Chapter 12. Introduction to Modeling Libraries in Python

In this book, I have focused on providing a programming foundation for doing data analysis in Python. Since data analysts and scientists often report spending a disproportionate amount of time with data wrangling and preparation, the book's structure reflects the importance of mastering these techniques.

Which library you use for developing models will depend on the application. Many statistical problems can be solved by simpler techniques like ordinary least squares regression, while other problems may call for more advanced machine learning methods. Fortunately, Python has become one of the languages of choice for implementing analytical methods, so there are many tools you can explore after completing this book.

In this chapter, I will review some features of pandas that may be helpful when you're crossing back and forth between data wrangling with pandas and model fitting and scoring. I will then give short introductions to two popular modeling toolkits, statsmodels and scikit-learn. Since each of these projects is large enough to warrant its own dedicated book, I make no effort to be comprehensive and instead direct you to both projects' online documentation along with some other Python-based books on data science, statistics, and machine learning.

12.1 Interfacing Between pandas and Model Code

A common workflow for model development is to use pandas for data loading and cleaning before switching over to a modeling library to build the model itself. An important part of the model development process is called *feature engineering* in machine learning. This can describe any data transformation or analytics that extract information from a raw dataset that may be useful in a modeling context. The data aggregation and GroupBy tools we have explored in this book are used often in a feature engineering context.

While details of “good” feature engineering are out of scope for this book, I will show some methods to make switching between data manipulation with pandas and modeling as painless as possible.

The point of contact between pandas and other analysis libraries is usually NumPy arrays. To turn a DataFrame into a NumPy array, use the `to_numpy` method:

```
In [12]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.to_numpy()
Out[15]:
array([[ 1.  ,  0.01, -1.5 ],
       [ 2.  , -0.01,  0.  ],
       [ 3.  ,  0.25,  3.6 ],
       [ 4.  , -4.1 ,  1.3 ],
       [ 5.  ,  0.  , -2.  ]])
```

To convert back to a DataFrame, as you may recall from earlier chapters, you can pass a two-dimensional ndarray with optional column names:

```
In [16]: df2 = pd.DataFrame(data.to_numpy(), columns=['one', 'two', 'three']

In [17]: df2
Out[17]:
   one   two  three
0  1.0  0.01   -1.5
1  2.0 -0.01    0.0
2  3.0  0.25    3.6
```

```
3  4.0 -4.10    1.3
4  5.0  0.00   -2.0
```

The `to_numpy` method is intended to be used when your data is homogeneous—for example, all numeric types. If you have heterogeneous data, the result will be an ndarray of Python objects:

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
   x0    x1    y strings
0   1  0.01 -1.5       a
1   2 -0.01  0.0       b
2   3  0.25  3.6       c
3   4 -4.10  1.3       d
4   5  0.00 -2.0       e

In [21]: df3.to_numpy()
Out[21]:
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

For some models, you may wish to use only a subset of the columns. I recommend using `loc` indexing with `to_numpy`:

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].to_numpy()
Out[23]:
array([[ 1.  ,  0.01],
       [ 2.  , -0.01],
       [ 3.  ,  0.25],
       [ 4.  , -4.1 ],
       [ 5.  ,  0.  ]])
```

Some libraries have native support for pandas and do some of this work for you automatically: converting to NumPy from DataFrame and attaching model parameter names to the columns of output tables or Series. In other cases, you will have to perform this "metadata management" manually.

In Section 7.5, "Categorical Data,", we looked at pandas's `Categorical` type and the `pandas.get_dummies` function. Suppose we had a nonnumeric column in our example dataset:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
   ....:                                    categories=['a', 'b'])

In [25]: data
Out[25]:
   x0    x1    y category
0   1  0.01 -1.5        a
1   2 -0.01  0.0        b
2   3  0.25  3.6        a
3   4 -4.10  1.3        a
4   5  0.00 -2.0        b
```

If we wanted to replace the `'category'` column with dummy variables, we create dummy variables, drop the `'category'` column, and then join the result:

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies
Out[28]:
   x0    x1    y  category_a  category_b
0   1  0.01 -1.5           1           0
1   2 -0.01  0.0           0           1
2   3  0.25  3.6           1           0
3   4 -4.10  1.3           1           0
4   5  0.00 -2.0           0           1
```

There are some nuances to fitting certain statistical models with dummy variables. It may be simpler and less error-prone to use Patsy (the subject of the next section) when you have more than simple numeric columns.

12.2 Creating Model Descriptions with Patsy

Patsy is a Python library for describing statistical models (especially linear models) with a string-based "formula syntax," which is inspired by (but not exactly the same as) the formula syntax used by the R and S statistical programming languages. It is installed automatically when you install statsmodels:

```
conda install statsmodels
```

Patsy is well supported for specifying linear models in statsmodels, so I will focus on some of the main features to help you get up and running. Patsy's *formulas* are a special string syntax that looks like:

```
y ~ x0 + x1
```

The syntax `a + b` does not mean to add `a` to `b`, but rather that these are *terms* in the *design matrix* created for the model. The `patsy.dmatrices` function takes a formula string along with a dataset (which can be a DataFrame or a dictionary of arrays) and produces design matrices for a linear model:

```
In [29]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Now we have:

```
In [33]: y
Out[33]:
```

```
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

These Patsy `DesignMatrix` instances are NumPy ndarrays with additional metadata:

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1.  ,  1.  ,  0.01],
       [ 1.  ,  2.  , -0.01],
       [ 1.  ,  3.  ,  0.25],
       [ 1.  ,  4.  , -4.1 ],
       [ 1.  ,  5.  ,  0.  ]])
```

You might wonder where the `Intercept` term came from. This is a convention for linear models like ordinary least squares (OLS) regression. You can suppress the intercept by adding the term `+ 0` to the model:

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

The Patsy objects can be passed directly into algorithms like `numpy.linalg.lstsq`, which performs an ordinary least squares regression:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

The model metadata is retained in the `design_info` attribute, so you can reattach the model column names to the fitted coefficients to obtain a Series, for example:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)

In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

Data Transformations in Patsy Formulas

You can mix Python code into your Patsy formulas; when evaluating the formula, the library will try to find the functions you use in the enclosing scope:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

Some commonly used variable transformations include *standardizing* (to mean 0 and variance 1) and *centering* (subtracting the mean). Patsy has built-in functions for this purpose:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
  Intercept  standardize(x0)  center(x1)
          1         -1.41421        0.78
          1         -0.70711        0.76
          1          0.00000        1.02
          1          0.70711       -3.33
          1          1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

As part of a modeling process, you may fit a model on one dataset, then evaluate the model based on another. This might be a *hold-out* portion or new data that is observed later. When applying transformations like center and standardize, you should be careful when using the model to form predications based on new data. These are called *stateful* transformations, because you must use statistics like the mean or standard deviation of the original dataset when transforming a new dataset.

The `patsy.build_design_matrices` function can apply transformations to new *out-of-sample* data using the saved information from the original *in-sample* dataset:

```
In [46]: new_data = pd.DataFrame({
   ....:     'x0': [6, 7, 8, 9],
   ....:     'x1': [3.1, -0.5, 0, 2.3],
   ....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
   Intercept  standardize(x0)  center(x1)
           1          2.12132        3.87
           1          2.82843        0.27
           1          3.53553        0.77
           1          4.24264        3.07
   Terms:
     'Intercept' (column 0)
     'standardize(x0)' (column 1)
     'center(x1)' (column 2)]
```

Because the plus symbol (`+`) in the context of Patsy formulas does not mean addition, when you want to add columns from a dataset by name, you must wrap them in the special `I` function:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
  Intercept  I(x0 + x1)
          1        1.01
          1        1.99
          1        3.25
          1       -0.10
          1        5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

Patsy has several other built-in transforms in the `patsy.builtins` module. See the online documentation for more.

Categorical data has a special class of transformations, which I explain next.

Categorical Data and Patsy

Nonnumeric data can be transformed for a model design matrix in many different ways. A complete treatment of this topic is outside the scope of this book and would be studied best along with a course in statistics.

When you use nonnumeric terms in a Patsy formula, they are converted to dummy variables by default. If there is an intercept, one of the levels will be left out to avoid collinearity:

```
In [51]: data = pd.DataFrame({
   ....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
   ....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
   ....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
   ....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
   ....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
  Intercept  key1[T.b]
          1          0
          1          0
          1          1
          1          1
          1          0
          1          1
          1          0
          1          1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
```

If you omit the intercept from the model, then columns for each category value will be included in the model design matrix:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
```

```
Out[55]:
DesignMatrix with shape (8, 2)
  key1[a]  key1[b]
        1        0
        1        0
        0        1
        0        1
        1        0
        0        1
        1        0
        0        1
  Terms:
    'key1' (columns 0:2)
```

Numeric columns can be interpreted as categorical with the `C` function:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept  C(key2)[T.1]
          1             0
          1             1
          1             0
          1             1
          1             0
          1             1
          1             0
          1             0
  Terms:
    'Intercept' (column 0)
    'C(key2)' (column 1)
```

When you’re using multiple categorical terms in a model, things can be more complicated, as you can include interaction terms of the form `key1:key2`, which can be used, for example, in analysis of variance (ANOVA) models:

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})

In [59]: data
Out[59]:
  key1  key2  v1   v2
0    a  zero   1 -1.0
```

```
1    a   one   2  0.0
2    b  zero   3  2.5
3    b   one   4 -0.5
4    a  zero   5  4.0
5    b   one   6 -1.2
6    a  zero   7  0.2
7    b  zero   8 -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept  key1[T.b]  key2[T.zero]
          1          0             1
          1          0             0
          1          1             1
          1          1             0
          1          0             1
          1          1             0
          1          0             1
          1          1             1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)

In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
  Intercept  key1[T.b]  key2[T.zero]  key1[T.b]:key2[T.zero]
          1          0             1                       0
          1          0             0                       0
          1          1             1                       1
          1          1             0                       0
          1          0             1                       0
          1          1             0                       0
          1          0             1                       0
          1          1             1                       1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)
    'key1:key2' (column 3)
```

Patsy provides for other ways to transform categorical data, including transformations for terms with a particular ordering. See the online documentation for more.

12.3 Introduction to statsmodels

statsmodels is a Python library for fitting many kinds of statistical models, performing statistical tests, and data exploration and visualization. statsmodels contains more "classical" frequentist statistical methods, while Bayesian methods and machine learning models are found in other libraries.

Some kinds of models found in statsmodels include:

- Linear models, generalized linear models, and robust linear models
- Linear mixed effects models
- Analysis of variance (ANOVA) methods
- Time series processes and state space models
- Generalized method of moments

In the next few pages, we will use a few basic tools in statsmodels and explore how to use the modeling interfaces with Patsy formulas and pandas DataFrame objects. If you didn't install statsmodels in the Patsy discussion earlier, you can install it now with:

```
conda install statsmodels
```

Estimating Linear Models

There are several kinds of linear regression models in statsmodels, from the more basic (e.g., ordinary least squares) to more complex (e.g., iteratively reweighted least squares).

Linear models in statsmodels have two different main interfaces: array based and formula based. These are accessed through these API module imports:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

To show how to use these, we generate a linear model from some random data. Run the following code in a Jupyter cell:

```
# To make the example reproducible
rng = np.random.default_rng(seed=12345)

def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * rng.standard_normal(*size)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

Here, I wrote down the "true" model with known parameters `beta`. In this case, `dnorm` is a helper function for generating normally distributed data with a particular mean and variance. So now we have:

```
In [66]: X[:5]
Out[66]:
array([[-0.9005, -0.1894, -1.0279],
       [ 0.7993, -1.546 , -0.3274],
       [-0.5507, -0.1203,  0.3294],
       [-0.1639,  0.824 ,  0.2083],
       [-0.0477, -0.2131, -0.0482]])

In [67]: y[:5]
Out[67]: array([-0.5995, -0.5885,  0.1856, -0.0075, -0.0154])
```

A linear model is generally fitted with an intercept term, as we saw before with Patsy. The `sm.add_constant` function can add an intercept column to an existing matrix:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1.    , -0.9005, -0.1894, -1.0279],
       [ 1.    ,  0.7993, -1.546 , -0.3274],
       [ 1.    , -0.5507, -0.1203,  0.3294],
       [ 1.    , -0.1639,  0.824 ,  0.2083],
       [ 1.    , -0.0477, -0.2131, -0.0482]])
```

The `sm.OLS` class can fit an ordinary least squares linear regression:

```
In [70]: model = sm.OLS(y, X)
```

The model's `fit` method returns a regression results object containing estimated model parameters and other diagnostics:

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([0.0668, 0.268 , 0.4505])
```

The `summary` method on `results` can print a model detailing diagnostic output of the model:

```
In [73]: print(results.summary())
OLS Regression Results
============================================================================
======
Dep. Variable:                      y   R-squared (uncentered):
 0.469
Model:                            OLS   Adj. R-squared (uncentered):
 0.452
Method:                 Least Squares   F-statistic:
 28.51
Date:                Mon, 15 Aug 2022   Prob (F-statistic):
66e-13
Time:                        09:17:35   Log-Likelihood:
25.611
No. Observations:                 100   AIC:
 57.22
Df Residuals:                      97   BIC:
 65.04
Df Model:                           3

Covariance Type:            nonrobust

============================================================================
                 coef    std err          t      P>|t|      [0.025      0.97
```

```
------------------------------------------------------------------------------
x1             0.0668      0.054      1.243      0.217      -0.040       0.1
x2             0.2680      0.042      6.313      0.000       0.184       0.3
x3             0.4505      0.068      6.605      0.000       0.315       0.5
==============================================================================
Omnibus:                        0.435   Durbin-Watson:                   1.8
Prob(Omnibus):                  0.805   Jarque-Bera (JB):                0.3
Skew:                           0.134   Prob(JB):                        0.8
Kurtosis:                       2.995   Cond. No.                         1.
==============================================================================
Notes:
[1] R² is computed without centering (uncentered) since the model does not c
n a constant.
[2] Standard Errors assume that the covariance matrix of the errors is corre
specified.
```

The parameter names here have been given the generic names `x1, x2`, and so on. Suppose instead that all of the model parameters are in a DataFrame:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
       col0      col1      col2         y
0 -0.900506 -0.189430 -1.027870 -0.599527
1  0.799252 -1.545984 -0.327397 -0.588454
2 -0.550655 -0.120254  0.329359  0.185634
3 -0.163916  0.824040  0.208275 -0.007477
4 -0.047651 -0.213147 -0.048244 -0.015374
```

Now we can use the statsmodels formula API and Patsy formula strings:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()

In [78]: results.params
Out[78]:
Intercept   -0.020799
col0         0.065813
col1         0.268970
col2         0.449419
dtype: float64

In [79]: results.tvalues
```

```
Out[79]:
Intercept   -0.652501
col0         1.219768
col1         6.312369
col2         6.567428
dtype: float64
```

Observe how statsmodels has returned results as Series with the DataFrame column names attached. We also do not need to use `add_constant` when using formulas and pandas objects.

Given new out-of-sample data, you can compute predicted values given the estimated model parameters:

```
In [80]: results.predict(data[:5])
Out[80]:
0   -0.592959
1   -0.531160
2    0.058636
3    0.283658
4   -0.102947
dtype: float64
```

There are many additional tools for analysis, diagnostics, and visualization of linear model results in statsmodels that you can explore. There are also other kinds of linear models beyond ordinary least squares.

Estimating Time Series Processes

Another class of models in statsmodels is for time series analysis. Among these are autoregressive processes, Kalman filtering and other state space models, and multivariate autoregressive models.

Let’s simulate some time series data with an autoregressive structure and noise. Run the following in Jupyter:

```
init_x = 4

values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
```

```
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

This data has an AR(2) structure (two *lags*) with parameters `0.8` and `-0.4`. When you fit an AR model, you may not know the number of lagged terms to include, so you can fit the model with some larger number of lags:

```
In [82]: from statsmodels.tsa.ar_model import AutoReg

In [83]: MAXLAGS = 5

In [84]: model = AutoReg(values, MAXLAGS)

In [85]: results = model.fit()
```

The estimated parameters in the results have the intercept first, and the estimates for the first two lags next:

```
In [86]: results.params
Out[86]: array([ 0.0235,  0.8097, -0.4287, -0.0334,  0.0427, -0.0567])
```

Deeper details of these models and how to interpret their results are beyond what I can cover in this book, but there's plenty more to discover in the statsmodels documentation.

12.4 Introduction to scikit-learn

scikit-learn is one of the most widely used and trusted general-purpose Python machine learning toolkits. It contains a broad selection of standard supervised and unsupervised machine learning methods, with tools for model selection and evaluation, data transformation, data loading, and model persistence. These models can be used for classification, clustering, prediction, and other common tasks. You can install scikit-learn from conda like so:

```
conda install scikit-learn
```

There are excellent online and print resources for learning about machine learning and how to apply libraries like scikit-learn to solve real-world problems. In this section, I will give a brief flavor of the scikit-learn API style.

pandas integration in scikit-learn has improved significantly in recent years, and by the time you are reading this it may have improved even more. I encourage you to check out the latest project documentation.

As an example for this chapter, I use a now-classic dataset from a Kaggle competition about passenger survival rates on the *Titanic* in 1912. We load the training and test datasets using pandas:

```
In [87]: train = pd.read_csv('datasets/titanic/train.csv')

In [88]: test = pd.read_csv('datasets/titanic/test.csv')

In [89]: train.head(4)
Out[89]:
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
                                                Name     Sex   Age  SibSp
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Thayer)  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
```

Libraries like statsmodels and scikit-learn generally cannot be fed missing data, so we look at the columns to see if there are any that contain missing data:

```
In [90]: train.isna().sum()
Out[90]:
PassengerId      0
Survived         0
Pclass           0
Name             0
```

```
Sex             0
Age           177
SibSp           0
Parch           0
Ticket          0
Fare            0
Cabin         687
Embarked        2
dtype: int64

In [91]: test.isna().sum()
Out[91]:
PassengerId     0
Pclass          0
Name            0
Sex             0
Age            86
SibSp           0
Parch           0
Ticket          0
Fare            1
Cabin         327
Embarked        0
dtype: int64
```

In statistics and machine learning examples like this one, a typical task is to predict whether a passenger would survive based on features in the data. A model is fitted on a *training* dataset and then evaluated on an out-of-sample *testing* dataset.

I would like to use `Age` as a predictor, but it has missing data. There are a number of ways to do missing data imputation, but I will do a simple one and use the median of the training dataset to fill the nulls in both tables:

```
In [92]: impute_value = train['Age'].median()

In [93]: train['Age'] = train['Age'].fillna(impute_value)

In [94]: test['Age'] = test['Age'].fillna(impute_value)
```

Now we need to specify our models. I add a column `IsFemale` as an encoded version of the `'Sex'` column:

```
In [95]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [96]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Then we decide on some model variables and create NumPy arrays:

```
In [97]: predictors = ['Pclass', 'IsFemale', 'Age']

In [98]: X_train = train[predictors].to_numpy()

In [99]: X_test = test[predictors].to_numpy()

In [100]: y_train = train['Survived'].to_numpy()

In [101]: X_train[:5]
Out[101]:
array([[ 3.,  0., 22.],
       [ 1.,  1., 38.],
       [ 3.,  1., 26.],
       [ 1.,  1., 35.],
       [ 3.,  0., 35.]])

In [102]: y_train[:5]
Out[102]: array([0, 1, 1, 1, 0])
```

I make no claims that this is a good model or that these features are engineered properly. We use the `LogisticRegression` model from scikit-learn and create a model instance:

```
In [103]: from sklearn.linear_model import LogisticRegression

In [104]: model = LogisticRegression()
```

We can fit this model to the training data using the model's `fit` method:

```
In [105]: model.fit(X_train, y_train)
Out[105]: LogisticRegression()
```

Now, we can form predictions for the test dataset using `model.predict`:

```
In [106]: y_predict = model.predict(X_test)
In [107]: y_predict[:10]
Out[107]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

If you had the true values for the test dataset, you could compute an accuracy percentage or some other error metric:

```
(y_true == y_predict).mean()
```

In practice, there are often many additional layers of complexity in model training. Many models have parameters that can be tuned, and there are techniques such as *cross-validation* that can be used for parameter tuning to avoid overfitting to the training data. This can often yield better predictive performance or robustness on new data.

Cross-validation works by splitting the training data to simulate out-of-sample prediction. Based on a model accuracy score like mean squared error, you can perform a grid search on model parameters. Some models, like logistic regression, have estimator classes with built-in cross-validation. For example, the `LogisticRegressionCV` class can be used with a parameter indicating how fine-grained of a grid search to do on the model regularization parameter `C`:

```
In [108]: from sklearn.linear_model import LogisticRegressionCV

In [109]: model_cv = LogisticRegressionCV(Cs=10)

In [110]: model_cv.fit(X_train, y_train)
Out[110]: LogisticRegressionCV()
```

To do cross-validation by hand, you can use the `cross_val_score` helper function, which handles the data splitting process. For example, to cross-validate our model with four nonoverlapping splits of the training data, we can do:

```
In [111]: from sklearn.model_selection import cross_val_score

In [112]: model = LogisticRegression(C=10)

In [113]: scores = cross_val_score(model, X_train, y_train, cv=4)

In [114]: scores
Out[114]: array([0.7758, 0.7982, 0.7758, 0.7883])
```

The default scoring metric is model dependent, but it is possible to choose an explicit scoring function. Cross-validated models take longer to train but can often yield better model performance.

12.5 Conclusion

While I have only skimmed the surface of some Python modeling libraries, there are more and more frameworks for various kinds of statistics and machine learning either implemented in Python or with a Python user interface.

This book is focused especially on data wrangling, but there are many others dedicated to modeling and data science tools. Some excellent ones are:

- *Introduction to Machine Learning with Python* by Andreas Müller and Sarah Guido (O'Reilly)
- *Python Data Science Handbook* by Jake VanderPlas (O'Reilly)
- *Data Science from Scratch: First Principles with Python* by Joel Grus (O'Reilly)
- *Python Machine Learning* by Sebastian Raschka and Vahid Mirjalili (Packt Publishing)
- *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron (O'Reilly)

While books can be valuable resources for learning, they can sometimes grow out of date when the underlying open source software changes. It's a good idea to be familiar with the documentation for the various statistics or machine learning frameworks to stay up to date on the latest features and API.