

FIGURE 4.11. Boxplots of the test error rates for each of the linear scenarios described in the main text.

Scenario 1: There were 20 training observations in each of two classes. The observations within each class were uncorrelated random normal variables with a different mean in each class. The left-hand panel of Figure 4.11 shows that LDA performed well in this setting, as one would expect since this is the model assumed by LDA. Logistic regression also performed quite well, since it assumes a linear decision boundary. KNN performed poorly because it paid a price in terms of variance that was not offset by a reduction in bias. QDA also performed worse than LDA, since it fit a more flexible classifier than necessary. The performance of naive Bayes was slightly better than QDA, because the naive Bayes assumption of independent predictors is correct.

Scenario 2: Details are as in Scenario 1, except that within each class, the two predictors had a correlation of -0.5 . The center panel of Figure 4.11 indicates that the performance of most methods is similar to the previous scenario. The notable exception is naive Bayes, which performs very poorly here, since the naive Bayes assumption of independent predictors is violated.

Scenario 3: As in the previous scenario, there is substantial negative correlation between the predictors within each class. However, this time we generated X_1 and X_2 from the t -distribution, with 50 observations per class. The t -distribution has a similar shape to the normal distribution, but it has a tendency to yield more extreme points—that is, more points that are far from the mean. In this setting, the decision boundary was still linear, and so fit into the logistic regression framework. The set-up violated the assumptions of LDA, since the observations were not drawn from a normal distribution. The right-hand panel of Figure 4.11 shows that logistic regression outperformed LDA, though both methods were superior to the other approaches. In particular, the QDA results deteriorated considerably as a consequence of non-normality. Naive Bayes performed very poorly because the independence assumption is violated.

Scenario 4: The data were generated from a normal distribution, with a correlation of 0.5 between the predictors in the first class, and correlation of -0.5 between the predictors in the second class. This setup corresponded to the QDA assumption, and resulted in quadratic decision boundaries. The left-hand panel of Figure 4.12 shows that QDA outperformed all of the

t -
distribution

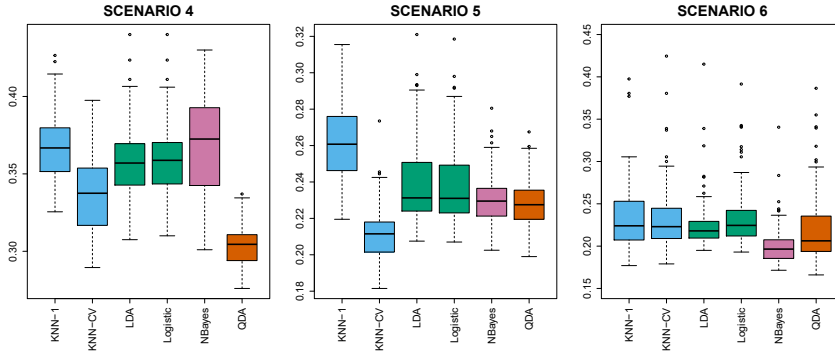


FIGURE 4.12. Boxplots of the test error rates for each of the non-linear scenarios described in the main text.

other approaches. The naive Bayes assumption of independent predictors is violated, so naive Bayes performs poorly.

Scenario 5: The data were generated from a normal distribution with uncorrelated predictors. Then the responses were sampled from the logistic function applied to a complicated non-linear function of the predictors. The center panel of Figure 4.12 shows that both QDA and naive Bayes gave slightly better results than the linear methods, while the much more flexible KNN-CV method gave the best results. But KNN with $K = 1$ gave the worst results out of all methods. This highlights the fact that even when the data exhibits a complex non-linear relationship, a non-parametric method such as KNN can still give poor results if the level of smoothness is not chosen correctly.

Scenario 6: The observations were generated from a normal distribution with a different diagonal covariance matrix for each class. However, the sample size was *very* small: just $n = 6$ in each class. Naive Bayes performed very well, because its assumptions are met. LDA and logistic regression performed poorly because the true decision boundary is non-linear, due to the unequal covariance matrices. QDA performed a bit worse than naive Bayes, because given the very small sample size, the former incurred too much variance in estimating the correlation between the predictors within each class. KNN's performance also suffered due to the very small sample size.

These six examples illustrate that no one method will dominate the others in every situation. When the true decision boundaries are linear, then the LDA and logistic regression approaches will tend to perform well. When the boundaries are moderately non-linear, QDA or naive Bayes may give better results. Finally, for much more complicated decision boundaries, a non-parametric approach such as KNN can be superior. But the level of smoothness for a non-parametric approach must be chosen carefully. In the next chapter we examine a number of approaches for choosing the correct level of smoothness and, in general, for selecting the best overall method.

Finally, recall from Chapter 3 that in the regression setting we can accommodate a non-linear relationship between the predictors and the response by performing regression using transformations of the predictors. A similar approach could be taken in the classification setting. For instance, we could

	Coefficient	Std. error	<i>t</i> -statistic	<i>p</i> -value
Intercept	73.60	5.13	14.34	0.00
workingday	1.27	1.78	0.71	0.48
temp	157.21	10.26	15.32	0.00
weathersit[cloudy/misty]	-12.89	1.96	-6.56	0.00
weathersit[light rain/snow]	-66.49	2.97	-22.43	0.00
weathersit[heavy rain/snow]	-109.75	76.67	-1.43	0.15

TABLE 4.10. Results for a least squares linear model fit to predict **bikers** in the Bikeshare data. The predictors **mnth** and **hr** are omitted from this table due to space constraints, and can be seen in Figure 4.13. For the qualitative variable **weathersit**, the baseline level corresponds to clear skies.

create a more flexible version of logistic regression by including X^2 , X^3 , and even X^4 as predictors. This may or may not improve logistic regression's performance, depending on whether the increase in variance due to the added flexibility is offset by a sufficiently large reduction in bias. We could do the same for LDA. If we added all possible quadratic terms and cross-products to LDA, the form of the model would be the same as the QDA model, although the parameter estimates would be different. This device allows us to move somewhere between an LDA and a QDA model.

4.6 Generalized Linear Models

In Chapter 3, we assumed that the response Y is quantitative, and explored the use of least squares linear regression to predict Y . Thus far in this chapter, we have instead assumed that Y is qualitative. However, we may sometimes be faced with situations in which Y is neither qualitative nor quantitative, and so neither linear regression from Chapter 3 nor the classification approaches covered in this chapter is applicable.

As a concrete example, we consider the Bikeshare data set. The response is **bikers**, the number of hourly users of a bike sharing program in Washington, DC. This response value is neither qualitative nor quantitative: instead, it takes on non-negative integer values, or *counts*. We will consider predicting **bikers** using the covariates **mnth** (month of the year), **hr** (hour of the day, from 0 to 23), **workingday** (an indicator variable that equals 1 if it is neither a weekend nor a holiday), **temp** (the normalized temperature, in Celsius), and **weathersit** (a qualitative variable that takes on one of four possible values: clear; misty or cloudy; light rain or light snow; or heavy rain or heavy snow.)

In the analyses that follow, we will treat **mnth**, **hr**, and **weathersit** as qualitative variables.

4.6.1 Linear Regression on the Bikeshare Data

To begin, we consider predicting **bikers** using linear regression. The results are shown in Table 4.10.

We see, for example, that a progression of weather from clear to cloudy results in, on average, 12.89 fewer bikers per hour; however, if the weather progresses further to rain or snow, then this further results in 53.60 fewer bikers per hour. Figure 4.13 displays the coefficients associated with **mnth**

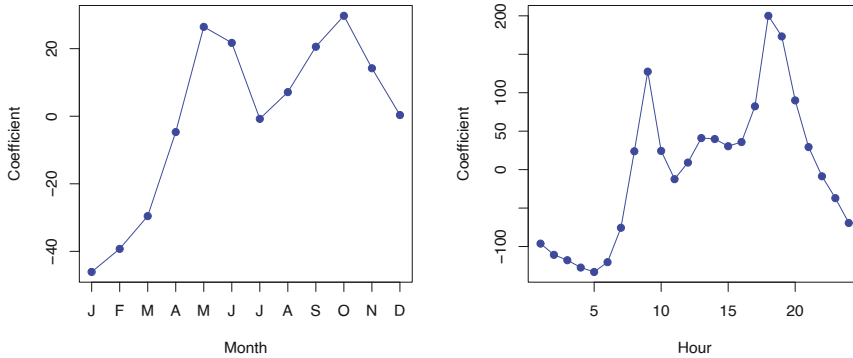


FIGURE 4.13. A least squares linear regression model was fit to predict **bikers** in the **Bikeshare** data set. Left: The coefficients associated with the month of the year. Bike usage is highest in the spring and fall, and lowest in the winter. Right: The coefficients associated with the hour of the day. Bike usage is highest during peak commute times, and lowest overnight.

and the coefficients associated with **hr**. We see that bike usage is highest in the spring and fall, and lowest during the winter months. Furthermore, bike usage is greatest around rush hour (9 AM and 6 PM), and lowest overnight. Thus, at first glance, fitting a linear regression model to the **Bikeshare** data set seems to provide reasonable and intuitive results.

But upon more careful inspection, some issues become apparent. For example, 9.6% of the fitted values in the **Bikeshare** data set are negative: that is, the linear regression model predicts a *negative* number of users during 9.6% of the hours in the data set. This calls into question our ability to perform meaningful predictions on the data, and it also raises concerns about the accuracy of the coefficient estimates, confidence intervals, and other outputs of the regression model.

Furthermore, it is reasonable to suspect that when the expected value of **bikers** is small, the variance of **bikers** should be small as well. For instance, at 2 AM during a heavy December snow storm, we expect that extremely few people will use a bike, and moreover that there should be little variance associated with the number of users during those conditions. This is borne out in the data: between 1 AM and 4 AM, in December, January, and February, when it is raining, there are 5.05 users, on average, with a standard deviation of 3.73. By contrast, between 7 AM and 10 AM, in April, May, and June, when skies are clear, there are 243.59 users, on average, with a standard deviation of 131.7. The mean-variance relationship is displayed in the left-hand panel of Figure 4.14. This is a major violation of the assumptions of a linear model, which state that $Y = \sum_{j=1}^p X_j \beta_j + \epsilon$, where ϵ is a mean-zero error term with variance σ^2 that is *constant*, and not a function of the covariates. Therefore, the heteroscedasticity of the data calls into question the suitability of a linear regression model.

Finally, the response **bikers** is integer-valued. But under a linear model, $Y = \beta_0 + \sum_{j=1}^p X_j \beta_j + \epsilon$, where ϵ is a continuous-valued error term. This means that in a linear model, the response Y is necessarily continuous-valued (quantitative). Thus, the integer nature of the response **bikers** suggests that a linear regression model is not entirely satisfactory for this data set.

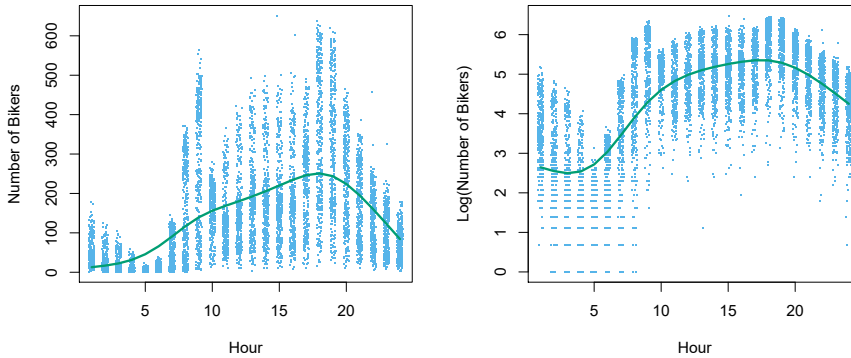


FIGURE 4.14. Left: On the **Bikeshare** dataset, the number of bikers is displayed on the y-axis, and the hour of the day is displayed on the x-axis. Jitter was applied for ease of visualization. For the most part, as the mean number of bikers increases, so does the variance in the number of bikers. A smoothing spline fit is shown in green. Right: The log of the number of bikers is now displayed on the y-axis.

Some of the problems that arise when fitting a linear regression model to the **Bikeshare** data can be overcome by transforming the response; for instance, we can fit the model

$$\log(Y) = \sum_{j=1}^p X_j \beta_j + \epsilon.$$

Transforming the response avoids the possibility of negative predictions, and it overcomes much of the heteroscedasticity in the untransformed data, as is shown in the right-hand panel of Figure 4.14. However, it is not quite a satisfactory solution, since predictions and inference are made in terms of the log of the response, rather than the response. This leads to challenges in interpretation, e.g. “a one-unit increase in X_j is associated with an increase in the mean of the log of Y by an amount β_j ”. Furthermore, a log transformation of the response cannot be applied in settings where the response can take on a value of 0. Thus, while fitting a linear model to a transformation of the response may be an adequate approach for some count-valued data sets, it often leaves something to be desired. We will see in the next section that a Poisson regression model provides a much more natural and elegant approach for this task.

4.6.2 Poisson Regression on the Bikeshare Data

To overcome the inadequacies of linear regression for analyzing the **Bikeshare** data set, we will make use of an alternative approach, called *Poisson regression*. Before we can talk about Poisson regression, we must first introduce the *Poisson distribution*.

Suppose that a random variable Y takes on nonnegative integer values, i.e. $Y \in \{0, 1, 2, \dots\}$. If Y follows the Poisson distribution, then

$$\Pr(Y = k) = \frac{e^{-\lambda} \lambda^k}{k!} \quad \text{for } k = 0, 1, 2, \dots \quad (4.35)$$



Poisson
regression
Poisson
distribution

Here, $\lambda > 0$ is the expected value of Y , i.e. $E(Y)$. It turns out that λ also equals the variance of Y , i.e. $\lambda = E(Y) = \text{Var}(Y)$. This means that if Y follows the Poisson distribution, then the larger the mean of Y , the larger its variance. (In (4.35), the notation $k!$, pronounced “ k factorial”, is defined as $k! = k \times (k-1) \times (k-2) \times \dots \times 3 \times 2 \times 1$.)

The Poisson distribution is typically used to model *counts*; this is a natural choice for a number of reasons, including the fact that counts, like the Poisson distribution, take on nonnegative integer values. To see how we might use the Poisson distribution in practice, let Y denote the number of users of the bike sharing program during a particular hour of the day, under a particular set of weather conditions, and during a particular month of the year. We might model Y as a Poisson distribution with mean $E(Y) = \lambda = 5$. This means that the probability of no users during this particular hour is $\Pr(Y = 0) = \frac{e^{-5}5^0}{0!} = e^{-5} = 0.0067$ (where $0! = 1$ by convention). The probability that there is exactly one user is $\Pr(Y = 1) = \frac{e^{-5}5^1}{1!} = 5e^{-5} = 0.034$, the probability of two users is $\Pr(Y = 2) = \frac{e^{-5}5^2}{2!} = 0.084$, and so on.

Of course, in reality, we expect the mean number of users of the bike sharing program, $\lambda = E(Y)$, to vary as a function of the hour of the day, the month of the year, the weather conditions, and so forth. So rather than modeling the number of bikers, Y , as a Poisson distribution with a fixed mean value like $\lambda = 5$, we would like to allow the mean to vary as a function of the covariates. In particular, we consider the following model for the mean $\lambda = E(Y)$, which we now write as $\lambda(X_1, \dots, X_p)$ to emphasize that it is a function of the covariates X_1, \dots, X_p :

$$\log(\lambda(X_1, \dots, X_p)) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p \quad (4.36)$$

or equivalently

$$\lambda(X_1, \dots, X_p) = e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}. \quad (4.37)$$

Here, $\beta_0, \beta_1, \dots, \beta_p$ are parameters to be estimated. Together, (4.35) and (4.36) define the Poisson regression model. Notice that in (4.36), we take the *log* of $\lambda(X_1, \dots, X_p)$ to be linear in X_1, \dots, X_p , rather than having $\lambda(X_1, \dots, X_p)$ itself be linear in X_1, \dots, X_p ; this ensures that $\lambda(X_1, \dots, X_p)$ takes on nonnegative values for all values of the covariates.

To estimate the coefficients $\beta_0, \beta_1, \dots, \beta_p$, we use the same maximum likelihood approach that we adopted for logistic regression in Section 4.3.2. Specifically, given n independent observations from the Poisson regression model, the likelihood takes the form

$$\ell(\beta_0, \beta_1, \dots, \beta_p) = \prod_{i=1}^n \frac{e^{-\lambda(x_i)} \lambda(x_i)^{y_i}}{y_i!}, \quad (4.38)$$

where $\lambda(x_i) = e^{\beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip}}$, due to (4.37). We estimate the coefficients that maximize the likelihood $\ell(\beta_0, \beta_1, \dots, \beta_p)$, i.e. that make the observed data as likely as possible.

We now fit a Poisson regression model to the **Bikeshare** data set. The results are shown in Table 4.11 and Figure 4.15. Qualitatively, the results are similar to those from linear regression in Section 4.6.1. We again see that bike usage is highest in the spring and fall and during rush hour,

	Coefficient	Std. error	z-statistic	p-value
Intercept	4.12	0.01	683.96	0.00
workingday	0.01	0.00	7.5	0.00
temp	0.79	0.01	68.43	0.00
weathersit[cloudy/misty]	-0.08	0.00	-34.53	0.00
weathersit[light rain/snow]	-0.58	0.00	-141.91	0.00
weathersit[heavy rain/snow]	-0.93	0.17	-5.55	0.00

TABLE 4.11. Results for a Poisson regression model fit to predict **bikers** in the **Bikeshare** data. The predictors **mnth** and **hr** are omitted from this table due to space constraints, and can be seen in Figure 4.15. For the qualitative variable **weathersit**, the baseline corresponds to clear skies.

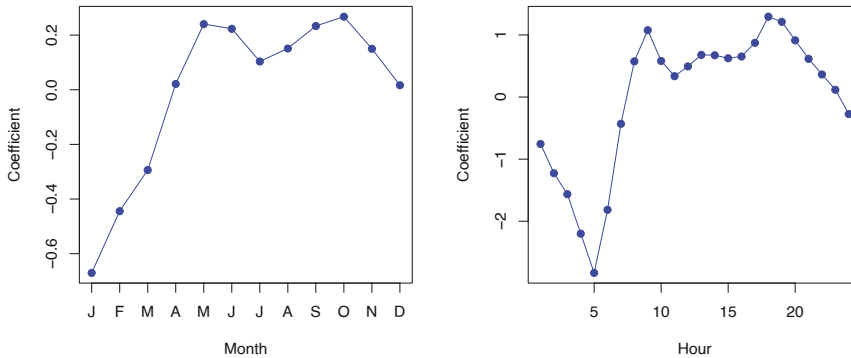


FIGURE 4.15. A Poisson regression model was fit to predict **bikers** in the **Bikeshare** data set. Left: The coefficients associated with the month of the year. Bike usage is highest in the spring and fall, and lowest in the winter. Right: The coefficients associated with the hour of the day. Bike usage is highest during peak commute times, and lowest overnight.

and lowest during the winter and in the early morning hours. Moreover, bike usage increases as the temperature increases, and decreases as the weather worsens. Interestingly, the coefficient associated with **workingday** is statistically significant under the Poisson regression model, but not under the linear regression model.

Some important distinctions between the Poisson regression model and the linear regression model are as follows:

- *Interpretation:* To interpret the coefficients in the Poisson regression model, we must pay close attention to (4.37), which states that an increase in X_j by one unit is associated with a change in $E(Y) = \lambda$ by a factor of $\exp(\beta_j)$. For example, a change in weather from clear to cloudy skies is associated with a change in mean bike usage by a factor of $\exp(-0.08) = 0.923$, i.e. on average, only 92.3% as many people will use bikes when it is cloudy relative to when it is clear. If the weather worsens further and it begins to rain, then the mean bike usage will further change by a factor of $\exp(-0.5) = 0.607$, i.e. on average only 60.7% as many people will use bikes when it is rainy relative to when it is cloudy.

- *Mean-variance relationship*: As mentioned earlier, under the Poisson model, $\lambda = E(Y) = \text{Var}(Y)$. Thus, by modeling bike usage with a Poisson regression, we implicitly assume that mean bike usage in a given hour equals the variance of bike usage during that hour. By contrast, under a linear regression model, the variance of bike usage always takes on a constant value. Recall from Figure 4.14 that in the **Bikeshare** data, when biking conditions are favorable, both the mean *and* the variance in bike usage are much higher than when conditions are unfavorable. Thus, the Poisson regression model is able to handle the mean-variance relationship seen in the **Bikeshare** data in a way that the linear regression model is not.⁵
- *nonnegative fitted values*: There are no negative predictions using the Poisson regression model. This is because the Poisson model itself only allows for nonnegative values; see (4.35). By contrast, when we fit a linear regression model to the **Bikeshare** data set, almost 10% of the predictions were negative.

overdispersion

4.6.3 Generalized Linear Models in Greater Generality

We have now discussed three types of regression models: linear, logistic and Poisson. These approaches share some common characteristics:



1. Each approach uses predictors X_1, \dots, X_p to predict a response Y . We assume that, conditional on X_1, \dots, X_p , Y belongs to a certain family of distributions. For linear regression, we typically assume that Y follows a Gaussian or normal distribution. For logistic regression, we assume that Y follows a Bernoulli distribution. Finally, for Poisson regression, we assume that Y follows a Poisson distribution.
2. Each approach models the mean of Y as a function of the predictors. In linear regression, the mean of Y takes the form

$$E(Y|X_1, \dots, X_p) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p, \quad (4.39)$$

i.e. it is a linear function of the predictors. For logistic regression, the mean instead takes the form

$$\begin{aligned} E(Y|X_1, \dots, X_p) &= \Pr(Y = 1|X_1, \dots, X_p) \\ &= \frac{e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}, \end{aligned} \quad (4.40)$$

while for Poisson regression it takes the form

$$E(Y|X_1, \dots, X_p) = \lambda(X_1, \dots, X_p) = e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}. \quad (4.41)$$

Equations (4.39)–(4.41) can be expressed using a *link function*, η , which

link function

⁵In fact, the variance in the **Bikeshare** data appears to be much higher than the mean, a situation referred to as *overdispersion*. This causes the Z-values to be inflated in Table 4.11. A more careful analysis should account for this overdispersion to obtain more accurate Z-values, and there are a variety of methods for doing this. But they are beyond the scope of this book.

applies a transformation to $E(Y|X_1, \dots, X_p)$ so that the transformed mean is a linear function of the predictors. That is,

$$\eta(E(Y|X_1, \dots, X_p)) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p. \quad (4.42)$$

The link functions for linear, logistic and Poisson regression are $\eta(\mu) = \mu$, $\eta(\mu) = \log(\mu/(1 - \mu))$, and $\eta(\mu) = \log(\mu)$, respectively.

The Gaussian, Bernoulli and Poisson distributions are all members of a wider class of distributions, known as the *exponential family*. Other well-known members of this family are the *exponential* distribution, the *Gamma* distribution, and the *negative binomial* distribution. In general, we can perform a regression by modeling the response Y as coming from a particular member of the exponential family, and then transforming the mean of the response so that the transformed mean is a linear function of the predictors via (4.42). Any regression approach that follows this very general recipe is known as a *generalized linear model* (GLM). Thus, linear regression, logistic regression, and Poisson regression are three examples of GLMs. Other examples not covered here include *Gamma regression* and *negative binomial regression*.

exponential
family
exponential
Gamma
negative
binomial

generalized
linear model

4.7 Lab: Logistic Regression, LDA, QDA, and KNN

4.7.1 The Stock Market Data

In this lab we will examine the `Smarket` data, which is part of the `ISLP` library. This data set consists of percentage returns for the S&P 500 stock index over 1,250 days, from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, `Lag1` through `Lag5`. We have also recorded `Volume` (the number of shares traded on the previous day, in billions), `Today` (the percentage return on the date in question) and `Direction` (whether the market was `Up` or `Down` on this date).

We start by importing our libraries at this top level; these are all imports we have seen in previous labs.

```
In [1]: import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                        summarize)
```

We also collect together the new imports needed for this lab.

```
In [2]: from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
    (LinearDiscriminantAnalysis as LDA,
     QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

Now we are ready to load the `Smarket` data.

```
In [3]: Smarket = load_data('Smarket')
Smarket
```

This gives a truncated listing of the data, which we do not show here. We can see what the variable names are.

```
In [4]: Smarket.columns
```

```
Out[4]: Index(['Year', 'Lag1', 'Lag2', 'Lag3', 'Lag4', 'Lag5', 'Volume',
              'Today', 'Direction'],
              dtype='object')
```

We compute the correlation matrix using the `corr()` method for data frames, which produces a matrix that contains all of the pairwise correlations among the variables. (We suppress the output here.) The `pandas` library does not report a correlation for the `Direction` variable because it is qualitative. `.corr()`

```
In [5]: Smarket.corr()
```

As one would expect, the correlations between the lagged return variables and today's return are close to zero. The only substantial correlation is between `Year` and `Volume`. By plotting the data we see that `Volume` is increasing over time. In other words, the average number of shares traded daily increased from 2001 to 2005.

```
In [6]: Smarket.plot(y='Volume');
```

4.7.2 Logistic Regression

Next, we will fit a logistic regression model in order to predict `Direction` using `Lag1` through `Lag5` and `Volume`. The `sm.GLM()` function fits *generalized linear models*, a class of models that includes logistic regression. Alternatively, the function `sm.Logit()` fits a logistic regression model directly. The syntax of `sm.GLM()` is similar to that of `sm.OLS()`, except that we must pass in the argument `family=sm.families.Binomial()` in order to tell `statsmodels` to run a logistic regression rather than some other type of generalized linear model.

`sm.GLM()`
generalized
linear model

```
In [7]: allvars = Smarket.columns.drop(['Today', 'Direction', 'Year'])
design = MS(allvars)
X = design.fit_transform(Smarket)
y = Smarket.Direction == 'Up'
glm = sm.GLM(y,
             X,
             family=sm.families.Binomial())
results = glm.fit()
summarize(results)
```

```
Out[7]:
```

	coef	std err	z	P> z
intercept	-0.1260	0.241	-0.523	0.601
Lag1	-0.0731	0.050	-1.457	0.145
Lag2	-0.0423	0.050	-0.845	0.398
Lag3	0.0111	0.050	0.222	0.824
Lag4	0.0094	0.050	0.187	0.851
Lag5	0.0103	0.050	0.208	0.835
Volume	0.1354	0.158	0.855	0.392

The smallest p -value here is associated with **Lag1**. The negative coefficient for this predictor suggests that if the market had a positive return yesterday, then it is less likely to go up today. However, at a value of 0.15, the p -value is still relatively large, and so there is no clear evidence of a real association between **Lag1** and **Direction**.

We use the `params` attribute of `results` in order to access just the coefficients for this fitted model.

```
In [8]: results.params
```

```
Out[8]: intercept    -0.126000
Lag1                -0.073074
Lag2                -0.042301
Lag3                 0.011085
Lag4                 0.009359
Lag5                 0.010313
Volume              0.135441
dtype: float64
```

Likewise we can use the `pvalues` attribute to access the p -values for the coefficients (not shown).

```
In [9]: results.pvalues
```

The `predict()` method of `results` can be used to predict the probability that the market will go up, given values of the predictors. This method returns predictions on the probability scale. If no data set is supplied to the `predict()` function, then the probabilities are computed for the training data that was used to fit the logistic regression model. As with linear regression, one can pass an optional `exog` argument consistent with a design matrix if desired. Here we have printed only the first ten probabilities.

```
In [10]: probs = results.predict()
probs[:10]
```

```
Out[10]: array([0.5070841, 0.4814679, 0.4811388, 0.5152223, 0.5107812,
0.5069565, 0.4926509, 0.5092292, 0.5176135, 0.4888378])
```

In order to make a prediction as to whether the market will go up or down on a particular day, we must convert these predicted probabilities into class labels, **Up** or **Down**. The following two commands create a vector of class predictions based on whether the predicted probability of a market increase is greater than or less than 0.5.

```
In [11]: labels = np.array(['Down']*1250)
labels[probs>0.5] = "Up"
```

The `confusion_table()` function from the `ISLP` package summarizes these predictions, showing how many observations were correctly or incorrectly classified. Our function, which is adapted from a similar function in the module `sklearn.metrics`, transposes the resulting matrix and includes row and column labels. The `confusion_table()` function takes as first argument the predicted labels, and second argument the true labels.

`confusion_`
`table()`

```
In [12]: confusion_table(labels, Smarket.Direction)
```

```
Out[12]:      Truth   Down   Up
Predicted
      Down   145   141
      Up    457   507
```

The diagonal elements of the confusion matrix indicate correct predictions, while the off-diagonals represent incorrect predictions. Hence our model correctly predicted that the market would go up on 507 days and that it would go down on 145 days, for a total of $507 + 145 = 652$ correct predictions. The `np.mean()` function can be used to compute the fraction of days for which the prediction was correct. In this case, logistic regression correctly predicted the movement of the market 52.2% of the time.

```
In [13]: (507+145)/1250, np.mean(labels == Smarket.Direction)
```

```
Out[13]: (0.5216, 0.5216)
```

At first glance, it appears that the logistic regression model is working a little better than random guessing. However, this result is misleading because we trained and tested the model on the same set of 1,250 observations. In other words, $100 - 52.2 = 47.8\%$ is the *training* error rate. As we have seen previously, the training error rate is often overly optimistic — it tends to underestimate the test error rate. In order to better assess the accuracy of the logistic regression model in this setting, we can fit the model using part of the data, and then examine how well it predicts the *held out* data. This will yield a more realistic error rate, in the sense that in practice we will be interested in our model's performance not on the data that we used to fit the model, but rather on days in the future for which the market's movements are unknown.

To implement this strategy, we first create a Boolean vector corresponding to the observations from 2001 through 2004. We then use this vector to create a held out data set of observations from 2005.

```
In [14]: train = (Smarket.Year < 2005)
Smarket_train = Smarket.loc[train]
Smarket_test = Smarket.loc[~train]
Smarket_test.shape
```

```
Out[14]: (252, 9)
```

The object `train` is a vector of 1,250 elements, corresponding to the observations in our data set. The elements of the vector that correspond to observations that occurred before 2005 are set to `True`, whereas those that correspond to observations in 2005 are set to `False`. Hence `train` is a *boolean* array, since its elements are `True` and `False`. Boolean arrays can be used to obtain a subset of the rows or columns of a data frame using the

`loc` method. For instance, the command `Smarket.loc[train]` would pick out a submatrix of the stock market data set, corresponding only to the dates before 2005, since those are the ones for which the elements of `train` are `True`. The `~` symbol can be used to negate all of the elements of a Boolean vector. That is, `~train` is a vector similar to `train`, except that the elements that are `True` in `train` get swapped to `False` in `~train`, and vice versa. Therefore, `Smarket.loc[~train]` yields a subset of the rows of the data frame of the stock market data containing only the observations for which `train` is `False`. The output above indicates that there are 252 such observations.

We now fit a logistic regression model using only the subset of the observations that correspond to dates before 2005. We then obtain predicted probabilities of the stock market going up for each of the days in our test set — that is, for the days in 2005.

```
In [15]: X_train, X_test = X.loc[train], X.loc[~train]
y_train, y_test = y.loc[train], y.loc[~train]
glm_train = sm.GLM(y_train,
                    X_train,
                    family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
```

Notice that we have trained and tested our model on two completely separate data sets: training was performed using only the dates before 2005, and testing was performed using only the dates in 2005.

Finally, we compare the predictions for 2005 to the actual movements of the market over that time period. We will first store the test and training labels (recall `y_test` is binary).

```
In [16]: D = Smarket.Direction
L_train, L_test = D.loc[train], D.loc[~train]
```

Now we threshold the fitted probability at 50% to form our predicted labels.

```
In [17]: labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_table(labels, L_test)
```

```
Out[17]:      Truth   Down   Up
Predicted
        Down    77    97
         Up    34    44
```

The test accuracy is about 48% while the error rate is about 52%

```
In [18]: np.mean(labels == L_test), np.mean(labels != L_test)
```

```
Out[18]: (0.4802, 0.5198)
```

The `!=` notation means *not equal to*, and so the last command computes the test set error rate. The results are rather disappointing: the test error rate is 52%, which is worse than random guessing! Of course this result is not all that surprising, given that one would not generally expect to be able to use previous days' returns to predict future market performance. (After all, if it were possible to do so, then the authors of this book would be out striking it rich rather than writing a statistics textbook.)

We recall that the logistic regression model had very underwhelming p -values associated with all of the predictors, and that the smallest p -value, though not very small, corresponded to `Lag1`. Perhaps by removing the variables that appear not to be helpful in predicting `Direction`, we can obtain a more effective model. After all, using predictors that have no relationship with the response tends to cause a deterioration in the test error rate (since such predictors cause an increase in variance without a corresponding decrease in bias), and so removing such predictors may in turn yield an improvement. Below we refit the logistic regression using just `Lag1` and `Lag2`, which seemed to have the highest predictive power in the original logistic regression model.

```
In [19]: model = MS(['Lag1', 'Lag2']).fit(Smarket)
X = model.transform(Smarket)
X_train, X_test = X.loc[train], X.loc[~train]
glm_train = sm.GLM(y_train,
                    X_train,
                    family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_table(labels, L_test)
```

```
Out[19]:      Truth  Down  Up
Predicted
Down      35    35
Up       76   106
```

Let's evaluate the overall accuracy as well as the accuracy within the days when logistic regression predicts an increase.

```
In [20]: (35+106)/252, 106/(106+76)
```

```
Out[20]: (0.5595, 0.5824)
```

Now the results appear to be a little better: 56% of the daily movements have been correctly predicted. It is worth noting that in this case, a much simpler strategy of predicting that the market will increase every day will also be correct 56% of the time! Hence, in terms of overall error rate, the logistic regression method is no better than the naive approach. However, the confusion matrix shows that on days when logistic regression predicts an increase in the market, it has a 58% accuracy rate. This suggests a possible trading strategy of buying on days when the model predicts an increasing market, and avoiding trades on days when a decrease is predicted. Of course one would need to investigate more carefully whether this small improvement was real or just due to random chance.

Suppose that we want to predict the returns associated with particular values of `Lag1` and `Lag2`. In particular, we want to predict `Direction` on a day when `Lag1` and `Lag2` equal 1.2 and 1.1, respectively, and on a day when they equal 1.5 and -0.8 . We do this using the `predict()` function.

```
In [21]: newdata = pd.DataFrame({'Lag1':[1.2, 1.5],
                                'Lag2':[1.1, -0.8]});
```

```
newX = model.transform(newdata)
results.predict(newX)
```

```
Out[21]: 0    0.4791
         1    0.4961
         dtype: float64
```

4.7.3 Linear Discriminant Analysis

We begin by performing LDA on the `Smarket` data, using the function `LinearDiscriminantAnalysis()`, which we have abbreviated `LDA()`. We fit the model using only the observations before 2005.

Linear
Discriminant
Analysis()

```
In [22]: lda = LDA(store_covariance=True)
```

Since the `LDA` estimator automatically adds an intercept, we should remove the column corresponding to the intercept in both `X_train` and `X_test`. We can also directly use the labels rather than the Boolean vectors `y_train`.

```
In [23]: X_train, X_test = [M.drop(columns=['intercept'])
                           for M in [X_train, X_test]]
         lda.fit(X_train, L_train)
```

```
Out[23]: LinearDiscriminantAnalysis(store_covariance=True)
```

Here we have used the list comprehensions introduced in Section 3.6.4. Looking at our first line above, we see that the right-hand side is a list of length two. This is because the code `for M in [X_train, X_test]` iterates over a list of length two. While here we loop over a list, the list comprehension method works when looping over any iterable object. We then apply the `drop()` method to each element in the iteration, collecting the result in a list. The left-hand side tells `Python` to unpack this list of length two, assigning its elements to the variables `X_train` and `X_test`. Of course, this overwrites the previous values of `X_train` and `X_test`.

.drop()

Having fit the model, we can extract the means in the two classes with the `means_` attribute. These are the average of each predictor within each class, and are used by LDA as estimates of μ_k . These suggest that there is a tendency for the previous 2 days' returns to be negative on days when the market increases, and a tendency for the previous days' returns to be positive on days when the market declines.

```
In [24]: lda.means_
```

```
Out[24]: array([[ 0.04,  0.03],
               [-0.04, -0.03]])
```

The estimated prior probabilities are stored in the `priors_` attribute. The package `sklearn` typically uses this trailing `_` to denote a quantity estimated when using the `fit()` method. We can be sure of which entry corresponds to which label by looking at the `classes_` attribute.

```
In [25]: lda.classes_
```

```
Out[25]: array(['Down', 'Up'], dtype='<U4')
```

The LDA output indicates that $\hat{\pi}_{\text{Down}} = 0.492$ and $\hat{\pi}_{\text{Up}} = 0.508$.

```
In [26]: lda.priors_
```

```
Out[26]: array([0.492, 0.508])
```

The linear discriminant vectors can be found in the `scalings_` attribute:

```
In [27]: lda.scalings_
```

```
Out[27]: array([[ -0.642],
               [ -0.513]])
```

These values provide the linear combination of `Lag1` and `Lag2` that are used to form the LDA decision rule. In other words, these are the multipliers of the elements of $X = x$ in (4.24). If $-0.64 \times \text{Lag1} - 0.51 \times \text{Lag2}$ is large, then the LDA classifier will predict a market increase, and if it is small, then the LDA classifier will predict a market decline.

```
In [28]: lda_pred = lda.predict(X_test)
```

As we observed in our comparison of classification methods (Section 4.5), the LDA and logistic regression predictions are almost identical.

```
In [29]: confusion_table(lda_pred, L_test)
```

```
Out[29]:      Truth   Down   Up
Predicted
      Down    35    35
      Up     76   106
```

We can also estimate the probability of each class for each point in a training set. Applying a 50% threshold to the posterior probabilities of being in class one allows us to recreate the predictions contained in `lda_pred`.

```
In [30]: lda_prob = lda.predict_proba(X_test)
np.all(
    np.where(lda_prob[:,1] >= 0.5, 'Up','Down') == lda_pred
)
```

```
Out[30]: True
```

Above, we used the `np.where()` function that creates an array with value 'Up' for indices where the second column of `lda_prob` (the estimated posterior probability of 'Up') is greater than 0.5. For problems with more than two classes the labels are chosen as the class whose posterior probability is highest:

```
In [31]: np.all(
    [lda.classes_[i] for i in np.argmax(lda_prob, 1)] ==
    lda_pred
)
```

```
Out[31]: True
```

If we wanted to use a posterior probability threshold other than 50% in order to make predictions, then we could easily do so. For instance, suppose that we wish to predict a market decrease only if we are very certain that the

`np.where()`

market will indeed decrease on that day — say, if the posterior probability is at least 90%. We know that the first column of `lda_prob` corresponds to the label `Down` after having checked the `classes_` attribute, hence we use the column index 0 rather than 1 as we did above.

```
In [32]: np.sum(lda_prob[:,0] > 0.9)
```

```
Out [32]: 0
```

No days in 2005 meet that threshold! In fact, the greatest posterior probability of decrease in all of 2005 was 52.02%.

The LDA classifier above is the first classifier from the `sklearn` library. We will use several other objects from this library. The objects follow a common structure that simplifies tasks such as cross-validation, which we will see in Chapter 5. Specifically, the methods first create a generic classifier without referring to any data. This classifier is then fit to data with the `fit()` method and predictions are always produced with the `predict()` method. This pattern of first instantiating the classifier, followed by fitting it, and then producing predictions is an explicit design choice of `sklearn`. This uniformity makes it possible to cleanly copy the classifier so that it can be fit on different data; e.g. different training sets arising in cross-validation. This standard pattern also allows for a predictable formation of workflows.

4.7.4 Quadratic Discriminant Analysis

We will now fit a QDA model to the `Smarket` data. QDA is implemented via `QuadraticDiscriminantAnalysis()` in the `sklearn` package, which we abbreviate to `QDA()`. The syntax is very similar to `LDA()`.

Quadratic
Discriminant
Analysis()

```
In [33]: qda = QDA(store_covariance=True)
         qda.fit(X_train, L_train)
```

```
Out [33]: QuadraticDiscriminantAnalysis(store_covariance=True)
```

The `QDA()` function will again compute `means_` and `priors_`.

```
In [34]: qda.means_, qda.priors_
```

```
Out [34]: (array([[ 0.04279022,  0.03389409],
                  [-0.03954635, -0.03132544]]),
          array([0.49198397, 0.50801603]))
```

The `QDA()` classifier will estimate one covariance per class. Here is the estimated covariance in the first class:

```
In [35]: qda.covariance_[0]
```

```
Out [35]: array([[ 1.50662277, -0.03924806],
                  [-0.03924806,  1.53559498]])
```

The output contains the group means. But it does not contain the coefficients of the linear discriminants, because the QDA classifier involves a quadratic, rather than a linear, function of the predictors. The `predict()` function works in exactly the same fashion as for LDA.

```
In [36]: qda_pred = qda.predict(X_test)
         confusion_table(qda_pred, L_test)
```

```
Out[36]:      Truth    Down    Up
         Predicted
           Down     30     20
           Up      81    121
```

Interestingly, the QDA predictions are accurate almost 60% of the time, even though the 2005 data was not used to fit the model.

```
In [37]: np.mean(qda_pred == L_test)
```

```
Out[37]: 0.599
```

This level of accuracy is quite impressive for stock market data, which is known to be quite hard to model accurately. This suggests that the quadratic form assumed by QDA may capture the true relationship more accurately than the linear forms assumed by LDA and logistic regression. However, we recommend evaluating this method's performance on a larger test set before betting that this approach will consistently beat the market!

4.7.5 Naive Bayes

Next, we fit a naive Bayes model to the `Smarket` data. The syntax is similar to that of `LDA()` and `QDA()`. By default, this implementation `GaussianNB()` of the naive Bayes classifier models each quantitative feature using a Gaussian distribution. However, a kernel density method can also be used to estimate the distributions.

`GaussianNB()`

```
In [38]: NB = GaussianNB()
         NB.fit(X_train, L_train)
```

```
Out[38]: GaussianNB()
```

The classes are stored as `classes_`.

```
In [39]: NB.classes_
```

```
Out[39]: array(['Down', 'Up'], dtype='<U4')
```

The class prior probabilities are stored in the `class_prior_` attribute.

```
In [40]: NB.class_prior_
```

```
Out[40]: array([0.49, 0.51])
```

The parameters of the features can be found in the `theta_` and `var_` attributes. The number of rows is equal to the number of classes, while the number of columns is equal to the number of features. We see below that the mean for feature `Lag1` in the `Down` class is 0.043.

```
In [41]: NB.theta_
```

```
Out[41]: array([[ 0.043,  0.034],
                [-0.040, -0.031]])
```

Its variance is 1.503.

```
In [42]: NB.var_
```

```
Out[42]: array([[1.503, 1.532],
               [1.514, 1.487]])
```

How do we know the names of these attributes? We use `NB?` (or `?NB`).

We can easily verify the mean computation:

```
In [43]: X_train[L_train == 'Down'].mean()
```

```
Out[43]: Lag1    0.042790
         Lag2    0.033894
         dtype: float64
```

Similarly for the variance:

```
In [44]: X_train[L_train == 'Down'].var(ddof=0)
```

```
Out[44]: Lag1    1.503554
         Lag2    1.532467
         dtype: float64
```

The `GaussianNB()` function calculates variances using the $1/n$ formula.⁶ Since `NB()` is a classifier in the `sklearn` library, making predictions uses the same syntax as for `LDA()` and `QDA()` above.

```
In [45]: nb_labels = NB.predict(X_test)
         confusion_table(nb_labels, L_test)
```

```
Out[45]:   Truth   Down   Up
         Predicted
         Down    29    20
         Up     82   121
```

Naive Bayes performs well on these data, with accurate predictions over 59% of the time. This is slightly worse than QDA, but much better than LDA.

As for LDA, the `predict_proba()` method estimates the probability that each observation belongs to a particular class.

```
In [46]: NB.predict_proba(X_test)[:5]
```

```
Out[46]: array([[0.4873, 0.5127],
               [0.4762, 0.5238],
               [0.4653, 0.5347],
               [0.4748, 0.5252],
               [0.4902, 0.5098]])
```

4.7.6 K-Nearest Neighbors

We will now perform KNN using the `KNeighborsClassifier()` function. This

`KNeighborsClassifier()`

⁶There are two formulas for computing the sample variance of n observations x_1, \dots, x_n : $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$ and $\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ where \bar{x} is the sample mean. In most cases the distinction is not important.

function works similarly to the other model-fitting functions that we have encountered thus far.

As is the case for LDA and QDA, we fit the classifier using the `fit` method. New predictions are formed using the `predict` method of the object returned by `fit()`.

```
In [47]: knn1 = KNeighborsClassifier(n_neighbors=1)
knn1.fit(X_train, L_train)
knn1_pred = knn1.predict(X_test)
confusion_table(knn1_pred, L_test)
```

```
Out[47]:      Truth    Down  Up
Predicted
      Down    43   58
      Up     68   83
```

The results using $K = 1$ are not very good, since only 50% of the observations are correctly predicted. Of course, it may be that $K = 1$ results in an overly-flexible fit to the data.

```
In [48]: (83+43)/252, np.mean(knn1_pred == L_test)
```

```
Out[48]: (0.5, 0.5)
```

We repeat the analysis below using $K = 3$.

```
In [49]: knn3 = KNeighborsClassifier(n_neighbors=3)
knn3_pred = knn3.fit(X_train, L_train).predict(X_test)
np.mean(knn3_pred == L_test)
```

```
Out[49]: 0.532
```

The results have improved slightly. But increasing K further provides no further improvements. It appears that for these data, and this train/test split, QDA gives the best results of the methods that we have examined so far.

KNN does not perform well on the `Smarket` data, but it often does provide impressive results. As an example we will apply the KNN approach to the `Caravan` data set, which is part of the `ISLP` library. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is `Purchase`, which indicates whether or not a given individual purchases a caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

```
In [50]: Caravan = load_data('Caravan')
Purchase = Caravan.Purchase
Purchase.value_counts()
```

```
Out[50]: No      5474
Yes       348
Name: Purchase, dtype: int64
```

The method `value_counts()` takes a `pd.Series` or `pd.DataFrame` and returns a `pd.Series` with the corresponding counts for each unique element. In this case `Purchase` has only `Yes` and `No` values and returns how many values of each there are.

```
In [51]: 348 / 5822
```

```
Out [51]: 0.0598
```

Our features will include all columns except `Purchase`.

```
In [52]: feature_df = Caravan.drop(columns=['Purchase'])
```

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the *distance* between the observations, and hence on the KNN classifier, than variables that are on a small scale. For instance, imagine a data set that contains two variables, `salary` and `age` (measured in dollars and years, respectively). As far as KNN is concerned, a difference of 1,000 USD in salary is enormous compared to a difference of 50 years in age. Consequently, `salary` will drive the KNN classification results, and `age` will have almost no effect. This is contrary to our intuition that a salary difference of 1,000 USD is quite small compared to an age difference of 50 years. Furthermore, the importance of scale to the KNN classifier leads to another issue: if we measured `salary` in Japanese yen, or if we measured `age` in minutes, then we'd get quite different classification results from what we get if these two variables are measured in dollars and years.

A good way to handle this problem is to *standardize* the data so that all variables are given a mean of zero and a standard deviation of one. Then all variables will be on a comparable scale. This is accomplished using the `StandardScaler()` transformation.

standardize

```
In [53]: scaler = StandardScaler(with_mean=True,
                                with_std=True,
                                copy=True)
```

Standard
Scaler()

The argument `with_mean` indicates whether or not we should subtract the mean, while `with_std` indicates whether or not we should scale the columns to have standard deviation of 1 or not. Finally, the argument `copy=True` indicates that we will always copy data, rather than trying to do calculations in place where possible.

This transformation can be fit and then applied to arbitrary data. In the first line below, the parameters for the scaling are computed and stored in `scaler`, while the second line actually constructs the standardized set of features.

```
In [54]: scaler.fit(feature_df)
X_std = scaler.transform(feature_df)
```

Now every column of `feature_std` below has a standard deviation of one and a mean of zero.

```
In [55]: feature_std = pd.DataFrame(
            X_std,
            columns=feature_df.columns);
feature_std.std()
```

```
Out [55]: MOSTYPE      1.000086
          MAANTHUI     1.000086
```

```

MGEMOMV      1.000086
MGEMLEEF      1.000086
MOSHOOFD      1.000086
...
AZEILPL       1.000086
APLEZIER      1.000086
AFIETS        1.000086
AINBOED       1.000086
ABYSTAND      1.000086
Length: 85, dtype: float64

```

Notice that the standard deviations are not quite 1 here; this is again due to some procedures using the $1/n$ convention for variances (in this case `scaler()`), while others use $1/(n-1)$ (the `std()` method). See the footnote on page 183. In this case it does not matter, as long as the variables are all on the same scale.

Using the function `train_test_split()` we now split the observations into a test set, containing 1000 observations, and a training set containing the remaining observations. The argument `random_state=0` ensures that we get the same split each time we rerun the code.

```

In [56]: (X_train,
          X_test,
          y_train,
          y_test) = train_test_split(feature_std,
                                     Purchase,
                                     test_size=1000,
                                     random_state=0)

```

`?train_test_split` reveals that the non-keyword arguments can be `lists`, `arrays`, `pandas dataframes` etc that all have the same length (`shape[0]`) and hence are *indexable*. In this case they are the dataframe `feature_std` and the response variable `Purchase`. We fit a KNN model on the training data using $K = 1$, and evaluate its performance on the test data.

```

In [57]: knn1 = KNeighborsClassifier(n_neighbors=1)
knn1_pred = knn1.fit(X_train, y_train).predict(X_test)
np.mean(y_test != knn1_pred), np.mean(y_test != "No")

```

```
Out[57]: (0.111, 0.067)
```

The KNN error rate on the 1,000 test observations is about 11%. At first glance, this may appear to be fairly good. However, since just over 6% of customers purchased insurance, we could get the error rate down to almost 6% by always predicting `No` regardless of the values of the predictors! This is known as the *null rate*.

Suppose that there is some non-trivial cost to trying to sell insurance to a given individual. For instance, perhaps a salesperson must visit each potential customer. If the company tries to sell insurance to a random selection of customers, then the success rate will be only 6%, which may be far too low given the costs involved. Instead, the company would like to try to sell insurance only to customers who are likely to buy it. So the overall error rate is not of interest. Instead, the fraction of individuals that are correctly predicted to buy insurance is of interest.

```

In [58]: confusion_table(knn1_pred, y_test)

```

```
Out[58]:      Truth   No   Yes
      Predicted
         No   880   58
         Yes   53    9
```

It turns out that KNN with $K = 1$ does far better than random guessing among the customers that are predicted to buy insurance. Among 62 such customers, 9, or 14.5%, actually do purchase insurance. This is double the rate that one would obtain from random guessing.

```
In [59]: 9/(53+9)
```

```
Out[59]: 0.145
```

Tuning Parameters

The number of neighbors in KNN is referred to as a *tuning parameter*, also referred to as a *hyperparameter*. We do not know *a priori* what value to use. It is therefore of interest to see how the classifier performs on test data as we vary these parameters. This can be achieved with a **for** loop, described in Section 2.3.8. Here we use a for loop to look at the accuracy of our classifier in the group predicted to purchase insurance as we vary the number of neighbors from 1 to 5:

tuning
parameter
hyper-
parameter

```
In [60]: for K in range(1,6):
          knn = KNeighborsClassifier(n_neighbors=K)
          knn_pred = knn.fit(X_train, y_train).predict(X_test)
          C = confusion_table(knn_pred, y_test)
          templ = ('K={0:d}: # predicted to rent: {1:>2}, ' +
                  ' # who did rent {2:d}, accuracy {3:.1%} ')
          pred = C.loc['Yes'].sum()
          did_rent = C.loc['Yes', 'Yes']
          print(templ.format(
              K,
              pred,
              did_rent,
              did_rent / pred))
```

```
K=1: # predicted to rent: 62, # who did rent 9, accuracy 14.5%
K=2: # predicted to rent: 6, # who did rent 1, accuracy 16.7%
K=3: # predicted to rent: 20, # who did rent 3, accuracy 15.0%
K=4: # predicted to rent: 3, # who did rent 0, accuracy 0.0%
K=5: # predicted to rent: 7, # who did rent 1, accuracy 14.3%
```

We see some variability — the numbers for $K=4$ are very different from the rest.

Comparison to Logistic Regression

As a comparison, we can also fit a logistic regression model to the data. This can also be done with **sklearn**, though by default it fits something like the *ridge regression* version of logistic regression, which we introduce in Chapter 6. This can be modified by appropriately setting the argument **C** below. Its default value is 1 but by setting it to a very large number, the algorithm converges to the same solution as the usual (unregularized) logistic regression estimator discussed above.

Unlike the `statsmodels` package, `sklearn` focuses less on inference and more on classification. Hence, the `summary` methods seen in `statsmodels` and our simplified version seen with `summarize` are not generally available for the classifiers in `sklearn`.

```
In [61]: logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict_proba(X_test)
logit_labels = np.where(logit_pred[:,1] > 5, 'Yes', 'No')
confusion_table(logit_labels, y_test)
```

```
Out[61]:      Truth    No  Yes
Predicted
        No   933   67
        Yes    0    0
```

We used the argument `solver='liblinear'` above to avoid a warning with the default solver which would indicate that the algorithm does not converge.

If we use 0.5 as the predicted probability cut-off for the classifier, then we have a problem: none of the test observations are predicted to purchase insurance. However, we are not required to use a cut-off of 0.5. If we instead predict a purchase any time the predicted probability of purchase exceeds 0.25, we get much better results: we predict that 29 people will purchase insurance, and we are correct for about 31% of these people. This is almost five times better than random guessing!

```
In [62]: logit_labels = np.where(logit_pred[:,1]>0.25, 'Yes', 'No')
confusion_table(logit_labels, y_test)
```

```
Out[62]:      Truth    No  Yes
Predicted
        No   913   58
        Yes    20    9
```

```
In [63]: 9/(20+9)
```

```
Out[63]: 0.310
```

4.7.7 Linear and Poisson Regression on the Bikeshare Data

Here we fit linear and Poisson regression models to the `Bikeshare` data, as described in Section 4.6. The response `bikers` measures the number of bike rentals per hour in Washington, DC in the period 2010–2012.

```
In [64]: Bike = load_data('Bikeshare')
```

Let's have a peek at the dimensions and names of the variables in this dataframe.

```
In [65]: Bike.shape, Bike.columns
```



```
Out[65]: ((8645, 15),
Index(['season', 'mnth', 'day', 'hr', 'holiday', 'weekday',
      'workingday', 'weathersit', 'temp', 'atemp', 'hum',
      'windspeed', 'casual', 'registered', 'bikers'],
      dtype='object'))
```

Linear Regression

We begin by fitting a linear regression model to the data.

```
In [66]: X = MS(['mnth',
                'hr',
                'workingday',
                'temp',
                'weathersit']).fit_transform(Bike)
Y = Bike['bikers']
M_lm = sm.OLS(Y, X).fit()
summarize(M_lm)
```

```
Out[66]:
```

	coef	std err	t	P> t
intercept	-68.6317	5.307	-12.932	0.000
mnth[Feb]	6.8452	4.287	1.597	0.110
mnth[March]	16.5514	4.301	3.848	0.000
mnth[April]	41.4249	4.972	8.331	0.000
mnth[May]	72.5571	5.641	12.862	0.000
mnth[June]	67.8187	6.544	10.364	0.000
mnth[July]	45.3245	7.081	6.401	0.000
mnth[Aug]	53.2430	6.640	8.019	0.000
mnth[Sept]	66.6783	5.925	11.254	0.000
mnth[Oct]	75.8343	4.950	15.319	0.000
mnth[Nov]	60.3100	4.610	13.083	0.000
mnth[Dec]	46.4577	4.271	10.878	0.000
hr[1]	-14.5793	5.699	-2.558	0.011
hr[2]	-21.5791	5.733	-3.764	0.000
hr[3]	-31.1408	5.778	-5.389	0.000
.....

There are 24 levels in `hr` and 40 rows in all, so we have truncated the summary. In `M_lm`, the first levels `hr[0]` and `mnth[Jan]` are treated as the baseline values, and so no coefficient estimates are provided for them: implicitly, their coefficient estimates are zero, and all other levels are measured relative to these baselines. For example, the Feb coefficient of 6.845 signifies that, holding all other variables constant, there are on average about 7 more riders in February than in January. Similarly there are about 16.5 more riders in March than in January.

The results seen in Section 4.6.1 used a slightly different coding of the variables `hr` and `mnth`, as follows:

```
In [67]: hr_encode = contrast('hr', 'sum')
mnth_encode = contrast('mnth', 'sum')
```

Refitting again:

```
In [68]: X2 = MS([mnth_encode,
                  hr_encode,
                  'workingday',
                  'temp',
```

```
'weathersit']).fit_transform(Bike)
M2_lm = sm.OLS(Y, X2).fit()
S2 = summarize(M2_lm)
S2
```

```
Out[68]:
```

	coef	std err	t	P> t
intercept	73.5974	5.132	14.340	0.000
mnth[Jan]	-46.0871	4.085	-11.281	0.000
mnth[Feb]	-39.2419	3.539	-11.088	0.000
mnth[March]	-29.5357	3.155	-9.361	0.000
mnth[April]	-4.6622	2.741	-1.701	0.089
mnth[May]	26.4700	2.851	9.285	0.000
mnth[June]	21.7317	3.465	6.272	0.000
mnth[July]	-0.7626	3.908	-0.195	0.845
mnth[Aug]	7.1560	3.535	2.024	0.043
mnth[Sept]	20.5912	3.046	6.761	0.000
mnth[Oct]	29.7472	2.700	11.019	0.000
mnth[Nov]	14.2229	2.860	4.972	0.000
hr[0]	-96.1420	3.955	-24.307	0.000
hr[1]	-110.7213	3.966	-27.916	0.000
hr[2]	-117.7212	4.016	-29.310	0.000
.....

What is the difference between the two codings? In `M2_lm`, a coefficient estimate is reported for all but level `23` of `hr` and level `Dec` of `mnth`. Importantly, in `M2_lm`, the (unreported) coefficient estimate for the last level of `mnth` is not zero: instead, it equals the negative of the sum of the coefficient estimates for all of the other levels. Similarly, in `M2_lm`, the coefficient estimate for the last level of `hr` is the negative of the sum of the coefficient estimates for all of the other levels. This means that the coefficients of `hr` and `mnth` in `M2_lm` will always sum to zero, and can be interpreted as the difference from the mean level. For example, the coefficient for January of -46.087 indicates that, holding all other variables constant, there are typically 46 fewer riders in January relative to the yearly average.

It is important to realize that the choice of coding really does not matter, provided that we interpret the model output correctly in light of the coding used. For example, we see that the predictions from the linear model are the same regardless of coding:

```
In[69]: np.sum((M_lm.fittedvalues - M2_lm.fittedvalues)**2)
```

```
Out[69]: 1.53e-20
```

The sum of squared differences is zero. We can also see this using the `np.allclose()` function:

```
In[70]: np.allclose(M_lm.fittedvalues, M2_lm.fittedvalues)
```

```
np.allclose()
```

```
Out[70]: True
```

To reproduce the left-hand side of Figure 4.13 we must first obtain the coefficient estimates associated with `mnth`. The coefficients for January through November can be obtained directly from the `M2_lm` object. The coefficient for December must be explicitly computed as the negative sum of all the other months. We first extract all the coefficients for month from the coefficients of `M2_lm`.

```
In [71]: coef_month = S2[S2.index.str.contains('mnth')]['coef']
coef_month
```

```
Out [71]: mnth[Jan]      -46.0871
          mnth[Feb]     -39.2419
          mnth[March]   -29.5357
          mnth[April]   -4.6622
          mnth[May]     26.4700
          mnth[June]    21.7317
          mnth[July]    -0.7626
          mnth[Aug]      7.1560
          mnth[Sept]    20.5912
          mnth[Oct]     29.7472
          mnth[Nov]     14.2229
          Name: coef, dtype: float64
```

Next, we append Dec as the negative of the sum of all other months.

```
In [72]: months = Bike['mnth'].dtype.categories
coef_month = pd.concat([
    coef_month,
    pd.Series([-coef_month.sum()],
              index=['mnth[Dec]'],
              dtype=coef_month.dtype),
])
coef_month
```

```
Out [72]: mnth[Jan]      -46.0871
          mnth[Feb]     -39.2419
          mnth[March]   -29.5357
          mnth[April]   -4.6622
          mnth[May]     26.4700
          mnth[June]    21.7317
          mnth[July]    -0.7626
          mnth[Aug]      7.1560
          mnth[Sept]    20.5912
          mnth[Oct]     29.7472
          mnth[Nov]     14.2229
          mnth[Dec]      0.3705
          Name: coef, dtype: float64
```

Finally, to make the plot neater, we'll just use the first letter of each month, which is the 6th entry of each of the labels in the index.

```
In [73]: fig_month, ax_month = subplots(figsize=(8,8))
x_month = np.arange(coef_month.shape[0])
ax_month.plot(x_month, coef_month, marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20);
```

Reproducing the right-hand plot in Figure 4.13 follows a similar process.

```
In [74]: coef_hr = S2[S2.index.str.contains('hr')]['coef']
coef_hr = coef_hr.reindex(['hr[{0}]'.format(h) for h in range(23)])
coef_hr = pd.concat([coef_hr,
```

```
pd.Series([-coef_hr.sum()], index=['hr[23]'])
])
```

We now make the hour plot.

```
In [75]: fig_hr, ax_hr = subplots(figsize=(8,8))
x_hr = np.arange(coef_hr.shape[0])
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticks(x_hr[::2])
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20);
```

Poisson Regression

Now we fit instead a Poisson regression model to the **Bikeshare** data. Very little changes, except that we now use the function `sm.GLM()` with the Poisson family specified:

```
In [76]: M_pois = sm.GLM(Y, X2, family=sm.families.Poisson()).fit()
```

We can plot the coefficients associated with `mnth` and `hr`, in order to reproduce Figure 4.15. We first complete these coefficients as before.

```
In [77]: S_pois = summarize(M_pois)
coef_month = S_pois[S_pois.index.str.contains('mnth']]['coef']
coef_month = pd.concat([coef_month,
                        pd.Series([-coef_month.sum()],
                                index=['mnth[Dec]'])])
coef_hr = S_pois[S_pois.index.str.contains('hr']]['coef']
coef_hr = pd.concat([coef_hr,
                    pd.Series([-coef_hr.sum()],
                                index=['hr[23]'])])
```

The plotting is as before.

```
In [78]: fig_pois, (ax_month, ax_hr) = subplots(1, 2, figsize=(16,8))
ax_month.plot(x_month, coef_month, marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20)
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20);
```

We compare the fitted values of the two models. The fitted values are stored in the `fittedvalues` attribute returned by the `fit()` method for both the linear regression and the Poisson fits. The linear predictors are stored as the attribute `lin_pred`.

```
In [79]: fig, ax = subplots(figsize=(8, 8))
ax.scatter(M2_lm.fittedvalues,
          M_pois.fittedvalues,
          s=20)
ax.set_xlabel('Linear Regression Fit', fontsize=20)
```

```
ax.set_ylabel('Poisson Regression Fit', fontsize=20)
ax.axline([0,0], c='black', linewidth=3,
          linestyle='--', slope=1);
```

The predictions from the Poisson regression model are correlated with those from the linear model; however, the former are non-negative. As a result the Poisson regression predictions tend to be larger than those from the linear model for either very low or very high levels of ridership.

In this section, we fit Poisson regression models using the `sm.GLM()` function with the argument `family=sm.families.Poisson()`. Earlier in this lab we used the `sm.GLM()` function with `family=sm.families.Binomial()` to perform logistic regression. Other choices for the `family` argument can be used to fit other types of GLMs. For instance, `family=sm.families.Gamma()` fits a Gamma regression model.

4.8 Exercises

Conceptual

1. Using a little bit of algebra, prove that (4.2) is equivalent to (4.3). In other words, the logistic function representation and logit representation for the logistic regression model are equivalent.
2. It was stated in the text that classifying an observation to the class for which (4.17) is largest is equivalent to classifying an observation to the class for which (4.18) is largest. Prove that this is the case. In other words, under the assumption that the observations in the k th class are drawn from a $N(\mu_k, \sigma^2)$ distribution, the Bayes classifier assigns an observation to the class for which the discriminant function is maximized.
3. This problem relates to the QDA model, in which the observations within each class are drawn from a normal distribution with a class-specific mean vector and a class specific covariance matrix. We consider the simple case where $p = 1$; i.e. there is only one feature.

Suppose that we have K classes, and that if an observation belongs to the k th class then X comes from a one-dimensional normal distribution, $X \sim N(\mu_k, \sigma_k^2)$. Recall that the density function for the one-dimensional normal distribution is given in (4.16). Prove that in this case, the Bayes classifier is *not* linear. Argue that it is in fact quadratic.

Hint: For this problem, you should follow the arguments laid out in Section 4.4.1, but without making the assumption that $\sigma_1^2 = \dots = \sigma_K^2$.

4. When the number of features p is large, there tends to be a deterioration in the performance of KNN and other *local* approaches that perform prediction using only observations that are *near* the test observation for which a prediction must be made. This phenomenon is known as the *curse of dimensionality*, and it ties into the fact that non-parametric approaches often perform poorly when p is large. We will now investigate this curse.



curse of dimensionality

- (a) Suppose that we have a set of observations, each with measurements on $p = 1$ feature, X . We assume that X is uniformly (evenly) distributed on $[0, 1]$. Associated with each observation is a response value. Suppose that we wish to predict a test observation's response using only observations that are within 10 % of the range of X closest to that test observation. For instance, in order to predict the response for a test observation with $X = 0.6$, we will use observations in the range $[0.55, 0.65]$. On average, what fraction of the available observations will we use to make the prediction?
- (b) Now suppose that we have a set of observations, each with measurements on $p = 2$ features, X_1 and X_2 . We assume that (X_1, X_2) are uniformly distributed on $[0, 1] \times [0, 1]$. We wish to predict a test observation's response using only observations that are within 10 % of the range of X_1 *and* within 10 % of the range of X_2 closest to that test observation. For instance, in order to predict the response for a test observation with $X_1 = 0.6$ and $X_2 = 0.35$, we will use observations in the range $[0.55, 0.65]$ for X_1 and in the range $[0.3, 0.4]$ for X_2 . On average, what fraction of the available observations will we use to make the prediction?
- (c) Now suppose that we have a set of observations on $p = 100$ features. Again the observations are uniformly distributed on each feature, and again each feature ranges in value from 0 to 1. We wish to predict a test observation's response using observations within the 10 % of each feature's range that is closest to that test observation. What fraction of the available observations will we use to make the prediction?
- (d) Using your answers to parts (a)–(c), argue that a drawback of KNN when p is large is that there are very few training observations “near” any given test observation.
- (e) Now suppose that we wish to make a prediction for a test observation by creating a p -dimensional hypercube centered around the test observation that contains, on average, 10 % of the training observations. For $p = 1, 2$, and 100, what is the length of each side of the hypercube? Comment on your answer.

Note: A hypercube is a generalization of a cube to an arbitrary number of dimensions. When $p = 1$, a hypercube is simply a line segment, when $p = 2$ it is a square, and when $p = 100$ it is a 100-dimensional cube.



5. We now examine the differences between LDA and QDA.

- (a) If the Bayes decision boundary is linear, do we expect LDA or QDA to perform better on the training set? On the test set?
- (b) If the Bayes decision boundary is non-linear, do we expect LDA or QDA to perform better on the training set? On the test set?

- (c) In general, as the sample size n increases, do we expect the test prediction accuracy of QDA relative to LDA to improve, decline, or be unchanged? Why?
 - (d) True or False: Even if the Bayes decision boundary for a given problem is linear, we will probably achieve a superior test error rate using QDA rather than LDA because QDA is flexible enough to model a linear decision boundary. Justify your answer.
6. Suppose we collect data for a group of students in a statistics class with variables X_1 = hours studied, X_2 = undergrad GPA, and Y = receive an A. We fit a logistic regression and produce estimated coefficient, $\hat{\beta}_0 = -6, \hat{\beta}_1 = 0.05, \hat{\beta}_2 = 1$.
- (a) Estimate the probability that a student who studies for 40 h and has an undergrad GPA of 3.5 gets an A in the class.
 - (b) How many hours would the student in part (a) need to study to have a 50 % chance of getting an A in the class?
7. Suppose that we wish to predict whether a given stock will issue a dividend this year (“Yes” or “No”) based on X , last year’s percent profit. We examine a large number of companies and discover that the mean value of X for companies that issued a dividend was $\bar{X} = 10$, while the mean for those that didn’t was $\bar{X} = 0$. In addition, the variance of X for these two sets of companies was $\hat{\sigma}^2 = 36$. Finally, 80 % of companies issued dividends. Assuming that X follows a normal distribution, predict the probability that a company will issue a dividend this year given that its percentage profit was $X = 4$ last year.

Hint: Recall that the density function for a normal random variable is $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$. You will need to use Bayes’ theorem.

8. Suppose that we take a data set, divide it into equally-sized training and test sets, and then try out two different classification procedures. First we use logistic regression and get an error rate of 20 % on the training data and 30 % on the test data. Next we use 1-nearest neighbors (i.e. $K = 1$) and get an average error rate (averaged over both test and training data sets) of 18 %. Based on these results, which method should we prefer to use for classification of new observations? Why?
9. This problem has to do with *odds*.
- (a) On average, what fraction of people with an odds of 0.37 of defaulting on their credit card payment will in fact default?
 - (b) Suppose that an individual has a 16 % chance of defaulting on her credit card payment. What are the odds that she will default?

10. Equation 4.32 derived an expression for $\log \left(\frac{\Pr(Y=k|X=x)}{\Pr(Y=K|X=x)} \right)$ in the setting where $p > 1$, so that the mean for the k th class, μ_k , is a p -dimensional vector, and the shared covariance Σ is a $p \times p$ matrix. However, in the setting with $p = 1$, (4.32) takes a simpler form, since the means μ_1, \dots, μ_K and the variance σ^2 are scalars. In this simpler setting, repeat the calculation in (4.32), and provide expressions for a_k and b_{kj} in terms of π_k , π_K , μ_k , μ_K , and σ^2 . 
11. Work out the detailed forms of a_k , b_{kj} , and b_{kjl} in (4.33). Your answer should involve π_k , π_K , μ_k , μ_K , Σ_k , and Σ_K . 
12. Suppose that you wish to classify an observation $X \in \mathbb{R}$ into **apples** and **oranges**. You fit a logistic regression model and find that

$$\widehat{\Pr}(Y = \text{orange} | X = x) = \frac{\exp(\hat{\beta}_0 + \hat{\beta}_1 x)}{1 + \exp(\hat{\beta}_0 + \hat{\beta}_1 x)}.$$

Your friend fits a logistic regression model to the same data using the *softmax* formulation in (4.13), and finds that

$$\widehat{\Pr}(Y = \text{orange} | X = x) = \frac{\exp(\hat{\alpha}_{\text{orange}0} + \hat{\alpha}_{\text{orange}1}x)}{\exp(\hat{\alpha}_{\text{orange}0} + \hat{\alpha}_{\text{orange}1}x) + \exp(\hat{\alpha}_{\text{apple}0} + \hat{\alpha}_{\text{apple}1}x)}.$$

- What is the log odds of **orange** versus **apple** in your model?
- What is the log odds of **orange** versus **apple** in your friend's model?
- Suppose that in your model, $\hat{\beta}_0 = 2$ and $\hat{\beta}_1 = -1$. What are the coefficient estimates in your friend's model? Be as specific as possible.
- Now suppose that you and your friend fit the same two models on a different data set. This time, your friend gets the coefficient estimates $\hat{\alpha}_{\text{orange}0} = 1.2$, $\hat{\alpha}_{\text{orange}1} = -2$, $\hat{\alpha}_{\text{apple}0} = 3$, $\hat{\alpha}_{\text{apple}1} = 0.6$. What are the coefficient estimates in your model?
- Finally, suppose you apply both models from (d) to a data set with 2,000 test observations. What fraction of the time do you expect the predicted class labels from your model to agree with those from your friend's model? Explain your answer.

Applied

13. This question should be answered using the **Weekly** data set, which is part of the **ISLP** package. This data is similar in nature to the **Smarket** data from this chapter's lab, except that it contains 1,089 weekly returns for 21 years, from the beginning of 1990 to the end of 2010.
- Produce some numerical and graphical summaries of the **Weekly** data. Do there appear to be any patterns?

- (b) Use the full data set to perform a logistic regression with **Direction** as the response and the five lag variables plus **Volume** as predictors. Use the summary function to print the results. Do any of the predictors appear to be statistically significant? If so, which ones?
 - (c) Compute the confusion matrix and overall fraction of correct predictions. Explain what the confusion matrix is telling you about the types of mistakes made by logistic regression.
 - (d) Now fit the logistic regression model using a training data period from 1990 to 2008, with **Lag2** as the only predictor. Compute the confusion matrix and the overall fraction of correct predictions for the held out data (that is, the data from 2009 and 2010).
 - (e) Repeat (d) using LDA.
 - (f) Repeat (d) using QDA.
 - (g) Repeat (d) using KNN with $K = 1$.
 - (h) Repeat (d) using naive Bayes.
 - (i) Which of these methods appears to provide the best results on this data?
 - (j) Experiment with different combinations of predictors, including possible transformations and interactions, for each of the methods. Report the variables, method, and associated confusion matrix that appears to provide the best results on the held out data. Note that you should also experiment with values for K in the KNN classifier.
14. In this problem, you will develop a model to predict whether a given car gets high or low gas mileage based on the **Auto** data set.
- (a) Create a binary variable, **mpg01**, that contains a 1 if **mpg** contains a value above its median, and a 0 if **mpg** contains a value below its median. You can compute the median using the `median()` method of the data frame. Note you may find it helpful to add a column **mpg01** to the data frame by assignment. Assuming you have stored the data frame as **Auto**, this can be done as follows:
- ```
Auto['mpg01'] = mpg01
```
- (b) Explore the data graphically in order to investigate the association between **mpg01** and the other features. Which of the other features seem most likely to be useful in predicting **mpg01**? Scatterplots and boxplots may be useful tools to answer this question. Describe your findings.
  - (c) Split the data into a training set and a test set.
  - (d) Perform LDA on the training data in order to predict **mpg01** using the variables that seemed most associated with **mpg01** in (b). What is the test error of the model obtained?

- (e) Perform QDA on the training data in order to predict `mpg01` using the variables that seemed most associated with `mpg01` in (b). What is the test error of the model obtained?
  - (f) Perform logistic regression on the training data in order to predict `mpg01` using the variables that seemed most associated with `mpg01` in (b). What is the test error of the model obtained?
  - (g) Perform naive Bayes on the training data in order to predict `mpg01` using the variables that seemed most associated with `mpg01` in (b). What is the test error of the model obtained?
  - (h) Perform KNN on the training data, with several values of  $K$ , in order to predict `mpg01`. Use only the variables that seemed most associated with `mpg01` in (b). What test errors do you obtain? Which value of  $K$  seems to perform the best on this data set?
15. This problem involves writing functions.
- (a) Write a function, `Power()`, that prints out the result of raising 2 to the 3rd power. In other words, your function should compute  $2^3$  and print out the results.  
*Hint: Recall that `x**a` raises `x` to the power `a`. Use the `print()` function to display the result.*
  - (b) Create a new function, `Power2()`, that allows you to pass *any* two numbers, `x` and `a`, and prints out the value of `x**a`. You can do this by beginning your function with the line
 

```
def Power2(x, a):
```

You should be able to call your function by entering, for instance,

```
Power2(3, 8)
```

on the command line. This should output the value of  $3^8$ , namely, 6,561.
  - (c) Using the `Power2()` function that you just wrote, compute  $10^3$ ,  $8^{17}$ , and  $131^3$ .
  - (d) Now create a new function, `Power3()`, that actually *returns* the result `x**a` as a `Python` object, rather than simply printing it to the screen. That is, if you store the value `x**a` in an object called `result` within your function, then you can simply `return` this result, using the following line:
 

```
return result
```

Note that the line above should be the last line in your function, and it should be indented 4 spaces.
  - (e) Now using the `Power3()` function, create a plot of  $f(x) = x^2$ . The  $x$ -axis should display a range of integers from 1 to 10, and the  $y$ -axis should display  $x^2$ . Label the axes appropriately, and use an appropriate title for the figure. Consider displaying either the  $x$ -axis, the  $y$ -axis, or both on the log-scale. You can do this by using the `ax.set_xscale()` and `ax.set_yscale()` methods of the axes you are plotting to.

- (f) Create a function, `PlotPower()`, that allows you to create a plot of `x` against `x**a` for a fixed `a` and a sequence of values of `x`. For instance, if you call

```
PlotPower(np.arange(1, 11), 3)
```

then a plot should be created with an  $x$ -axis taking on values  $1, 2, \dots, 10$ , and a  $y$ -axis taking on values  $1^3, 2^3, \dots, 10^3$ .

16. Using the `Boston` data set, fit classification models in order to predict whether a given suburb has a crime rate above or below the median. Explore logistic regression, LDA, naive Bayes, and KNN models using various subsets of the predictors. Describe your findings.

*Hint: You will have to create the response variable yourself, using the variables that are contained in the `Boston` data set.*