CHAPTER 4

Regression and Prediction

Perhaps the most common goal in statistics is to answer the question "Is the variable X (or more likely, $X_1, ..., X_p$) associated with a variable Y, and if so, what is the relationship and can we use it to predict Y?"

Nowhere is the nexus between statistics and data science stronger than in the realm of prediction—specifically, the prediction of an outcome (target) variable based on the values of other "predictor" variables. This process of training a model on data where the outcome is known, for subsequent application to data where the outcome is not known, is termed *supervised learning*. Another important connection between data science and statistics is in the area of *anomaly detection*, where regression diagnostics originally intended for data analysis and improving the regression model can be used to detect unusual records.

Simple Linear Regression

Simple linear regression provides a model of the relationship between the magnitude of one variable and that of a second—for example, as X increases, Y also increases. Or as X increases, Y decreases.[1] Correlation is another way to measure how two variables are related—see the section "Correlation" on page 30. The difference is that while correlation measures the *strength* of an association between two variables, regression quantifies the *nature* of the relationship.

1 This and subsequent sections in this chapter © 2020 Datastats, LLC, Peter Bruce, Andrew Bruce, and Peter Gedeck; used by permission.

Key Terms for Simple Linear Regression

Response
The variable we are trying to predict.

Synonyms
dependent variable, Y variable, target, outcome

Independent variable
The variable used to predict the response.

Synonyms
X variable, feature, attribute, predictor

Record
The vector of predictor and outcome values for a specific individual or case.

Synonyms
row, case, instance, example

Intercept
The intercept of the regression line—that is, the predicted value when $X = 0$.

Synonyms
b_0, β_0

Regression coefficient
The slope of the regression line.

Synonyms
slope, b_1, β_1, parameter estimates, weights

Fitted values
The estimates $\hat{Y}_i$ obtained from the regression line.

Synonym
predicted values

Residuals
The difference between the observed values and the fitted values.

Synonym
errors

Least squares

The method of fitting a regression by minimizing the sum of squared residuals.

Synonyms

ordinary least squares, OLS

The Regression Equation

Simple linear regression estimates how much Y will change when X changes by a certain amount. With the correlation coefficient, the variables X and Y are interchangeable. With regression, we are trying to predict the Y variable from X using a linear relationship (i.e., a line):

$$Y = b_0 + b_1 X$$

We read this as "Y equals b_1 times X, plus a constant b_0." The symbol b_0 is known as the *intercept* (or constant), and the symbol b_1 as the *slope* for X. Both appear in *R* output as *coefficients*, though in general use the term *coefficient* is often reserved for b_1. The Y variable is known as the *response* or *dependent* variable since it depends on X. The X variable is known as the *predictor* or *independent* variable. The machine learning community tends to use other terms, calling Y the *target* and X a *feature* vector. Throughout this book, we will use the terms *predictor* and *feature* interchangeably.

Consider the scatterplot in Figure 4-1 displaying the number of years a worker was exposed to cotton dust (`Exposure`) versus a measure of lung capacity (`PEFR` or "peak expiratory flow rate"). How is `PEFR` related to `Exposure`? It's hard to tell based just on the picture.

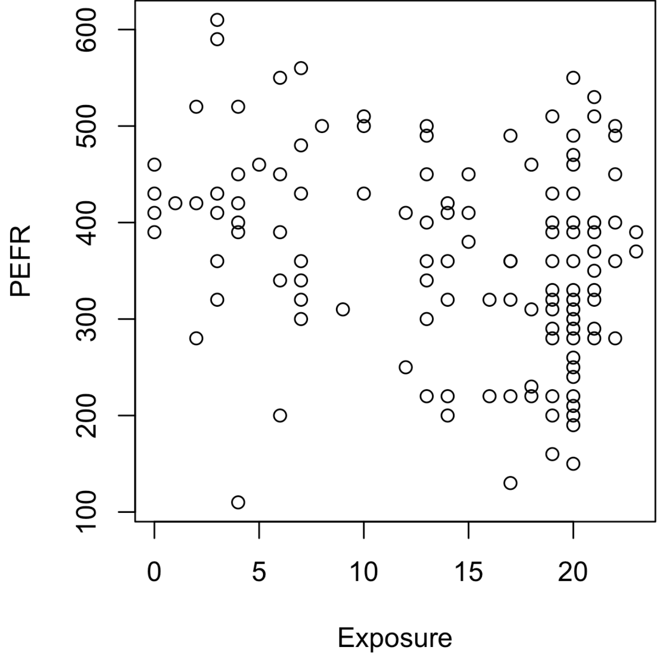


Figure 4-1. Cotton exposure versus lung capacity

Simple linear regression tries to find the "best" line to predict the response `PEFR` as a function of the predictor variable `Exposure`:

$$\text{PEFR} = b_0 + b_1\text{Exposure}$$

The `lm` function in *R* can be used to fit a linear regression:

```
model <- lm(PEFR ~ Exposure, data=lung)
```

`lm` stands for *linear model*, and the ~ symbol denotes that `PEFR` is predicted by `Exposure`. With this model definition, the intercept is automatically included and fitted. If you want to exclude the intercept from the model, you need to write the model definition as follows:

```
PEFR ~ Exposure - 1
```

Printing the `model` object produces the following output:

```
Call:
lm(formula = PEFR ~ Exposure, data = lung)

Coefficients:
(Intercept)     Exposure
    424.583       -4.185
```

The intercept, or b_0, is 424.583 and can be interpreted as the predicted `PEFR` for a worker with zero years exposure. The regression coefficient, or b_1, can be interpreted as follows: for each additional year that a worker is exposed to cotton dust, the worker's `PEFR` measurement is reduced by –4.185.

In *Python*, we can use `LinearRegression` from the `scikit-learn` package. (the `statsmodels` package has a linear regression implementation that is more similar to *R* (`sm.OLS`); we will use it later in this chapter):

```
predictors = ['Exposure']
outcome = 'PEFR'

model = LinearRegression()
model.fit(lung[predictors], lung[outcome])

print(f'Intercept: {model.intercept_:.3f}')
print(f'Coefficient Exposure: {model.coef_[0]:.3f}')
```

The regression line from this model is displayed in Figure 4-2.

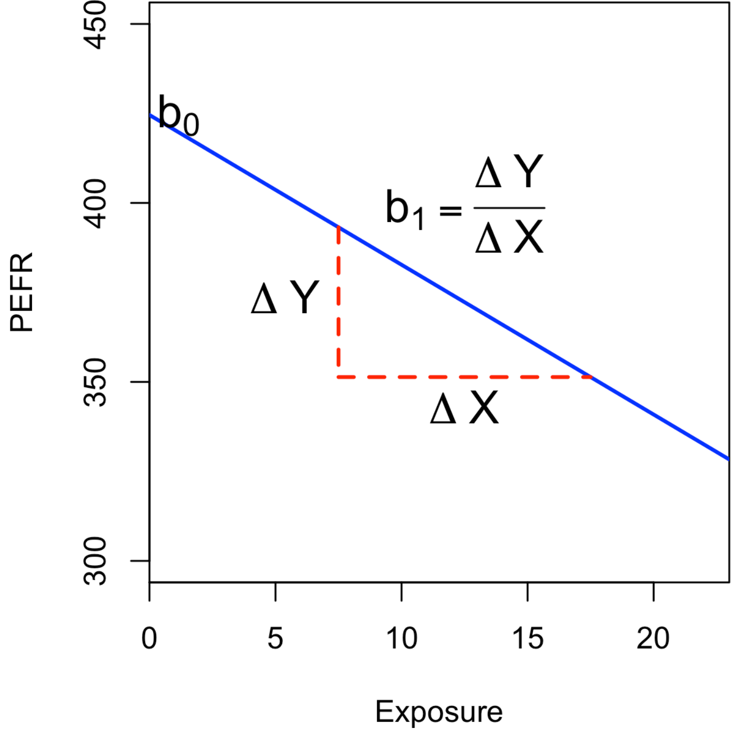


Figure 4-2. Slope and intercept for the regression fit to the lung data

Fitted Values and Residuals

Important concepts in regression analysis are the *fitted values* (the predictions) and *residuals* (prediction errors). In general, the data doesn't fall exactly on a line, so the regression equation should include an explicit error term e_i:

$$Y_i = b_0 + b_1 X_i + e_i$$

The fitted values, also referred to as the *predicted values*, are typically denoted by $\hat{Y}_i$ (Y-hat). These are given by:

$$\hat{Y}_i = \hat{b}_0 + \hat{b}_1 X_i$$

The notation $\hat{b}_0$ and $\hat{b}_1$ indicates that the coefficients are estimated versus known.

Hat Notation: Estimates Versus Known Values

The "hat" notation is used to differentiate between estimates and known values. So the symbol $\hat{b}$ ("b-hat") is an estimate of the unknown parameter b. Why do statisticians differentiate between the estimate and the true value? The estimate has uncertainty, whereas the true value is fixed.[2]

We compute the residuals $\hat{e}_i$ by subtracting the *predicted* values from the original data:

$$\hat{e}_i = Y_i - \hat{Y}_i$$

In *R*, we can obtain the fitted values and residuals using the functions `predict` and `residuals`:

```
fitted <- predict(model)
resid <- residuals(model)
```

With `scikit-learn`'s `LinearRegression` model, we use the `predict` method on the training data to get the `fitted` values and subsequently the `residuals`. As we will see, this is a general pattern that all models in `scikit-learn` follow:

```
fitted = model.predict(lung[predictors])
residuals = lung[outcome] - fitted
```

Figure 4-3 illustrates the residuals from the regression line fit to the lung data. The residuals are the length of the vertical dashed lines from the data to the line.

2 In Bayesian statistics, the true value is assumed to be a random variable with a specified distribution. In the Bayesian context, instead of estimates of unknown parameters, there are posterior and prior distributions.

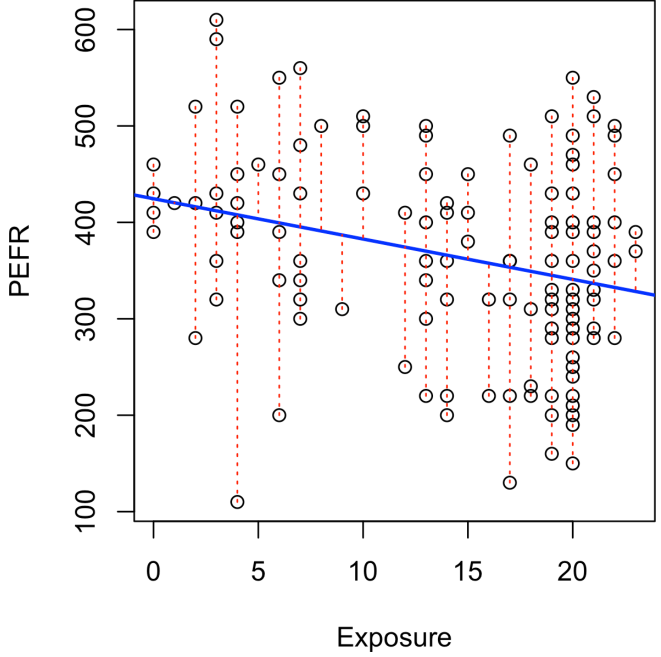


Figure 4-3. Residuals from a regression line (to accommodate all the data, the y-axis scale differs from Figure 4-2, hence the apparently different slope)

Least Squares

How is the model fit to the data? When there is a clear relationship, you could imagine fitting the line by hand. In practice, the regression line is the estimate that minimizes the sum of squared residual values, also called the *residual sum of squares* or *RSS*:

$$\begin{aligned} RSS &= \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i\right)^2 \\ &= \sum_{i=1}^{n} \left(Y_i - \hat{b}_0 - \hat{b}_1 X_i\right)^2 \end{aligned}$$

The estimates $\hat{b}_0$ and $\hat{b}_1$ are the values that minimize RSS.

The method of minimizing the sum of the squared residuals is termed *least squares* regression, or *ordinary least squares* (OLS) regression. It is often attributed to Carl Friedrich Gauss, the German mathematician, but was first published by the French mathematician Adrien-Marie Legendre in 1805. Least squares regression can be computed quickly and easily with any standard statistical software.

Historically, computational convenience is one reason for the widespread use of least squares in regression. With the advent of big data, computational speed is still an important factor. Least squares, like the mean (see "Median and Robust Estimates" on page 10), are sensitive to outliers, although this tends to be a significant problem only in small or moderate-sized data sets. See "Outliers" on page 177 for a discussion of outliers in regression.

Regression Terminology

When analysts and researchers use the term *regression* by itself, they are typically referring to linear regression; the focus is usually on developing a linear model to explain the relationship between predictor variables and a numeric outcome variable. In its formal statistical sense, regression also includes nonlinear models that yield a functional relationship between predictors and outcome variables. In the machine learning community, the term is also occasionally used loosely to refer to the use of any predictive model that produces a predicted numeric outcome (as opposed to classification methods that predict a binary or categorical outcome).

Prediction Versus Explanation (Profiling)

Historically, a primary use of regression was to illuminate a supposed linear relationship between predictor variables and an outcome variable. The goal has been to understand a relationship and explain it using the data that the regression was fit to. In this case, the primary focus is on the estimated slope of the regression equation, $\hat{b}$. Economists want to know the relationship between consumer spending and GDP growth. Public health officials might want to understand whether a public information campaign is effective in promoting safe sex practices. In such cases, the focus is not on predicting individual cases but rather on understanding the overall relationship among variables.

With the advent of big data, regression is widely used to form a model to predict individual outcomes for new data (i.e., a predictive model) rather than explain data in hand. In this instance, the main items of interest are the fitted values $\hat{Y}$. In marketing, regression can be used to predict the change in revenue in response to the size of an ad campaign. Universities use regression to predict students' GPA based on their SAT scores.

A regression model that fits the data well is set up such that changes in X lead to changes in Y. However, by itself, the regression equation does not prove the direction of causation. Conclusions about causation must come from a broader understanding about the relationship. For example, a regression equation might show a definite relationship between number of clicks on a web ad and number of conversions. It is our knowledge of the marketing process, not the regression equation, that leads us to the conclusion that clicks on the ad lead to sales, and not vice versa.

Key Ideas

- The regression equation models the relationship between a response variable *Y* and a predictor variable *X* as a line.
- A regression model yields fitted values and residuals—predictions of the response and the errors of the predictions.
- Regression models are typically fit by the method of least squares.
- Regression is used both for prediction and explanation.

Further Reading

For an in-depth treatment of prediction versus explanation, see Galit Shmueli's article "To Explain or to Predict?".

Multiple Linear Regression

When there are multiple predictors, the equation is simply extended to accommodate them:

$$Y = b_0 + b_1 X_1 + b_2 X_2 + \ldots + b_p X_p + e$$

Instead of a line, we now have a linear model—the relationship between each coefficient and its variable (feature) is linear.

Key Terms for Multiple Linear Regression

Root mean squared error
The square root of the average squared error of the regression (this is the most widely used metric to compare regression models).

Synonym
RMSE

Residual standard error
The same as the root mean squared error, but adjusted for degrees of freedom.

Synonym
: RSE

R-squared
: The proportion of variance explained by the model, from 0 to 1.

Synonyms
: coefficient of determination, R^2

t-statistic
: The coefficient for a predictor, divided by the standard error of the coefficient, giving a metric to compare the importance of variables in the model. See "t-Tests" on page 110.

Weighted regression
: Regression with the records having different weights.

All of the other concepts in simple linear regression, such as fitting by least squares and the definition of fitted values and residuals, extend to the multiple linear regression setting. For example, the fitted values are given by:

$$\hat{Y}_i = \hat{b}_0 + \hat{b}_1 X_{1,i} + \hat{b}_2 X_{2,i} + \ldots + \hat{b}_p X_{p,i}$$

Example: King County Housing Data

An example of using multiple linear regression is in estimating the value of houses. County assessors must estimate the value of a house for the purposes of assessing taxes. Real estate professionals and home buyers consult popular websites such as Zillow to ascertain a fair price. Here are a few rows of housing data from King County (Seattle), Washington, from the `house data.frame`:

```
head(house[, c('AdjSalePrice', 'SqFtTotLiving', 'SqFtLot', 'Bathrooms',
               'Bedrooms', 'BldgGrade')])
Source: local data frame [6 x 6]

  AdjSalePrice SqFtTotLiving SqFtLot Bathrooms Bedrooms BldgGrade
         (dbl)         (int)   (int)     (dbl)    (int)     (int)
1       300805          2400    9373      3.00        6         7
2      1076162          3764   20156      3.75        4        10
3       761805          2060   26036      1.75        4         8
4       442065          3200    8618      3.75        5         7
5       297065          1720    8620      1.75        4         7
6       411781           930    1012      1.50        2         8
```

The `head` method of `pandas` data frame lists the top rows:

```
subset = ['AdjSalePrice', 'SqFtTotLiving', 'SqFtLot', 'Bathrooms',
          'Bedrooms', 'BldgGrade']
house[subset].head()
```

The goal is to predict the sales price from the other variables. The `lm` function handles the multiple regression case simply by including more terms on the righthand side of the equation; the argument `na.action=na.omit` causes the model to drop records that have missing values:

```
house_lm <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
               Bedrooms + BldgGrade,
               data=house, na.action=na.omit)
```

`scikit-learn`'s `LinearRegression` can be used for multiple linear regression as well:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms', 'BldgGrade']
outcome = 'AdjSalePrice'

house_lm = LinearRegression()
house_lm.fit(house[predictors], house[outcome])
```

Printing `house_lm` object produces the following output:

```
house_lm

Call:
lm(formula = AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
    Bedrooms + BldgGrade, data = house, na.action = na.omit)

Coefficients:
  (Intercept)  SqFtTotLiving        SqFtLot      Bathrooms
   -5.219e+05      2.288e+02     -6.047e-02     -1.944e+04
     Bedrooms      BldgGrade
   -4.777e+04      1.061e+05
```

For a `LinearRegression` model, intercept and coefficients are the fields `intercept_` and `coef_` of the fitted model:

```
print(f'Intercept: {house_lm.intercept_:.3f}')
print('Coefficients:')
for name, coef in zip(predictors, house_lm.coef_):
    print(f' {name}: {coef}')
```

The interpretation of the coefficients is as with simple linear regression: the predicted value $\hat{Y}$ changes by the coefficient b_j for each unit change in X_j assuming all the other variables, X_k for $k \neq j$, remain the same. For example, adding an extra finished square foot to a house increases the estimated value by roughly $229; adding 1,000 finished square feet implies the value will increase by $228,800.

Assessing the Model

The most important performance metric from a data science perspective is *root mean squared error*, or *RMSE*. RMSE is the square root of the average squared error in the predicted $\hat{y}_i$ values:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2}{n}}$$

This measures the overall accuracy of the model and is a basis for comparing it to other models (including models fit using machine learning techniques). Similar to RMSE is the *residual standard error*, or *RSE*. In this case we have p predictors, and the RSE is given by:

$$RSE = \sqrt{\frac{\sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2}{(n - p - 1)}}$$

The only difference is that the denominator is the degrees of freedom, as opposed to number of records (see "Degrees of Freedom" on page 116). In practice, for linear regression, the difference between RMSE and RSE is very small, particularly for big data applications.

The `summary` function in *R* computes RSE as well as other metrics for a regression model:

```
summary(house_lm)

Call:
lm(formula = AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
    Bedrooms + BldgGrade, data = house, na.action = na.omit)

Residuals:
     Min       1Q   Median       3Q      Max
-1199479  -118908   -20977    87435  9473035

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)   -5.219e+05  1.565e+04 -33.342  < 2e-16 ***
SqFtTotLiving  2.288e+02  3.899e+00  58.694  < 2e-16 ***
SqFtLot       -6.047e-02  6.118e-02  -0.988    0.323
Bathrooms     -1.944e+04  3.625e+03  -5.363 8.27e-08 ***
Bedrooms      -4.777e+04  2.490e+03 -19.187  < 2e-16 ***
BldgGrade      1.061e+05  2.396e+03  44.277  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 261300 on 22681 degrees of freedom
Multiple R-squared:  0.5406,    Adjusted R-squared:  0.5405
F-statistic:  5338 on 5 and 22681 DF,  p-value: < 2.2e-16
```

`scikit-learn` provides a number of metrics for regression and classification. Here, we use `mean_squared_error` to get RMSE and `r2_score` for the coefficient of determination:

```
fitted = house_lm.predict(house[predictors])
RMSE = np.sqrt(mean_squared_error(house[outcome], fitted))
r2 = r2_score(house[outcome], fitted)
print(f'RMSE: {RMSE:.0f}')
print(f'r2: {r2:.4f}')
```

Use `statsmodels` to get a more detailed analysis of the regression model in *Python*:

```
model = sm.OLS(house[outcome], house[predictors].assign(const=1))
results = model.fit()
results.summary()
```

The `pandas` method `assign`, as used here, adds a constant column with value 1 to the predictors. This is required to model the intercept.

Another useful metric that you will see in software output is the *coefficient of determination*, also called the *R-squared* statistic or R^2. R-squared ranges from 0 to 1 and measures the proportion of variation in the data that is accounted for in the model. It is useful mainly in explanatory uses of regression where you want to assess how well the model fits the data. The formula for R^2 is:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{n} \left(y_i - \bar{y}\right)^2}$$

The denominator is proportional to the variance of Y. The output from *R* also reports an *adjusted R-squared*, which adjusts for the degrees of freedom, effectively penalizing the addition of more predictors to a model. Seldom is this significantly different from *R-squared* in multiple regression with large data sets.

Along with the estimated coefficients, *R* and `statsmodels` report the standard error of the coefficients (SE) and a *t-statistic*:

$$t_b = \frac{\hat{b}}{\mathrm{SE}\left(\hat{b}\right)}$$

The t-statistic—and its mirror image, the p-value—measures the extent to which a coefficient is “statistically significant”—that is, outside the range of what a random chance arrangement of predictor and target variable might produce. The higher the t-statistic (and the lower the p-value), the more significant the predictor. Since parsimony is a valuable model feature, it is useful to have a tool like this to guide choice of variables to include as predictors (see "Model Selection and Stepwise Regression" on page 156).

In addition to the t-statistic, *R* and other packages will often report a *p-value* (`Pr(>|t|)` in the *R* output) and *F-statistic*. Data scientists do not generally get too involved with the interpretation of these statistics, nor with the issue of statistical significance. Data scientists primarily focus on the t-statistic as a useful guide for whether to include a predictor in a model or not. High t-statistics (which go with p-values near 0) indicate a predictor should be retained in a model, while very low t-statistics indicate a predictor could be dropped. See "p-Value" on page 106 for more discussion.

Cross-Validation

Classic statistical regression metrics (R^2, F-statistics, and p-values) are all "in-sample" metrics—they are applied to the same data that was used to fit the model. Intuitively, you can see that it would make a lot of sense to set aside some of the original data, not use it to fit the model, and then apply the model to the set-aside (holdout) data to see how well it does. Normally, you would use a majority of the data to fit the model and use a smaller portion to test the model.

This idea of "out-of-sample" validation is not new, but it did not really take hold until larger data sets became more prevalent; with a small data set, analysts typically want to use all the data and fit the best possible model.

Using a holdout sample, though, leaves you subject to some uncertainty that arises simply from variability in the small holdout sample. How different would the assessment be if you selected a different holdout sample?

Cross-validation extends the idea of a holdout sample to multiple sequential holdout samples. The algorithm for basic *k-fold cross-validation* is as follows:

1. Set aside *1/k* of the data as a holdout sample.
2. Train the model on the remaining data.
3. Apply (score) the model to the *1/k* holdout, and record needed model assessment metrics.
4. Restore the first *1/k* of the data, and set aside the next *1/k* (excluding any records that got picked the first time).
5. Repeat steps 2 and 3.

6. Repeat until each record has been used in the holdout portion.
7. Average or otherwise combine the model assessment metrics.

The division of the data into the training sample and the holdout sample is also called a *fold*.

Model Selection and Stepwise Regression

In some problems, many variables could be used as predictors in a regression. For example, to predict house value, additional variables such as the basement size or year built could be used. In *R*, these are easy to add to the regression equation:

```
house_full <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
                   Bedrooms + BldgGrade + PropertyType + NbrLivingUnits +
                   SqFtFinBasement + YrBuilt + YrRenovated +
                   NewConstruction,
                 data=house, na.action=na.omit)
```

In *Python*, we need to convert the categorical and boolean variables into numbers:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms', 'BldgGrade',
              'PropertyType', 'NbrLivingUnits', 'SqFtFinBasement', 'YrBuilt',
              'YrRenovated', 'NewConstruction']

X = pd.get_dummies(house[predictors], drop_first=True)
X['NewConstruction'] = [1 if nc else 0 for nc in X['NewConstruction']]

house_full = sm.OLS(house[outcome], X.assign(const=1))
results = house_full.fit()
results.summary()
```

Adding more variables, however, does not necessarily mean we have a better model. Statisticians use the principle of *Occam's razor* to guide the choice of a model: all things being equal, a simpler model should be used in preference to a more complicated model.

Including additional variables always reduces RMSE and increases R^2 for the training data. Hence, these are not appropriate to help guide the model choice. One approach to including model complexity is to use the adjusted R^2:

$$R_{adj}^2 = 1 - (1 - R^2)\frac{n-1}{n-P-1}$$

Here, n is the number of records and P is the number of variables in the model.

In the 1970s, Hirotugu Akaike, the eminent Japanese statistician, developed a metric called *AIC* (Akaike's Information Criteria) that penalizes adding terms to a model. In the case of regression, AIC has the form:

$$\text{AIC} = 2P + n\log(\texttt{RSS}/n)$$

where P is the number of variables and n is the number of records. The goal is to find the model that minimizes AIC; models with k more extra variables are penalized by $2k$.

AIC, BIC, and Mallows Cp

The formula for AIC may seem a bit mysterious, but in fact it is based on asymptotic results in information theory. There are several variants to AIC:

AICc
: A version of AIC corrected for small sample sizes.

BIC or Bayesian information criteria
: Similar to AIC, with a stronger penalty for including additional variables to the model.

Mallows Cp
: A variant of AIC developed by Colin Mallows.

These are typically reported as in-sample metrics (i.e., on the training data), and data scientists using holdout data for model assessment do not need to worry about the differences among them or the underlying theory behind them.

How do we find the model that minimizes AIC or maximizes adjusted R^2? One way is to search through all possible models, an approach called *all subset regression*. This is computationally expensive and is not feasible for problems with large data and many variables. An attractive alternative is to use *stepwise regression*. It could start with a full model and successively drop variables that don't contribute meaningfully. This is called *backward elimination*. Alternatively one could start with a constant model and successively add variables (*forward selection*). As a third option we can also successively add and drop predictors to find a model that lowers AIC or adjusted R^2. The `MASS` in *R* package by Venebles and Ripley offers a stepwise regression function called `stepAIC`:

```
library(MASS)
step <- stepAIC(house_full, direction="both")
step

Call:
```

```
lm(formula = AdjSalePrice ~ SqFtTotLiving + Bathrooms + Bedrooms +
    BldgGrade + PropertyType + SqFtFinBasement + YrBuilt, data = house,
    na.action = na.omit)

Coefficients:
              (Intercept)              SqFtTotLiving
                6.179e+06                  1.993e+02
                Bathrooms                   Bedrooms
                4.240e+04                 -5.195e+04
                BldgGrade  PropertyTypeSingle Family
                1.372e+05                  2.291e+04
    PropertyTypeTownhouse            SqFtFinBasement
                8.448e+04                  7.047e+00
                  YrBuilt
               -3.565e+03
```

scikit-learn has no implementation for stepwise regression. We implemented functions stepwise_selection, forward_selection, and backward_elimination in our dmba package:

```
y = house[outcome]

def train_model(variables): ❶
    if len(variables) == 0:
        return None
    model = LinearRegression()
    model.fit(X[variables], y)
    return model

def score_model(model, variables): ❷
    if len(variables) == 0:
        return AIC_score(y, [y.mean()] * len(y), model, df=1)
    return AIC_score(y, model.predict(X[variables]), model)

best_model, best_variables = stepwise_selection(X.columns, train_model,
                                                score_model, verbose=True)

print(f'Intercept: {best_model.intercept_:.3f}')
print('Coefficients:')
for name, coef in zip(best_variables, best_model.coef_):
    print(f' {name}: {coef}')
```

❶ Define a function that returns a fitted model for a given set of variables.

❷ Define a function that returns a score for a given model and set of variables. In this case, we use the AIC_score implemented in the dmba package.

The function chose a model in which several variables were dropped from house_full: SqFtLot, NbrLivingUnits, YrRenovated, and NewConstruction.

Simpler yet are *forward selection* and *backward selection*. In forward selection, you start with no predictors and add them one by one, at each step adding the predictor that has the largest contribution to R^2, and stopping when the contribution is no longer statistically significant. In backward selection, or *backward elimination*, you start with the full model and take away predictors that are not statistically significant until you are left with a model in which all predictors are statistically significant.

Penalized regression is similar in spirit to AIC. Instead of explicitly searching through a discrete set of models, the model-fitting equation incorporates a constraint that penalizes the model for too many variables (parameters). Rather than eliminating predictor variables entirely—as with stepwise, forward, and backward selection—penalized regression applies the penalty by reducing coefficients, in some cases to near zero. Common penalized regression methods are *ridge regression* and *lasso regression*.

Stepwise regression and all subset regression are *in-sample* methods to assess and tune models. This means the model selection is possibly subject to overfitting (fitting the noise in the data) and may not perform as well when applied to new data. One common approach to avoid this is to use cross-validation to validate the models. In linear regression, overfitting is typically not a major issue, due to the simple (linear) global structure imposed on the data. For more sophisticated types of models, particularly iterative procedures that respond to local data structure, cross-validation is a very important tool; see "Cross-Validation" on page 155 for details.

Weighted Regression

Weighted regression is used by statisticians for a variety of purposes; in particular, it is important for analysis of complex surveys. Data scientists may find weighted regression useful in two cases:

- Inverse-variance weighting when different observations have been measured with different precision; the higher variance ones receiving lower weights.
- Analysis of data where rows represent multiple cases; the weight variable encodes how many original observations each row represents.

For example, with the housing data, older sales are less reliable than more recent sales. Using the `DocumentDate` to determine the year of the sale, we can compute a `Weight` as the number of years since 2005 (the beginning of the data):

R

```
library(lubridate)
house$Year = year(house$DocumentDate)
house$Weight = house$Year - 2005
```

Python

```
house['Year'] = [int(date.split('-')[0]) for date in house.DocumentDate]
house['Weight'] = house.Year - 2005
```

We can compute a weighted regression with the `lm` function using the `weight` argument:

```
house_wt <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
                 Bedrooms + BldgGrade,
               data=house, weight=Weight)
round(cbind(house_lm=house_lm$coefficients,
            house_wt=house_wt$coefficients), digits=3)

                house_lm    house_wt
(Intercept)  -521871.368 -584189.329
SqFtTotLiving    228.831     245.024
SqFtLot           -0.060      -0.292
Bathrooms     -19442.840  -26085.970
Bedrooms      -47769.955  -53608.876
BldgGrade     106106.963  115242.435
```

The coefficients in the weighted regression are slightly different from the original regression.

Most models in `scikit-learn` accept weights as the keyword argument `sample_weight` in the call of the `fit` method:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms', 'BldgGrade']
outcome = 'AdjSalePrice'

house_wt = LinearRegression()
house_wt.fit(house[predictors], house[outcome], sample_weight=house.Weight)
```

Key Ideas

- Multiple linear regression models the relationship between a response variable Y and multiple predictor variables $X_1, ..., X_p$.
- The most important metrics to evaluate a model are root mean squared error (RMSE) and R-squared (R^2).
- The standard error of the coefficients can be used to measure the reliability of a variable's contribution to a model.
- Stepwise regression is a way to automatically determine which variables should be included in the model.
- Weighted regression is used to give certain records more or less weight in fitting the equation.

Further Reading

An excellent treatment of cross-validation and resampling can be found in *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (Springer, 2013).

Prediction Using Regression

The primary purpose of regression in data science is prediction. This is useful to keep in mind, since regression, being an old and established statistical method, comes with baggage that is more relevant to its traditional role as a tool for explanatory modeling than to prediction.

Key Terms for Prediction Using Regression

Prediction interval
An uncertainty interval around an individual predicted value.

Extrapolation
Extension of a model beyond the range of the data used to fit it.

The Dangers of Extrapolation

Regression models should not be used to extrapolate beyond the range of the data (leaving aside the use of regression for time series forecasting.). The model is valid only for predictor values for which the data has sufficient values (even in the case that sufficient data is available, there could be other problems—see "Regression Diagnostics" on page 176). As an extreme case, suppose `model_lm` is used to predict the value of a 5,000-square-foot empty lot. In such a case, all the predictors related to the building would have a value of 0, and the regression equation would yield an absurd prediction of $-521{,}900 + 5{,}000 \times -.0605 = -\$522{,}202$. Why did this happen? The data contains only parcels with buildings—there are no records corresponding to vacant land. Consequently, the model has no information to tell it how to predict the sales price for vacant land.

Confidence and Prediction Intervals

Much of statistics involves understanding and measuring variability (uncertainty). The t-statistics and p-values reported in regression output deal with this in a formal way, which is sometimes useful for variable selection (see "Assessing the Model" on page 153). More useful metrics are confidence intervals, which are uncertainty intervals placed around regression coefficients and predictions. An easy way to understand this is via the bootstrap (see "The Bootstrap" on page 61 for more details about the general bootstrap procedure). The most common regression confidence intervals encountered in software output are those for regression parameters (coefficients). Here is a bootstrap algorithm for generating confidence intervals for regression parameters (coefficients) for a data set with *P* predictors and *n* records (rows):

1. Consider each row (including outcome variable) as a single "ticket" and place all the *n* tickets in a box.
2. Draw a ticket at random, record the values, and replace it in the box.
3. Repeat step 2 *n* times; you now have one bootstrap resample.
4. Fit a regression to the bootstrap sample, and record the estimated coefficients.
5. Repeat steps 2 through 4, say, 1,000 times.
6. You now have 1,000 bootstrap values for each coefficient; find the appropriate percentiles for each one (e.g., 5th and 95th for a 90% confidence interval).

You can use the `Boot` function in *R* to generate actual bootstrap confidence intervals for the coefficients, or you can simply use the formula-based intervals that are a routine *R* output. The conceptual meaning and interpretation are the same, and not of central importance to data scientists, because they concern the regression coefficients. Of greater interest to data scientists are intervals around predicted *y* values ($\hat{Y}_i$). The uncertainty around $\hat{Y}_i$ comes from two sources:

- Uncertainty about what the relevant predictor variables and their coefficients are (see the preceding bootstrap algorithm)
- Additional error inherent in individual data points

The individual data point error can be thought of as follows: even if we knew for certain what the regression equation was (e.g., if we had a huge number of records to fit it), the *actual* outcome values for a given set of predictor values will vary. For example, several houses—each with 8 rooms, a 6,500-square-foot lot, 3 bathrooms, and a basement—might have different values. We can model this individual error with the residuals from the fitted values. The bootstrap algorithm for modeling both the regression model error and the individual data point error would look as follows:

1. Take a bootstrap sample from the data (spelled out in greater detail earlier).
2. Fit the regression, and predict the new value.
3. Take a single residual at random from the original regression fit, add it to the predicted value, and record the result.
4. Repeat steps 1 through 3, say, 1,000 times.
5. Find the 2.5th and the 97.5th percentiles of the results.

Key Ideas

- Extrapolation beyond the range of the data can lead to error.
- Confidence intervals quantify uncertainty around regression coefficients.
- Prediction intervals quantify uncertainty in individual predictions.
- Most software, *R* included, will produce prediction and confidence intervals in default or specified output, using formulas.
- The bootstrap can also be used to produce prediction and confidence intervals; the interpretation and idea are the same.

Prediction Interval or Confidence Interval?

A prediction interval pertains to uncertainty around a single value, while a confidence interval pertains to a mean or other statistic calculated from multiple values. Thus, a prediction interval will typically be much wider than a confidence interval for the same value. We model this individual value error in the bootstrap model by selecting an individual residual to tack on to the predicted value. Which should you use? That depends on the context and the purpose of the analysis, but, in general, data scientists are interested in specific individual predictions, so a prediction interval would be more appropriate. Using a confidence interval when you should be using a prediction interval will greatly underestimate the uncertainty in a given predicted value.

Factor Variables in Regression

Factor variables, also termed *categorical* variables, take on a limited number of discrete values. For example, a loan purpose can be "debt consolidation," "wedding," "car," and so on. The binary (yes/no) variable, also called an *indicator* variable, is a special case of a factor variable. Regression requires numerical inputs, so factor variables need to be recoded to use in the model. The most common approach is to convert a variable into a set of binary *dummy* variables.

Key Terms for Factor Variables

Dummy variables
Binary 0–1 variables derived by recoding factor data for use in regression and other models.

Reference coding
The most common type of coding used by statisticians, in which one level of a factor is used as a reference and other factors are compared to that level.

Synonym
treatment coding

One hot encoder
A common type of coding used in the machine learning community in which all factor levels are retained. While useful for certain machine learning algorithms, this approach is not appropriate for multiple linear regression.

Deviation coding
A type of coding that compares each level against the overall mean as opposed to the reference level.

Synonym
sum contrasts

Dummy Variables Representation

In the King County housing data, there is a factor variable for the property type; a small subset of six records is shown below:

R:

```
head(house[, 'PropertyType'])
Source: local data frame [6 x 1]

   PropertyType
         (fctr)
1     Multiplex
2 Single Family
3 Single Family
4 Single Family
5 Single Family
6     Townhouse
```

Python:

```
house.PropertyType.head()
```

There are three possible values: `Multiplex`, `Single Family`, and `Townhouse`. To use this factor variable, we need to convert it to a set of binary variables. We do this by creating a binary variable for each possible value of the factor variable. To do this in *R*, we use the `model.matrix` function:[3]

```
prop_type_dummies <- model.matrix(~PropertyType -1, data=house)
head(prop_type_dummies)
  PropertyTypeMultiplex PropertyTypeSingle Family PropertyTypeTownhouse
1                     1                         0                     0
2                     0                         1                     0
3                     0                         1                     0
4                     0                         1                     0
5                     0                         1                     0
6                     0                         0                     1
```

The function `model.matrix` converts a data frame into a matrix suitable to a linear model. The factor variable `PropertyType`, which has three distinct levels, is represented as a matrix with three columns. In the machine learning community, this representation is referred to as *one hot encoding* (see "One Hot Encoder" on page 242).

In *Python*, we can convert categorical variables to dummies using the `pandas` method `get_dummies`:

```
pd.get_dummies(house['PropertyType']).head() ❶
pd.get_dummies(house['PropertyType'], drop_first=True).head() ❷
```

❶ By default, returns one hot encoding of the categorical variable.

❷ The keyword argument `drop_first` will return *P* – 1 columns. Use this to avoid the problem of multicollinearity.

In certain machine learning algorithms, such as nearest neighbors and tree models, one hot encoding is the standard way to represent factor variables (for example, see "Tree Models" on page 249).

In the regression setting, a factor variable with *P* distinct levels is usually represented by a matrix with only *P* – 1 columns. This is because a regression model typically includes an intercept term. With an intercept, once you have defined the values for *P* – 1 binaries, the value for the *P*th is known and could be considered redundant. Adding the *P*th column will cause a multicollinearity error (see "Multicollinearity" on page 172).

3 The `-1` argument in the `model.matrix` produces one hot encoding representation (by removing the intercept, hence the "-"). Otherwise, the default in *R* is to produce a matrix with *P* – 1 columns with the first factor level as a reference.

The default representation in *R* is to use the first factor level as a *reference* and interpret the remaining levels relative to that factor:

```
lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
       Bedrooms + BldgGrade + PropertyType, data=house)

Call:
lm(formula = AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
    Bedrooms + BldgGrade + PropertyType, data = house)

Coefficients:
              (Intercept)              SqFtTotLiving
               -4.468e+05                  2.234e+02
                  SqFtLot                  Bathrooms
               -7.037e-02                 -1.598e+04
                 Bedrooms                  BldgGrade
               -5.089e+04                  1.094e+05
PropertyTypeSingle Family       PropertyTypeTownhouse
               -8.468e+04                 -1.151e+05
```

The method `get_dummies` takes the optional keyword argument `drop_first` to exclude the first factor as *reference*:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms',
              'BldgGrade', 'PropertyType']

X = pd.get_dummies(house[predictors], drop_first=True)

house_lm_factor = LinearRegression()
house_lm_factor.fit(X, house[outcome])

print(f'Intercept: {house_lm_factor.intercept_:.3f}')
print('Coefficients:')
for name, coef in zip(X.columns, house_lm_factor.coef_):
    print(f' {name}: {coef}')
```

The output from the *R* regression shows two coefficients corresponding to `PropertyType`: `PropertyTypeSingle Family` and `PropertyTypeTownhouse`. There is no coefficient of `Multiplex` since it is implicitly defined when `PropertyTypeSingle Family == 0` and `PropertyTypeTownhouse == 0`. The coefficients are interpreted as relative to `Multiplex`, so a home that is `Single Family` is worth almost $85,000 less, and a home that is `Townhouse` is worth over $150,000 less.[4]

4 This is unintuitive, but can be explained by the impact of location as a confounding variable; see "Confounding Variables" on page 172.

Different Factor Codings

There are several different ways to encode factor variables, known as *contrast coding* systems. For example, *deviation coding*, also known as *sum contrasts*, compares each level against the overall mean. Another contrast is *polynomial coding*, which is appropriate for ordered factors; see the section "Ordered Factor Variables" on page 169. With the exception of ordered factors, data scientists will generally not encounter any type of coding besides reference coding or one hot encoder.

Factor Variables with Many Levels

Some factor variables can produce a huge number of binary dummies—zip codes are a factor variable, and there are 43,000 zip codes in the US. In such cases, it is useful to explore the data, and the relationships between predictor variables and the outcome, to determine whether useful information is contained in the categories. If so, you must further decide whether it is useful to retain all factors, or whether the levels should be consolidated.

In King County, there are 80 zip codes with a house sale:

```
table(house$ZipCode)

98001 98002 98003 98004 98005 98006 98007 98008 98010 98011 98014 98019
  358   180   241   293   133   460   112   291    56   163    85   242
98022 98023 98024 98027 98028 98029 98030 98031 98032 98033 98034 98038
  188   455    31   366   252   475   263   308   121   517   575   788
98039 98040 98042 98043 98045 98047 98050 98051 98052 98053 98055 98056
   47   244   641     1   222    48     7    32   614   499   332   402
98057 98058 98059 98065 98068 98070 98072 98074 98075 98077 98092 98102
    4   420   513   430     1    89   245   502   388   204   289   106
98103 98105 98106 98107 98108 98109 98112 98113 98115 98116 98117 98118
  671   313   361   296   155   149   357     1   620   364   619   492
98119 98122 98125 98126 98133 98136 98144 98146 98148 98155 98166 98168
  260   380   409   473   465   310   332   287    40   358   193   332
98177 98178 98188 98198 98199 98224 98288 98354
  216   266   101   225   393     3     4     9
```

The `value_counts` method of `pandas` data frames returns the same information:

```
pd.DataFrame(house['ZipCode'].value_counts()).transpose()
```

`ZipCode` is an important variable, since it is a proxy for the effect of location on the value of a house. Including all levels requires 79 coefficients corresponding to 79 degrees of freedom. The original model `house_lm` has only 5 degrees of freedom; see "Assessing the Model" on page 153. Moreover, several zip codes have only one sale. In some problems, you can consolidate a zip code using the first two or three digits, corresponding to a submetropolitan geographic region. For King County, almost all of the sales occur in 980xx or 981xx, so this doesn't help.

An alternative approach is to group the zip codes according to another variable, such as sale price. Even better is to form zip code groups using the residuals from an initial model. The following `dplyr` code in *R* consolidates the 80 zip codes into five groups based on the median of the residual from the `house_lm` regression:

```
zip_groups <- house %>%
  mutate(resid = residuals(house_lm)) %>%
  group_by(ZipCode) %>%
  summarize(med_resid = median(resid),
            cnt = n()) %>%
  arrange(med_resid) %>%
  mutate(cum_cnt = cumsum(cnt),
         ZipGroup = ntile(cum_cnt, 5))
house <- house %>%
  left_join(select(zip_groups, ZipCode, ZipGroup), by='ZipCode')
```

The median residual is computed for each zip, and the `ntile` function is used to split the zip codes, sorted by the median, into five groups. See "Confounding Variables" on page 172 for an example of how this is used as a term in a regression improving upon the original fit.

In *Python* we can calculate this information as follows:

```
zip_groups = pd.DataFrame([
    *pd.DataFrame({
        'ZipCode': house['ZipCode'],
        'residual' : house[outcome] - house_lm.predict(house[predictors]),
    })
    .groupby(['ZipCode'])
    .apply(lambda x: {
        'ZipCode': x.iloc[0,0],
        'count': len(x),
        'median_residual': x.residual.median()
    })
]).sort_values('median_residual')
zip_groups['cum_count'] = np.cumsum(zip_groups['count'])
zip_groups['ZipGroup'] = pd.qcut(zip_groups['cum_count'], 5, labels=False,
                                 retbins=False)

to_join = zip_groups[['ZipCode', 'ZipGroup']].set_index('ZipCode')
house = house.join(to_join, on='ZipCode')
house['ZipGroup'] = house['ZipGroup'].astype('category')
```

The concept of using the residuals to help guide the regression fitting is a fundamental step in the modeling process; see "Regression Diagnostics" on page 176.

Ordered Factor Variables

Some factor variables reflect levels of a factor; these are termed *ordered factor variables* or *ordered categorical variables*. For example, the loan grade could be A, B, C, and so on—each grade carries more risk than the prior grade. Often, ordered factor variables can be converted to numerical values and used as is. For example, the variable `BldgGrade` is an ordered factor variable. Several of the types of grades are shown in Table 4-1. While the grades have specific meaning, the numeric value is ordered from low to high, corresponding to higher-grade homes. With the regression model `house_lm`, fit in "Multiple Linear Regression" on page 150, `BldgGrade` was treated as a numeric variable.

Table 4-1. Building grades and numeric equivalents

Value	Description
1	Cabin
2	Substandard
5	Fair
10	Very good
12	Luxury
13	Mansion

Treating ordered factors as a numeric variable preserves the information contained in the ordering that would be lost if it were converted to a factor.

Key Ideas

- Factor variables need to be converted into numeric variables for use in a regression.
- The most common method to encode a factor variable with P distinct values is to represent them using P – 1 dummy variables.
- A factor variable with many levels, even in very big data sets, may need to be consolidated into a variable with fewer levels.
- Some factors have levels that are ordered and can be represented as a single numeric variable.

Interpreting the Regression Equation

In data science, the most important use of regression is to predict some dependent (outcome) variable. In some cases, however, gaining insight from the equation itself to understand the nature of the relationship between the predictors and the outcome can be of value. This section provides guidance on examining the regression equation and interpreting it.

Key Terms for Interpreting the Regression Equation

Correlated variables
: Variables that tend to move in the same direction—when one goes up so does the other, and vice-versa (with negative correlation, when one goes up the other does down). When the predictor variables are highly correlated, it is difficult to interpret the individual coefficients.

Multicollinearity
: When the predictor variables have perfect, or near-perfect, correlation, the regression can be unstable or impossible to compute.

  Synonym
  : collinearity

Confounding variables
: An important predictor that, when omitted, leads to spurious relationships in a regression equation.

Main effects
: The relationship between a predictor and the outcome variable, independent of other variables.

Interactions
: An interdependent relationship between two or more predictors and the response.

Correlated Predictors

In multiple regression, the predictor variables are often correlated with each other. As an example, examine the regression coefficients for the model `step_lm`, fit in "Model Selection and Stepwise Regression" on page 156.

R:

```
step_lm$coefficients
           (Intercept)             SqFtTotLiving                 Bathrooms
          6.178645e+06              1.992776e+02              4.239616e+04
              Bedrooms                 BldgGrade PropertyTypeSingle Family
         -5.194738e+04              1.371596e+05              2.291206e+04
 PropertyTypeTownhouse             SqFtFinBasement                   YrBuilt
          8.447916e+04              7.046975e+00             -3.565425e+03
```

Python:

```
print(f'Intercept: {best_model.intercept_:.3f}')
print('Coefficients:')
for name, coef in zip(best_variables, best_model.coef_):
    print(f' {name}: {coef}')
```

The coefficient for `Bedrooms` is negative! This implies that adding a bedroom to a house will reduce its value. How can this be? This is because the predictor variables are correlated: larger houses tend to have more bedrooms, and it is the size that drives house value, not the number of bedrooms. Consider two homes of the exact same size: it is reasonable to expect that a home with more but smaller bedrooms would be considered less desirable.

Having correlated predictors can make it difficult to interpret the sign and value of regression coefficients (and can inflate the standard error of the estimates). The variables for bedrooms, house size, and number of bathrooms are all correlated. This is illustrated by the following example in *R*, which fits another regression removing the variables `SqFtTotLiving`, `SqFtFinBasement`, and `Bathrooms` from the equation:

```
update(step_lm, . ~ . - SqFtTotLiving - SqFtFinBasement - Bathrooms)

Call:
lm(formula = AdjSalePrice ~ Bedrooms + BldgGrade + PropertyType +
    YrBuilt, data = house, na.action = na.omit)

Coefficients:
              (Intercept)                   Bedrooms
                  4913973                      27151
                BldgGrade  PropertyTypeSingle Family
                   248998                     -19898
    PropertyTypeTownhouse                    YrBuilt
                   -47355                      -3212
```

The `update` function can be used to add or remove variables from a model. Now the coefficient for bedrooms is positive—in line with what we would expect (though it is really acting as a proxy for house size, now that those variables have been removed).

In *Python*, there is no equivalent to *R*'s `update` function. We need to refit the model with the modified predictor list:

```
predictors = ['Bedrooms', 'BldgGrade', 'PropertyType', 'YrBuilt']
outcome = 'AdjSalePrice'

X = pd.get_dummies(house[predictors], drop_first=True)

reduced_lm = LinearRegression()
reduced_lm.fit(X, house[outcome])
```

Correlated variables are only one issue with interpreting regression coefficients. In `house_lm`, there is no variable to account for the location of the home, and the model is mixing together very different types of regions. Location may be a *confounding* variable; see "Confounding Variables" on page 172 for further discussion.

Multicollinearity

An extreme case of correlated variables produces multicollinearity—a condition in which there is redundance among the predictor variables. Perfect multicollinearity occurs when one predictor variable can be expressed as a linear combination of others. Multicollinearity occurs when:

- A variable is included multiple times by error.
- *P* dummies, instead of *P* – 1 dummies, are created from a factor variable (see "Factor Variables in Regression" on page 163).
- Two variables are nearly perfectly correlated with one another.

Multicollinearity in regression must be addressed—variables should be removed until the multicollinearity is gone. A regression does not have a well-defined solution in the presence of perfect multicollinearity. Many software packages, including *R* and *Python*, automatically handle certain types of multicollinearity. For example, if `SqFtTotLiving` is included twice in the regression of the `house` data, the results are the same as for the `house_lm` model. In the case of nonperfect multicollinearity, the software may obtain a solution, but the results may be unstable.

Multicollinearity is not such a problem for nonlinear regression methods like trees, clustering, and nearest-neighbors, and in such methods it may be advisable to retain *P* dummies (instead of *P* – 1). That said, even in those methods, nonredundancy in predictor variables is still a virtue.

Confounding Variables

With correlated variables, the problem is one of commission: including different variables that have a similar predictive relationship with the response. With *confounding variables*, the problem is one of omission: an important variable is not included in the regression equation. Naive interpretation of the equation coefficients can lead to invalid conclusions.

Take, for example, the King County regression equation `house_lm` from "Example: King County Housing Data" on page 151. The regression coefficients of `SqFtLot`, `Bathrooms`, and `Bedrooms` are all negative. The original regression model does not contain a variable to represent location—a very important predictor of house price.

To model location, include a variable `ZipGroup` that categorizes the zip code into one of five groups, from least expensive (1) to most expensive (5):[5]

```
lm(formula = AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
    Bedrooms + BldgGrade + PropertyType + ZipGroup, data = house,
    na.action = na.omit)

Coefficients:
              (Intercept)            SqFtTotLiving
               -6.666e+05                2.106e+02
                  SqFtLot                Bathrooms
                4.550e-01                5.928e+03
                 Bedrooms                BldgGrade
               -4.168e+04                9.854e+04
PropertyTypeSingle Family    PropertyTypeTownhouse
                1.932e+04               -7.820e+04
                ZipGroup2                ZipGroup3
                5.332e+04                1.163e+05
                ZipGroup4                ZipGroup5
                1.784e+05                3.384e+05
```

The same model in *Python*:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms',
              'BldgGrade', 'PropertyType', 'ZipGroup']
outcome = 'AdjSalePrice'

X = pd.get_dummies(house[predictors], drop_first=True)

confounding_lm = LinearRegression()
confounding_lm.fit(X, house[outcome])

print(f'Intercept: {confounding_lm.intercept_:.3f}')
print('Coefficients:')
for name, coef in zip(X.columns, confounding_lm.coef_):
    print(f' {name}: {coef}')
```

`ZipGroup` is clearly an important variable: a home in the most expensive zip code group is estimated to have a higher sales price by almost $340,000. The coefficients of `SqFtLot` and `Bathrooms` are now positive, and adding a bathroom increases the sale price by $5,928.

The coefficient for `Bedrooms` is still negative. While this is unintuitive, this is a well-known phenomenon in real estate. For homes of the same livable area and number of bathrooms, having more and therefore smaller bedrooms is associated with less valuable homes.

5 There are 80 zip codes in King County, several with just a handful of sales. An alternative to directly using zip code as a factor variable, `ZipGroup` clusters similar zip codes into a single group. See "Factor Variables with Many Levels" on page 167 for details.

Interactions and Main Effects

Statisticians like to distinguish between *main effects*, or independent variables, and the *interactions* between the main effects. Main effects are what are often referred to as the *predictor variables* in the regression equation. An implicit assumption when only main effects are used in a model is that the relationship between a predictor variable and the response is independent of the other predictor variables. This is often not the case.

For example, the model fit to the King County Housing Data in "Confounding Variables" on page 172 includes several variables as main effects, including `ZipCode`. Location in real estate is everything, and it is natural to presume that the relationship between, say, house size and the sale price depends on location. A big house built in a low-rent district is not going to retain the same value as a big house built in an expensive area. You include interactions between variables in *R* using the `*` operator. For the King County data, the following fits an interaction between `SqFtTotLiving` and `ZipGroup`:

```
lm(formula = AdjSalePrice ~ SqFtTotLiving * ZipGroup + SqFtLot +
    Bathrooms + Bedrooms + BldgGrade + PropertyType, data = house,
    na.action = na.omit)

Coefficients:
                 (Intercept)               SqFtTotLiving
                  -4.853e+05                   1.148e+02
                   ZipGroup2                   ZipGroup3
                  -1.113e+04                   2.032e+04
                   ZipGroup4                   ZipGroup5
                   2.050e+04                  -1.499e+05
                     SqFtLot                   Bathrooms
                   6.869e-01                  -3.619e+03
                    Bedrooms                   BldgGrade
                  -4.180e+04                   1.047e+05
PropertyTypeSingle Family     PropertyTypeTownhouse
                   1.357e+04                  -5.884e+04
     SqFtTotLiving:ZipGroup2     SqFtTotLiving:ZipGroup3
                   3.260e+01                   4.178e+01
     SqFtTotLiving:ZipGroup4     SqFtTotLiving:ZipGroup5
                   6.934e+01                   2.267e+02
```

The resulting model has four new terms: `SqFtTotLiving:ZipGroup2`, `SqFtTotLiving:ZipGroup3`, and so on.

In *Python*, we need to use the `statsmodels` package to train linear regression models with interactions. This package was designed similar to *R* and allows defining models using a formula interface:

```
model = smf.ols(formula='AdjSalePrice ~ SqFtTotLiving*ZipGroup + SqFtLot + ' +
     'Bathrooms + Bedrooms + BldgGrade + PropertyType', data=house)
```

```
    results = model.fit()
    results.summary()
```

The `statsmodels` package takes care of categorical variables (e.g., `ZipGroup[T.1]`, `PropertyType[T.Single Family]`) and interaction terms (e.g., `SqFtTotLiving:ZipGroup[T.1]`).

Location and house size appear to have a strong interaction. For a home in the lowest `ZipGroup`, the slope is the same as the slope for the main effect `SqFtTotLiving`, which is $118 per square foot (this is because *R* uses *reference* coding for factor variables; see "Factor Variables in Regression" on page 163). For a home in the highest `ZipGroup`, the slope is the sum of the main effect plus `SqFtTotLiving:ZipGroup5`, or $115 + $227 = $342 per square foot. In other words, adding a square foot in the most expensive zip code group boosts the predicted sale price by a factor of almost three, compared to the average boost from adding a square foot.

Model Selection with Interaction Terms

In problems involving many variables, it can be challenging to decide which interaction terms should be included in the model. Several different approaches are commonly taken:

- In some problems, prior knowledge and intuition can guide the choice of which interaction terms to include in the model.
- Stepwise selection (see "Model Selection and Stepwise Regression" on page 156) can be used to sift through the various models.
- Penalized regression can automatically fit to a large set of possible interaction terms.
- Perhaps the most common approach is to use *tree models*, as well as their descendants, *random forest* and *gradient boosted trees*. This class of models automatically searches for optimal interaction terms; see "Tree Models" on page 249.

Key Ideas

- Because of correlation between predictors, care must be taken in the interpretation of the coefficients in multiple linear regression.
- Multicollinearity can cause numerical instability in fitting the regression equation.
- A confounding variable is an important predictor that is omitted from a model and can lead to a regression equation with spurious relationships.

- An interaction term between two variables is needed if the relationship between the variables and the response is interdependent.

Regression Diagnostics

In explanatory modeling (i.e., in a research context), various steps, in addition to the metrics mentioned previously (see "Assessing the Model" on page 153), are taken to assess how well the model fits the data; most are based on analysis of the residuals. These steps do not directly address predictive accuracy, but they can provide useful insight in a predictive setting.

Key Terms for Regression Diagnostics

Standardized residuals
: Residuals divided by the standard error of the residuals.

Outliers
: Records (or outcome values) that are distant from the rest of the data (or the predicted outcome).

Influential value
: A value or record whose presence or absence makes a big difference in the regression equation.

Leverage
: The degree of influence that a single record has on a regression equation.

  Synonym
  : hat-value

Non-normal residuals
: Non-normally distributed residuals can invalidate some technical requirements of regression but are usually not a concern in data science.

Heteroskedasticity
: When some ranges of the outcome experience residuals with higher variance (may indicate a predictor missing from the equation).

Partial residual plots
: A diagnostic plot to illuminate the relationship between the outcome variable and a single predictor.

  Synonym
  : added variables plot

Outliers

Generally speaking, an extreme value, also called an *outlier*, is one that is distant from most of the other observations. Just as outliers need to be handled for estimates of location and variability (see "Estimates of Location" on page 7 and "Estimates of Variability" on page 13), outliers can cause problems with regression models. In regression, an outlier is a record whose actual *y* value is distant from the predicted value. You can detect outliers by examining the *standardized residual*, which is the residual divided by the standard error of the residuals.

There is no statistical theory that separates outliers from nonoutliers. Rather, there are (arbitrary) rules of thumb for how distant from the bulk of the data an observation needs to be in order to be called an outlier. For example, with the boxplot, outliers are those data points that are too far above or below the box boundaries (see "Percentiles and Boxplots" on page 20), where "too far" = "more than 1.5 times the interquartile range." In regression, the standardized residual is the metric that is typically used to determine whether a record is classified as an outlier. Standardized residuals can be interpreted as "the number of standard errors away from the regression line."

Let's fit a regression to the King County house sales data for all sales in zip code 98105 in *R*:

```
house_98105 <- house[house$ZipCode == 98105,]
lm_98105 <- lm(AdjSalePrice ~ SqFtTotLiving + SqFtLot + Bathrooms +
                 Bedrooms + BldgGrade, data=house_98105)
```

In *Python*:

```
house_98105 = house.loc[house['ZipCode'] == 98105, ]

predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms', 'Bedrooms', 'BldgGrade']
outcome = 'AdjSalePrice'

house_outlier = sm.OLS(house_98105[outcome],
                       house_98105[predictors].assign(const=1))
result_98105 = house_outlier.fit()
```

We extract the standardized residuals in *R* using the `rstandard` function and obtain the index of the smallest residual using the `order` function:

```
sresid <- rstandard(lm_98105)
idx <- order(sresid)
sresid[idx[1]]
    20429
-4.326732
```

In `statsmodels`, use `OLSInfluence` to analyze the residuals:

```
influence = OLSInfluence(result_98105)
sresiduals = influence.resid_studentized_internal
sresiduals.idxmin(), sresiduals.min()
```

The biggest overestimate from the model is more than four standard errors above the regression line, corresponding to an overestimate of $757,754. The original data record corresponding to this outlier is as follows in *R*:

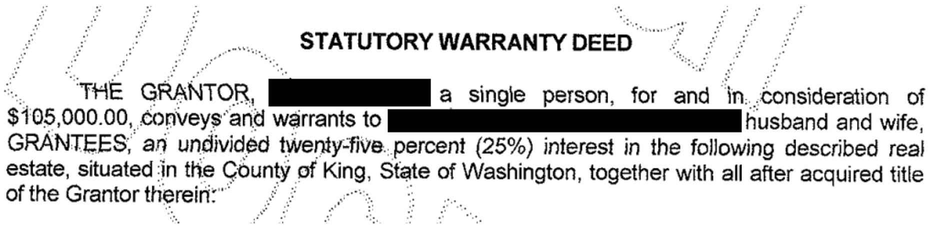
```
house_98105[idx[1], c('AdjSalePrice', 'SqFtTotLiving', 'SqFtLot',
              'Bathrooms', 'Bedrooms', 'BldgGrade')]

AdjSalePrice SqFtTotLiving SqFtLot Bathrooms Bedrooms BldgGrade
         (dbl)          (int)   (int)     (dbl)    (int)     (int)
20429   119748          2900    7276         3        6         7
```

In *Python*:

```
outlier = house_98105.loc[sresiduals.idxmin(), :]
print('AdjSalePrice', outlier[outcome])
print(outlier[predictors])
```

In this case, it appears that there is something wrong with the record: a house of that size typically sells for much more than $119,748 in that zip code. Figure 4-4 shows an excerpt from the statutory deed from this sale: it is clear that the sale involved only partial interest in the property. In this case, the outlier corresponds to a sale that is anomalous and should not be included in the regression. Outliers could also be the result of other problems, such as a "fat-finger" data entry or a mismatch of units (e.g., reporting a sale in thousands of dollars rather than simply in dollars).



STATUTORY WARRANTY DEED

THE GRANTOR, [redacted] a single person, for and in consideration of $105,000.00, conveys and warrants to [redacted] husband and wife, GRANTEES, an undivided twenty-five percent (25%) interest in the following described real estate, situated in the County of King, State of Washington, together with all after acquired title of the Grantor therein:

Figure 4-4. Statutory warrany deed for the largest negative residual

For big data problems, outliers are generally not a problem in fitting the regression to be used in predicting new data. However, outliers are central to anomaly detection, where finding outliers is the whole point. The outlier could also correspond to a case of fraud or an accidental action. In any case, detecting outliers can be a critical business need.

Influential Values

A value whose absence would significantly change the regression equation is termed an *influential observation*. In regression, such a value need not be associated with a large residual. As an example, consider the regression lines in Figure 4-5. The solid line corresponds to the regression with all the data, while the dashed line corresponds to the regression with the point in the upper-right corner removed. Clearly, that data value has a huge influence on the regression even though it is not associated with a large outlier (from the full regression). This data value is considered to have high *leverage* on the regression.

In addition to standardized residuals (see "Outliers" on page 177), statisticians have developed several metrics to determine the influence of a single record on a regression. A common measure of leverage is the *hat-value*; values above $2(P + 1)/n$ indicate a high-leverage data value.[6]

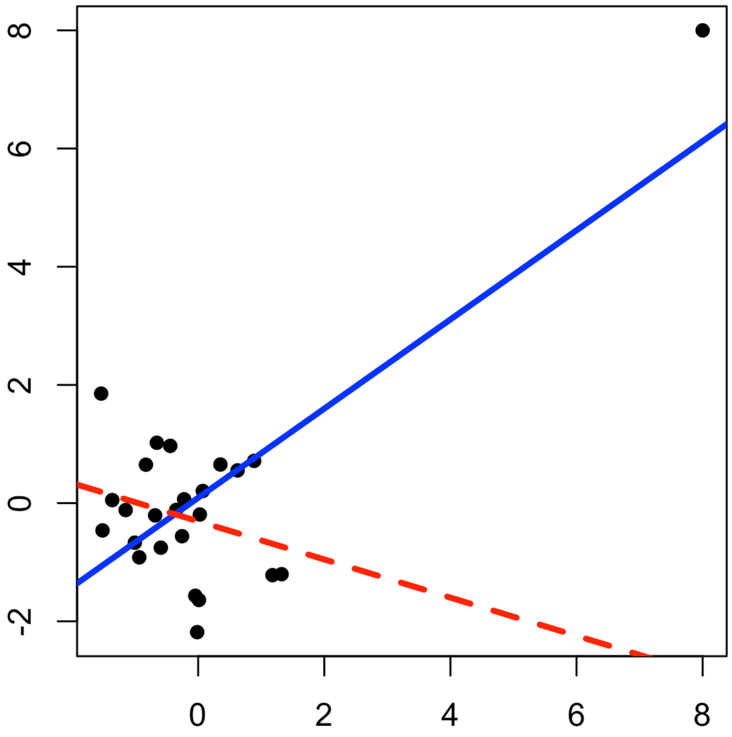


Figure 4-5. An example of an influential data point in regression

[6] The term *hat-value* comes from the notion of the hat matrix in regression. Multiple linear regression can be expressed by the formula $\hat{Y} = HY$ where H is the hat matrix. The hat-values correspond to the diagonal of H.

Another metric is *Cook's distance*, which defines influence as a combination of leverage and residual size. A rule of thumb is that an observation has high influence if Cook's distance exceeds $4/(n - P - 1)$.

An *influence plot* or *bubble plot* combines standardized residuals, the hat-value, and Cook's distance in a single plot. Figure 4-6 shows the influence plot for the King County house data and can be created by the following *R* code:

```
std_resid <- rstandard(lm_98105)
cooks_D <- cooks.distance(lm_98105)
hat_values <- hatvalues(lm_98105)
plot(subset(hat_values, cooks_D > 0.08), subset(std_resid, cooks_D > 0.08),
     xlab='hat_values', ylab='std_resid',
     cex=10*sqrt(subset(cooks_D, cooks_D > 0.08)), pch=16, col='lightgrey')
points(hat_values, std_resid, cex=10*sqrt(cooks_D))
abline(h=c(-2.5, 2.5), lty=2)
```

Here is the *Python* code to create a similar figure:

```
influence = OLSInfluence(result_98105)
fig, ax = plt.subplots(figsize=(5, 5))
ax.axhline(-2.5, linestyle='--', color='C1')
ax.axhline(2.5, linestyle='--', color='C1')
ax.scatter(influence.hat_matrix_diag, influence.resid_studentized_internal,
           s=1000 * np.sqrt(influence.cooks_distance[0]),
           alpha=0.5)
ax.set_xlabel('hat values')
ax.set_ylabel('studentized residuals')
```

There are apparently several data points that exhibit large influence in the regression. Cook's distance can be computed using the function `cooks.distance`, and you can use `hatvalues` to compute the diagnostics. The hat values are plotted on the x-axis, the residuals are plotted on the y-axis, and the size of the points is related to the value of Cook's distance.

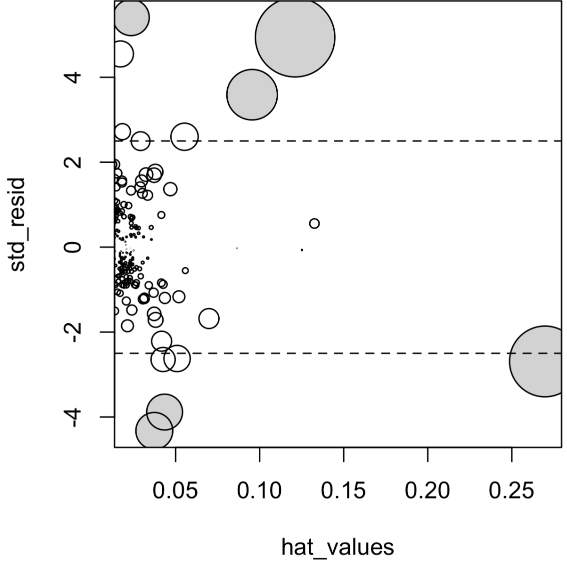


Figure 4-6. A plot to determine which observations have high influence; points with Cook's distance greater than 0.08 are highlighted in grey

Table 4-2 compares the regression with the full data set and with highly influential data points removed (Cook's distance > 0.08).

The regression coefficient for `Bathrooms` changes quite dramatically.[7]

Table 4-2. Comparison of regression coefficients with the full data and with influential data removed

	Original	Influential removed
(Intercept)	–772,550	–647,137
SqFtTotLiving	210	230
SqFtLot	39	33
Bathrooms	2282	–16,132
Bedrooms	–26,320	–22,888
BldgGrade	130,000	114,871

7 The coefficient for `Bathrooms` becomes negative, which is unintuitive. Location has not been taken into account, and the zip code 98105 contains areas of disparate types of homes. See "Confounding Variables" on page 172 for a discussion of confounding variables.

For purposes of fitting a regression that reliably predicts future data, identifying influential observations is useful only in smaller data sets. For regressions involving many records, it is unlikely that any one observation will carry sufficient weight to cause extreme influence on the fitted equation (although the regression may still have big outliers). For purposes of anomaly detection, though, identifying influential observations can be very useful.

Heteroskedasticity, Non-Normality, and Correlated Errors

Statisticians pay considerable attention to the distribution of the residuals. It turns out that ordinary least squares (see "Least Squares" on page 148) are unbiased, and in some cases are the "optimal" estimator, under a wide range of distributional assumptions. This means that in most problems, data scientists do not need to be too concerned with the distribution of the residuals.

The distribution of the residuals is relevant mainly for the validity of formal statistical inference (hypothesis tests and p-values), which is of minimal importance to data scientists concerned mainly with predictive accuracy. Normally distributed errors are a sign that the model is complete; errors that are not normally distributed indicate the model may be missing something. For formal inference to be fully valid, the residuals are assumed to be normally distributed, have the same variance, and be independent. One area where this may be of concern to data scientists is the standard calculation of confidence intervals for predicted values, which are based upon the assumptions about the residuals (see "Confidence and Prediction Intervals" on page 161).

Heteroskedasticity is the lack of constant residual variance across the range of the predicted values. In other words, errors are greater for some portions of the range than for others. Visualizing the data is a convenient way to analyze residuals.

The following code in *R* plots the absolute residuals versus the predicted values for the `lm_98105` regression fit in "Outliers" on page 177:

```
df <- data.frame(resid = residuals(lm_98105), pred = predict(lm_98105))
ggplot(df, aes(pred, abs(resid))) + geom_point() + geom_smooth()
```

Figure 4-7 shows the resulting plot. Using `geom_smooth`, it is easy to superpose a smooth of the absolute residuals. The function calls the `loess` method (locally estimated scatterplot smoothing) to produce a smoothed estimate of the relationship between the variables on the x-axis and y-axis in a scatterplot (see "Scatterplot Smoothers" on page 185).

In *Python*, the `seaborn` package has the `regplot` function to create a similar figure:

```
fig, ax = plt.subplots(figsize=(5, 5))
sns.regplot(result_98105.fittedvalues, np.abs(result_98105.resid),
            scatter_kws={'alpha': 0.25}, line_kws={'color': 'C1'},
            lowess=True, ax=ax)
```

```
    ax.set_xlabel('predicted')
    ax.set_ylabel('abs(residual)')
```

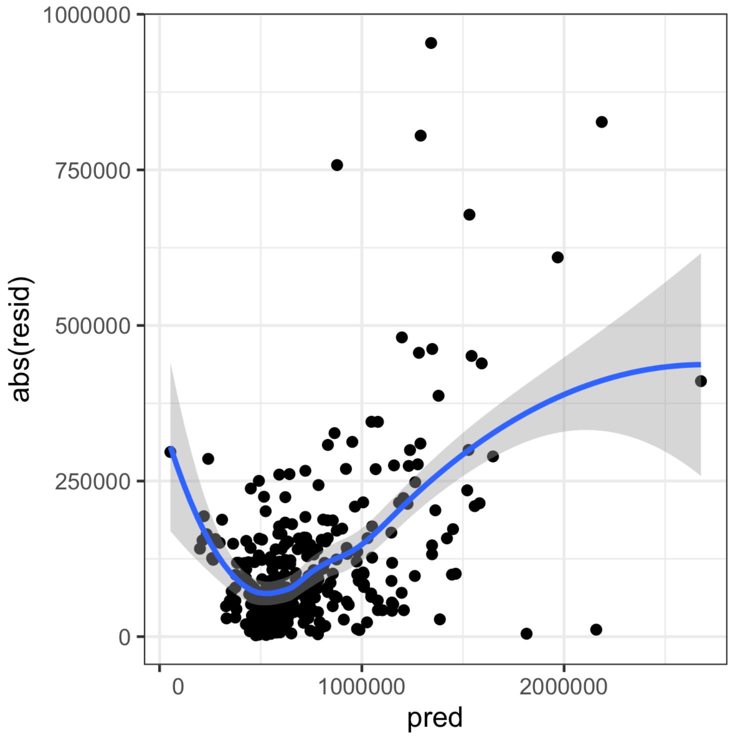


Figure 4-7. A plot of the absolute value of the residuals versus the predicted values

Evidently, the variance of the residuals tends to increase for higher-valued homes but is also large for lower-valued homes. This plot indicates that `lm_98105` has *heteroskedastic* errors.

Why Would a Data Scientist Care About Heteroskedasticity?

Heteroskedasticity indicates that prediction errors differ for different ranges of the predicted value, and may suggest an incomplete model. For example, the heteroskedasticity in `lm_98105` may indicate that the regression has left something unaccounted for in high- and low-range homes.

Figure 4-8 is a histogram of the standardized residuals for the `lm_98105` regression. The distribution has decidedly longer tails than the normal distribution and exhibits mild skewness toward larger residuals.

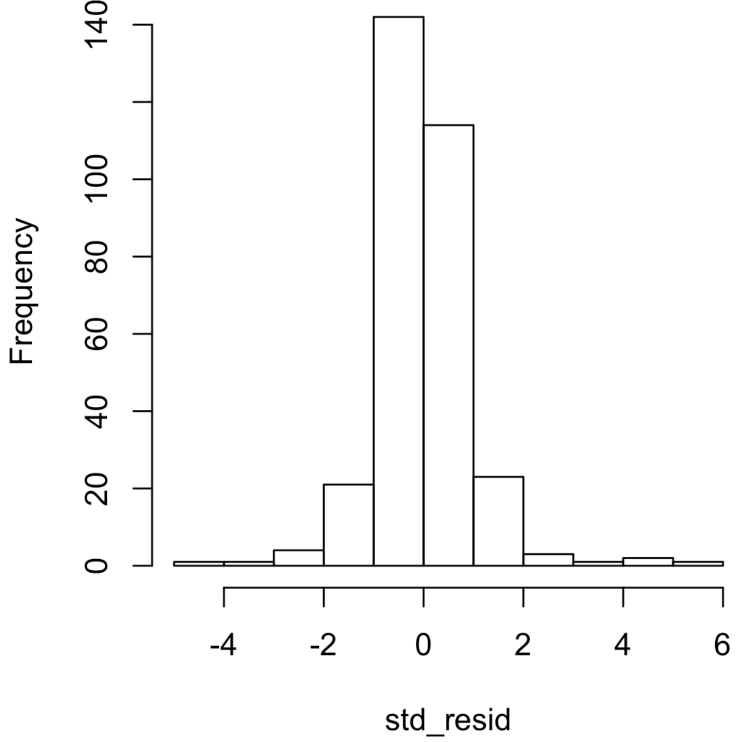


Figure 4-8. A histogram of the residuals from the regression of the housing data

Statisticians may also check the assumption that the errors are independent. This is particularly true for data that is collected over time or space. The *Durbin-Watson* statistic can be used to detect if there is significant autocorrelation in a regression involving time series data. If the errors from a regression model are correlated, then this information can be useful in making short-term forecasts and should be built into the model. See *Practical Time Series Forecasting with R*, 2nd ed., by Galit Shmueli and Kenneth Lichtendahl (Axelrod Schnall, 2018) to learn more about how to build autocorrelation information into regression models for time series data. If longer-term forecasts or explanatory models are the goal, excess autocorrelated data at the microlevel may distract. In that case, smoothing, or less granular collection of data in the first place, may be in order.

Even though a regression may violate one of the distributional assumptions, should we care? Most often in data science, the interest is primarily in predictive accuracy, so some review of heteroskedasticity may be in order. You may discover that there is some signal in the data that your model has not captured. However, satisfying distributional assumptions simply for the sake of validating formal statistical inference (p-values, F-statistics, etc.) is not that important for the data scientist.

Scatterplot Smoothers

Regression is about modeling the relationship between the response and predictor variables. In evaluating a regression model, it is useful to use a *scatterplot smoother* to visually highlight relationships between two variables.

For example, in Figure 4-7, a smooth of the relationship between the absolute residuals and the predicted value shows that the variance of the residuals depends on the value of the residual. In this case, the `loess` function was used; `loess` works by repeatedly fitting a series of local regressions to contiguous subsets to come up with a smooth. While `loess` is probably the most commonly used smoother, other scatterplot smoothers are available in *R*, such as super smooth (`supsmu`) and kernel smoothing (`ksmooth`). In *Python*, we can find additional smoothers in `scipy` (`wiener` or `sav`) and `statsmodels` (`kernel_regression`). For the purposes of evaluating a regression model, there is typically no need to worry about the details of these scatterplot smooths.

Partial Residual Plots and Nonlinearity

Partial residual plots are a way to visualize how well the estimated fit explains the relationship between a predictor and the outcome. The basic idea of a partial residual plot is to isolate the relationship between a predictor variable and the response, *taking into account all of the other predictor variables*. A partial residual might be thought of as a "synthetic outcome" value, combining the prediction based on a single predictor with the actual residual from the full regression equation. A partial residual for predictor X_i is the ordinary residual plus the regression term associated with X_i:

$$\text{Partial residual} = \text{Residual} + \hat{b}_i X_i$$

where $\hat{b}_i$ is the estimated regression coefficient. The `predict` function in *R* has an option to return the individual regression terms $\hat{b}_i X_i$:

```r
terms <- predict(lm_98105, type='terms')
partial_resid <- resid(lm_98105) + terms
```

The partial residual plot displays the X_i predictor on the x-axis and the partial residuals on the y-axis. Using `ggplot2` makes it easy to superpose a smooth of the partial residuals:

```r
df <- data.frame(SqFtTotLiving = house_98105[, 'SqFtTotLiving'],
                 Terms = terms[, 'SqFtTotLiving'],
                 PartialResid = partial_resid[, 'SqFtTotLiving'])
ggplot(df, aes(SqFtTotLiving, PartialResid)) +
    geom_point(shape=1) + scale_shape(solid = FALSE) +
    geom_smooth(linetype=2) +
    geom_line(aes(SqFtTotLiving, Terms))
```

The `statsmodels` package has the method `sm.graphics.plot_ccpr` that creates a similar partial residual plot:

```
sm.graphics.plot_ccpr(result_98105, 'SqFtTotLiving')
```

The *R* and *Python* graphs differ by a constant shift. In *R*, a constant is added so that the mean of the terms is zero.

The resulting plot is shown in Figure 4-9. The partial residual is an estimate of the contribution that `SqFtTotLiving` adds to the sales price. The relationship between `SqFtTotLiving` and the sales price is evidently nonlinear (dashed line). The regression line (solid line) underestimates the sales price for homes less than 1,000 square feet and overestimates the price for homes between 2,000 and 3,000 square feet. There are too few data points above 4,000 square feet to draw conclusions for those homes.

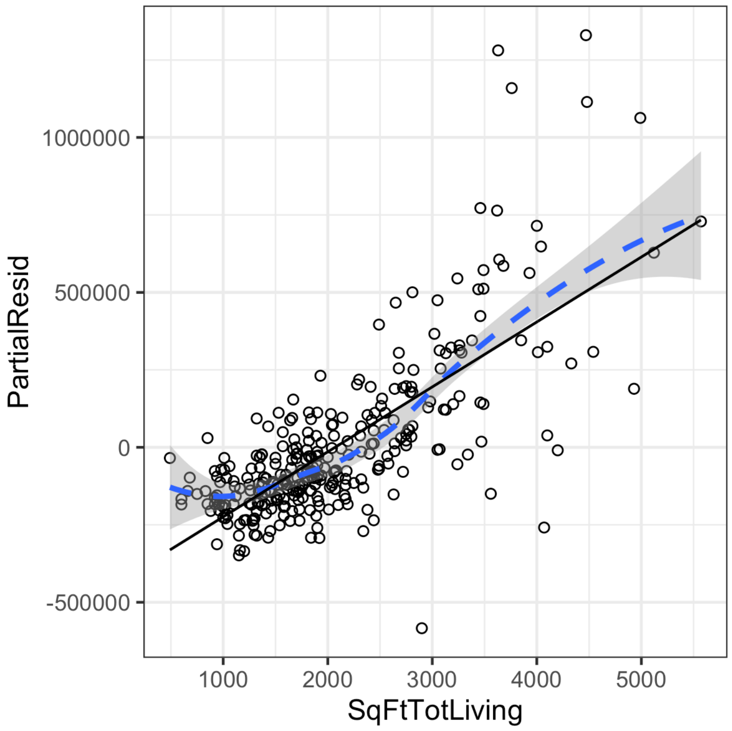


Figure 4-9. A partial residual plot for the variable `SqFtTotLiving`

This nonlinearity makes sense in this case: adding 500 feet in a small home makes a much bigger difference than adding 500 feet in a large home. This suggests that, instead of a simple linear term for `SqFtTotLiving`, a nonlinear term should be considered (see "Polynomial and Spline Regression" on page 187).

Key Ideas

- While outliers can cause problems for small data sets, the primary interest with outliers is to identify problems with the data, or locate anomalies.
- Single records (including regression outliers) can have a big influence on a regression equation with small data, but this effect washes out in big data.
- If the regression model is used for formal inference (p-values and the like), then certain assumptions about the distribution of the residuals should be checked. In general, however, the distribution of residuals is not critical in data science.
- The partial residuals plot can be used to qualitatively assess the fit for each regression term, possibly leading to alternative model specification.

Polynomial and Spline Regression

The relationship between the response and a predictor variable isn't necessarily linear. The response to the dose of a drug is often nonlinear: doubling the dosage generally doesn't lead to a doubled response. The demand for a product isn't a linear function of marketing dollars spent; at some point, demand is likely to be saturated. There are many ways that regression can be extended to capture these nonlinear effects.

Key Terms for Nonlinear Regression

Polynomial regression
Adds polynomial terms (squares, cubes, etc.) to a regression.

Spline regression
Fitting a smooth curve with a series of polynomial segments.

Knots
Values that separate spline segments.

Generalized additive models
Spline models with automated selection of knots.

Synonym
GAM

Nonlinear Regression

When statisticians talk about *nonlinear regression*, they are referring to models that can't be fit using least squares. What kind of models are nonlinear? Essentially all models where the response cannot be expressed as a linear combination of the predictors or some transform of the predictors. Nonlinear regression models are harder and computationally more intensive to fit, since they require numerical optimization. For this reason, it is generally preferred to use a linear model if possible.

Polynomial

Polynomial regression involves including polynomial terms in a regression equation. The use of polynomial regression dates back almost to the development of regression itself with a paper by Gergonne in 1815. For example, a quadratic regression between the response Y and the predictor X would take the form:

$$Y = b_0 + b_1X + b_2X^2 + e$$

Polynomial regression can be fit in *R* through the `poly` function. For example, the following fits a quadratic polynomial for `SqFtTotLiving` with the King County housing data:

```
lm(AdjSalePrice ~  poly(SqFtTotLiving, 2) + SqFtLot +
                BldgGrade + Bathrooms + Bedrooms,
                    data=house_98105)

Call:
lm(formula = AdjSalePrice ~ poly(SqFtTotLiving, 2) + SqFtLot +
   BldgGrade + Bathrooms + Bedrooms, data = house_98105)

Coefficients:
           (Intercept)  poly(SqFtTotLiving, 2)1  poly(SqFtTotLiving, 2)2
            -402530.47                3271519.49                776934.02
               SqFtLot                 BldgGrade                Bathrooms
                 32.56                 135717.06                 -1435.12
              Bedrooms
              -9191.94
```

In `statsmodels`, we add the squared term to the model definition using `I(SqFtTot Living**2)`:

```
model_poly = smf.ols(formula='AdjSalePrice ~  SqFtTotLiving + ' +
                '+ I(SqFtTotLiving**2) + ' +
                'SqFtLot + Bathrooms + Bedrooms + BldgGrade', data=house_98105)
result_poly = model_poly.fit()
result_poly.summary()  ❶
```

❶ The intercept and the polynomial coefficients are different compared to *R*. This is due to different implementations. The remaining coefficients and the predictions are equivalent.

There are now two coefficients associated with `SqFtTotLiving`: one for the linear term and one for the quadratic term.

The partial residual plot (see "Partial Residual Plots and Nonlinearity" on page 185) indicates some curvature in the regression equation associated with `SqFtTotLiving`. The fitted line more closely matches the smooth (see "Splines" on page 189) of the partial residuals as compared to a linear fit (see Figure 4-10).

The `statsmodels` implementation works only for linear terms. The accompanying source code gives an implementation that will work for polynomial regression as well.

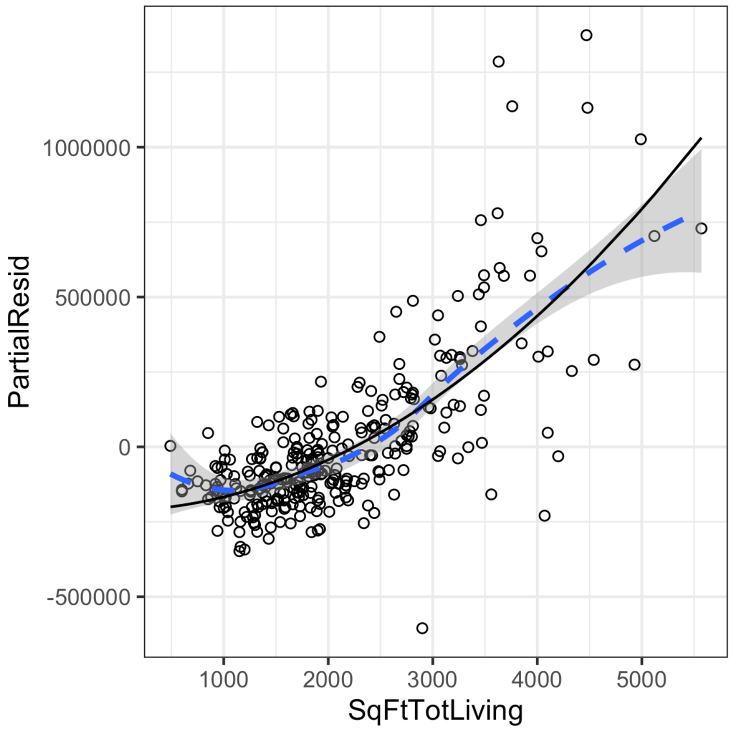


Figure 4-10. A polynomial regression fit for the variable `SqFtTotLiving` (solid line) versus a smooth (dashed line; see the following section about splines)

Splines

Polynomial regression captures only a certain amount of curvature in a nonlinear relationship. Adding in higher-order terms, such as a cubic quartic polynomial, often leads to undesirable "wiggliness" in the regression equation. An alternative, and often superior, approach to modeling nonlinear relationships is to use *splines*. *Splines* provide a way to smoothly interpolate between fixed points. Splines were originally used by draftsmen to draw a smooth curve, particularly in ship and aircraft building.

The splines were created by bending a thin piece of wood using weights, referred to as "ducks"; see Figure 4-11.

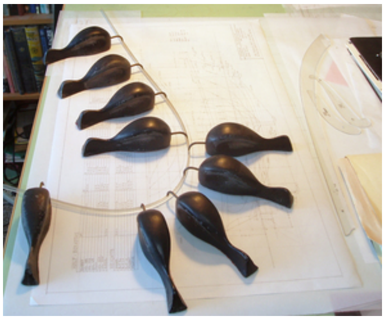


Figure 4-11. Splines were originally created using bendable wood and "ducks" and were used as a draftsman's tool to fit curves (photo courtesy of Bob Perry)

The technical definition of a spline is a series of piecewise continuous polynomials. They were first developed during World War II at the US Aberdeen Proving Grounds by I. J. Schoenberg, a Romanian mathematician. The polynomial pieces are smoothly connected at a series of fixed points in a predictor variable, referred to as *knots*. Formulation of splines is much more complicated than polynomial regression; statistical software usually handles the details of fitting a spline. The *R* package `splines` includes the function `bs` to create a *b-spline* (basis spline) term in a regression model. For example, the following adds a b-spline term to the house regression model:

```
library(splines)
knots <- quantile(house_98105$SqFtTotLiving, p=c(.25, .5, .75))
lm_spline <- lm(AdjSalePrice ~ bs(SqFtTotLiving, knots=knots, degree=3) +
  SqFtLot + Bathrooms + Bedrooms + BldgGrade,  data=house_98105)
```

Two parameters need to be specified: the degree of the polynomial and the location of the knots. In this case, the predictor `SqFtTotLiving` is included in the model using a cubic spline (`degree=3`). By default, `bs` places knots at the boundaries; in addition, knots were also placed at the lower quartile, the median quartile, and the upper quartile.

The `statsmodels` formula interface supports the use of splines in a similar way to *R*. Here, we specify the *b-spline* using `df`, the degrees of freedom. This will create `df` – `degree` = 6 – 3 = 3 internal knots with positions calculated in the same way as in the *R* code above:

```
formula = 'AdjSalePrice ~ bs(SqFtTotLiving, df=6, degree=3) ' +
          'SqFtLot + Bathrooms + Bedrooms + BldgGrade'
model_spline = smf.ols(formula=formula, data=house_98105)
result_spline = model_spline.fit()
```

In contrast to a linear term, for which the coefficient has a direct meaning, the coefficients for a spline term are not interpretable. Instead, it is more useful to use the visual display to reveal the nature of the spline fit. Figure 4-12 displays the partial residual plot from the regression. In contrast to the polynomial model, the spline model more closely matches the smooth, demonstrating the greater flexibility of splines. In this case, the line more closely fits the data. Does this mean the spline regression is a better model? Not necessarily: it doesn't make economic sense that very small homes (less than 1,000 square feet) would have higher value than slightly larger homes. This is possibly an artifact of a confounding variable; see "Confounding Variables" on page 172.

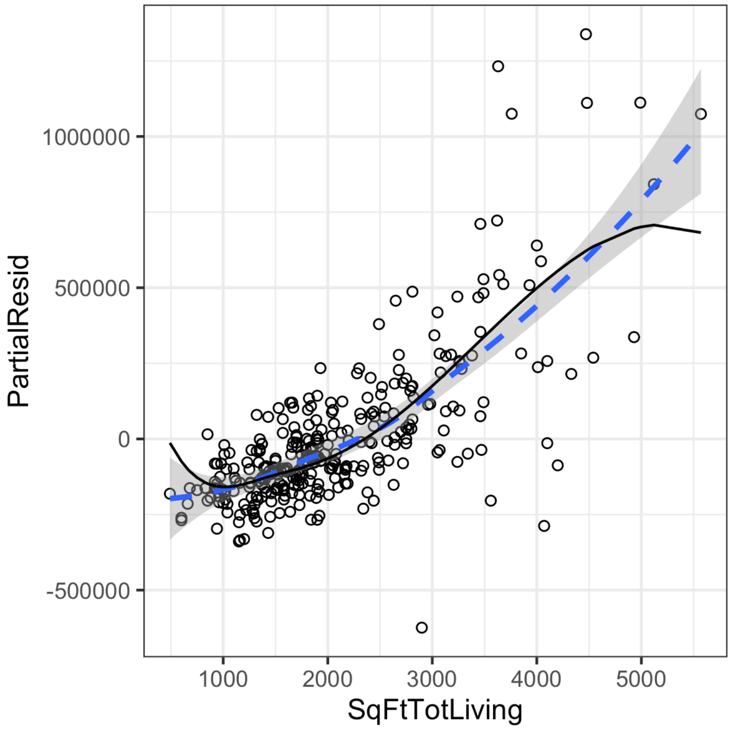


Figure 4-12. A spline regression fit for the variable `SqFtTotLiving` (solid line) compared to a smooth (dashed line)

Generalized Additive Models

Suppose you suspect a nonlinear relationship between the response and a predictor variable, either by a priori knowledge or by examining the regression diagnostics. Polynomial terms may not be flexible enough to capture the relationship, and spline terms require specifying the knots. *Generalized additive models*, or *GAM*, are a flexible modeling technique that can be used to automatically fit a spline regression. The `mgcv` package in *R* can be used to fit a GAM model to the housing data:

```
library(mgcv)
lm_gam <- gam(AdjSalePrice ~ s(SqFtTotLiving) + SqFtLot +
                    Bathrooms +  Bedrooms + BldgGrade,
                    data=house_98105)
```

The term `s(SqFtTotLiving)` tells the `gam` function to find the "best" knots for a spline term (see Figure 4-13).

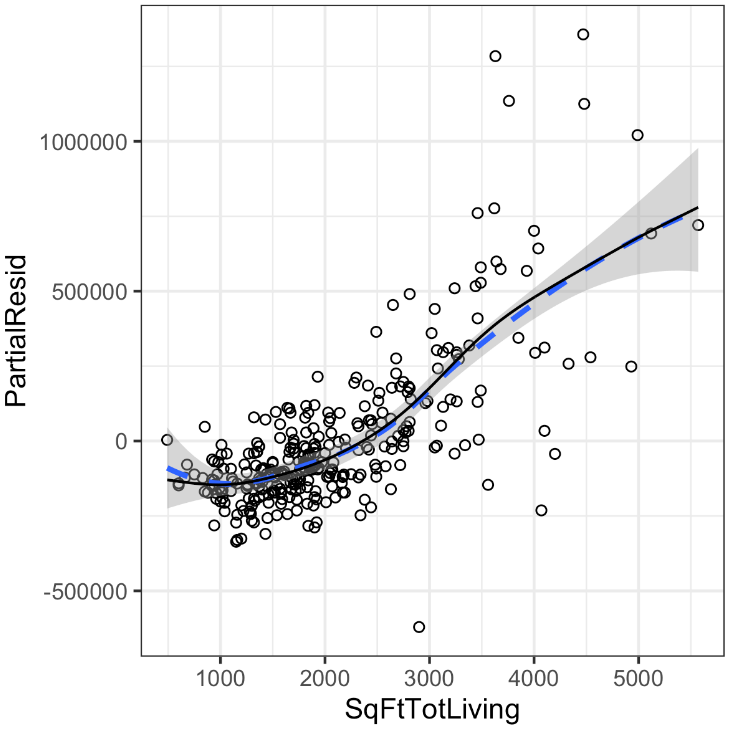


Figure 4-13. A GAM regression fit for the variable `SqFtTotLiving` (solid line) compared to a smooth (dashed line)

In *Python*, we can use the `pyGAM` package. It provides methods for regression and classification. Here, we use `LinearGAM` to create a regression model:

```
predictors = ['SqFtTotLiving', 'SqFtLot', 'Bathrooms',  'Bedrooms', 'BldgGrade']
outcome = 'AdjSalePrice'
X = house_98105[predictors].values
y = house_98105[outcome]

gam = LinearGAM(s(0, n_splines=12) + l(1) + l(2) + l(3) + l(4))  ❶
gam.gridsearch(X, y)
```

❶ The default value for `n_splines` is 20. This leads to overfitting for larger `SqFtTotLiving` values. A value of 12 leads to a more reasonable fit.

Key Ideas

- Outliers in a regression are records with a large residual.
- Multicollinearity can cause numerical instability in fitting the regression equation.
- A confounding variable is an important predictor that is omitted from a model and can lead to a regression equation with spurious relationships.
- An interaction term between two variables is needed if the effect of one variable depends on the level or magnitude of the other.
- Polynomial regression can fit nonlinear relationships between predictors and the outcome variable.
- Splines are series of polynomial segments strung together, joining at knots.
- We can automate the process of specifying the knots in splines using generalized additive models (GAM).

Further Reading

- For more on spline models and GAMs, see *The Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (2009), and its shorter cousin based on R, *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (2013); both are Springer books.
- To learn more about using regression models for time series forecasting, see *Practical Time Series Forecasting with R* by Galit Shmueli and Kenneth Lichtendahl (Axelrod Schnall, 2018).

Summary

Perhaps no other statistical method has seen greater use over the years than regression—the process of establishing a relationship between multiple predictor variables and an outcome variable. The fundamental form is linear: each predictor variable has a coefficient that describes a linear relationship between the predictor and the outcome. More advanced forms of regression, such as polynomial and spline regression, permit the relationship to be nonlinear. In classical statistics, the emphasis is on finding a good fit to the observed data to explain or describe some phenomenon, and the strength of this fit is how traditional *in-sample* metrics are used to assess the model. In data science, by contrast, the goal is typically to predict values for new data, so metrics based on predictive accuracy for out-of-sample data are used. Variable selection methods are used to reduce dimensionality and create more compact models.