Chapter_13_Introduction_to_Modeling_Libraries_in_Python

March 7, 2024

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 dispro- portionate 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 imple- menting 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, stats- models 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.

0.1   13.1 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 fea- ture 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 mod- eling 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 .values property:

```python
[19]: import pandas as pd
import numpy as np
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.]})
data
```

```
[19]:    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
```

```
[20]: data.columns
```

```
[20]: Index(['x0', 'x1', 'y'], dtype='object')
```

```
[21]: data.values
```

```
[21]: 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:

```
[22]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])
      df2
```

```
[22]:    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 .values attribute is intended to be used when your data is homogeneous—for example, all numeric types. If you have hetero- geneous data, the result will be an ndarray of Python objects:

```
[23]: df3 = data.copy()
      df3['strings'] = ['a', 'b', 'c', 'd', 'e']
      df3
```

```
[23]:    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
```

```
[24]: df3.values
```

```
[24]: 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 only wish to use a subset of the columns. I recommend using loc indexing with values:

```
[25]: model_cols = ['x0', 'x1']
      data.loc[:, model_cols].values
```

```
[25]: 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 auto- matically: 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 per- form this "metadata management" manually.

In Chapter 12 we looked at pandas's Categorical type and the pandas.get_dummies function. Suppose we had a non-numeric column in our example dataset:

```
[26]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],categories=['a',␣
       ↪'b'])
      data
```

```
[26]:    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:

```
[27]: dummies = pd.get_dummies(data.category, prefix='category')
      data_with_dummies = data.drop('category', axis=1).join(dummies)
      data_with_dummies
```

```
[27]:    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.

0.2 13.2 Creating Model Descriptions with Patsy

Patsy is a Python library for describing statistical models (especially linear models) with a small 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.

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 spe- cial 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 dict of arrays) and pro- duces design matrices for a linear model:

```
[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.]})
      data
```

```
[29]:    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
```

```
[30]: import patsy
```

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

Now we have:

```
[32]: y
```

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

4

```
[33]: X
```

```
[33]: 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:

```
[34]: np.asarray(y)
```

```
[34]: array([[-1.5],
             [ 0. ],
             [ 3.6],
             [ 1.3],
             [-2. ]])
```

```
[35]: np.asarray(X)
```

```
[35]: 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:

```
[36]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
```

```
[36]: 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:

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

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Pra
ctice_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-
packages\ipykernel_launcher.py:1: FutureWarning: `rcond` parameter will change
to the default of machine precision times ``max(M, N)`` where M and N are the
input matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  """Entry point for launching an IPython kernel.
```

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:

```
[38]: coef
```

```
[38]: array([[ 0.31290976],
             [-0.07910564],
             [-0.26546384]])
```

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

```
[40]: coef
```

```
[40]: Intercept    0.312910
      x0          -0.079106
      x1          -0.265464
      dtype: float64
```

0.3   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:

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

```
[41]: 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:

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

```
[42]: 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 outof-sample data using the saved information from the original in-sample dataset:

```
[43]: new_data = pd.DataFrame({'x0': [6, 7, 8, 9],'x1': [3.1, -0.5, 0, 2.3],'y': [1,␣
      ↪2, 3, 4]})
      new_X = patsy.build_design_matrices([X.design_info], new_data)
      new_X
```

```
[43]: [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:

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

```
[44]: 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.

0.4   Categorical Data and Patsy

Non-numeric 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 best studied along with a course in statistics

When you use non-numeric 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:

```
[45]: 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]})
      y, X = patsy.dmatrices('v2 ~ key1', data)
      X
```

```
[45]: 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:

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

```
[46]: 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:

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

```
[47]: 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 compli- cated, as you can include interaction terms of the form key1:key2, which can be used, for example, in analysis of variance (ANOVA) models:

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

```
[48]:    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
```

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

```
[49]: 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)
```

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

```
[50]: 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 transforma- tions for terms with a particular ordering. See the online documentation for more.

0.5   13.3 Introduction to statsmodels

statsmodels is a Python library for fitting many kinds of statistical models, perform- ing statistical tests, and data exploration and visualization. Statsmodels contains more "classical" frequentist

statistical methods, while Bayesian methods and machine learn- ing 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.

0.5.1   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:

```
[51]: 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:

```
[52]: def dnorm(mean, variance, size=1):
          if isinstance(size, int):
              size = size,
          return mean + np.sqrt(variance) * np.random.randn(*size)
```

```
[53]: # For reproducibility
      np.random.seed(12345)
      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:

```
[54]: X[:5]
```

```
[54]: array([[-0.12946849, -1.21275292,  0.50422488],
             [ 0.30291036, -0.43574176, -0.25417986],
             [-0.32852189, -0.02530153,  0.13835097],
```

11

```
           [-0.35147471, -0.71960511, -0.25821463],
           [ 1.2432688 , -0.37379916, -0.52262905]])
```

[55]: `y[:5]`

[55]: `array([ 0.42786349, -0.67348041, -0.09087764, -0.48949442, -0.12894109])`

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:

[56]:
```
X_model = sm.add_constant(X)
X_model[:5]
```

[56]:
```
array([[ 1.        , -0.12946849, -1.21275292,  0.50422488],
       [ 1.        ,  0.30291036, -0.43574176, -0.25417986],
       [ 1.        , -0.32852189, -0.02530153,  0.13835097],
       [ 1.        , -0.35147471, -0.71960511, -0.25821463],
       [ 1.        ,  1.2432688 , -0.37379916, -0.52262905]])
```

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

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

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

[59]:
```
result = model.fit()
result.params
```

[59]: `array([0.17826108, 0.22303962, 0.50095093])`

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

[60]: `print(results.summary())`

```
                          OLS Regression Results
================================================================================
=======
Dep. Variable:                      y   R-squared (uncentered):
0.430
Model:                            OLS   Adj. R-squared (uncentered):
0.413
Method:                 Least Squares   F-statistic:
24.42
Date:                Fri, 02 Feb 2024   Prob (F-statistic):
7.44e-12
Time:                        04:18:10   Log-Likelihood:
-34.305
No. Observations:                 100   AIC:
74.61
```

```
Df Residuals:                         97    BIC:
82.42
Df Model:                              3
Covariance Type:               nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.1783      0.053      3.364      0.001       0.073       0.283
x2             0.2230      0.046      4.818      0.000       0.131       0.315
x3             0.5010      0.080      6.237      0.000       0.342       0.660
==============================================================================
Omnibus:                        4.662    Durbin-Watson:                   2.201
Prob(Omnibus):                  0.097    Jarque-Bera (JB):                4.098
Skew:                           0.481    Prob(JB):                        0.129
Kurtosis:                       3.243    Cond. No.                        1.74
==============================================================================
```

Notes:
[1] R^2 is computed without centering (uncentered) since the model does not contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly 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:

```
[61]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
      data['y'] = y
      data[:5]
```

```
[61]:        col0       col1       col2          y
      0 -0.129468 -1.212753  0.504225   0.427863
      1  0.302910 -0.435742 -0.254180  -0.673480
      2 -0.328522 -0.025302  0.138351  -0.090878
      3 -0.351475 -0.719605 -0.258215  -0.489494
      4  1.243269 -0.373799 -0.522629  -0.128941
```

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

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

```
[62]: Intercept    0.033559
      col0         0.176149
      col1         0.224826
      col2         0.514808
      dtype: float64
```

```
[63]: results.tvalues
```

```
[63]: Intercept    0.952188
      col0         3.319754
      col1         4.850730
      col2         6.303971
      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:

```
[64]: results.predict(data[:5])
```

```
[64]: 0   -0.002327
      1   -0.141904
      2    0.041226
      3   -0.323070
      4   -0.100535
      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

0.6 Estimating Time Series Processes

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

Let's simulate some time series data with an autoregressive structure and noise:

```
[65]: init_x = 4
      import random
      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:

```
[66]: MAXLAGS = 5
      model = sm.tsa.AR(values)
      results = model.fit(MAXLAGS)
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Pra
ctice_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-
packages\statsmodels\tsa\ar_model.py:791: FutureWarning:
statsmodels.tsa.AR has been deprecated in favor of statsmodels.tsa.AutoReg and
statsmodels.tsa.SARIMAX.

AutoReg adds the ability to specify exogenous variables, include time trends,
and add seasonal dummies. The AutoReg API differs from AR since the model is
treated as immutable, and so the entire specification including the lag
length must be specified when creating the model. This change is too
substantial to incorporate into the existing AR api. The function
ar_select_order performs lag length selection for AutoReg models.

AutoReg only estimates parameters using conditional MLE (OLS). Use SARIMAX to
estimate ARX and related models using full MLE via the Kalman Filter.

To silence this warning and continue using AR until it is removed, use:

import warnings
warnings.filterwarnings('ignore', 'statsmodels.tsa.ar_model.AR', FutureWarning)

  warnings.warn(AR_DEPRECATION_WARN, FutureWarning)
```

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

```
[67]: results.params
```

```
[67]: array([-0.00616093,  0.78446347, -0.40847891, -0.01364148,  0.01496872,
              0.01429462])
```

0.7   13.4 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 evalua- tion, data transformation, data loading, and model persistence. These models can be used for classification, clustering, prediction, and other common tasks.

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

At the time of this writing, scikit-learn does not have deep pandas integration, though there are some add-on third-party packages that are still in development. pandas can be very useful for massaging datasets prior to model fitting, though.

As an example, I use a now-classic dataset from a Kaggle competition about passen- ger survival rates on the Titanic, which sank in 1912. We load the test and training dataset using pandas:

```
[68]: train = pd.read_csv('titanic/train.csv')
      test = pd.read_csv('titanic/test.csv')
      train[:4]
```

```
[68]:    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 Th…  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:

```
[69]: train.isnull().sum()
```

```
[69]: 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
```

```
[70]: test.isnull().sum()
```

```
[70]: 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:

```
[71]: impute_value = train['Age'].median()
      train['Age'] = train['Age'].fillna(impute_value)
      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:

```
[72]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)
      test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

```
[73]: train
```

```
[73]:      PassengerId  Survived  Pclass  \
      0              1         0       3
      1              2         1       1
      2              3         1       3
      3              4         1       1
      4              5         0       3
      ..           ...       ...     ...
      886          887         0       2
      887          888         1       1
      888          889         0       3
      889          890         1       1
      890          891         0       3

                                                     Name     Sex   Age  SibSp  \
      0                            Braund, Mr. Owen Harris    male  22.0      1
      1    Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0      1
      2                             Heikkinen, Miss. Laina  female  26.0      0
      3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
      4                           Allen, Mr. William Henry    male  35.0      0
      ..                                               ...     ...   ...    ...
      886                            Montvila, Rev. Juozas    male  27.0      0
```

```
887                     Graham, Miss. Margaret Edith  female  19.0      0
888         Johnston, Miss. Catherine Helen "Carrie"  female  28.0      1
889                            Behr, Mr. Karl Howell    male  26.0      0
890                              Dooley, Mr. Patrick    male  32.0      0

     Parch          Ticket     Fare Cabin Embarked  IsFemale
0        0       A/5 21171   7.2500   NaN        S         0
1        0        PC 17599  71.2833   C85        C         1
2        0  STON/O2. 3101282   7.9250   NaN        S         1
3        0          113803  53.1000  C123        S         1
4        0          373450   8.0500   NaN        S         0
..     ...             ...      ...   ...      ...       ...
886      0          211536  13.0000   NaN        S         0
887      0          112053  30.0000   B42        S         1
888      2       W./C. 6607  23.4500   NaN        S         1
889      0          111369  30.0000  C148        C         0
890      0          370376   7.7500   NaN        Q         0

[891 rows x 13 columns]
```

[74]: `test`

```
[74]:      PassengerId  Pclass                                          Name  \
0              892       3                              Kelly, Mr. James
1              893       3              Wilkes, Mrs. James (Ellen Needs)
2              894       2                     Myles, Mr. Thomas Francis
3              895       3                              Wirz, Mr. Albert
4              896       3  Hirvonen, Mrs. Alexander (Helga E Lindqvist)
..             ...     ...                                           ...
413           1305       3                            Spector, Mr. Woolf
414           1306       1                  Oliva y Ocana, Dona. Fermina
415           1307       3                  Saether, Mr. Simon Sivertsen
416           1308       3                           Ware, Mr. Frederick
417           1309       3                     Peter, Master. Michael J

        Sex   Age  SibSp  Parch              Ticket      Fare Cabin Embarked  \
0      male  34.5      0      0              330911    7.8292   NaN        Q
1    female  47.0      1      0              363272    7.0000   NaN        S
2      male  62.0      0      0              240276    9.6875   NaN        Q
3      male  27.0      0      0              315154    8.6625   NaN        S
4    female  22.0      1      1             3101298   12.2875   NaN        S
..      ...   ...    ...    ...                 ...       ...   ...      ...
413    male  28.0      0      0           A.5. 3236    8.0500   NaN        S
414  female  39.0      0      0            PC 17758  108.9000  C105        C
415    male  38.5      0      0  SOTON/O.Q. 3101262    7.2500   NaN        S
416    male  28.0      0      0              359309    8.0500   NaN        S
417    male  28.0      1      1                2668   22.3583   NaN        C
```

```
         IsFemale
    0           0
    1           1
    2           0
    3           0
    4           1
    ..          …
    413         0
    414         1
    415         0
    416         0
    417         0

    [418 rows x 12 columns]
```

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

```
[75]: predictors = ['Pclass', 'IsFemale', 'Age']
      X_train = train[predictors].values
      X_test = test[predictors].values
      y_train = train['Survived'].values
      X_train[:5]
```

```
[75]: array([[ 3.,  0., 22.],
             [ 1.,  1., 38.],
             [ 3.,  1., 26.],
             [ 1.,  1., 35.],
             [ 3.,  0., 35.]])
```

```
[76]: y_train[:5]
```

```
[76]: array([0, 1, 1, 1, 0], dtype=int64)
```

I make no claims that this is a good model nor that these features are engineered properly. We use the LogisticRegression model from scikit-learn and create a model instance:

```
[77]: from sklearn.linear_model import LogisticRegression
      model = LogisticRegression()
```

Similar to statsmodels, we can fit this model to the training data using the model's fit method:

```
[78]: model.fit(X_train, y_train)
```

```
[78]: LogisticRegression()
```

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

```
[79]: y_predict = model.predict(X_test)
      y_predict[:10]
```

19

```
[79]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0], dtype=int64)
```

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

```
[81]: # (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 pre- diction. Based on a model accuracy score like mean squared error, one can perform a grid search on model param- eters. Some models, like logistic regression, have estima- tor 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:

```
[82]: from sklearn.linear_model import LogisticRegressionCV
      model_cv = LogisticRegressionCV(10)
      model_cv.fit(X_train, y_train)
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Pra
ctice_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-
packages\sklearn\utils\validation.py:72: FutureWarning: Pass Cs=10 as keyword
args. From version 1.0 (renaming of 0.25) passing these as positional arguments
will result in an error
  "will result in an error", FutureWarning)
```

```
[82]: 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 non-overlapping splits of the training data, we can do:

```
[83]: from sklearn.model_selection import cross_val_score
      model = LogisticRegression(C=10)
      scores = cross_val_score(model, X_train, y_train, cv=4)
      scores
```

```
[83]: array([0.77578475, 0.79820628, 0.77578475, 0.78828829])
```

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

0.8   13.5 Continuing Your Education

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 dedica- ted to modeling and data science tools. Some excellent ones are:

- Introduction to Machine Learning with Python by Andreas Mueller 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 (Packt Publishing)

- Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron (O'Reilly)

[ ]: