# 10 Predictive Modeling (10)

## 10.1 Introduction to Predictive Modeling and Machine Learning

- ***Machine learning*** is suite of methods for data-based **prediction**, **classification**, and **grouping of observations**.
- There are two varieties of **machine learning**:

  1. ***Supervised learning*** – The goal is **predicting** values of a *response variable* from the values of *explanatory variables*.

     When the *response variable* is **categorical**, prediction is called ***classification***.

     For supervised learning, the data must contain values of **explanatory variables** *and* a **response variable**.

     Regression and logistic regression are examples of supervised learning.

  2. ***Unsupervised learning*** (covered in Class Notes 7) – The goal is **identifying groups** (i.e. *clusters*) of individuals based *only* on the values of *explanatory variables*.

     For unsupervised learning, the data *only* needs to contain values of **explanatory variables**, but *not* a **response variable**.

     Cluster analysis is an example of unsupervised learning.

## 10.2 Logistic Regression: A Simple Classification Model

**Classification Using Logistic Regression**

- We'll introduce **classification** concepts using **logistic regression**.
- Recall that **classification** (a type of **supervised learning**) can be viewed as **predicting** a **categorical** response variable from the values of **explanatory variables**.

  When the response variable is **dichotomous** (takes only *two* categorical values, coded as **0** or **1**), we can use **logistic regression** for **classification**.

  An individual is **classified** as a **0** or **1** according to the following rule:

$$\text{Classification} = \begin{cases} \mathbf{1} & \text{if} \quad p(X) \geq 0.5 \\ \mathbf{0} & \text{if} \quad p(X) < 0.5 \end{cases}$$

  where $\boldsymbol{p(X)}$ is the (estimated) **probability** of an individual being a **1** based on their $\boldsymbol{X}$ value.

- For example, consider (again) the `dues` data set (Class Notes 5):

```
head(dues)

##   NotRenew DuesIncr
## 1        0       25
## 2        0       27
## 3        0       30
## 4        0       30
## 5        0       31
## 6        0       32
```

  We might want to **classify** an individual as a membership **non-renewer** or **renewer**, i.e. **predict** whether they **would** or **wouldn't renew** their membership.

  Here's the **fitted logistic regression model** (Fig. 1):

```
## Not-renew vs dues increase:
g1 <- ggplot(data = dues, mapping = aes(x = DuesIncr, y = NotRenew)) +
  geom_point() +
  labs(title = "Not Renew vs Dues Increase", x = "Dues Increase", y = "Not Renew") +
  geom_smooth(method = "glm",
              method.args = list(family = "binomial"),
              se = FALSE) +
  scale_y_continuous(breaks = seq(-0.25, 1.25, 0.25),
                     labels = seq(-0.25, 1.25, 0.25),
                     limits = c(-0.25, 1.25)) +
  geom_hline(mapping = aes(yintercept = 0.5),
             linetype = 3)
g1
```

```
## Not-renew vs dues increase with classification split-point:
g2 <- g1 + geom_vline(xintercept = 39.54,
                      color = "dodgerblue", linetype = 2)
g2
```

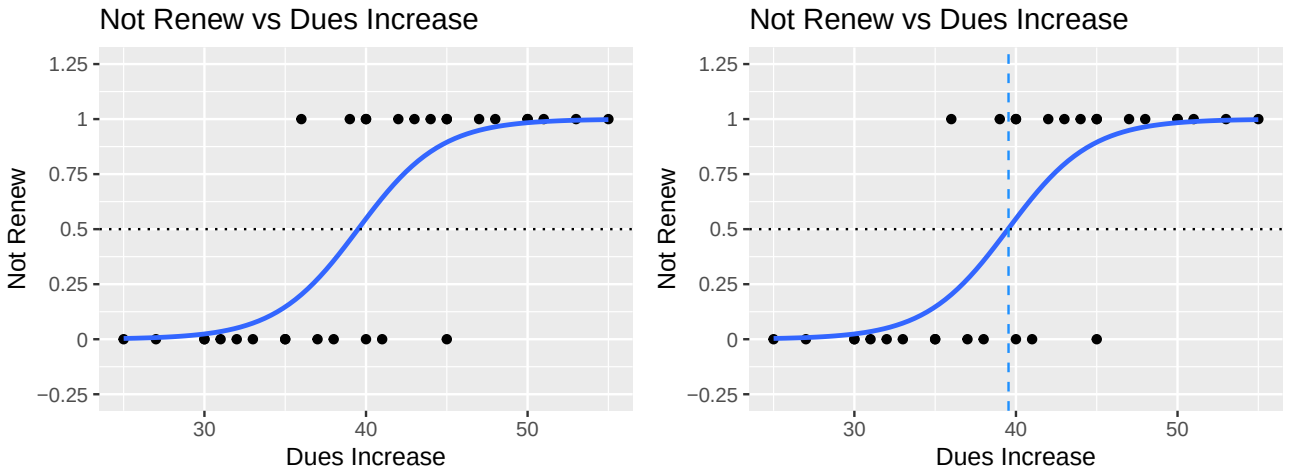



Figure 1

The curves in Fig. 1 give the (estimated) **probability** of **not renewing** for any given value of the **dues increase** $\boldsymbol{X}$.

Recall (Class Notes 5) we carried out the **logistic regression analysis** by typing:

```
my.logreg <- glm(NotRenew ~ DuesIncr, data = dues, family = "binomial")

summary(my.logreg)
```

and the **equation** of the *fitted logistic regression model* is:

$$p(X) = \frac{e^{-15.42+0.39X}}{1+e^{-15.42+0.39X}}. \tag{1}$$

This is the curve graphed in Fig. 1.

If we plug **$42** into the equation for the **dues increase** $X$, we get the (estimated) **probability** that an individual **won't renew** their membership:

$$p(42) = \frac{e^{-15.42+0.39(42)}}{1+e^{-15.42+0.39(42)}} = 0.72,$$

i.e.

```
exp(-15.42 + 0.39 * 42)/(1 + exp(-15.42 + 0.39 * 42))

## [1] 0.7231218
```

There's a **72% chance** that an individual whose **dues increase** is **$42 won't renew** their membership.

We could also get this value using `predict()`:

```
newDues <- data.frame(DuesIncr = 42)

predict(my.logreg, newDues, type = "response")

##         1
## 0.7254854
```

Because there's a **0.72 probability** that an individual whose **dues increase** is **$42 won't renew** their membership, we **predict** (i.e. **classify**) that individual as being a **non-renewer**.

By setting $\boldsymbol{p(X) = 0.5}$ in the **equation** (1) and solving for $X$, it can be shown that $\boldsymbol{p(X) \geq 0.5}$ above the value $\boldsymbol{X = \$39.54}$, and $\boldsymbol{p(X) < 0.5}$ below it. This is the vertical dashed line in the right plot of Fig. 1.

- In practice, $\boldsymbol{p(X)}$ could be a function of **multiple** explanatory variables, i.e. $\boldsymbol{p(X_1, X_2, \ldots, X_p)}$.
- Suppose, for example, a **pet** is **9** inches tall and weighs **10** pounds. Can we **predict** whether it's a **dog** or **cat**?

To make it easier, suppose also we have data on **weights** (pounds) and **heights** (inches) of **19 other pets**:

```
type <- c("dog", "dog", "cat", "dog", "cat", "dog", "cat", "dog", "cat",
          "dog", "cat", "dog", "dog", "cat", "dog", "cat", "dog", "cat", "dog")
wt <- c(8, 17, 8, 18, 7, 22, 6, 16, 7, 20, 10, 15, 14, 11, 13, 13, 15, 17, 10)
ht <- c(7.5, 10, 8, 15, 7, 15, 7, 13, 11, 16, 7, 10.5, 9, 9.5, 9, 8, 9, 8, 12)
pets <- data.frame(Type = type, Ht = ht, Wt = wt)
```

```
pets

##    Type   Ht Wt
## 1   dog  7.5  8
## 2   dog 10.0 17
## 3   cat  8.0  8
## 4   dog 15.0 18
## 5   cat  7.0  7
## 6   dog 15.0 22
## 7   cat  7.0  6
## 8   dog 13.0 16
## 9   cat 11.0  7
## 10  dog 16.0 20
## 11  cat  7.0 10
## 12  dog 10.5 15
## 13  dog  9.0 14
## 14  cat  9.5 11
## 15  dog  9.0 13
## 16  cat  8.0 13
## 17  dog  9.0 15
## 18  cat  8.0 17
## 19  dog 12.0 10
```

and we want to use the data to **predict** the **type** of the **9-inch**, **10-pound** animal. In other words, we want to **classify** the new animal as either a **dog** or a **cat**.

Note there are **11 dogs** and **8 cats** in the `pets` data set.

We carry out the **logistic regression analysis**, with **Type** (**0** or **1**) as the response ($Y$) and **Ht** and **Wt** as the explanatory variables ($X_1$ and $X_2$), using the `pets` data set by typing:

```
library(dplyr)       # For mutate() and case_when()

# Create a 0 or 1 Type variable for logistic regression:
pets <- mutate(pets, Type01 = case_when(Type == "cat" ~ 0,
                                        Type == "dog" ~ 1))

my.logreg <- glm(Type01 ~ Ht + Wt, data = pets, family = "binomial")

summary(my.logreg)

##
## Call:
## glm(formula = Type01 ~ Ht + Wt, family = "binomial", data = pets)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4684  -0.5364   0.0448   0.5805   2.1218
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -9.5477     4.9398  -1.933   0.0533 .
## Ht            0.6969     0.4557   1.529   0.1262
## Wt            0.2726     0.1928   1.414   0.1573
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 25.864  on 18  degrees of freedom
## Residual deviance: 15.083  on 16  degrees of freedom
## AIC: 21.083
##
## Number of Fisher Scoring iterations: 6
```

- To understand the **fitted logistic regression model**, when there are **two explanatory variables**, consider the function $\boldsymbol{p(X_1, X_2)}$ defined as

$$p(X_1, X_2) \;=\; P(Y = 1 \text{ when the values of the explanatory variables are } X_1 \text{ and } X_2)$$

The **fitted logistic regression model** has the form

$$p(X_1, X_2) \;=\; \frac{e^{\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2}}\,.$$

From the output of `summary(my.logreg)` above, the **coefficients** of the **equation** of the *fitted logistic regression model* are in the `Estimate` column. Thus

$$\hat{\boldsymbol{\beta}}_{\mathbf{0}} \mathbf{= -9.55}, \quad \hat{\boldsymbol{\beta}}_{\mathbf{1}} \mathbf{= 0.70}, \quad \text{and} \quad \hat{\boldsymbol{\beta}}_{\mathbf{2}} \mathbf{= 0.27},$$

so the **equation** is:

$$p(X_1, X_2) \;=\; \frac{e^{-9.55 + 0.70 X_1 + 0.27 X_2}}{1 + e^{-9.55 + 0.70 X_1 + 0.27 X_2}}\,.$$

The (estimated) **probability** that the **9-inch**, **10-pound** animal is a **dog** is obtained by plugging **9** in for the **Ht** $X_1$ and **10** in for the **Wt** $X_2$:

$$p(9, 10) \;=\; \frac{e^{-9.55 + 0.70\,(9) + 0.27\,(10)}}{1 + e^{-9.55 + 0.70\,(9) + 0.27\,(10)}} \;=\; 0.37\,,$$

i.e.

```
exp(-9.55 + 0.70*9 + 0.27*10)/(1 + exp(-9.55 + 0.70*9 + 0.27*10))

## [1] 0.3658644
```

There's a **37% chance** that a given **9-inch**, **10-pound** animal is a **dog**.

We could also get this value using `predict()`:

```
newPets <- data.frame(Ht = 9, Wt = 10)
newPets

##   Ht Wt
## 1  9 10

predict(my.logreg, newPets, type = "response")

##         1
## 0.3660569
```

Because there's only a **0.37 probability** that a given **9-inch**, **10-pound** animal is a **dog**, we **predict** (i.e. **classify**) that animal as being a **cat**.

It can be shown that $\boldsymbol{p(X_1, X_2) \geq 0.5}$ above the line $X_2 = 9.55/0.27 - 0.70/0.27 X_1$, and $\boldsymbol{p(X_1, X_2) < 0.5}$ below it. The following set of scatterplots shows this **split-line**.

```
## Scatterplot of pets heights and weights:
g1 <- ggplot(data = pets, mapping = aes(x = Ht, y = Wt, color = Type)) +
  geom_point() +
  labs(title = "Wts and Hts of Pets")
g1
```

```
## Scatterplot with logistic regression split-line:
g2 <- g1 + geom_abline(intercept = 9.55/0.27, slope = -0.70/0.27,
                       color = "dodgerblue", linetype = 2)
g2
```

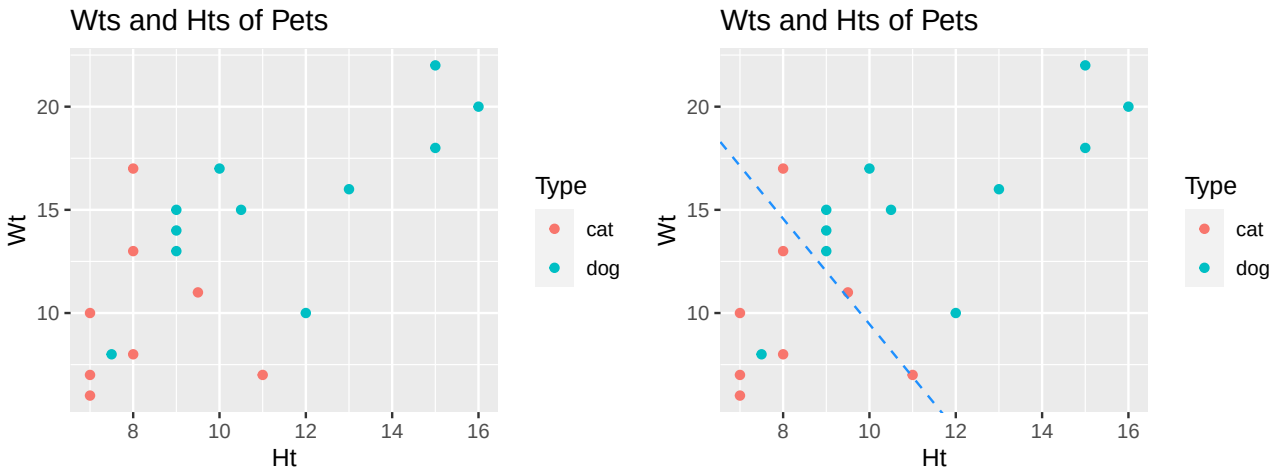



Figure 2

## 10.3 Evaluating Models

- We can **evaluate** the **accuracy** of a **classification** model using the following functions from the `"yardstick"` package.

```
accuracy()              #  Evaluate the classification accuracy.
conf_mat()              # Produce a confusion matrix.
```

- For example, to evaluate the (*in-sample*) **accuracy** of the **logistic regression model** for **classifying** pets as **dogs** or **cats**, we type:

```
# Get the (estimated) probabilities of being a "dog":
probs <- predict(my.logreg, newdata = pets, type = "response")

round(probs, digits = 2)

##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17
## 0.11 0.89 0.14 1.00 0.06 1.00 0.05 0.98 0.51 1.00 0.13 0.87 0.63 0.52 0.57 0.39 0.69
##   18   19
## 0.66 0.82

# Convert the probabilities to classifications ("cat" or "dog" predictions):
preds <- case_when(probs < 0.5 ~ "cat",
                   probs >= 0.5 ~ "dog")
preds

##  [1] "cat" "dog" "cat" "dog" "cat" "dog" "cat" "dog" "dog" "dog" "cat" "dog" "dog"
## [14] "dog" "dog" "cat" "dog" "dog" "dog"

# Insert the classifications ("cat" or "dog" predictions) as a new column in pets:
pets <- mutate(pets, predType = preds)

pets

##    Type   Ht Wt Type01 predType
## 1   dog  7.5  8      1      cat
## 2   dog 10.0 17      1      dog
## 3   cat  8.0  8      0      cat
## 4   dog 15.0 18      1      dog
## 5   cat  7.0  7      0      cat
## 6   dog 15.0 22      1      dog
## 7   cat  7.0  6      0      cat
## 8   dog 13.0 16      1      dog
## 9   cat 11.0  7      0      dog
## 10  dog 16.0 20      1      dog
## 11  cat  7.0 10      0      cat
## 12  dog 10.5 15      1      dog
## 13  dog  9.0 14      1      dog
## 14  cat  9.5 11      0      dog
## 15  dog  9.0 13      1      dog
## 16  cat  8.0 13      0      cat
## 17  dog  9.0 15      1      dog
## 18  cat  8.0 17      0      dog
## 19  dog 12.0 10      1      dog
```

One way to measure the **accuracy** of the **classification** model is via the ***correct classification rate***. Scanning above, we see that **15** of the **19** pets would be **classified** correctly, so the **correct classification rate** is **78.9%**:

$$\frac{15}{19} \;=\; \mathbf{0.789}\,.$$

We can also get this **accuracy** measure using `accuracy()` by typing:

```
library(yardstick)

# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = pets, truth = as.factor(Type), estimate = as.factor(predType))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.789
```

Another way to summarize the **accuracy** of the **classification** model is via the ***confusion matrix***:

```
# The conf_mat() function automatically converts Type and predType to factors:
conf_mat(data = pets, truth = Type, estimate = predType)

##            Truth
## Prediction cat dog
##        cat   5   1
##        dog   3  10
```

We see that **one dog** was **incorrectly classified** as a **cat**, and **three cats** were **incorrectly classified** as **dogs**.

- When **classifying** individuals into one of *two categories* (**positive** and **negative**, say), we can compute the ***true positive*** and ***true negative rates*** (sometimes called ***sensitivity*** and ***specificity***). For example, consider the **confusion matrix** form above:

| | | **Actual** | |
|---|---|---|---|
| | | Cat (Negative) | Dog (Positive) |
| **Prediction** | Cat (Negative) | 5 | 1 |
| | Dog (Positive) | 3 | 10 |

The **true positive** and **true negative rates** are:

$$\textbf{True Positive Rate} \;=\; \frac{10}{10+1} \;=\; \textbf{0.91}$$

$$\textbf{True Negative Rate} \;=\; \frac{5}{5+3} \;=\; \textbf{0.625}$$

We can also compute the ***false positive rate*** (which is **one minus** the **true negative rate**):

$$\textbf{False Positive Rate} \;=\; \frac{3}{5+3} \;=\; \textbf{0.375}\,.$$

- We've seen that for **classifying** individuals into one of *two categories* (**positive** and **negative**, say), we can **classify** an individual into the **positive** category if the (estimated) **probability** of the individual belonging to that category is greater than the ***threshold*** value **0.5**.

  But a *different* **threshold** value could be used.

  Using a value **smaller** than **0.5** will result in **more true positives** but also **more false positives**. Using a value **larger** than **0.5** will result in **fewer true positives** but also **fewer false positives**.

  We can assess a classifier's performance across the range of **threshold** values from **0.0** to **1.0** via a ***receiver operating characteristic*** (or ***ROC***) ***curve***.

  An **ROC curve** is a plot of the **true positive rate** ($y$-axis) versus the **false positive rate** ($x$-axis) for candidate **threshold** values ranging between **0.0** and **1.0**.

  For more details, see the text book.

- We'll look at other ways of **evaluating models** in Section 11.3.

# 11 Supervised Learning (11)

## 11.1 Supervised Learning

- Supervised learning methods involve developing a **function** of the explanatory variables $X_1, X_2, \ldots, X_p$ that can be used to **predict** a response value $Y$.

### 11.1.1 Classifiers

- Recall that **classification** can be viewed as **predicting** a **categorical** response variable from the values of explanatory variables.
- We'll look at several **classifiers**:
  - Logistic regression (already discussed above)
  - Decision trees
  - Random forests
  - Nearest neighbor
  - Naive Bayes
  - Artificial Neural Networks

**Decision Trees**

- Starting with the complete set of rows (or "records") of a data frame, a ***decision tree*** is produced by *recursively* using values of explanatory variables to **split** *sets of rows* (or **"nodes"**) into smaller *subsets* (which become the new **"nodes"**), so that the *subsets* are **"purer"** with respect to the **categorical** response variable than the original sets.
- The `"rpart"` package has functions for developing **decision trees** in R. Among them are the following.

```
rpart()              # Recursive partitioning for decision trees (and
                     # regression trees).
predict.rpart()      # Predict the response variable of a new observation
                     # using a tree.  This is the "rpart" method for the
                     # generic predict() function.
plot.rpart()         # Plot a decision tree.  This is the "rpart" method
                     # for the generic plot() function.
text.rpart()         # Add text to the plot of a decision tree.  This is
                     # the "rpart" method for the generic text() function.
```

- The main arguments passed to `rpart()` are a *formula* indicating the response and explanatory variables and a *data frame* in which to find those variables.
- For example, using the `pets` data set:

```
library(rpart)

my.tree <- rpart(Type ~ Wt + Ht,
                 data = pets,
                 control = rpart.control(minsplit = 7))
```

The `rpart.control()` function is used, via the `control` argument, to set various parameters that control details of the **decision tree** `rpart()` fits to the data, in this case the minimum number of observations that must exist in a node in order for a split to be attempted.

The object `my.tree` contains the results:

```
my.tree
```

```
## n= 19
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 19 8 dog (0.4210526 0.5789474)
##   2) Ht< 8.5 7 1 cat (0.8571429 0.1428571) *
##   3) Ht>=8.5 12 2 dog (0.1666667 0.8333333)
##     6) Wt< 12 3 1 cat (0.6666667 0.3333333) *
##     7) Wt>=12 9 0 dog (0.0000000 1.0000000) *
```

This tree uses two explanatory variables (`Wt` and `Ht`) to partition the `pets` data set into three parts (or ***terminal nodes***):

1. Pets shorter than 8.5 inches.
2. Pets taller than 8.5 inches but weighing less than 12 pounds.
3. Pets taller than 8.5 inches and weighing more than 12 pounds.

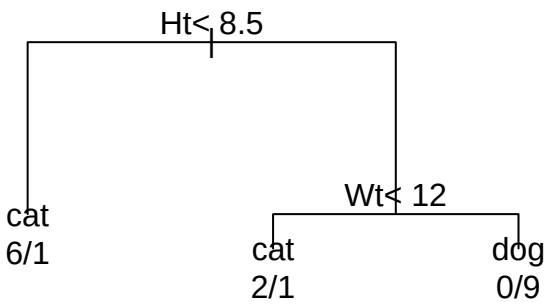
The object `my.tree` belongs to the `"rpart"` *class* of objects:

```
class(my.tree)

## [1] "rpart"
```

It's actually just a *list* object (type `is.list(my.tree)`).

We can plot the tree using `plot()` and add text using `text()`. Both are *generic* functions that pass `my.tree` along to the `plot.rpart()` and `text.rpart()` *methods* from the `"rpart"` package:

```
#This allows the tree diagram to extend outside the normal plot region:
par(xpd = TRUE)

plot(my.tree, compress = TRUE)
text(my.tree, use.n = TRUE)
```

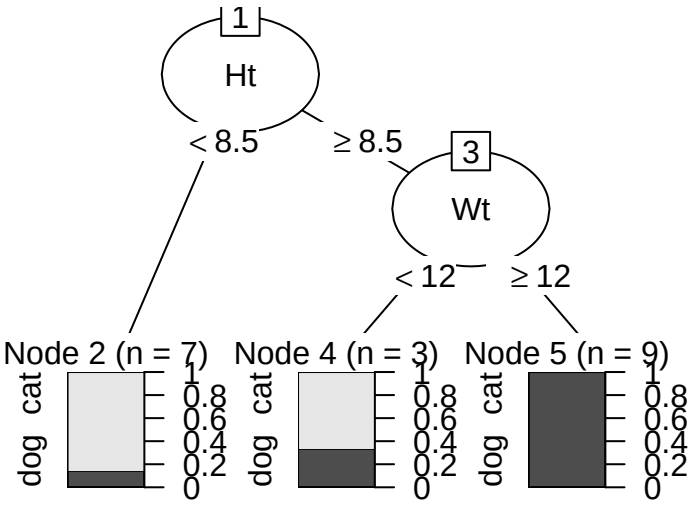



```
# Reset xpd to its default value:
par(xpd = FALSE)
```

Above, setting the graphical parameter `xpd` to `TRUE` allows the tree diagram to extend outside the normal plot region.

Another way to plot the tree is to use the `plot.party()` *method* from the `"partykit"` package (after converting `my.tree` to the `"party"` *class*):

```
library(partykit)

# Convert my.tree to the "party" class for plotting:
my.party.tree <- as.party(my.tree)

plot(my.party.tree)
```




In the plots, the numbers (or proportions) of **cats** and **dogs** are displayed at each **terminal node**.

The following sequence of scatterplots shows the **splits** used.

```
## Scatterplot of pets heights and weights:
g1 <- ggplot(data = pets, mapping = aes(x = Ht, y = Wt, color = Type)) +
  geom_point() +
  labs(title = "Wts and Hts of Pets")
g1
```

```
## Scatterplot with first split-line:
splitHt <- 8.5
g2 <- g1 + geom_vline(xintercept = splitHt, color = "dodgerblue", linetype = 2)
g2
```

```
## Scatterplot with first and second split-lines:
splitWt <- 12
splitLines <- data.frame(x1 = splitHt, x2 = 16, y1 = splitWt, y2 = splitWt)
g3 <- g2 + geom_segment(data = splitLines,
                        mapping = aes(x = x1, y = y1, xend = x2, yend = y2),
                        color = "rosybrown", linetype = 2)
g3
```

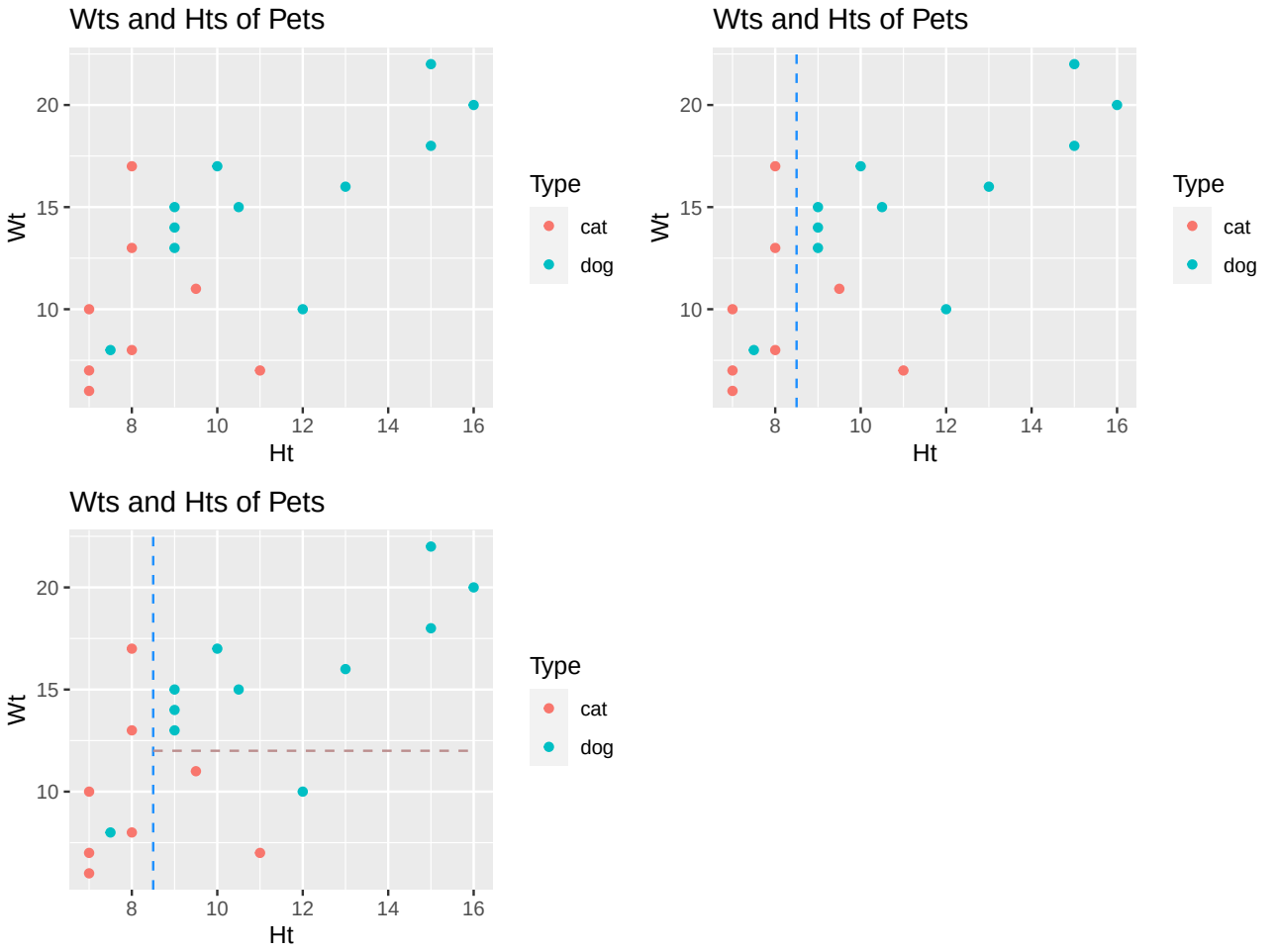



Figure 3

To get a more detailed summary of the fitted tree, we could use `summary()`:

```
summary(my.tree)
```

(Output not shown.)

- We can **evaluate** the **accuracy** of the **classification** model using the `accuracy()` and `conf_mat()` functions (from the `"yardstick"` package).

For example, to evaluate the (*in-sample*) **accuracy** of the **decision tree** for **classifying** pets as **dogs** or **cats**, we type:

```
# Get the classifications ("cat" or "dog" predictions):
preds <- predict(my.tree, type = "class")
preds

##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
## cat dog cat dog cat dog cat dog cat dog cat dog dog cat dog cat dog cat cat
## Levels: cat dog

# Insert the classifications ("cat" or "dog" predictions) as a new column in pets:
pets <- mutate(pets, predType = preds)

pets

##    Type   Ht Wt predType
```

```
## 1   dog  7.5  8      cat
## 2   dog 10.0 17      dog
## 3   cat  8.0  8      cat
## 4   dog 15.0 18      dog
## 5   cat  7.0  7      cat
## 6   dog 15.0 22      dog
## 7   cat  7.0  6      cat
## 8   dog 13.0 16      dog
## 9   cat 11.0  7      cat
## 10  dog 16.0 20      dog
## 11  cat  7.0 10      cat
## 12  dog 10.5 15      dog
## 13  dog  9.0 14      dog
## 14  cat  9.5 11      cat
## 15  dog  9.0 13      dog
## 16  cat  8.0 13      cat
## 17  dog  9.0 15      dog
## 18  cat  8.0 17      cat
## 19  dog 12.0 10      cat
```

Above, the `predict()` *generic* function passes the `"rpart"` object (`my.tree`) to the `predict.rpart()` *method*, which returns the tree's **predicted** type ("dog" or "cat" **classification**) for each animal in `pets` (based on its `Ht` and `Wt`).

Note that **17** of the **19** animals in `pets` were **classified correctly** by the model. Only the first and last animals were **classified incorrectly**.

Thus the **correct classification rate** is **89.5%**:

$$\frac{17}{19} = \mathbf{0.895}.$$

We can also get this **accuracy** measure using `accuracy()` by typing:

```
# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = pets, truth = as.factor(Type), estimate = as.factor(predType))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.895
```

Another way to summarize the **accuracy** of the **classification** model is via the **confusion matrix**:

```
# The conf_mat() function automatically converts Type and predType to factors:
conf_mat(data = pets, truth = Type, estimate = predType)

##           Truth
## Prediction cat dog
##        cat   8   2
##        dog   0   9
```

We see that **two dogs** were **incorrectly classified** as **cats**.

- At each step of the **decision tree model building** process, for each **node** (i.e. each *subset* of rows), all variables are tested for whether to split the **node** on that variable.

  For any given variable, the **optimal split** value is the one for which the (average) **"purity"** of the resulting new **nodes** (*subsets* of rows) is highest.

**"Impurity"** is measured by the ***Gini index***:

$$G = 1 - \sum_{i=1}^{k} p_i^2 , \tag{2}$$

where $\boldsymbol{k}$ is the number of categories of the response variable, and $\boldsymbol{p_i}$ is the proportion of rows, among the set under consideration for splitting, in the $i$th category of the response variable.

Each $p_i^2$ is the *probability* that two randomly selected rows will *both* be in the *ith category* of the response. Their sum is the *probability* that they'll *both* be in the *same category* (one or another of the $k$ categories).

$G$ takes values between 0 and 1, with **0** implying perfect **"purity"** (i.e. all the same category) and values closer to **1** implying more **"impurity"** (diversity).*

* The maximum value of $G$ is $1 - 1/k$.

- By default, in `rpart()` a split is only performed if it decreases the **misclassification** rate by at least **1%**.

This so-called ***complexity parameter*** (or ***tuning parameter***) value can be adjusted (see Exercise 6). A *smaller* value results in a tree that *fits* the **original data** better (i.e. has higher **complexity**, with more **terminal nodes** and **lower misclassification rate**), but risks **overfitting**.

## Section 11.1 Exercises

**Exercise 1** Refer to the **decision tree** based on the `pets` data (above). Answer the following questions *without* using `predict()`.

a) What type of animal (**cat** or **dog**) would you **predict** a **9**-inch, **10**-pound pet to be (using the **decision tree**)?

b) What type of animal (**cat** or **dog**) would you **predict** a **14**-inch, **21**-pound pet to be (using the **decision tree**)?

**Exercise 2** The function `predict()` can be used with an object of class `"rpart"` to (**classify**) a new observation.

Create the `pets` data frame:

```
type <- c("dog", "dog", "cat", "dog", "cat", "dog", "cat", "dog", "cat",
          "dog", "cat", "dog", "dog", "cat", "dog", "cat", "dog", "cat", "dog")
wt <- c(8, 17, 8, 18, 7, 22, 6, 16, 7, 20, 10, 15, 14, 11, 13, 13, 15, 17, 10)
ht <- c(7.5, 10, 8, 15, 7, 15, 7, 13, 11, 16, 7, 10.5, 9, 9.5, 9, 8, 9, 8, 12)
pets <- data.frame(Type = type, Ht = ht, Wt = wt)
```

Then fit the **decision tree** using:

```
library(rpart)

my.tree <- rpart(Type ~ Wt + Ht, data = pets,
                 control = rpart.control(minsplit = 7))
```

Create a data frame with *new* height and weight values for which we want **predict** the pet type:

```
newPets <- data.frame(Ht = c(9, 14), Wt = c(10, 21))
newPets

##   Ht Wt
## 1  9 10
## 2 14 21
```

Now type:

```
predict(my.tree, newdata = newPets, type = "class")
```

a) What type of animal (**cat** or **dog**) would you **classify** the **9**-inch, **10**-pound pet as?

b) What type would you **classify** the **14**-inch, **21**-pound pet as?

c) Are your answers using `predict()` consistent with your answers to Exercise 1?

**Exercise 3** Consider the (built-in) `iris` data set. Fit the following **decision tree** for **classifying** flowers into the different `Species` based on their `Petal.Length` and `Petal.Width`:

```
my.tree <- rpart(Species ~ Petal.Length + Petal.Width, data = iris)
my.tree
```

a) How many **terminal nodes** does the tree have?

b) Now create the **confusion matrix**:

```
preds <- predict(my.tree, type = "class")

#This places a copy of the built-in iris data set in the Workspace:
iris <- mutate(iris, predSpecies = preds)
```

```
# The conf_mat() function automatically converts Species and predSpecies to fac-
tors:
conf_mat(data = iris, truth = Species, estimate = predSpecies)
```

What is the tree's **correct classification rate** (as a percent)? What is it's **misclassification rate** (as a percent)?

c) Look at the following two plots.

```
## First plot:
par(xpd = TRUE)
plot(my.tree, compress = TRUE)
text(my.tree, use.n = TRUE)
par(xpd = FALSE)
```

```
## Second plot:
splitPetal.Length <- 2.45
splitPetal.Width <- 1.75
splitLines <- data.frame(x1 = splitPetal.Length, x2 = 7,
                         y1 = splitPetal.Width, y2 = splitPetal.Width)

g <- ggplot(data = iris,
            mapping = aes(x = Petal.Length, y = Petal.Width,
            color = Species)) +
     geom_point() +
     labs(title = "Widths and Lengths of Petals") +
     geom_vline(xintercept = splitPetal.Length,
                color = "dodgerblue",
                linetype = 2) +
     geom_segment(data = splitLines,
                  mapping = aes(x = x1, y = y1, xend = x2, yend = y2),
                  color = "rosybrown", linetype = 2)
g
```

Use the plots (*not* `predict()`) to **classify** the the following flowers into `Species`:

A flower whose `Petal.Length` is **3.0 cm** and whose `Petal.Width` is **1.5 cm**.

A flower whose `Petal.Length` is **4.0 cm** and whose `Petal.Width` is **2.1 cm**.

d) Create the following data frame of two *new* iris flowers:

```
newIris <- data.frame(Petal.Length = c(3.0, 4.0),
                      Petal.Width = c(1.5, 2.1))
newIris
```

Use `predict()`, with `my.tree`, to **predict** the species of the flowers in the `newIris` data set. Report the **two predictions** and compare them to those of Part *c*.

**Exercise 4** Consider again the (built-in) `iris` data set. This time fit the **decision tree** for **classifying** flowers into the different `Species` based on their `Sepal.Length` and `Sepal.Width`:

```
my.tree <- rpart(Species ~ Sepal.Length + Sepal.Width, data = iris)
my.tree
```

a) How many **terminal nodes** does this tree have?

b) Now create the **confusion matrix**:

```
preds <- predict(my.tree, type = "class")
iris <- mutate(iris, predSpecies = preds)
```

```
# The conf_mat() function automatically converts Species and predSpecies to fac-
tors:
conf_mat(data = iris, truth = Species, estimate = predSpecies)
```

What is the tree's **correct classification rate** (as a percent)? What is it's **misclassification rate** (as a percent)?

c) Look at the following two plots.

```
## First plot:
par(xpd = TRUE)
plot(my.tree, compress = TRUE)
text(my.tree, use.n = TRUE)
par(xpd = FALSE)
```

```
## Second plot:
split1Sepal.Length <- 5.45
split1Sepal.Width <- 2.8
splitLines1 <- data.frame(x1 = 3, x2 = split1Sepal.Length,
                          y1 = split1Sepal.Width, y2 = split1Sepal.Width)
split2Sepal.Length <- 6.15
split2Sepal.Width <- 3.1
splitLines2 <- data.frame(x1 = split1Sepal.Length, x2 = split2Sepal.Length,
                          y1 = split2Sepal.Width, y2 = split2Sepal.Width)

g <- ggplot(data = iris,
            mapping = aes(x = Sepal.Length, y = Sepal.Width,
            color = Species)) +
     geom_point() +
     labs(title = "Widths and Lengths of Sepals") +
     geom_vline(xintercept = split1Sepal.Length,
                color = "dodgerblue",
                linetype = 2)  +
     geom_vline(xintercept = split2Sepal.Length,
                color = "purple",
                linetype = 2)  +
     geom_segment(data = splitLines1,
                  mapping = aes(x = x1, y = y1, xend = x2, yend = y2),
                  color = "brown", linetype = 2) +
     geom_segment(data = splitLines2,
                  mapping = aes(x = x1, y = y1, xend = x2, yend = y2),
                  color = "red", linetype = 2)
g
```

Use the plots (*not* `predict()`) to **classify** the the following flowers into `Species`:

A flower whose `Sepal.Length` is **6.0 cm** and whose `Sepal.Width` is **3.5 cm**?

A flower whose `Sepal.Length` is **7.0 cm** and whose `Sepal.Width` is **3.0 cm**?

d) Create the following data frame of two *new* iris flowers:

```
newIris <- data.frame(Sepal.Length = c(6.0, 7.0),
                      Sepal.Width = c(3.5, 3.0))
newIris
```

Use `predict()`, with `my.tree`, to **predict** the species of the flowers in the `newIris` data set. Report the **two predictions** and compare them to those of Part *c*.

**Exercise 5** The **Gini index** takes values between 0 and 1, with 0 implying perfect **"purity"** (i.e. all individuals belonging to the same category) and larger values implying less "purity" (i.e. more diversity).

a) Guess which of the two data sets, `y1` and `y2`, is "purer" according to the **Gini index**, then check your answer by computing $G$ using expression 2 with the proportions $p_1, p_2$, and $p_3$ shown below.

```r
y1 <- c("A", "B", "C", "C", "C", "C", "C", "C", "C", "C", "C", "C")
round(prop.table(table(y1)), digits = 1)

## y1
##   A   B   C
## 0.1 0.1 0.8

y2 <- c("A", "B", "C", "C", "A", "C", "B", "A", "A", "B", "C", "B")
round(prop.table(table(y2)), digits = 1)

## y2
##   A   B   C
## 0.3 0.3 0.3
```

b) What is the **Gini index** value for the following data set?

```r
y <- c("A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A")
round(prop.table(table(y)), digits = 1)

## y
## A
## 1
```

**Exercise 6** In constructing a tree, optimal split values are chosen so that the (average) **"purity"** of the resulting **nodes** (subsets of rows) is highest, i.e. so that the (average) of the two **Gini index** values is *lowest*.

But in `rpart()`, by default the split is only performed if it decreases the **misclassification rate** by at least **1%**.

This so-called **complexity parameter** (or **tuning parameter**) can be adjusted (see the textbook).

A *smaller* value of this **complexity parameter** results in a tree that *fits* the **original data** better (i.e. has higher **complexity**, with more **terminal nodes** and **lower in-sample misclassification rate**), but risks **overfitting**.

Here's how to *lower* the complexity parameter to **0.2%** using the `control` argument:

```r
my.tree <- rpart(Species ~ Sepal.Length + Sepal.Width, data = iris,
                 control = rpart.control(cp = 0.002))
my.tree
```

```r
par(xpd = TRUE)
plot(my.tree, compress = TRUE)
text(my.tree, use.n = TRUE)
par(xpd = FALSE)
```

Here's how to *raise* it to **5%**:

```r
my.tree <- rpart(Species ~ Sepal.Length + Sepal.Width, data = iris,
    control = rpart.control(cp = 0.05))
my.tree
```

```r
par(xpd = TRUE)
plot(my.tree, compress = TRUE)
text(my.tree, use.n = TRUE)
par(xpd = FALSE)
```

Which threshold value, **0.2%** or **5%**, resulted in **more terminal nodes** (i.e. **higher complexity** in the tree)?

**Random Forests and Bagging**

- A ***random forest*** is a collection of **bootstrapped decision trees**. **Classification** of a new observation is based running the new observation through **every one** of the trees. The observation is **classified** according to **majority rule** (i.e. a **"vote"**) of the trees.

  Each tree is based on a **bootstrap sample** of **observations** (rows) from the original data frame. Furthermore:

  - ***Bagging*** ("bootstrap aggregating") is when each **bootstrapped tree** is based on **all** the **variables** (columns) in the original data frame.
  - ***Random forest*** is when each **bootstrapped tree** is based on its own **random sample** of **variables** (columns) from the original data frame.*

  *A separate random sample of variables is taken for consideration to be split upon at each split (node).

- The following function, from the `"randomForest"` package, will carry out the procedures.

```
randomForest()     # Carry out a random forest procedure on the data
                   # in a data frame.
importance()     #
```

  To construct a random forest:

  1. Choose `ntree`, the desired number of **trees** to make (e.g. 500), and choose `mtry`, the desired number of variables (columns) to use in each tree.
  2. Randomly select $n$ rows from the data frame *with replacement* (where $n$ is the number of rows in the original data frame), i.e. take a *bootstrap* sample of rows.
  3. Randomly select `mtry` variables (columns) *without replacement*.
  4. Build a tree on the resulting randomly selected data set.
  5. Repeat this procedure `ntree` times.

  A **classification** (**prediction**) for a new observation is made by **majority rule** (i.e. a **"vote"**) of the **classifications** of all `ntree` trees in the forest.

  Setting `mtry` equal to the total number of explanatory variables in the original data frame gives **bagging** (all explanatory variables used at each split).

- For example, using the `iris` data set:

```
library(randomForest)
```

```
my.forest <- randomForest(Species ~ Sepal.Length + Sepal.Width +
                                    Petal.Length + Petal.Width,
                          data = iris,
                          ntree = 500,
                          mtry = 2)
my.forest

##
## Call:
##  randomForest(formula = Species ~ Sepal.Length + Sepal.Width +      Petal.Length + Petal.Width,
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 4%
## Confusion matrix:
##            setosa versicolor virginica class.error
## setosa         50          0         0        0.00
## versicolor      0         47         3        0.06
## virginica       0          3        47        0.06
```

- The class of `my.forest` is `"randomForest"` (and also `"randomForest.formula"`):

```
class(my.forest)

## [1] "randomForest.formula" "randomForest"
```

Objects in this class are actually just a *lists* (type `is.list(my.forest)`)

To see the names of the objects stored in `my.forest`, type:

```
names(my.forest)
```

(Output not shown.)

- **correct classification rate** is obtained via:

```
library(yardstick)

# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = iris, truth = as.factor(Species), estimate = as.factor(predSpecies))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.793
```

The procedure has a **96% correct classification rate** for predicting an iris `Species`.

The **confusion matrix** is obtained via:

```
preds <- predict(my.forest, type = "class")
iris <- mutate(iris, predSpecies = preds)
```

```
# The conf_mat() function automatically converts Species and predSpecies to factors:
conf_mat(data = iris, truth = Species, estimate = predSpecies)
```

- The **importance** of each explanatory variable in **predicting** the iris `Species` is obtained by typing:

```
importance(my.forest)

##              MeanDecreaseGini
## Sepal.Length        10.075713
## Sepal.Width          2.022187
## Petal.Length        43.162783
## Petal.Width         43.963557
```

The output shows the **total decrease** in **node "impurities"** from splitting on the variable, averaged over all trees. A **larger importance** value implies the variable is **more important** in the **classification model**.

Thus `Petal.Width` is the *most* important for **classification** (predicting `Species`), followed closely by `Petal.Length`, `Sepal.Length`, and `Sepal.Width` (*least* important).

This is analogous to using p-values in multiple regression for deciding which explanatory variables are most important.

- In **random forest**, the main **complexity parameter** (i.e. **tuning parameter**) is the **number of variables** attempted for use at each split (i.e. `mtry`).

Using **more variables** results in **higher model complexity**, but risks **overfitting** the data.

## Section 11.1 Exercises

**Exercise 7** Recall that the argument `mtry` in `randomForest()` is the number of variables (columns) randomly sampled as candidate variables to split on at each node. It can be thought of as a **complexity parameter** – larger `mtry` values produce trees that conform to the data better but risk **overfitting**.

Compute the (**in-sample**) **correct classification rate** for each of the following **random forests** for predicting `Species` using the `iris` data and the `accuracy()` function (from the `"yardstick"` package):

a) Using `mtry = 1` in `randomForest()`:

```
my.forest <- randomForest(Species ~ Sepal.Length + Sepal.Width +
                                    Petal.Length + Petal.Width,
                          data = iris,
                          ntree = 500,
                          mtry = 1)
```

b) Using `mtry = 3` in `randomForest()`:

```
my.forest <- randomForest(Species ~ Sepal.Length + Sepal.Width +
                                    Petal.Length + Petal.Width,
                          data = iris,
                          ntree = 500,
                          mtry = 3)
```

**Exercise 8** When **all** of the variables (columns) in a data set are used as candidate variables to split on at each **node** in **random forest**, the procedure is sometimes called ***bagging***.

Carry out **bagging** for **classification** (predicting `Species`) using the `iris` data by setting `mtry = 4` in `randomForest()`:

```
my.forest <- randomForest(Species ~ Sepal.Length + Sepal.Width +
                                    Petal.Length + Petal.Width,
                          data = iris,
                          ntree = 500,
                          mtry = 4)
```

a) Then look at the **importance** of each explanatory variable:

```
importance(my.forest)
```

Which of the four explanatory variables is *most* important for predicting `Species`? Which is *least* important?

b) There is a `predict()` *method* for the `"randomForest"` class of objects called `predict.randomForest()`.

Create the following data frame of three *new* iris flowers:

```
newIris <- data.frame(Petal.Length = c(3.0, 2.2, 2.7),
                      Petal.Width = c(1.2, 2.1, 1.6),
                      Sepal.Length = c(5.5, 5.1, 5.9),
                      Sepal.Width = c(3.0, 2.7, 3.2))
newIris
```

Use `predict()`, with `my.forest`, to **predict** the species of the three flowers in the `newIris` data set:

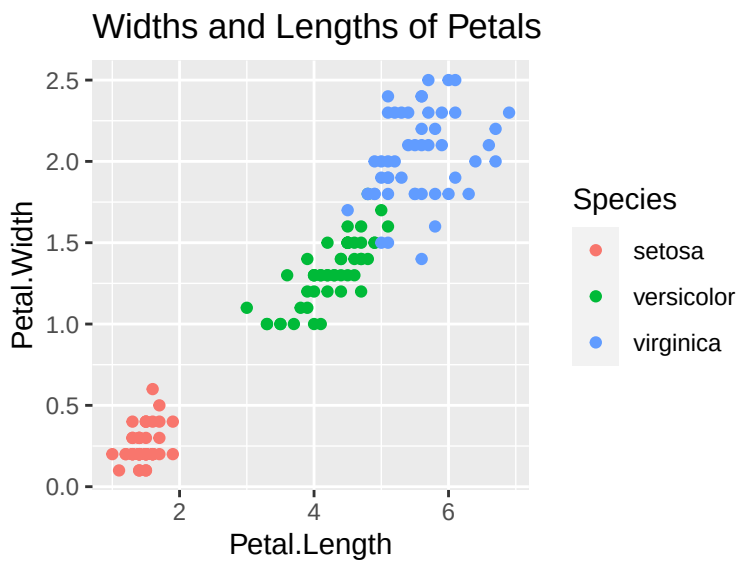
```
predict(my.forest, newdata = newIris, type = "class")
```

Report the **three species predictions**.

**Nearest Neighbor**

- We can think of an observation (row) with ***p* explanatory variables (*X*'s)** as a point in ***p*-dimensional** coordinate system. For example, a flower with given `Petal.Length` and `Petal.Width` is a point in **2-dimensional** coordinate system:

```
ggplot(data = iris,
       mapping = aes(x = Petal.Length, y = Petal.Width,
       color = Species)) +
  geom_point() +
  labs(title = "Widths and Lengths of Petals")
```




Likewise, a flower with given `Petal.Length`, `Petal.Width`, `Sepal.Length`, and `Sepal.Width` is a point in **4-dimensional** coordinate system.

The (Euclidean) **distance** between any two observations in ***p*-dimensional** space can be computed. For example, in **2-dimensional** space, the **distance** between $(x_1, y_1)$ and $(x_2, y_2)$ is

$$\text{Distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

(the *Pythagorean Theorem*), and in **3-dimensional** space, the **distance** between $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$ is

$$\text{Distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}.$$

- A ***k* *nearest neighbor*** procedure for **predicting** the (categorical) response for a new observation (i.e. for **classifying** the observation) is based on the **majority rule** (i.e. aka **"vote"**) of the ***k* nearest observations** (in Euclidean distance) to the new observation.

For example, a *new* flower with **petal length 4.35 cm** and **petal width 1.65 cm** is shown in Fig. 4 (black dot).

```
#This removes the copy of the built-in iris data that was placed in the
# Workspace earlier:
rm(iris)

newIris <- data.frame(Petal.Length = 4.35,
                      Petal.Width = 1.65)
```

```
g <- ggplot(data = iris,
            mapping = aes(x = Petal.Length, y = Petal.Width)) +
       geom_point(mapping = aes(color = Species)) +
```

```
        labs(title = "Widths and Lengths of Petals") +
        geom_point(data = newIris, color = "black")
g
```

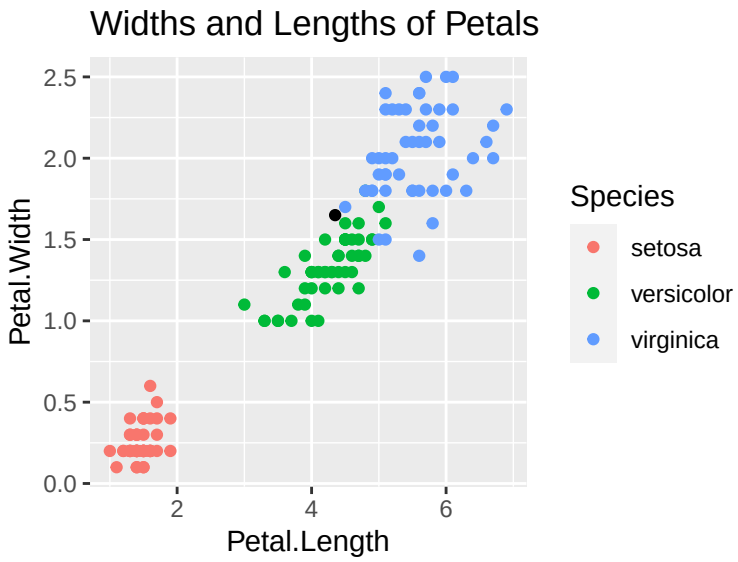



Figure 4

Of the $\boldsymbol{k = 3}$ nearest flowers (shown in the circle in Fig. 5), two are Versicolor, one is Setosa, and none are Virginica.

By a **majority rule** (**"vote"**), the *new* flower is **predicted** to be **Versicolor**.

```
g + geom_point(data = newIris, size = 9.1, shape = 1, color = "gold4")
```

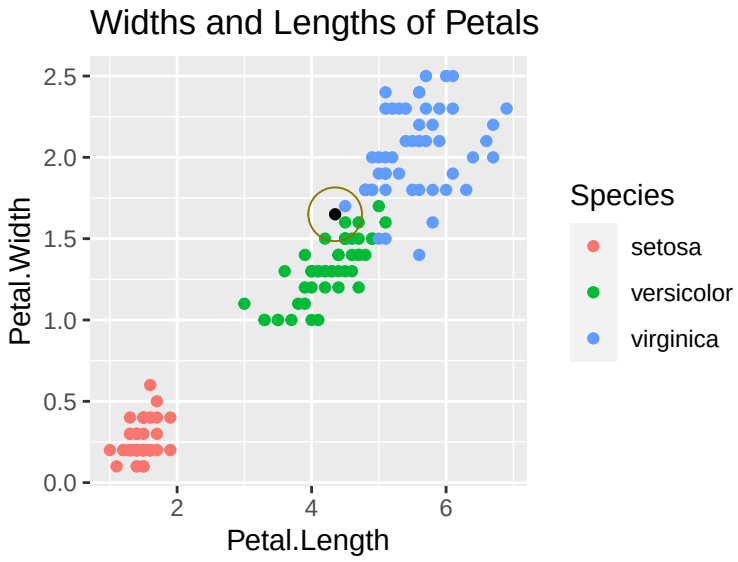



Figure 5

- To carry out ***k* nearest neighbor** for predicting the (categorical) response for a **new observation** (i.e. for **classifying** that observation):

  1. Choose $k$.
  2. Find the $k$ observations in the data set that are closest to the new observation (in ***p*-dimensional** space, where $\boldsymbol{p}$ is the number of explanatory variables).

3. **Classify** the new observation into response variable category shared by the **majority** (or **plurality**) of its ***k* nearest neighbors**.

The number of neighbors, ***k***, is a **complexity parameter**. A smaller value of $k$ leads to the procedure conforming more closely to the data (i.e. higher model **complexity** and **lower misclassification rate**), but runs the risk of **overfitting**.

- The following function, from the `"kknn"` package, will carry out the ***k* nearest neighbor** classification procedure.

```
kknn()    # Carry out a k nearest neighbor procedure on the data
          # in a data frame.
```

- For example, using the `iris` data set:

```
library(kknn)
library(dplyr)          # For select()

# Data frame containing new flower for classification:
newIris <- data.frame(Petal.Length = 4.35,
                      Petal.Width = 1.65)

# k nearest neighbor classification procedure:
my.knn <- kknn(Species ~ Petal.Length + Petal.Width,
               train = iris,
               test = newIris,
               k = 3)

summary(my.knn)

##
## Call:
## kknn(formula = Species ~ Petal.Length + Petal.Width, train = iris,     test = newIris, k = 3)
##
## Response: "nominal"
##          fit prob.setosa prob.versicolor prob.virginica
## 1 versicolor           0       0.6666667      0.3333333

# Get the classification of the new iris flower:
pred <- fitted(my.knn)
pred

## [1] versicolor
## Levels: setosa versicolor virginica
```

As expected (from Fig. 5), the **4.35 cm** long, **1.65 cm** wide flower is **predicted** to be **Versicolor**.

- We can determine the (**in-sample**) **correct classification rate** for **prediction** of `Species` for flowers in the *original data set* (`iris`) by typing:

```
my.knn <- kknn(Species ~ Petal.Length + Petal.Width,
               train = iris,
               test = iris,
               k = 3)

preds <- fitted(my.knn)
iris <- mutate(iris, predSpecies = preds)
```

```
# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = iris, truth = as.factor(Species), estimate = as.factor(predSpecies))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.993
```

Thus the **correct classification rate** is **99.3%**.

We can obtain the **confusion matrix** by typing:

```
# The conf_mat() function automatically converts Species and predSpecies to factors:
conf_mat(data = iris, truth = Species, estimate = predSpecies)
```

## Section 11.1 Exercises

**Exercise 9** The number of neighbors ***k*** is a **complexity parameter** (or **tuning parameter**) in the **nearest neighbor** procedure. Using $k = 1$ predicts a new observation to be the same class as its (one) nearest neighbor. A *larger* $k$ uses more data in each prediction.

Using $k = 1$ leads to a **100% correct classification rate** for the observations in the *original data set* (i.e. *in-sample* observations), but won't predict *new* (*out-of-sample*) observations, very well due to **overfitting**.

Up to a point, a *larger* $k$ will have a **lower** *in-sample* correct classification rate, but will predict *new* (*out-of-sample*) observations **better**. Beyond that point, the predictive accuracy for new (*out-of-sample*) observations deteriorates with larger $k$ values.

(The optimal $k$ can be determined using ***cross-validation***.)

a) Experiment with a few different values of $k$ by editing the code below. **Report** the **correct classification rate** (for observations in the *original data set*, i.e. *in-sample* observations) for the different values of $k$ you chose.

```
library(kknn)
library(dplyr)          # Contains select()

# Change this value to try different k, then run the code
# below for each choice of k:
ktry <- 3

# k nearest neighbor classification procedure:
my.knn <- kknn(Species ~ Petal.Length + Petal.Width,
               train = iris,
               test = iris,
               k = ktry)

preds <- fitted(my.knn)
iris <- mutate(iris, predSpecies = preds)
```

```
# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = iris, truth = as.factor(Species), estimate = as.factor(predSpecies))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.993
```

b) Do your **predictions** for the **4.35 cm** long, **1.65 cm** wide flower change with the choice of $k$? Edit the code below to find out.

```
# Change this value to try different k, then run the code
# below for each choice of k:
ktry <- 3

# Data frame containing new flower for classification:
newIris <- data.frame(Petal.Length = 4.35,
                      Petal.Width = 1.65)

# k nearest neighbor classification procedure:
my.knn <- kknn(Species ~ Petal.Length + Petal.Width,
               train = iris,
               test = newIris,
               k = ktry)

# Get the classification of the new iris flower:
pred <- fitted(my.knn)
pred
```

**Naive Bayes**

- The ***naive Bayes*** classifier predicts for an individual the class that the individual has the **highest probability** of belonging to, based on its values $X_1, X_2, \ldots, X_p$ of the explanatory variables.

  To estimate the (conditional) **probability** of each class, given the observed values of $X_1, X_2, \ldots, X_p$, it invokes ***Bayes' Rule*** (from MTH 3210 Probability and Statistics).

  For more details, see the textbook.

**Artificial Neural Networks**

- ***Artificial neural networks*** can be used for **prediction** and for **classification**.
- For **prediction** (i.e. when the response variable $Y$ is *numerical*), a **neural network** can be thought of as a generalization of multiple regression for **predicting** $Y$ from $\boldsymbol{p}$ explanatory variables $\boldsymbol{X_1, X_2, \ldots, X_p}$.

  The idea is to **"derive"** $\boldsymbol{k}$ **new** explanatory variables $\boldsymbol{H_1, H_2, \ldots, H_k}$ from the original $X$'s via a (non-linear) function $\boldsymbol{g}$,

  $$H_j \;=\; g(a_{j0} + a_{j1}X_1 + \cdots + a_{jp}X_p) \qquad \text{for } j = 1, 2, \ldots, k\,,$$

  then **predict** $Y$ from the $H$'s via another (possibly non-linear) function $\boldsymbol{g_Y}$,

  $$\hat{Y} \;=\; g_Y(b_0 + b_1H_1 + \cdots + b_kH_k)\,.$$

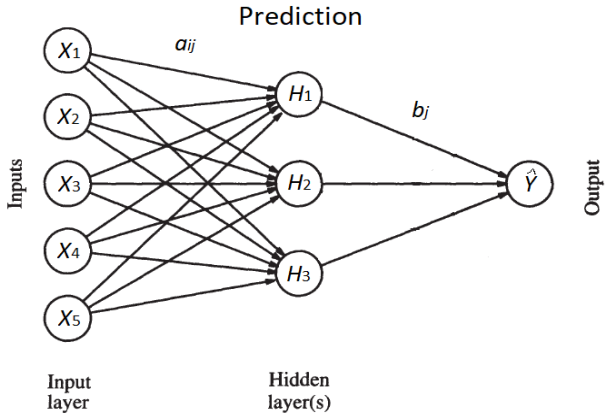
  Note that *if* the *linear* function $g_Y(z) = z$ was used, it would lead to a *multiple regression model* using the **"derived"** explanatory variables:

  $$\hat{Y} \;=\; b_0 + b_1H_1 + \cdots + b_kH_k\,.$$

  The **coefficients**,

  $$\begin{array}{lll} a_{j0}, a_{j1}, \ldots, a_{jp} & \text{for } j = 1, 2, \ldots, k & (\boldsymbol{k(p+1)}\textbf{ weights}) \\ b_0, b_1, \ldots, b_k & & (\boldsymbol{k+1}\textbf{ weights}) \end{array}$$

  are called ***weights***, and their values are determined simultaneously via the model fitting process.

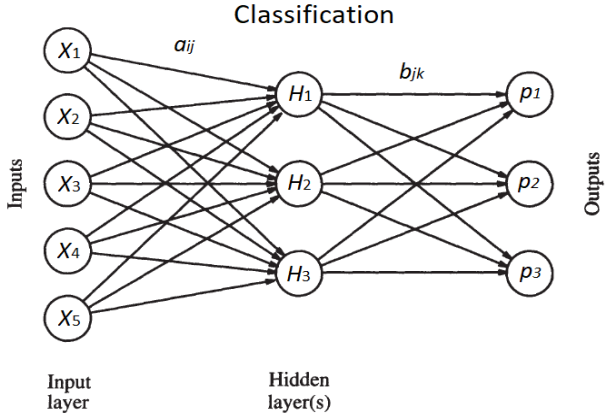



- For **classification** into one of $\boldsymbol{m}$ categories (i.e. when the response variable is *categorical* with $m$ possible values), individuals are **classified** according to **probabilities** $\boldsymbol{p_1, p_2, \ldots, p_m}$ estimated from the $H$'s via a (non-linear) function $\boldsymbol{g_Y}$,

$$p_\ell \;=\; g_Y(b_{\ell 0} + b_{\ell 1}H_1 + \cdots + b_{\ell k}H_k) \qquad \text{for } \ell = 1, 2, \ldots, m\,.$$

  Now the ***weights*** (**coefficients**) are:

$$\begin{array}{lll} a_{j0}, a_{j1}, \ldots, a_{jp} & \text{for } j = 1, 2, \ldots, k & (\boldsymbol{k(p+1)} \textbf{ weights}) \\ b_{\ell 0}, b_{\ell 1}, \ldots, b_{\ell k} & \text{for } \ell = 1, 2, \ldots, m & (\boldsymbol{m(k+1)} \textbf{ weights}) \end{array}$$




- In either case (**prediction** or **classification**), the fitted model has three ***layers***:
  - The explanatory variables, $X_1, X_2, \ldots, X_p$, are called ***input units***, and together comprise the so-called ***input layer*** of the neural network.
  - The "derived" explanatory variables, $H_1, H_2, \ldots, H_k$, are called ***hidden units***, and together comprise the so-called ***hidden layer***.
  - The **predicted** response $\hat{Y}$ or estimated **probabilities** $p_1, p_2, \ldots, p_m$ (for **classification**) are called the ***output units*** and make up the ***output layer***.

- The function $\boldsymbol{g}$ is called the ***hidden-layer activation function***, and is usually taken to be the **logistic function**

$$g(z) \;=\; \frac{e^z}{1+e^z}\,. \tag{3}$$

- The function $\boldsymbol{g_Y}$ is called the ***output-layer activation function*** and will be either a **linear function**, a **logistic function** (again), or a so-called ***softmax*** function, depending on whether $Y$ is *numerical*, *dichotomous*, or *categorical* with more than two categories.

  - The **linear** output function, used when $Y$ is *numerical*, is equivalent to the **"identity" function**, $g_Y(z) = z$, which leads to
$$\hat{Y} \;=\; b_0 + b_1 H_1 + \cdots + b_k H_k \,.$$
    As mentioned earlier, this is a *multiple regression model* using the **"derived"** explanatory variables.
  - The **logistic** output function, used when $Y$ is *dichotomous* (**0** or **1**), is given by (3). It returns the (estimated) **probability $p_1 = P(Y = 1)$**.
  - The **softmax** output function, used when $Y$ is *categorical* (taking more than two values), returns a set of (estimated) **probabilities $p_1, p_2, \ldots, p_m$**, one for each of the $m$ categories, that **sum to one**. It has the form
$$g_Y(z_\ell) \;=\; \frac{e^{z_\ell}}{\sum_{i=1}^{m} e^{z_i}} \qquad \text{for } \ell = 1, 2, \ldots, m \,.$$

- The number of hidden units $k$ is a **complexity parameter** (or **tuning parameter**) that can be adjusted (see Exercise 10). A *larger* value of $k$ results in a model that fits the (*in-sample*) **original observations** better (i.e. has higher **complexity**), but risks **overfitting**.

- The following functions, from the `"nnet"` package, will carry out the **artificial neural network** prediction or classification procedure.

```
nnet()            # Carry out an artificial neural network procedure on the data
                  # in a data frame.
predict.nnet()    # Predict the response variable of or classify a new observation
                  # using a neural network.  This is the "nnet" method for the
                  # generic predict() function.
```

**Artificial Neural Networks for Prediction**

- We'll first see how to use an **artificial neural network** for **prediction**. (Later we'll see how to use one for **classification**).

**Data Set:** `rock`

The `rock` data set, built in to R, contains measurements on $n = 48$ rock specimens from a petroleum reservoir.

The four *numerical* variables are:

| | |
|---|---|
| `area` | Area of pores space, in pixels out of 256 by 256. |
| `peri` | Perimeter of pores in pixels. |
| `shape` | Perimeter/sqrt(area). |
| `perm` | Permeability in milli-Darcies. |

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

- Here's an example of using an **artificial neural network** for **prediction** using the built-in `rocks` data set.

  We'll **predict** `perm` from the other three variables, using **$k = 3$ hidden units**, after **rescaling** the variables to be on roughly equal scales (which helps the model fitting process):

```
library(nnet)
library(dplyr)          # Contains mutate()


# Rescale the variables so they're on roughly equal scales:
rockRescaled <- rock %>% mutate(area = area/10000,
                                peri = peri/10000,
                                perm = log(perm))
```

```
# Artificial neural network procedure for prediction using k = 3 hidden units:
my.nn <- nnet(perm ~ area + peri + shape, data = rockRescaled,
              size = 3, linout = TRUE, maxit = 1000, trace = FALSE)

summary(my.nn)

## a 3-3-1 network with 16 weights
## options were - linear output units
##   b->h1  i1->h1  i2->h1  i3->h1
##   88.23  -61.01 -137.01   55.10
##   b->h2  i1->h2  i2->h2  i3->h2
##    1.79  -17.05    6.58   11.34
##   b->h3  i1->h3  i2->h3  i3->h3
##   -0.25   14.93  -20.44   -7.49
##    b->o   h1->o   h2->o   h3->o
##  -12.76    2.34   13.63   17.31
```

In the call to `nnet()`, the *formula* indicates `perm` is the response and the other three are the explanatory variables. `size = 3` indicates **3 hidden nodes**. Specifying `linout = TRUE` indicates a **linear output function** (appropriate for **prediction** of a *numerical* response $Y$), leading to a *multiple regression model* using the **"derived"** explanatory variables.

From the output of `summary()`, there are $\boldsymbol{p = 3}$ **input units**, $\boldsymbol{k = 3}$ **hidden units**, and **one output** (`3-3-1`). The **16 weights** (coefficients) are shown in the output.

Below, `predict()` returns the (*in-sample*) **predicted permeabilities** for the 48 specimens in the original (`rocks2`) data set:

```
# Get the predicted permeabilities:
preds <- predict(my.nn)

# Insert the predicted permeabilities as a new column in rock:
rockRescaled <- mutate(rockRescaled, predPerm = preds)

head(rockRescaled)

##     area     peri     shape     perm predPerm
## 1 0.4990 0.279190 0.0903296 1.840550 1.846892
## 2 0.7002 0.389260 0.1486220 1.840550 1.735394
## 3 0.7558 0.393066 0.1833120 1.840550 2.092956
## 4 0.7352 0.386932 0.1170630 1.840550 2.468653
## 5 0.7943 0.394854 0.1224170 2.839078 3.372033
## 6 0.7979 0.401015 0.1670450 2.839078 2.847247
```

We can measure the fit of the neural network to the data using a **sum of squared (prediction) errors** (or **"residual sum of squares"**) and a **mean squared (prediction) error** (or **"mean squared residual"**):

```
squaredPredErrors <- (rockRescaled$perm - rockRescaled$predPerm)^2

sum(squaredPredErrors)      # Sum of squared (prediction) errors

## [1] 10.30757

mean(squaredPredErrors)     # Mean squared (prediction) error

## [1] 0.214741
```

We can also use `predict()` to obtain **predictions** for **new** observations:

```
newRockRescaled <- data.frame(area = c(0.7, 1.0),
                              peri = c(0.15, 0.30),
                              shape = c(0.25, 0.35))

predict(my.nn, newdata = newRockRescaled)

##       [,1]
## 1 6.816064
## 2 6.825662
```

### Artificial Neural Networks for Classification

- We'll now see how to use an **artificial neural network** for **classification**.
- Here's an example of using an **artificial neural network**, with $k = 2$ **hidden units**, for **classification** using the built-in `iris` data set:

```
# Artificial neural network classification procedure with k = 2 hidden units:
my.nn <- nnet(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
              data = iris, size = 2, maxit = 200, trace = FALSE)

summary(my.nn)

## a 4-2-3 network with 19 weights
## options were - softmax modelling
##  b->h1 i1->h1 i2->h1 i3->h1 i4->h1
##  -2.14  -0.17  -0.43   0.62   1.17
##  b->h2 i1->h2 i2->h2 i3->h2 i4->h2
##  -0.90  -1.80  -1.44   0.16  -0.42
##  b->o1 h1->o1 h2->o1
##  23.37 -67.32   0.51
##  b->o2 h1->o2 h2->o2
##  10.75  -0.46   0.29
##  b->o3 h1->o3 h2->o3
## -33.67  67.69  -0.58
```

From the output of `summary()`, there are $p = 4$ **input units**, $k = 2$ **hidden units**, and $m = 3$ **outputs** (`4-2-3`). The **19 weights** (coefficients) are shown in the output.

We can obtain the **classifications** (**predicted** species) and insert them as a new column in `iris` using:

```
preds <- predict(my.nn, type = "class")

#This places a copy of the built-in iris data set in the Workspace:
iris <- mutate(iris, predSpecies = preds)

head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species predSpecies
## 1          5.1         3.5          1.4         0.2  setosa      setosa
## 2          4.9         3.0          1.4         0.2  setosa      setosa
## 3          4.7         3.2          1.3         0.2  setosa      setosa
## 4          4.6         3.1          1.5         0.2  setosa      setosa
## 5          5.0         3.6          1.4         0.2  setosa      setosa
## 6          5.4         3.9          1.7         0.4  setosa      setosa
```

In `predict()`, specifying `type = "class"` indicates that the `Species` **"classification"** (prediction) should be returned for each flower in `newIris`. Otherwise, the (estimated) **probabilities** $p_1, p_2, p_3$ for the three `Species` would be returned for each flower.

- The **correct classification rate** is obtained via:

```
library(yardstick)          # For the accuracy() function

# The accuracy() function requires the arguments truth and estimate to be factors:
accuracy(data = iris, truth = as.factor(Species), estimate = as.factor(predSpecies))

## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.987
```

The procedure has a **98.7% correct classification rate** for predicting an iris `Species`.

The **confusion matrix** is obtained via:

```
# The conf_mat() function automatically converts Species and predSpecies to factors:
conf_mat(data = iris, truth = Species, estimate = predSpecies)

##             Truth
## Prediction   setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         49         1
##   virginica       0          1        49
```

Here are **classifications** for two **new** observations:

```
# Data frame containing new flowers for classification:
newIris <- data.frame(Petal.Length = c(3.0, 4.0),
                      Petal.Width = c(1.5, 2.1),
                      Sepal.Length = c(5.0, 6.1),
                      Sepal.Width = c(2.6, 3.4))

predict(my.nn, newdata = newIris, type = "class")

## [1] "versicolor" "versicolor"
```

## Section 11.1 Exercises

**Exercise 10** In a **neural network**, the number of hidden units $k$ is a **complexity parameter** (or **tuning parameter**) that can be adjusted. A *larger* value of $k$ results in a model that fits the (*in-sample*) **original observations** better (i.e. has higher **complexity**), but risks **overfitting**.

Create the `rockRescaled` data frame from the built-in `rocks` data set (after loading the two packages):

```
library(nnet)
library(dplyr)          # Contains mutate()

# Rescale the variables in rock so they're on roughly equal scales:
rockRescaled <- rock %>% mutate(area = area/10000,
                                peri = peri/10000,
                                perm = log(perm))
```

Here's a **neural network** for **predicting** `perm` from the other three variables, using $k = 3$ **hidden units** (or **"derived"** explanatory variables, $H_1$, $H_2$, and $H_3$), indicated by specifying `size = 3`:

```
# Neural network procedure for prediction using k = 3 hidden units.
# Change this value to try different k, then run the code below
# for each choice of k:
ktry <- 3

my.nn <- nnet(perm ~ area + peri + shape, data = rockRescaled,
              size = ktry, linout = TRUE, maxit = 1000, trace = FALSE)
```

Here are the (*in-sample*) **predicted permeabilities** for the 48 rock specimens:

```
# Get the predicted permeabilities:
preds <- predict(my.nn)

# Insert the predicted permeabilities as a new column in rock:
rockRescaled <- mutate(rockRescaled, predPerm = preds)
```

We can measure the fit of the neural network to the data using a **mean squared (prediction) error** (or **"mean squared residual"**):

```
squaredPredErrors <- (rockRescaled$perm - rockRescaled$predPerm)^2

mean(squaredPredErrors)     # Mean squared (prediction) error
```

Run the **neural network** procedure as above, but altering $\boldsymbol{k}$ to be $\boldsymbol{k = 1, 3, 5}$, and **7**, and determine the **mean squared (prediction) error** each time.

Which value, $\boldsymbol{k = 1, 3, 5}$, or **7**, resulted in the **smallest mean squared (prediction) error**?

**Exercise 11** Consider again the (built-in) `iris` data set.

Here's a **neural network**, with $\boldsymbol{k = 3}$ **hidden units** (or **"derived"** explanatory variables, $\boldsymbol{H_1}, \boldsymbol{H_2}$, and $\boldsymbol{H_3}$), for **classification** (predicting `Species`):

```
# Neural network classification procedure with k = 3 hidden units:
my.nn <- nnet(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
              data = iris, size = 3, maxit = 200, trace = FALSE)

summary(my.nn)
```

Here three **new** flowers:

```
# Data frame containing new flowers for classification:
newIris <- data.frame(Petal.Length = c(1.5, 5.2, 5.5),
                      Petal.Width = c(0.2, 1.9, 2.0),
                      Sepal.Length = c(5.0, 6.3, 6.5),
                      Sepal.Width = c(3.4, 2.9, 2.9))
```

a) Use `predict()` (with `type = "class"`) to **classify** (predict the `Species` of) these three new flowers. Report your R command(s) and the three `Species` predictions.

b) Remove `type = "class"` from your `predict()` command of part *a*, so that the (estimated) **probabilities** $\boldsymbol{p_1, p_2, p_3}$ for each of the three `Species` are returned for each of the three flowers in `newIris`.

Report, for each of the three flowers, the `Species` whose **probability** is **highest**. **Hint**: They should be the same as those predicted in part *a*.

### 11.1.2 Acknowledgment

- The above notes (and examples) on **artificial neural networks** and the `"nnet"` package borrow heavily from the books:

  *Modern Applied Statistics with S, 4th Edition*, by Venables, W.N., and Ripley, B.D., Springer, 2002.
  *Applied Linear Regression Models, 4th Edition*, by Kutner, Nachtsheim, and Neter, McGraw-Hill, 2004.

## 11.2 Ensemble Methods

- ***Ensemble methods*** involve using **multiple** classification (or prediction) methods, and then classifying based on *majority vote* (or predicting based on *averaging* predictions).

  For example, we could perform *random forest*, *k nearest neighbor*, and *artificial neural network*, then classify an individual by *majority vote*, i.e. to the class the individual was predicted to be in by the majority of the three procedures.

  For more information, see the textbook.

## 11.3 Evaluating Models (10)

- Recall that a model ***overfits*** the **original data** if it predicts the *those* responses well, but *not* responses of **new** observations (not part of the original data set to which the model was fitted).

  **Overfitting** results when the **complexity** of the model is *too high*, resulting in a model conforming **too closely** to random fluctuations in the data.

- For example, here again are the data on **lengths** and **weights** of **nine** snakes (from Class Notes 5):

```
Ln <- c(85.7, 64.5, 84.1, 82.5, 78.0, 81.3, 71.0, 86.7, 78.7)
Wt <- c(331.9, 121.5, 382.2, 287.3, 224.3, 245.2, 208.2, 393.4, 228.3)

snakes <- data.frame(Length = Ln, Weight = Wt)
```

  We fit each of these ***polynomial regression models*** to the data:

$$\begin{array}{lrcl}
\text{Model 0:} & Y & = & b_0 \\
\text{Model 1:} & Y & = & b_0 + b_1 X \\
\text{Model 2:} & Y & = & b_0 + b_1 X + b_2 X^2 \\
\text{Model 3:} & Y & = & b_0 + b_1 X + b_2 X^2 + b_3 X^3 \\
\text{Model 4:} & Y & = & b_0 + b_1 X + b_2 X^2 + b_3 X^3 + b_4 X^4 \\
\text{Model 5:} & Y & = & b_0 + b_1 X + b_2 X^2 + b_3 X^3 + b_4 X^4 + b_5 X^5
\end{array}$$

  where $Y$ is the **weight** and $X$ the **length** of a snake.

```
g + stat_smooth(method = "lm", formula = y ~ 1, se = F) +
ggtitle(label = "Model 0")
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 1), se = F) +
ggtitle(label = "Model 1")
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 2), se = F) +
ggtitle(label = "Model 2")
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 3), se = F) +
ggtitle(label = "Model 3")
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 4), se = F) +
ggtitle(label = "Model 4")
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 5), se = F) +
ggtitle(label = "Model 5")
```

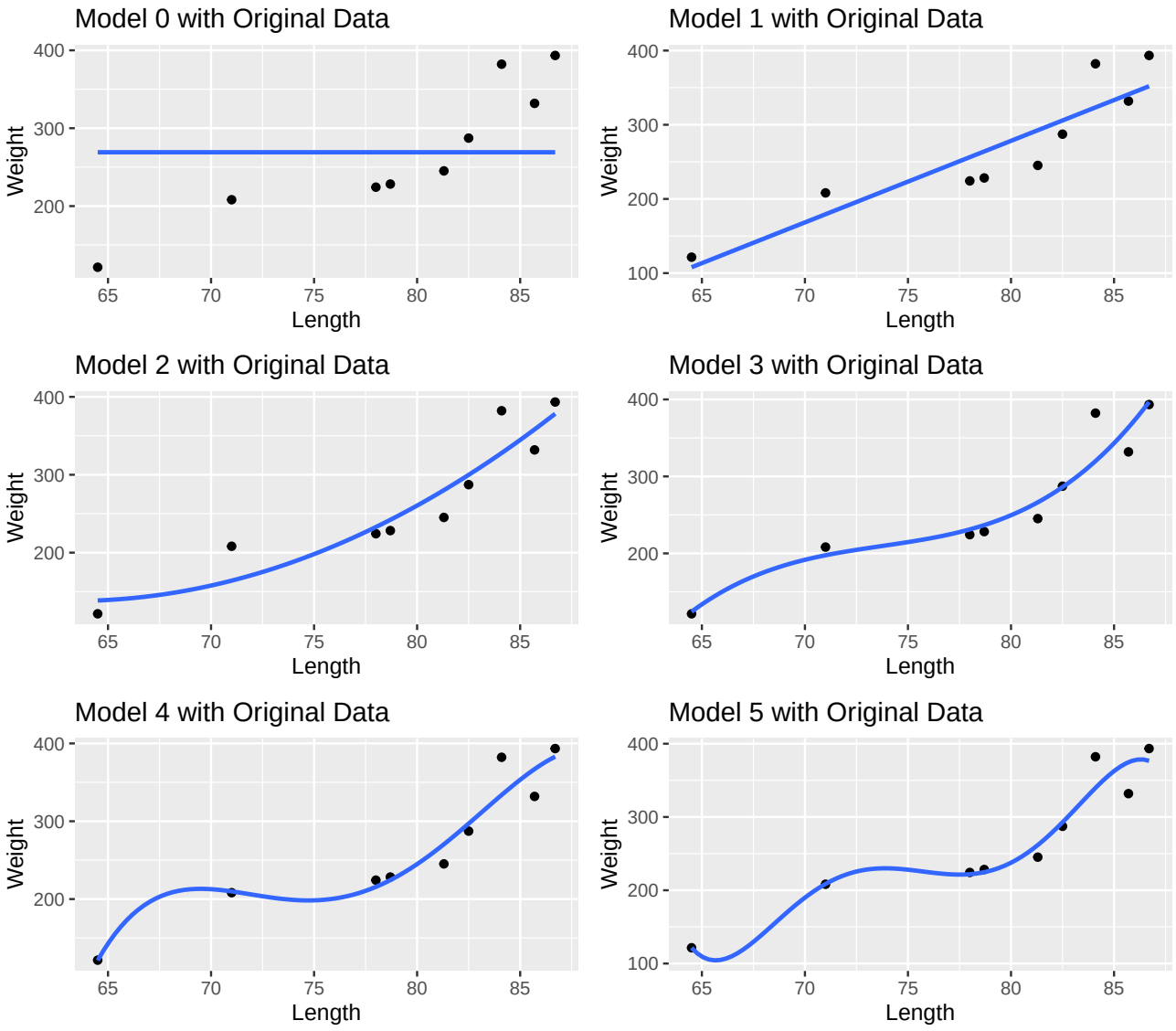



Figure 6

The models fit the **original data** progressively **better** as the model **complexity** (polynomial degree in this case) gets **higher**, i.e. they perform progressively better in ***in-sample*** accuracy (Fig. 6), but they **don't necessarily** predict **new** observations better, i.e. they don't necessarily perform better in ***out-of-sample*** accuracy.

For example, here are five **new** snakes:

```
newSnakes <- data.frame(Length = c(67, 72, 77, 81, 86),
                        Weight = c(127.9, 153.7, 204.7, 300.6, 291.4))
newSnakes
```

```
##   Length Weight
## 1     67  127.9
## 2     72  153.7
## 3     77  204.7
## 4     81  300.6
## 5     86  291.4
```

The **fifth-degree** polynomial fits the **original data** well, but doesn't predict **new** observations very well. The the **linear** doesn't fit the **original data** as well, but it predicts **new** observations better (Fig. 7).

```
g <- ggplot(snakes, aes(x = Length, y = Weight)) + geom_point(alpha = 0.05)
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 1), se = F) +
  ggtitle(label = "Model 5 with New Snakes") +
  geom_point(data = newSnakes)
```

```
g + stat_smooth(method = "lm", formula = y ~ poly(x, 5), se = F) +
  ggtitle(label = "Model 5 with New Snakes") +
  geom_point(data = newSnakes)
```

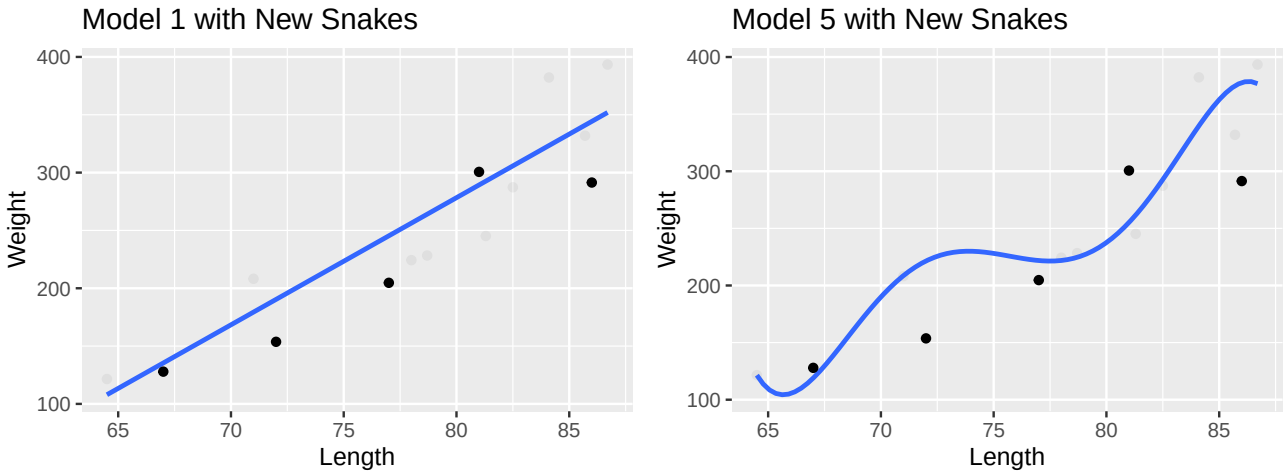



Figure 7

The degree of the polynomial is a kind of **complexity parameter** (or **tuning parameter**) that controls how "flexible" the model is. Higher-degree polynomials can "flex" to conform to the data better. Lower-degree ones are more "rigid".

We can use the five **new** observations to ***validate*** a given choice for the **polynomial degree**: The degree that leads to the **smallest prediction errors** for the **new observations** is preferred.

### 11.3.1 Cross-Validation

- In the absence of **new** observations for **validating** a model (e.g. to choosing a **complexity parameter** value), we can divide the **original** data set into *two parts*:
  - ***Training set***: Used to build (fit) the model.
  - ***Test set***: Used to test (or validate) the model.

  For example, **80%** of the (original) data set might be used as the **training set** to build the model and the other **20%** as the **test set** to test (or validate) the model.
- Another method is ***cross-validation***. Here's how to perform (***two-fold***) cross-validation:

1. Randomly separate the (original) data into two sets, a **training set** and a **test set**, with the same number of observations (i.e. **50%** in each set). Call them `my.data1` and `my.data2`.
2. Build (fit) the model using `my.data1` (the **training set**), then run `my.data2` (the **test set**) through the model and measure the model's (*out-of sample*) **prediction error**.
3. Now reverse the roles of `my.data1` and `my.data2`, i.e. use `my.data2` (as the **training set**) to build the model and `my.data1` (as the **test set**) to measure the (*out-of sample*) **prediction error**.
4. If your model **overfits** the data, it will likely have large **prediction errors** on the second (*out-of-sample*) data set.

- ***k-fold*** cross-validation is similar, but the (original) data set is separated into ***k*** smaller (equal-sized) sets that take turns serving as the **test set**.

  For example, for ***ten-fold*** cross-validation, the (original) data set is split into to **ten** smaller sets, each containing **10%** of the data. The smaller sets take turns serving as the **test set**, with the other **90%** of the data used to build (fit) the model.

### 11.3.2 Measuring Prediction Error: Numerical Response

- For **classification** (i.e. predicting a **categorical** response variable), we've seen that we can measure a model's accuracy by its **correct classification rate**.
- When the response ($Y$) is **numerical**, here are some ways to measure a model's **prediction** accuracy:
  - **MSE**: ***Mean squared (prediction) error*** (or ***"mean squared residual"***):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2\,,$$

    where $\boldsymbol{Y_i}$ is the *observed* response for the $i$th individual in the **test set**, and $\boldsymbol{\hat{Y}_i}$ is that individual's *predicted* value based on the model built from the **training set**. A *smaller* RMSE indicates a *better* model.
  - **MAE**: ***Mean absolute error***:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|\,,$$

    where (again) $\boldsymbol{Y_i}$ is the *observed* response for the $i$th individual in the **test set**, and $\boldsymbol{\hat{Y}_i}$ is that individual's *predicted* value based on the model built from the **training set**. A *smaller* MAE indicates a *better* model.

## Section 11.3 Exercises

**Exercise 12** We'll use the `snakes` data as the **training set** to *build* five **polynomial regression models** for **predicting** *weight* from *length* and `newSnakes` as the **test set** to choose a **polynomial degree** (the **complexity parameter** value).

Here are the (original) `snakes` data:

```
Ln <- c(85.7, 64.5, 84.1, 82.5, 78.0, 81.3, 71.0, 86.7, 78.7)
Wt <- c(331.9, 121.5, 382.2, 287.3, 224.3, 245.2, 208.2, 393.4, 228.3)

snakes <- data.frame(Length = Ln, Weight = Wt)
```

and here are the five **new** snakes:

```
newSnakes <- data.frame(Length = c(67, 72, 77, 81, 86),
                        Weight = c(127.9, 153.7, 204.7, 300.6, 291.4))
```

Fit these **six *polynomial models*** (using the "polynomial" function `poly()`):

```
mod0 <- lm(Weight ~ 1, data = snakes)
mod1 <- lm(Weight ~ Length, data = snakes)
mod2 <- lm(Weight ~ poly(Length, 2, raw = TRUE), data = snakes)
mod3 <- lm(Weight ~ poly(Length, 3, raw = TRUE), data = snakes)
mod4 <- lm(Weight ~ poly(Length, 4, raw = TRUE), data = snakes)
mod5 <- lm(Weight ~ poly(Length, 5, raw = TRUE), data = snakes)
```

Obtain the **predicted weights** for the five **new** snakes using each of the six polynomial regression models:

```
pred0 <- predict(mod0, newdata = newSnakes)
pred1 <- predict(mod1, newdata = newSnakes)
pred2 <- predict(mod2, newdata = newSnakes)
pred3 <- predict(mod3, newdata = newSnakes)
pred4 <- predict(mod4, newdata = newSnakes)
pred5 <- predict(mod5, newdata = newSnakes)
```

Add the six sets of predicted weights as new columns in `newSnakes`:

```
newSnakes <- mutate(newSnakes,
                    predWeight0 = pred0,
                    predWeight1 = pred1,
                    predWeight2 = pred2,
                    predWeight3 = pred3,
                    predWeight4 = pred4,
                    predWeight5 = pred5)
```

a) Obtain the **MSE** for each model (based on **prediction errors** for the five **new** snakes):

```
mse0 <- mean((newSnakes$Weight - newSnakes$predWeight0)^2)
mse1 <- mean((newSnakes$Weight - newSnakes$predWeight1)^2)
mse2 <- mean((newSnakes$Weight - newSnakes$predWeight2)^2)
mse3 <- mean((newSnakes$Weight - newSnakes$predWeight3)^2)
mse4 <- mean((newSnakes$Weight - newSnakes$predWeight4)^2)
mse5 <- mean((newSnakes$Weight - newSnakes$predWeight5)^2)
```

Which of the six **polynomial models** is best according to the (*out-of-sample*) **MSE**?

b) Obtain the **MAE** for each model (based on **prediction errors** for the five **new** snakes):

```
mae0 <- mean(abs(newSnakes$Weight - newSnakes$predWeight0))
mae1 <- mean(abs(newSnakes$Weight - newSnakes$predWeight1))
mae2 <- mean(abs(newSnakes$Weight - newSnakes$predWeight2))
mae3 <- mean(abs(newSnakes$Weight - newSnakes$predWeight3))
mae4 <- mean(abs(newSnakes$Weight - newSnakes$predWeight4))
mae5 <- mean(abs(newSnakes$Weight - newSnakes$predWeight5))
```

Which of the six **polynomial models** is best according to the (*out-of-sample*) **MAE**?