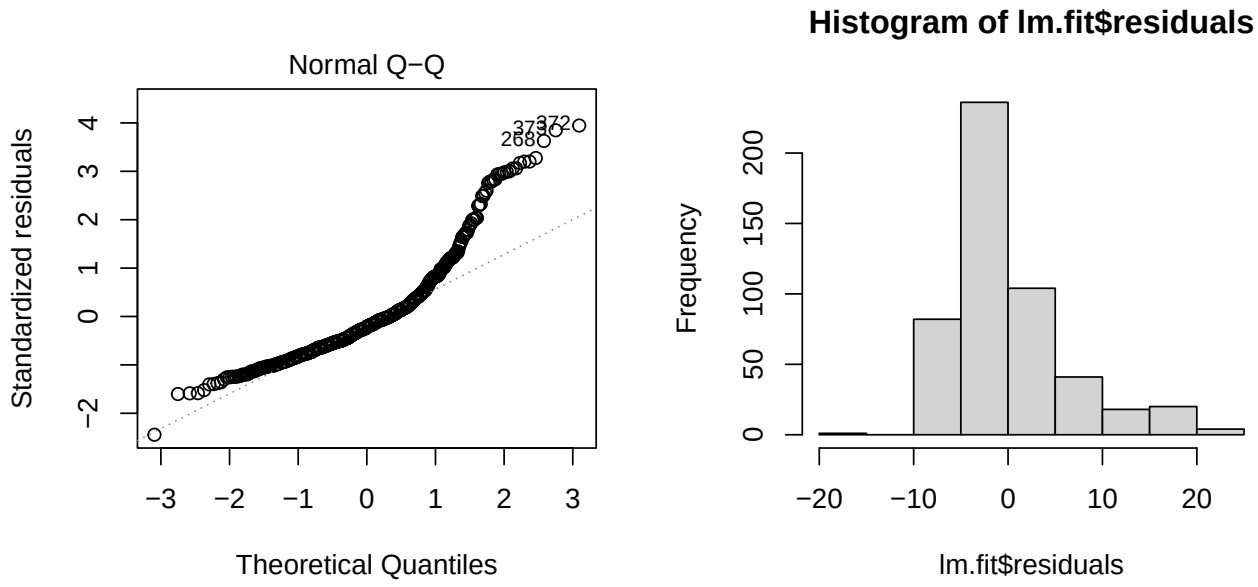
Contents

1. OLS Assumptions

Errors are Normally Distributed

QQ Plot

```
library(MASS) # To read the Boston housing market data
options(scipen = 4) # To minimize the use of scientific notation

# Fit the linear model

lm.fit <-lm(medv ~ lstat,
            data = Boston) # Fit the model and store results in lm.ols

# Then visually inspect the normality of the residuals

par(mfrow = c(1, 2))

plot(lm.fit, which = 2)
hist(lm.fit$residuals)
```



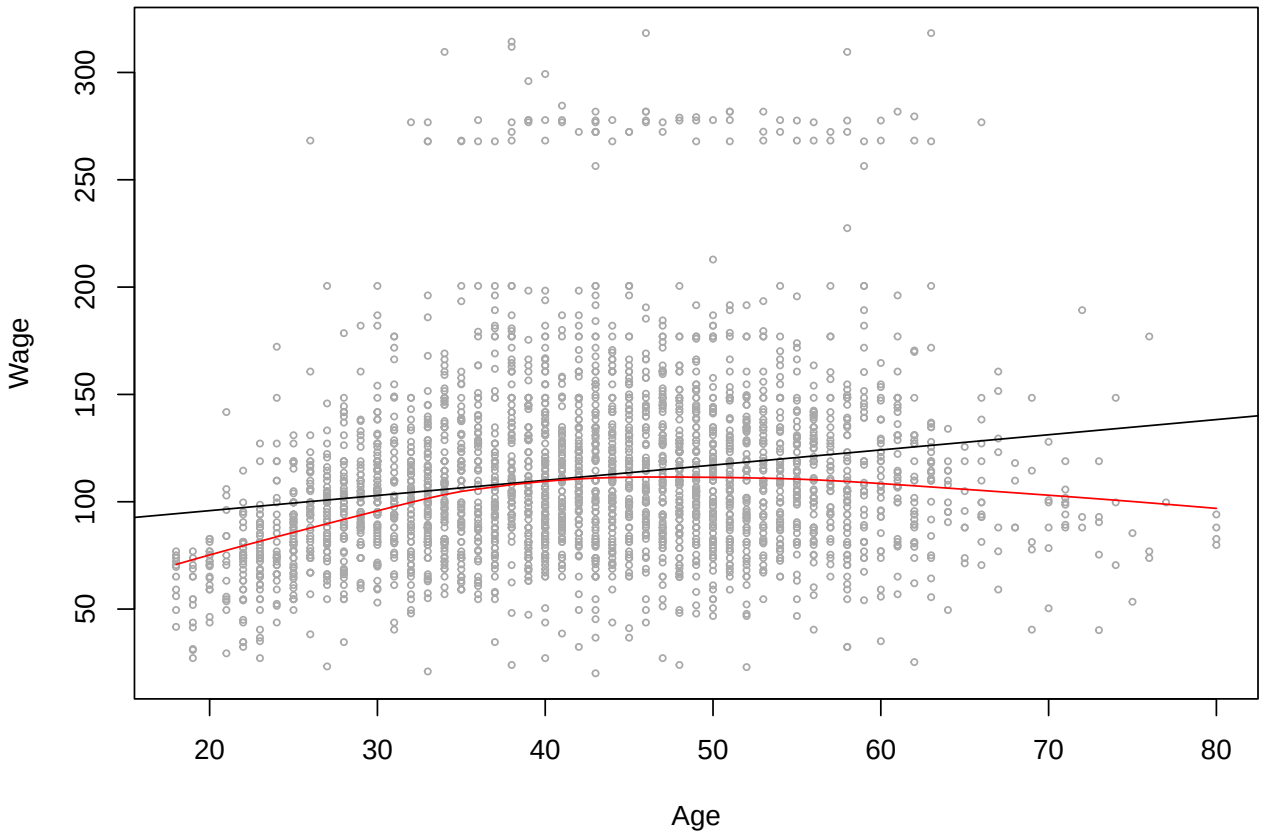

```
par(mfrow = c(1, 1))
```

Linearity

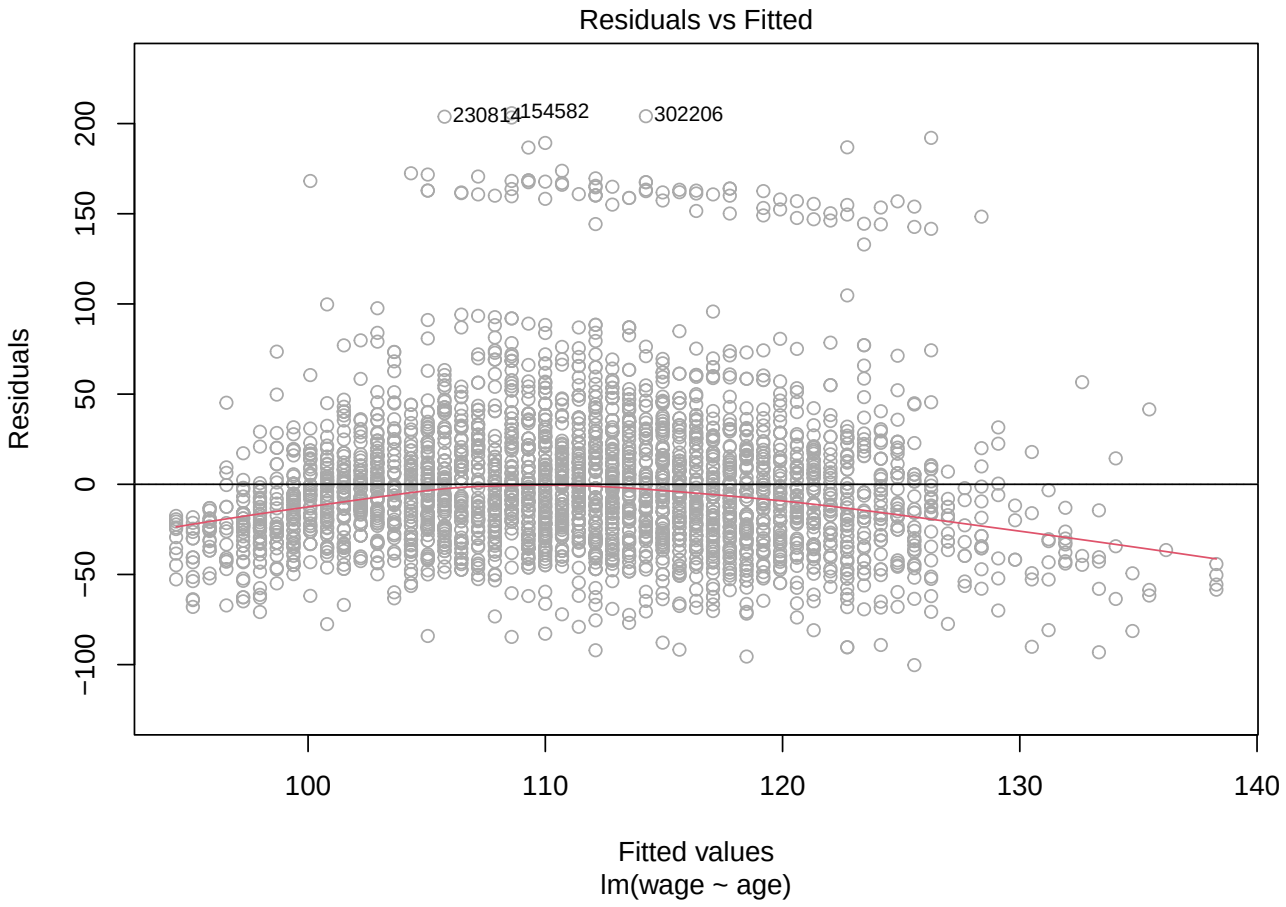
Let's use the Wage data set in the {ISLR} library to illustrate how to inspect for linearity visually.

```
library(ISLR)

lm.fit <- lm(wage ~ age, data = Wage)

plot(Wage$age, Wage$wage,
     xlab = "Age",
     ylab ="Wage",
     cex=.5,
```

```
      col="darkgrey")

abline(lm.fit) # Straight line

lines(lowess(Wage$age, Wage$wage),
       col = "red") # Trend curve
```

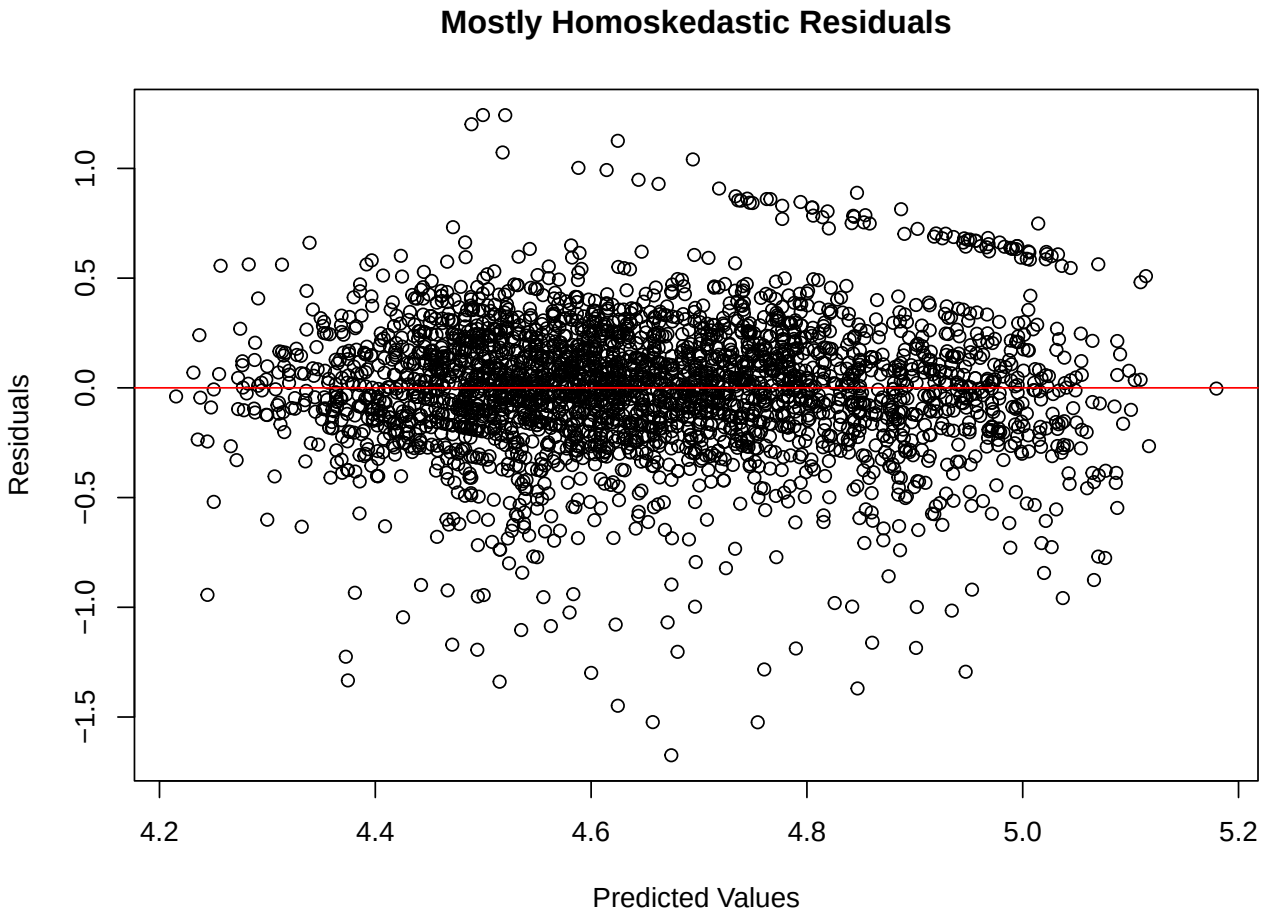



```
plot(lm.fit, which = 1,
     col = "darkgrey")

abline(h = 0) # Draw a horizontal line at 0
```

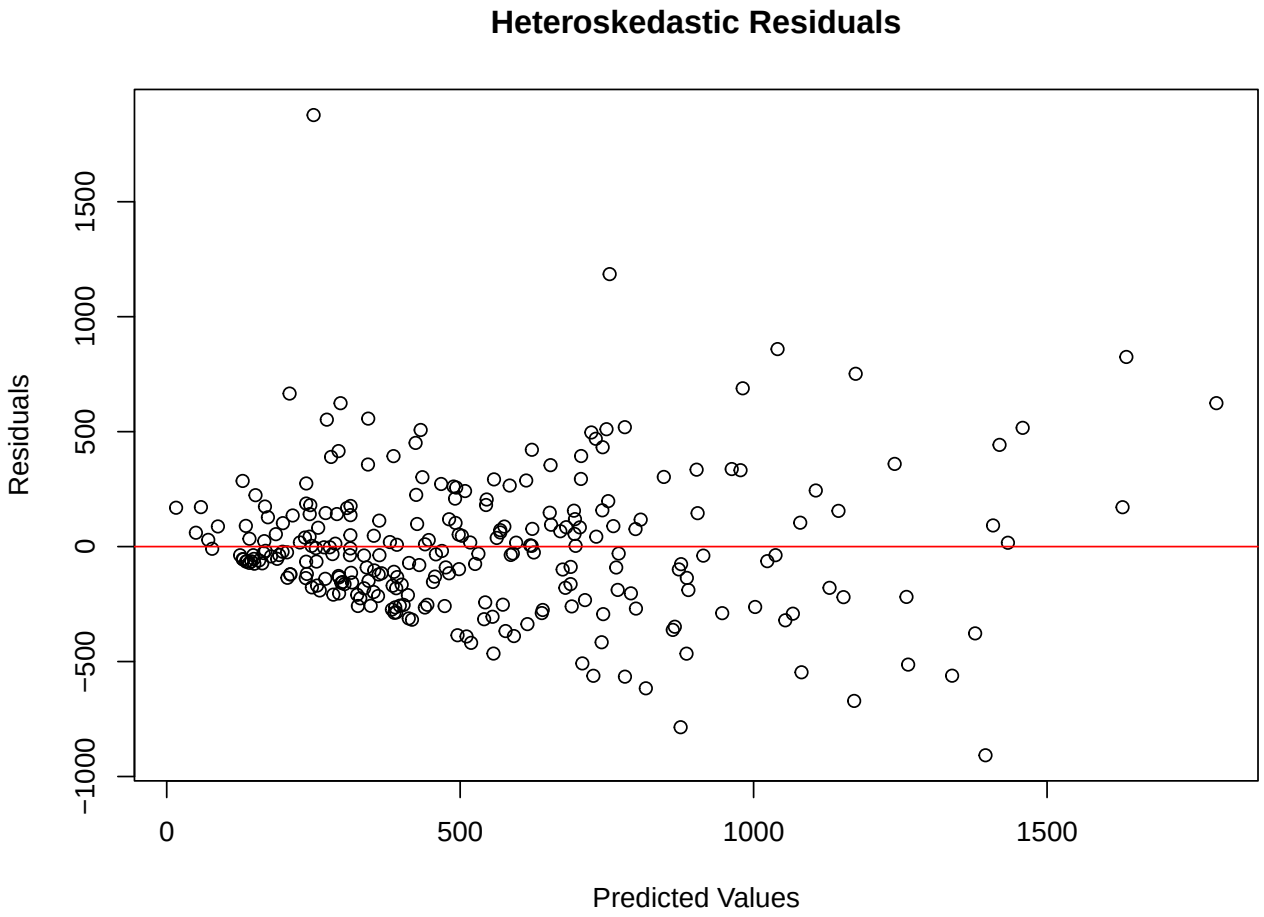


Heteroskedasticity

```
# Somewhat Homoskedastic Residuals

library(ISLR) # Contains Wage data set

lm.fit <- lm(logwage ~ year + age + race + education + jobclass,
             data = Wage)

# year + maritl + age + race + education + jobclass + health + health_ins,

plot(lm.fit$residuals ~ lm.fit$fitted.values,
     main = "Mostly Homoskedastic Residuals",
     xlab = "Predicted Values",
     ylab = "Residuals")

abline(h = 0, col = "red")
```




```r
# Heteroskedastic Residuals

library(ISLR) # Contains the Hitters data set

lm.fit <- lm(Salary ~ ., data = Hitters)

plot(lm.fit$residuals ~ lm.fit$fitted.values,
     main = "Heteroskedastic Residuals",
     xlab = "Predicted Values",
     ylab = "Residuals")

abline(h = 0, col = "red")
```



2. Weighted Least Squares (WLS) Regression

Testing for Heteroskedasticity

The OLS model assumes that residuals are even throughout the regression line. When you fit an OLS model, you need to inspect the first regression plot, which shows residuals against fitted values. The residuals should show a cloud of even data throughout. If it doesn't, then you need to test for **heteroskedasticity** or uneven residuals. If some residuals are much larger than others, then when you square them (to get the sum of error squares), the squared values will be even larger and those observations will pull the regression line disproportionately.

When hetoroskedasticy is present, we need to weight down the squared residuals and, instead of getting the regression line that minimizes the sum of squared errors, we fit a regression line that minimizes the "weighted" sum of squared errors.

There are various weights that can be used for WLS, but it is usually best to use the inverse of the OLS errors squaree, so that we weight down the sum proportionally to the size of the error squared. That is, the larger the error, the lower the weight we place when computing the regression line that best fits the data. This involves finding a weight vector, which we will call **wts**. This vector has one value for each observation, with the corresponding weight to apply to the residual associated with that observation.

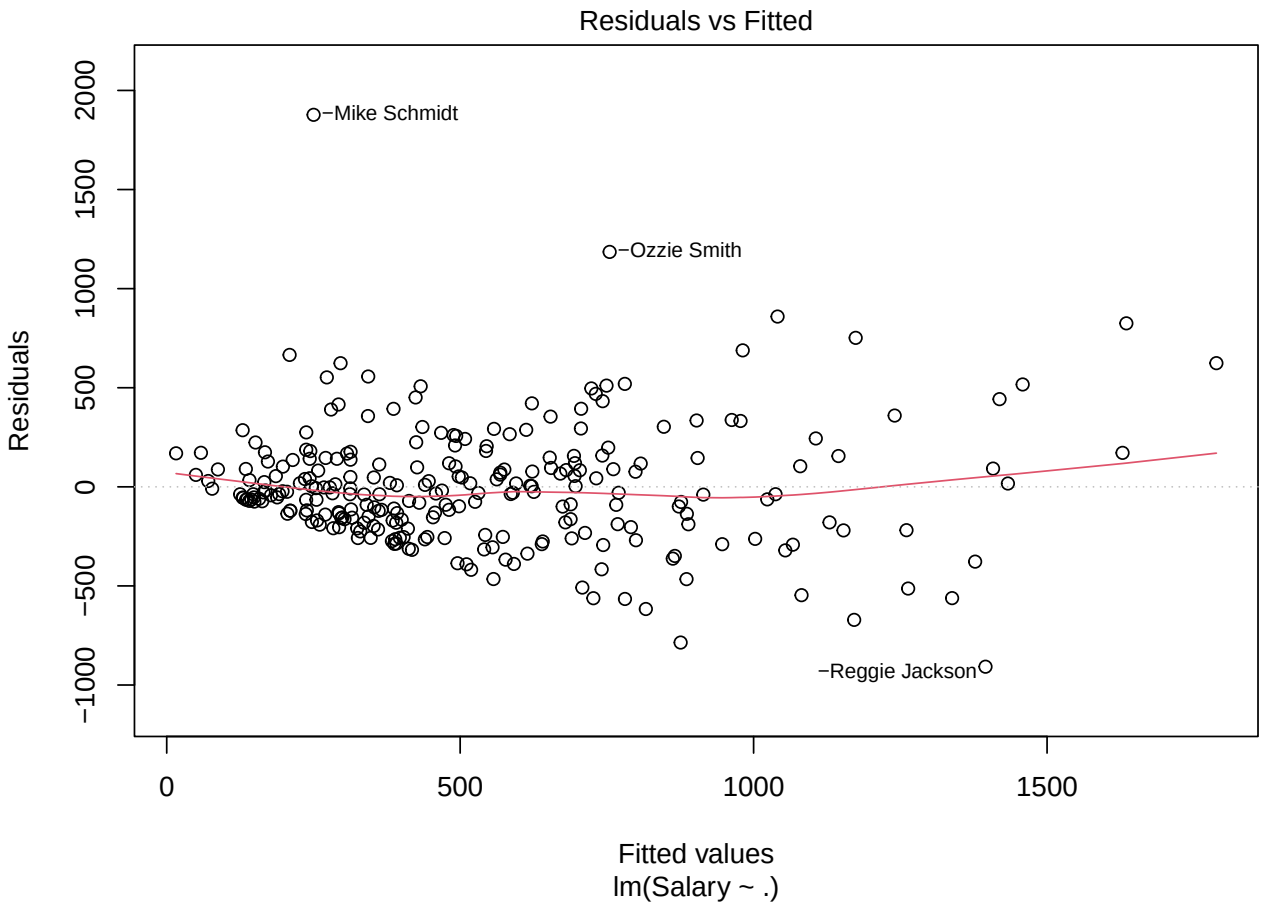
Let's start by fitting an OLS regression model to predict the median value **medv** of homes in the Boston area. There are a few libraries that contain functions to test for heteroskedasticity. We will use **{lmtest}**, which has a suite of tests for `lm()` objects.

```
library(ISLR) # To read the Boston housing market data
library(lmtest) # Contains bptest() and more

options(scipen = 4) # To minimize the use of scientific notation

Hitters <- na.omit(Hitters) # Need to remove several omitted values

lm.ols <- lm(Salary ~ ., data = Hitters)
summary(lm.ols) # Take a look
```

```
##
## Call:
## lm(formula = Salary ~ ., data = Hitters)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -907.62 -178.35  -31.11  139.09 1877.04
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  163.10359   90.77854   1.797 0.073622 .
## AtBat         -1.97987    0.63398  -3.123 0.002008 **
## Hits           7.50077    2.37753   3.155 0.001808 **
## HmRun          4.33088    6.20145   0.698 0.485616
## Runs          -2.37621    2.98076  -0.797 0.426122
## RBI           -1.04496    2.60088  -0.402 0.688204
## Walks          6.23129    1.82850   3.408 0.000766 ***
## Years         -3.48905   12.41219  -0.281 0.778874
## CAtBat        -0.17134    0.13524  -1.267 0.206380
## CHits          0.13399    0.67455   0.199 0.842713
## CHmRun        -0.17286    1.61724  -0.107 0.914967
## CRuns          1.45430    0.75046   1.938 0.053795 .
## CRBI           0.80771    0.69262   1.166 0.244691
## CWalks        -0.81157    0.32808  -2.474 0.014057 *
## LeagueN       62.59942   79.26140   0.790 0.430424
## DivisionW   -116.84925   40.36695  -2.895 0.004141 **
## PutOuts        0.28189    0.07744   3.640 0.000333 ***
## Assists        0.37107    0.22120   1.678 0.094723 .
## Errors        -3.36076    4.39163  -0.765 0.444857
## NewLeagueN   -24.76233   79.00263  -0.313 0.754218
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 315.6 on 243 degrees of freedom
## Multiple R-squared:  0.5461, Adjusted R-squared:  0.5106
## F-statistic: 15.39 on 19 and 243 DF,  p-value: < 2.2e-16
```

The first step is to inspect the residuals to see if the form a cloud of data around the predicted line, or whether the errors grow or shrink systematically.

```
plot(lm.ols, which = 1)
```



The residuals don't look totally even, but it is not entirely clear that there is heteroskedasticity. To be certain, we will perform a Breusch-Pagan test for Heteroscedasticity using the Breush-Pagan **bptest(){lmtest}** test. If the p-value is significant this means that the errors squared are highly correlated with the predicted values, which means that the errors increase or decrease systematically. That is, the errors are heteroskedsastic.

```
bptest(lm.ols, data = Hitters)
```

```
##
##  studentized Breusch-Pagan test
##
## data:  lm.ols
## BP = 32.196, df = 19, p-value = 0.0297
```

The p-values is highly significant, so we reject the null hypothesis of homoskedasticity and conclud the that the residuals are heteroskedastic. Heteroskedasticity does not affect bias of the coefficients. In fact, the

resulting coefficients will usually be similar in magnitude to OLS coefficients. But with heterorskedasticity, the OLS model is no longer BLUE. That is, it is not the most efficient (i.e., least variance) model and we can use other methods that will result in lower model variance. Thererfore, we correct for heteroskedasticity using the **Weighted Least Squares Method (WLS)**.

Fitting a WLS Model

There are many ways to compute the weighted sum of errors squared, but the most popular, and probably most effective weighting scheme is to use the actual errors of the OLS model to weight down the sum. However, with heteroskedasticity, the residuals will be all over the place, that is with high variance, so it is more effective and stable to use predicted residuals, rather than actual residuals for the weighted sum calculation. But since residuals can be positive or negative, it is better to use the magnitude of the residual, that is, the absolute value of the residual, i.e., `1 / fitted(abs(residuals)) ^ 2`. Let's do it slowly in steps:

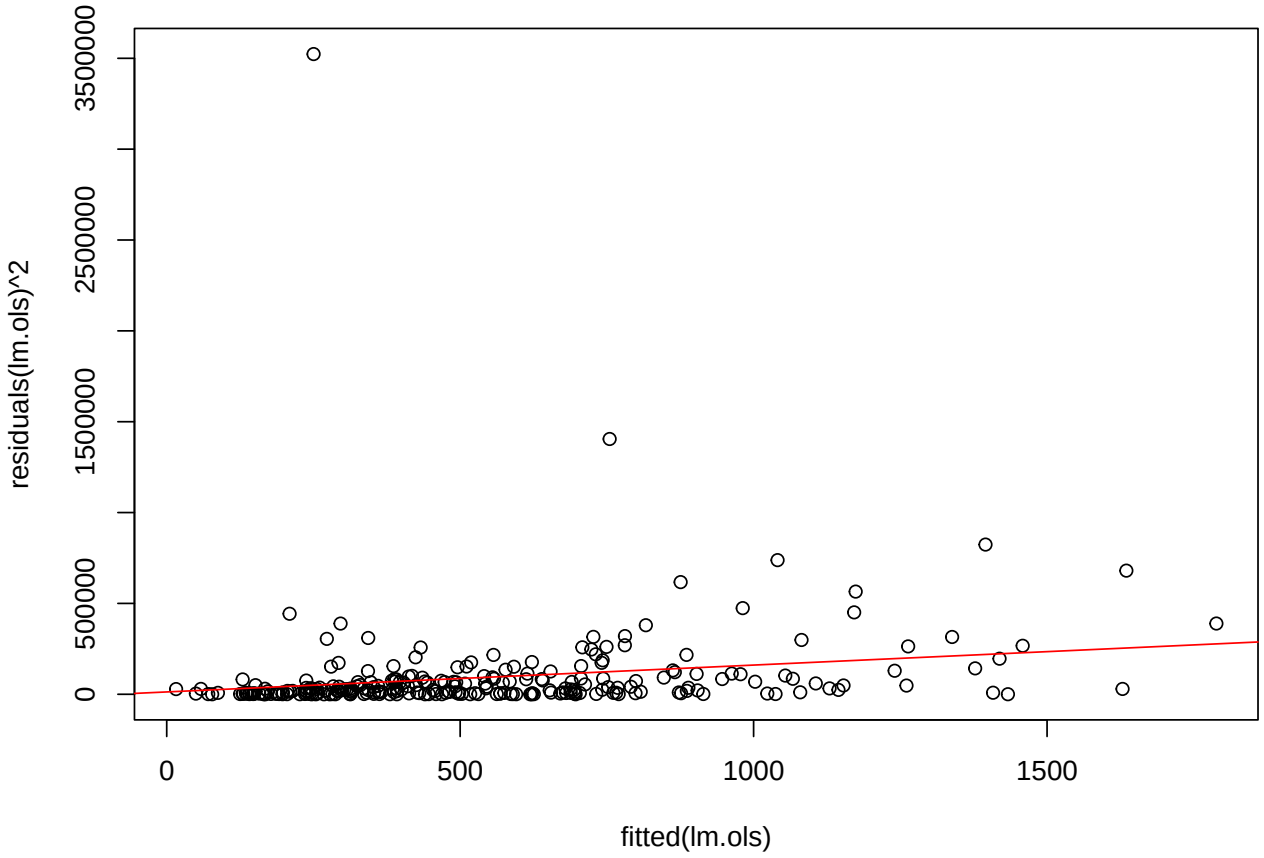
Step 1: Fit an OLS model

We did this above already

Step 2:

Fit another linear model to predict the squared errors with fitted (i.e., predicted) values of the OLS model above as predictors. I also provide a visualization for your reference.

```
lm.res2 <- lm(residuals(lm.ols) ^ 2 ~ fitted(lm.ols))

plot(fitted(lm.ols), residuals(lm.ols) ^ 2) # Take a look
abline(lm.res2, col = "red") # Draw regression line
```



We will be using as weights the values along the red regression line of the predicted squared errors, rather than the actual errors squared. We could use the actual error squares, but notice the wide variance of the errors in the plot above. You could still fit a WLS model with the actual error squares, but research has shown that using the predicted values instead yield more stable results. Naturally, we want to use the predicted error squares to **weight down** their influence in the SSE calculation, so we will use their inverse for that purpose:

```
wts <- 1 / fitted(lm.res2)
```

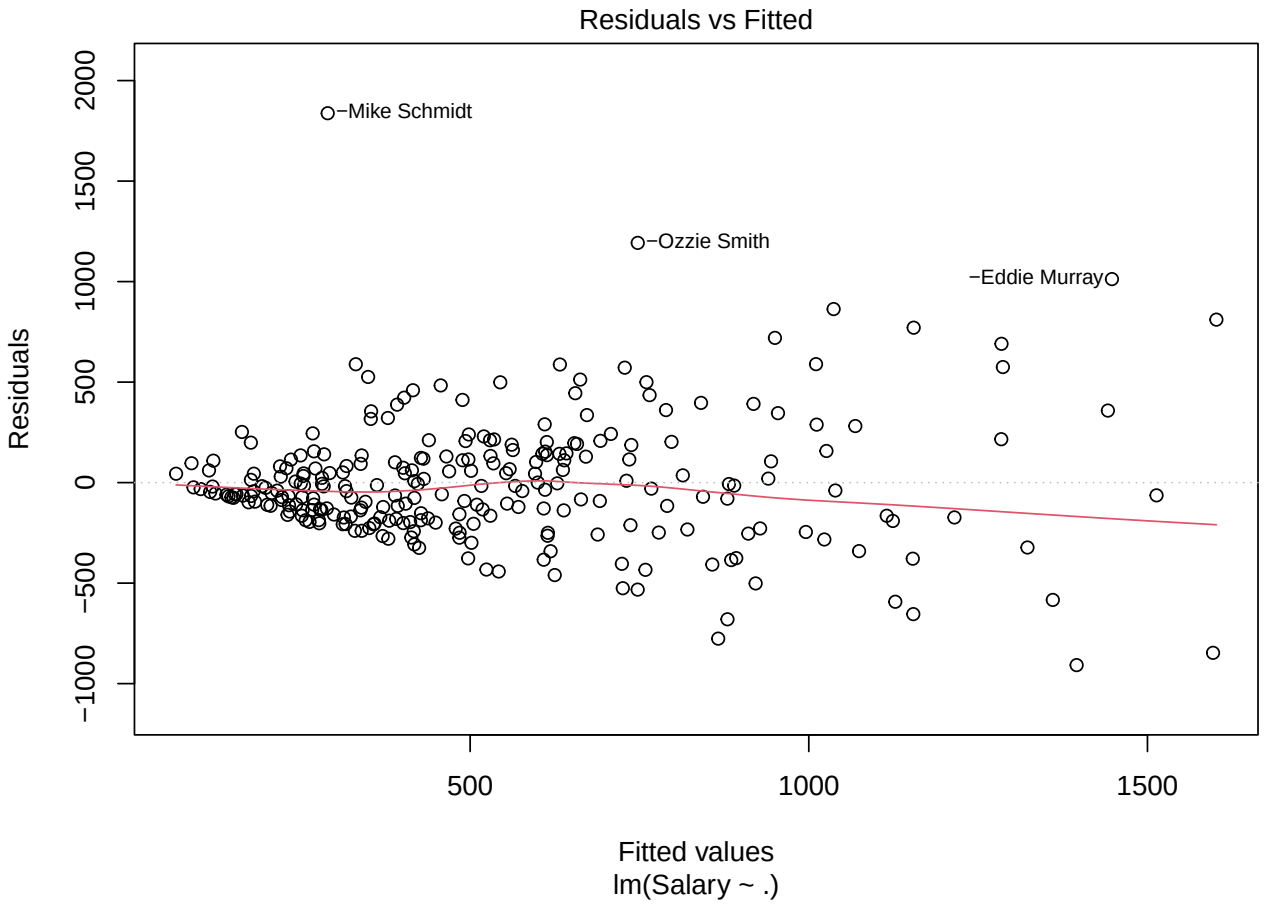
Then use this weight vector in the WLS model

```
lm.wls <- lm(Salary ~ .,
             data = Hitters,
             weights = wts)

summary(lm.wls)
```

```
##
## Call:
## lm(formula = Salary ~ ., data = Hitters, weights = wts)
##
```

```
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -2.0579 -0.6560 -0.1406  0.4243  8.2368
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 156.17644   73.43488   2.127 0.034450 *
## AtBat        -1.52287    0.56486  -2.696 0.007509 **
## Hits          5.39190    2.34770   2.297 0.022489 *
## HmRun         5.82599    5.75138   1.013 0.312082
## Runs         -1.36695    2.81557  -0.485 0.627761
## RBI          -1.34171    2.36811  -0.567 0.571526
## Walks         4.32732    1.69986   2.546 0.011525 *
## Years        -2.26035   10.64986  -0.212 0.832096
## CAtBat       -0.11967    0.14237  -0.841 0.401440
## CHits         0.37596    0.73073   0.515 0.607370
## CHmRun        0.95908    1.78117   0.538 0.590756
## CRuns         1.07441    0.77354   1.389 0.166119
## CRBI          0.06954    0.76022   0.091 0.927195
## CWalks       -0.70020    0.35192  -1.990 0.047751 *
## LeagueN      77.26497   68.38581   1.130 0.259658
## DivisionW   -84.35879   35.30112  -2.390 0.017626 *
## PutOuts       0.28081    0.08297   3.384 0.000831 ***
## Assists       0.52139    0.20664   2.523 0.012270 *
## Errors       -5.54329    3.91834  -1.415 0.158436
## NewLeagueN  -22.04302   68.91299  -0.320 0.749344
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.037 on 243 degrees of freedom
## Multiple R-squared:  0.4881, Adjusted R-squared:  0.4481
## F-statistic:  12.2 on 19 and 243 DF,  p-value: < 2.2e-16
```

```
plot(lm.wls, which = 1)
```




```
bptest(lm.wls, data = Hitters)
```

```
##
##  studentized Breusch-Pagan test
##
## data:  lm.wls
## BP = 0.00068335, df = 19, p-value = 1
```

Compared to the OLS regression model, the WLS model has a slightly lower R-squared, but it is still very high and significant. The WLS model generally yields a higher R-squared, but not always as in this case. Also notice that the coefficients have not changed much in termos of magniude, mainly because both models are unbiaseds, so they should yield similara predictor effects. But the significance of the predictors has changed more. Genearlly, the WLS will show smaller p-values, that is, more significant effects, although not always, as you will be able to see in the output. But the main thing is that the WLS model is more stable than the OLS model. That is, it has less variance. So, if we fit a model with multiple subsamples (as we do with machine learning), the WLS model will yield more consistent results than the OLS model.

A quick note about **Iterative Re-Weighted Least Squares (IRLS)**. The WLS method described above is pretty standard. But there are other methods and various weighting schemes you can use. Iterative Re-Weighted Least Squares (IRLS) is another popular method to compute **robust** estimators

when heteroscedasticity is present. The **{MASS}** library has a **Residual Linear Model** function `rlm()`, which does OLS with robust residuals. This method is similar to WLS, with one difference. Because the weights are calculated from residuals, but the final residuals are dependent on these weights, the IRLS solves this issue by itereating the model several times, calculating the residuals each subsequent time, re-weighting the model, and so on – i.e., running WLS multiple times in iterations until the residuals don't change any more (the model converges). Most of the time WLS will be sufficient. But if WLS does not solve the heteroskedasticity problem, IRLS will continue iterating further weitghted models until heteroskedasticity is resolved. You can fit an IRLS model this way:

```
rlm.fit <- rlm(Salary ~ ., data = Hitters)
summary(rlm.fit) # Take a look
```

3. Generalized Linear Method (GLM)

Overview

The `lm()` function fits a linear model using OLS. It assumes that the **residuals are normally distributed**. The outcome variable does not have to be normally distributed, as long as the residuals are normally distributed. However, when the outcome variable is not normally distibuted, the residuals are almost never normally distributed. Since you can inspect the normality of the outcome variable before you run your regression model, this is generally one of the first things to do.

If the outcome variable is not normally distributed, but you can ascertain the type of distribution it follows (e.g., poisson, logistic), then you can use the `glm()` function instead of the `lm()` to fit your model. But the `glm()` model requires that you specify the distribution family of the residuals.

The `glm()` function works just like the linear models function lm(), except that it can be used to fit models for a wider range of distributions, not just normal. It does not use the formulas to minimize the SSE, but it uses the **Maximum Likelihood Estimation (MLE)** method. Interesting, it can be shown mathematically that if you fit a `glm()` model using a **Gaussian** family distribution (omitted below because it is the default), you get the exact same results, except the `lm()` reports OLS fit statistics (e.g. R-Squared, ANOVA, etc.), whereas `glm()` reports **MLE** fit statistics (e.g., 2LL, Deviance, etc.). Take a look:

OLS Model

```
library(MASS) # Needed for the Boston data set

lm.fit <- lm(medv ~ lstat + age, data = Boston) # OLS
summary(lm.fit) # Check it out
```

```
##
## Call:
## lm(formula = medv ~ lstat + age, data = Boston)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.981  -3.978  -1.283   1.968  23.158
##
## Coefficients:
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 33.22276    0.73085  45.458  < 2e-16 ***
## lstat       -1.03207    0.04819 -21.416  < 2e-16 ***
## age          0.03454    0.01223   2.826  0.00491 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.173 on 503 degrees of freedom
## Multiple R-squared:  0.5513, Adjusted R-squared:  0.5495
## F-statistic:   309 on 2 and 503 DF,  p-value: < 2.2e-16
```

GLM Model

```
glm.fit = glm(medv ~ lstat + age,
              data = Boston) # GLM

summary(glm.fit)
```

```
##
## Call:
## glm(formula = medv ~ lstat + age, data = Boston)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -15.981   -3.978   -1.283    1.968   23.158
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 33.22276    0.73085  45.458  < 2e-16 ***
## lstat       -1.03207    0.04819 -21.416  < 2e-16 ***
## age          0.03454    0.01223   2.826  0.00491 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 38.10761)
##
##     Null deviance: 42716  on 505  degrees of freedom
## Residual deviance: 19168  on 503  degrees of freedom
## AIC: 3283
##
## Number of Fisher Scoring iterations: 2
```

Notice that You can derive the R-square of a GLM model by computing how much the deviance is improving as a percentage of the null model's deviance:

```
res.dev <- glm.fit$deviance
null.dev <- glm.fit$null.deviance

# Proportion of deviance explained by the model
```

```r
dev.Rsq <- (null.dev - res.dev) / null.dev

print(dev.Rsq, digits = 4)
```

```
## [1] 0.5513
```

Binomial Logistic Regression

For example, it is used to fit logistic regressions. For example, let's read the South Africa heart disease data set from the authors' data site:

```r
heart <- read.table("Heart.csv",
                    sep = ",",
                    head = T)
```

Notice that we deleted the first column of the dataset with `heart[,-1]` because it contains row names, which are not data variables per se, but just labels. I could have used the attribute `row.names = 1` in the `read.table()` function above, but I didn't because this data set contains duplicate row.names, which would give an error since you cannot have duplicate row or column names in a dataset. But deleting the column as I did above does the job.

The dataset documentation is available at https://web.stanford.edu/~hastie/ElemStatLearn/datasets/SAheart.info.txt

In the example above, I used a Heart.csv I had already downloaded from the ISLR website, but you could read the file directly from the web with this command (commented out for now):

```r
heart <- read.table("https://web.stanford.edu/~hastie/ElemStatLearn/datasets/SAheart.data",
                    sep = ",",
                    head = T,
                    row.names = 1)
```

You can view other datasets in the ISLR book's website: https://web.stanford.edu/~hastie/ElemStatLearn/datasets/

Now let's fit a binomial Logistic regression model:

```r
heart.fit <- glm(chd ~ .,
                 data = heart,
                 family = binomial(link = "logit"))
```

chd is 1 if patient has coronary heart disease, 0 if not

The family **distribution** of the outcome variable is `family = binomial`

The **link** function is what we use when we want to transform the outcome variable. When we fitted the OLS model with `glm()` above we used `link = "identity"`, which is the default and it means **do nothing**, so we omitted it. But to fit the binomial Logistic model we need to transform the outcome from binary to log-likelihoods using the Logistic function or `link = "logit"`. Let's take a look at the results:

```
summary(heart.fit)
```

```
##
## Call:
## glm(formula = chd ~ ., family = binomial(link = "logit"), data = heart)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.7781  -0.8213  -0.4387   0.8889   2.5435
##
## Coefficients:
##                   Estimate Std. Error z value   Pr(>|z|)
## (Intercept)    -6.1507209  1.3082600  -4.701 0.00000258 ***
## sbp             0.0065040  0.0057304   1.135   0.256374
## tobacco         0.0793764  0.0266028   2.984   0.002847 **
## ldl             0.1739239  0.0596617   2.915   0.003555 **
## adiposity       0.0185866  0.0292894   0.635   0.525700
## famhistPresent  0.9253704  0.2278940   4.061 0.00004896 ***
## typea           0.0395950  0.0123202   3.214   0.001310 **
## obesity        -0.0629099  0.0442477  -1.422   0.155095
## alcohol         0.0001217  0.0044832   0.027   0.978350
## age             0.0452253  0.0121298   3.728   0.000193 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 596.11  on 461  degrees of freedom
## Residual deviance: 472.14  on 452  degrees of freedom
## AIC: 492.14
##
## Number of Fisher Scoring iterations: 5
```

Interpretation

The coefficients represent the "change" in log-odds (i.e., logit) that the outcome variable is $= \mathbf{1}$. A positive coefficients indicates how much the log-odds (and the odds) of Y being 1 increase, when the variable goes up by 1. A negative coefficient indicates how much the log-odds decrease.

Again, it is always useful to convert the log-odds coefficients to odds, because log-odds are very hard to explain to a manager or business client. Let's convert the log-odds coefficients to odds and insert them in the summary output.

```
lm.sum <- summary(heart.fit) # Create and name the summary() object.

# Summary with odds effects. This output is a replica of the summary() output, but with the Odds

null.dev <- paste("Null Deviance",
                  round(lm.sum$null.deviance, digits = 2),
                  "on", lm.sum$df.null, "degrees of freedom")
```

```
res.dev <- paste("Residual Deviance",
                 round(lm.sum$deviance, digits = 2),
                 "on", lm.sum$df.residual, "degrees of freedom")

aic <- paste("AIC", round(lm.sum$aic, digits = 2))

round(
  cbind("Log Odds" = lm.sum$coefficients[,1],
        "Odds" = exp(lm.sum$coefficients[,1]),
         lm.sum$coefficients[,2:4]),
        digits = 4)
```

```
##                Log Odds   Odds Std. Error z value Pr(>|z|)
## (Intercept)     -6.1507 0.0021     1.3083 -4.7015   0.0000
## sbp              0.0065 1.0065     0.0057  1.1350   0.2564
## tobacco          0.0794 1.0826     0.0266  2.9838   0.0028
## ldl              0.1739 1.1900     0.0597  2.9152   0.0036
## adiposity        0.0186 1.0188     0.0293  0.6346   0.5257
## famhistPresent   0.9254 2.5228     0.2279  4.0605   0.0000
## typea            0.0396 1.0404     0.0123  3.2138   0.0013
## obesity         -0.0629 0.9390     0.0442 -1.4218   0.1551
## alcohol          0.0001 1.0001     0.0045  0.0271   0.9784
## age              0.0452 1.0463     0.0121  3.7285   0.0002
```

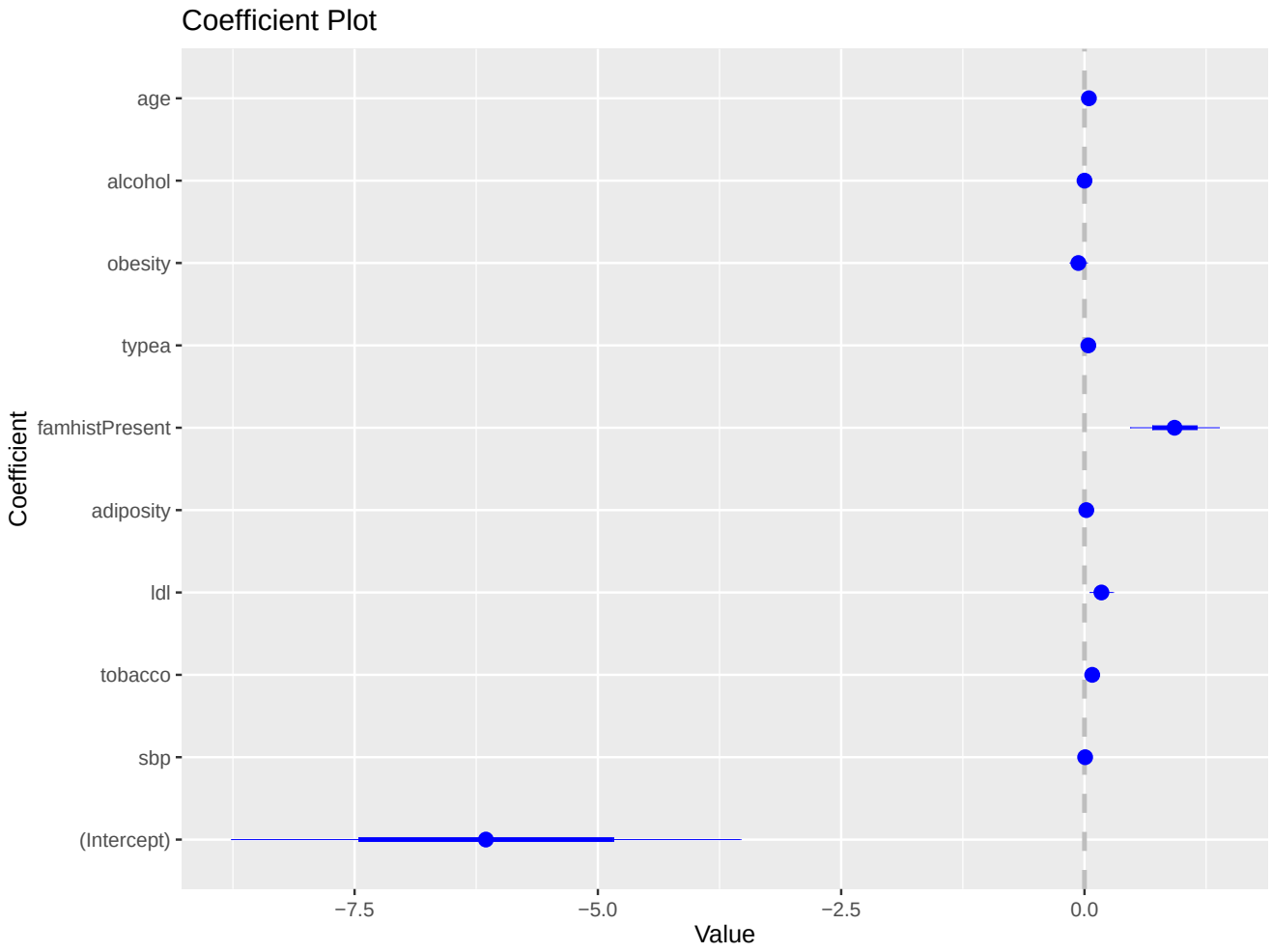
```
cat("\n")
```

```
cat(paste(null.dev, res.dev, aic, sep = "\n"))
```

```
## Null Deviance 596.11 on 461 degrees of freedom
## Residual Deviance 472.14 on 452 degrees of freedom
## AIC 492.14
```

You can inspect the coefficients visually too:

```
require(coefplot)
coefplot(heart.fit)
```



I will interpret a positive and a negative coefficient, as an example.

Effect of **tobacco**. This variable measures the cumulative consumption of tobacco in kilograms. Holding everything else constant, on average, an increase in 1 Kg. of cumulative consumption of tobacco increases the **log-odds** of developing coronary heart disease by 0.0794 and this effect is significant. Also, the **odds** of developing coronary heart disease increases by a **factor of 1.082**. Notice the use of the term **factor of**. This is a very important aspect of the interpretation. Log-odds are additive because they are in the linear model. So the log-odds effect reflect unit changes, just like any OLS coefficient. But when we convert log-odds to odds, we are using the `exp()` function, so the odds are now multiplicative, so their effect is a multiplication of the odds. So, odds > 1 than 1 increase the odds (just like positive log-odds) and odds < 1 reduce the odds (just like negative log-odds)

4. Decision Trees

Decision trees are non-parametric models that simply split the data into regions, called tree leaves, based on a predictor and the corresponding value that does the best job at reducing the deviance in the outcome variable predictions, and continue to split the leaves into further sub-regions recursively. A **regression tree** is the non-parametric equivalent to a linear regression model in which the prediction is a quantitative value. A **classification tree** is the non-parametric equivalent to a Logistic regression, and other classification methods, in which the prediction is the correct classification of the outcome into a category. When splitting a tree region into further sub-regions, a regression tree aims to reduce the mean

squared error within regions, whereas a classification tree aims to reduce the misclassification of outcomes into categories. In either case, the tree splits the data recursively based on predictors and threshold values that reduce deviance the most.

Regression Trees

Say, for example, that you have 2 predictors. A regression tree will find which of the two predictors does a better job at separating the outcome values into two regions, and finds the value of that predictor in which the mean squared error MSE within each region is minimized. The regression tree model assigns a predicted value for each region equal to the mean of the outcome value withing that region. It then continues recursively, taking each region and further splitting them into more regions or leaves.

The criteria for evaluating which variable to use for the split and at which value is based on minimizing the MSE. The process continues with further subdivisions of the data. Each subdividision portion is called a "leaf" and the splitting point is called a "node". In principle, one could continue the data splitting until each data point is a leaf, but this would overfit the data.

The next example fits a regression tree to predict median values of houses in Boston suburbs.We first need to load the **{tree}** library.

```
library(MASS) # Contains the Boston data set
library(tree) # Needed to fit decision trees

tree.boston <- tree(medv ~ ., Boston)
```

Let's view the summary results and splitting nodes and regions:

```
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston)
## Variables actually used in tree construction:
## [1] "rm"      "lstat"   "dis"     "crim"    "ptratio"
## Number of terminal nodes:  9
## Residual mean deviance:  13.55 = 6734 / 497
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -17.68000  -2.23000   0.07026   0.00000   2.22100  16.50000
```

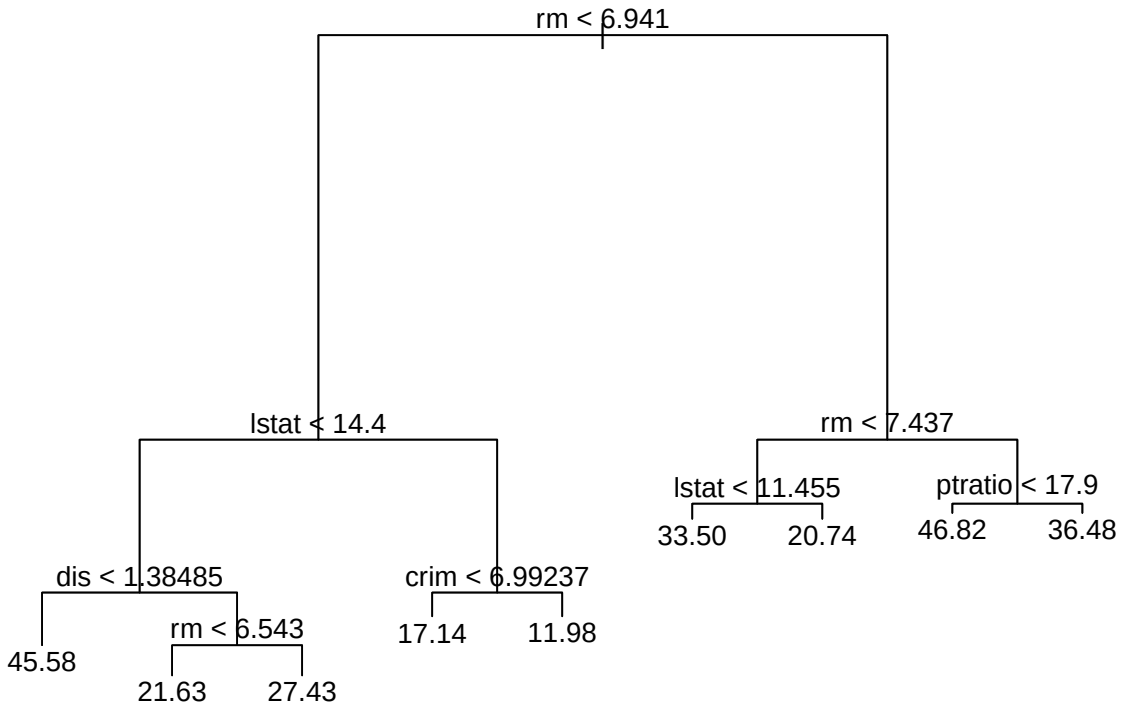
```
tree.boston
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 506 42720.0 22.53
##    2) rm < 6.941 430 17320.0 19.93
##      4) lstat < 14.4 255  6632.0 23.35
##        8) dis < 1.38485 5   390.7 45.58 *
```

```
##         9) dis > 1.38485 250  3721.0 22.91
##          18) rm < 6.543 195  1636.0 21.63 *
##          19) rm > 6.543 55   643.2 27.43 *
##       5) lstat > 14.4 175  3373.0 14.96
##        10) crim < 6.99237 101  1151.0 17.14 *
##        11) crim > 6.99237 74  1086.0 11.98 *
##     3) rm > 6.941 76  6059.0 37.24
##       6) rm < 7.437 46  1900.0 32.11
##        12) lstat < 11.455 41   844.2 33.50 *
##        13) lstat > 11.455 5   329.8 20.74 *
##       7) rm > 7.437 30  1099.0 45.10
##        14) ptratio < 17.9 25   340.7 46.82 *
##        15) ptratio > 17.9 5   312.7 36.48 *
```

Now lets visualize the tree:

```
plot(tree.boston) # Plot the tree
text(tree.boston, pretty = 0) # Let's make it pretty and add labels
```



Managing the Tree Size

Trees can grow quite large with large datasets. It is very important to figure out the optimal tree size to **grow** (forward) or **prune** (backward). I will discuss later how to find the optimal size of a tree using

cross-validation MSE testing, which will yield the tree with highest predictive accuarcy. But, for now, let's discuss how to control the size of the tree using the **mindev** attribute.

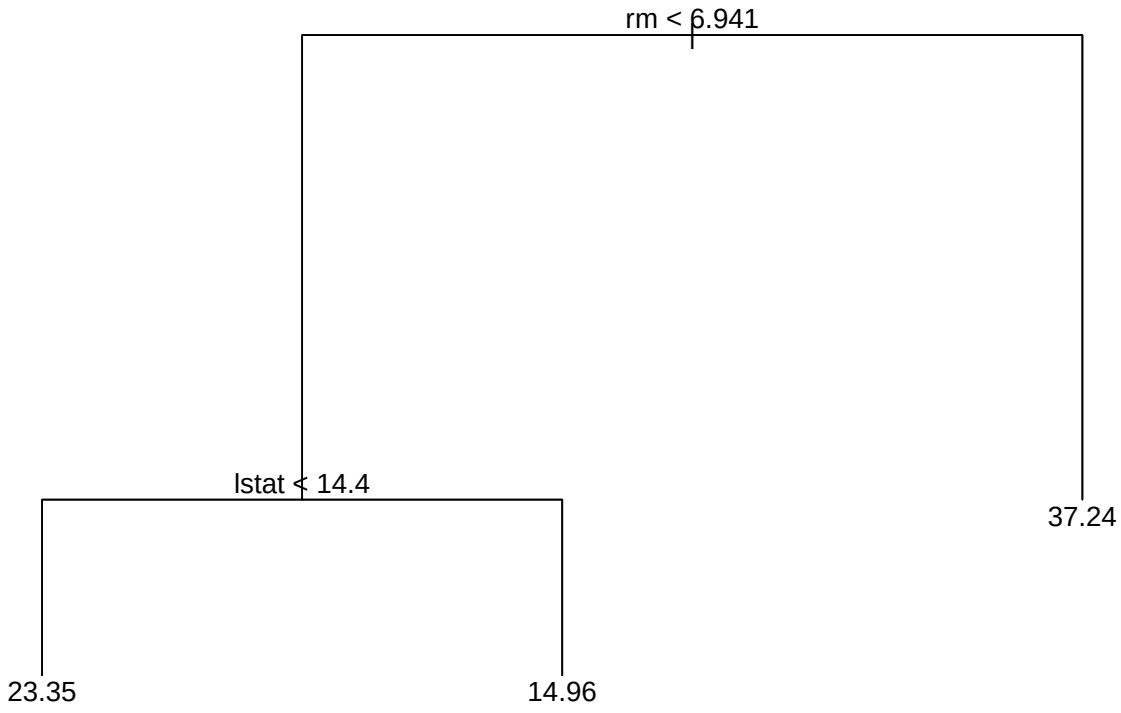
Before you start splitting the tree you have the root of the tree, which is equivalent to the **null** model with no predictors. At the root, the prediction for all data points is equal to the mean of the outcome variable for all the data points. Of course, we can do much better than this. After the first split, we have 2 leaves. We then predict all observations within a leaf to be equal to the mean of the outcome variable for all data points within that leaf. The MSE is calculated by squaring the differences between the actual outcome values and the outcome mean within the corresponding leaf.

As you would expect, the MSE gets reduced every time we split the tree into further regions. But if we keep splitting all the way till the end, we will end up with **terminal nodes**, in which we will have one leaf for each data point. The tree will be perfectly accurate because the predicted value will be identical to the actual value. Naturally, this accuracy is artificial because the tree is over-fitting and will not necessarily predict well when new data arrives. Cross-validation testing can help us find the optimal tree size in which predictions are most accurate with new data.

For now, we will simply use a parameter called **mindev** to control the size of the tree. By default the `tree()` function stops growing the tree at `mindev=0.01`. That is, when the MSE of the tree is 0.01 (or 1%) of the MSE at the root, the tree stops growing. We can **reduce** the **mindev** to let the tree grow **larger** (i.e., it will take longer to reduce the MSE to that level) or **increase** it to grow the tree smaller (i.e., it will take less time to reduce the MSE to that level).

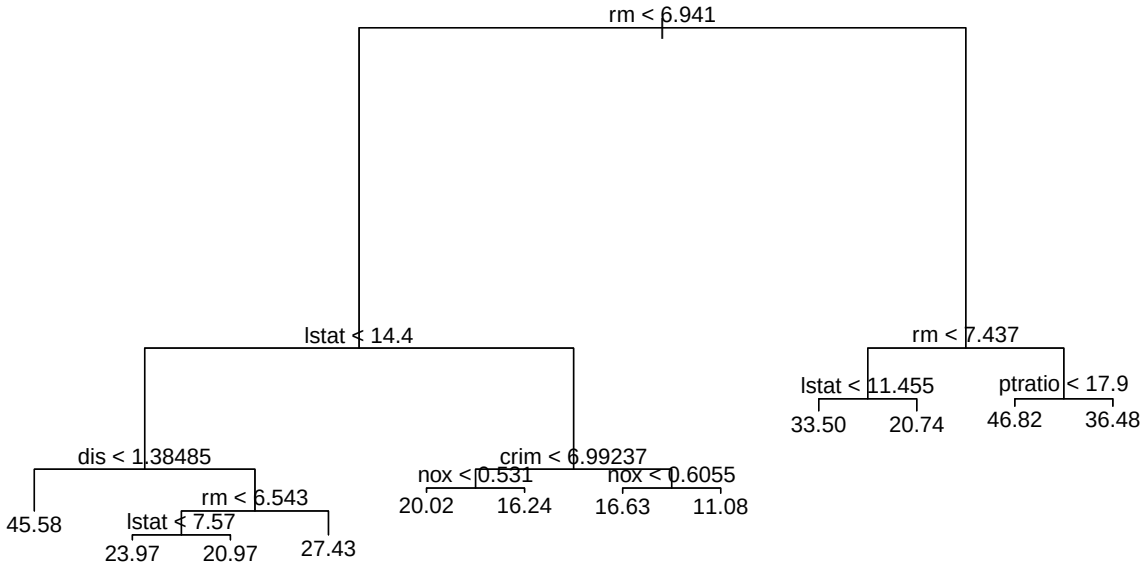
Small Tree Example, mindev $>$ 0.01

```
tree.boston.small <- tree(medv ~ ., Boston,
                          mindev = 0.1)

plot(tree.boston.small)
text(tree.boston.small, pretty = 0)
```



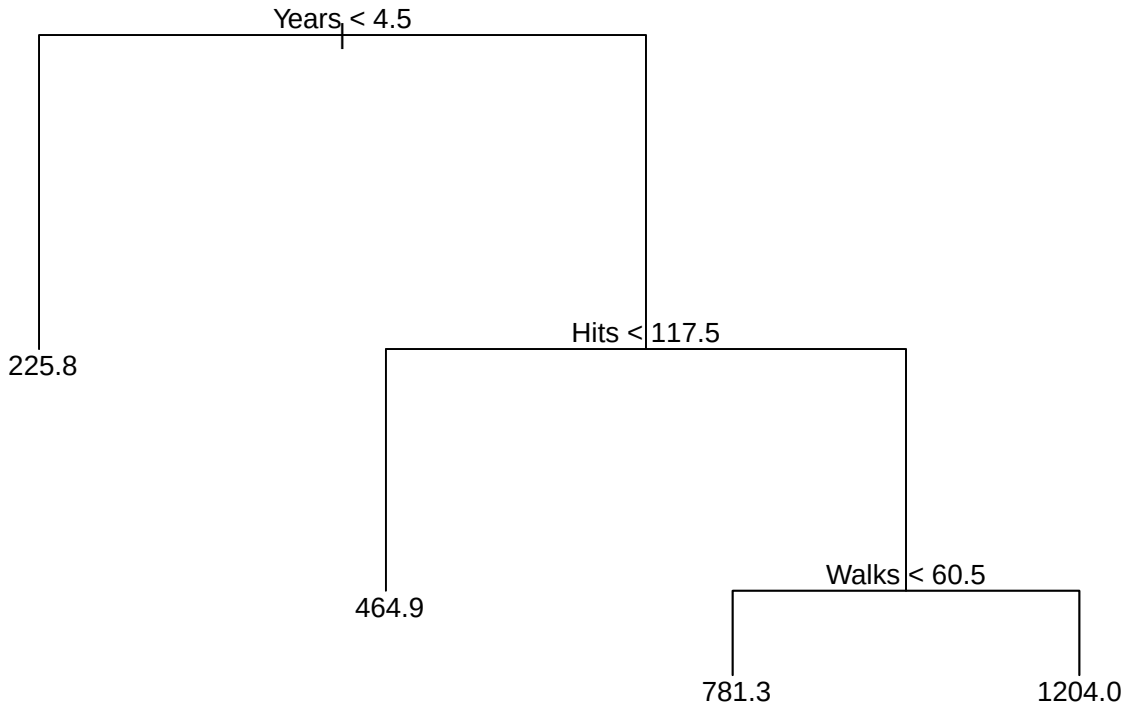
** Large Tree Example, mindev < 0.01**

```
tree.boston.large <- tree(medv ~ ., Boston,
                          mindev = 0.005)

plot(tree.boston.large)
text(tree.boston.large, pretty = 0)
```



Example with Baseball Player Salary Data

```
library(ISLR)

tree.sal <- tree(Salary ~
                 Years + Hits + RBI + PutOuts + Runs + Walks,
                 Hitters,
                 mindev = 0.05)

plot(tree.sal)
text(tree.sal, pretty = 0)
```



Classification Trees (Binomial)

Classification trees work just like regression trees, but the response variable is binary (i.e., a classification). While decision are generally not as precise as logistic regression models, and despite the fact that they are not very useful for interpretation there is an abundance of sophisticated decision tree methods (e.g., Bootstrap Aggregation, Random Forests, Boosting, etc.), which can be quite accurate for prediction.

Let's illustrate a classification tree with the **Carseats** dataset in the **{ISLR}** package. The dataset contains simulated data of child car seat sales in 400 stores. Since the outcome variable **Sales** is quantitative, let's create a binary named **HighSales** with a value of **No** if `Sales < 8000` units and **Yes** otherwise (note that Sales is in thousands of units). We then add the HighSales variabel vector to the Carseats data frame.

```
library(ISLR)
library(tree)

HighSales <- as.factor(ifelse(Carseats$Sales <= 8,
                              "No", "Yes"))

Carseats <- data.frame(Carseats, HighSales)
```

We can now fit the classification tree. But note that we need to exclude the variable **Sales** from the model because otherwise it would be redundant with HighSales. Also, I set `mindev` arbitrarily to get a good visual on the tree graph.

```
tree.carseats <- tree(HighSales ~ . -Sales,
                      Carseats,
                      mindev = 0.02)

summary(tree.carseats)
```

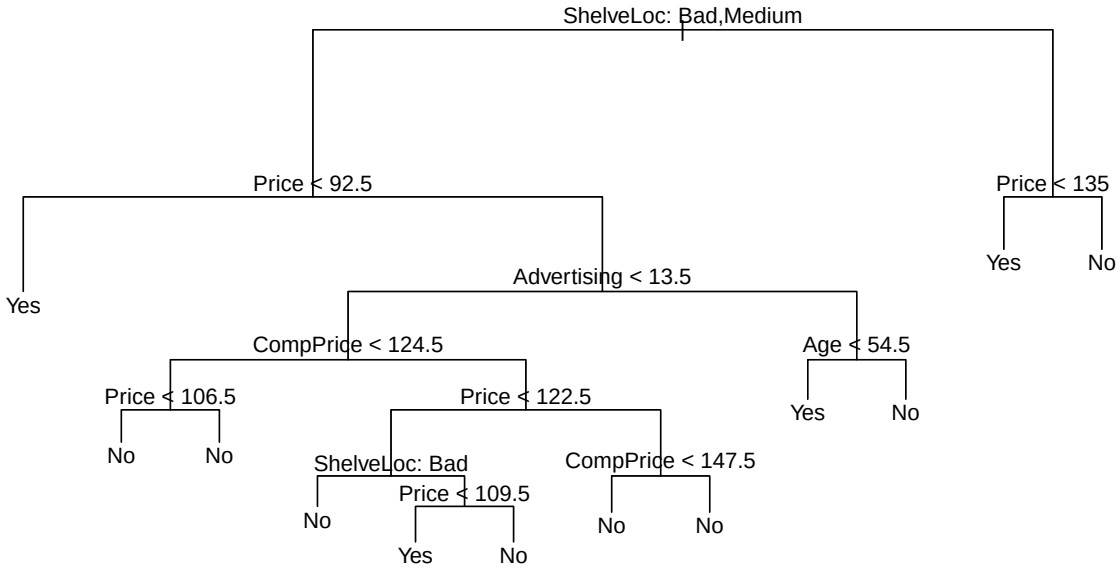
```
##
## Classification tree:
## tree(formula = HighSales ~ . - Sales, data = Carseats, mindev = 0.02)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Advertising" "CompPrice"   "Age"
## Number of terminal nodes:  12
## Residual mean deviance:  0.7673 = 297.7 / 388
## Misclassification error rate: 0.1625 = 65 / 400
```

Notes:

- The residual mean deviance is based on the log likelyhood function (2LL), the smaller the better. The residual mean deviance is the deviance divided by the number of observations minus terminal nodes. By itself, deviance or residual mean deviance are not very meaningful fit statistic, but are excellent to compare various tree modesl -> the model with the smallest deviance (between actual and predicted values is better)
- The misclassification error is more meaningful. It is the total number of misclassified observations, relative to the total number of observations.

Now let's analyze the tree visually

```
plot(tree.carseats) # Display tree
text(tree.carseats, pretty = 0) # Display data labels and make it pretty
```




```
tree.carseats # Display the data for every leaf
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 ) *
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 ) *
##        11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 ) *
##          23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 ) *
##     3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
```

```
##       6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 ) *
##       7) Price > 135 17  22.070 No ( 0.64706 0.35294 ) *
```