Linear Statistical Models

Basic Computations with Python and SciPy

Nicholas Bennett
nick@nickbenn.com

October 10, 2018

Copyright and License Information

Copyright © 2018 by Nicholas Bennett. All rights reserved.

“Linear Statistical Models: Basic Computations with Python and SciPy” by Nicholas Bennett is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. Permissions beyond the scope of this license may be available; for more information, contact nick@nickbenn.com.

Redistribution and use of accompanying program code in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

This software is provided by Nicholas Bennett "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall Nicholas Bennett be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

Linear Models

Much of the following is a quick review of the relevant sections from the companion document, "Mathematical Models & Linear Statistical Models: Basic Concepts & Computations". If you have not read that document, we recommend you do so now.

Definition

One of the simplest statistical models is applicable to a wide range of problems. In the *linear model*, a dependent variable is expressed as a linear combination of independent variables and an error term:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \varepsilon, \tag{1}$$

where

X_i are the independent variables;

Y is the dependent variable;

$\beta_i \in \mathbb{R}$, $i = 0, 1, 2, \ldots$ (the coefficients are real numbers);

ε is the error term, a quantity not explained by the model.

Formally, the independent variables are assumed to be continuous over real value ranges; in practice, this condition is often relaxed to allow for integral or other discrete numeric values.

Simple Linear Models

In a simple linear model, there's only one independent variable and one dependent variable, so (1) becomes

$$Y = \beta_0 + \beta_1 X + \varepsilon. \tag{2}$$

This is often written as

$$Y = \alpha + \beta X + \varepsilon. \tag{3}$$

As a rule, we don't know the values of α and β, but must estimate them. We denote our estimates of these two parameters by a and b, respectively; the formula for our estimated line is

$$\hat{Y} = a + bX. \tag{4}$$

In (4), X represents the actual values of the independent variable, while $\hat{Y}$ represents the *fitted* (estimated) values of the dependent variable Y.

As you've probably figured out already, a simple linear model is easy to show graphically, along with the actual data. It's essentially a straight line through the data points, fitting them as closely as possible—though we haven't yet said what "as closely as possible" really means.

Finding the Best Fit via Linear Regression

In linear regression, the values we choose for a and b are those that minimize the sum of squared differences between the fitted and actual values of Y. This sum is called the sum of squared errors (SSE), and is given by

$$\text{SSE} = \sum_{i=1}^{n} \left(y_i - \hat{y}\right)^2. \tag{5}$$

Using calculus, we find that for a simple linear model, a and b can be computed as

$$\begin{aligned} a &= \frac{\sum x^2 \sum y - \sum x \sum xy}{n\sum x^2 - \left(\sum x\right)^2} \\ b &= \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - \left(\sum x\right)^2} \end{aligned} \tag{6}$$

R^2: The Coefficient of Determination

When fitting a statistical model (not necessarily linear) to actual data, we often compute the coefficient of determination, denoted as R^2, as an indicator of how closely the model fits the data. We compute R^2 as

$$R^2 = \frac{\text{SSR}}{\text{SST}}, \tag{7}$$

where SSR (sum of squares of regression) and SST (sum of squares, total) are computed as follows:

$$SSR = \sum \left(\hat{y} - \bar{y}\right)^2 \tag{8}$$

$$\begin{aligned} \text{SST} &= \sum \left(y - \bar{y}\right)^2 \\ &= \sum y^2 - \frac{\left(\sum y\right)^2}{n} \end{aligned} \tag{9}$$

SST, SSR, and SSE (see (5)) can also be expressed in terms of each other:

$$\text{SST} = \text{SSE} + \text{SSR}. \tag{10}$$

The larger that SSR is in relation to SST, the more that the change in the dependent variable is explained by the model. More specifically, we can interpret R^2 as the fraction of the variation in the dependent variable that's determined or explained by the model.

Implementation of Simple Least-Squares Linear Regression

General Features and Components

To tackle the cricket chirp problem described in "Mathematical Models & Linear Statistical Models: Basic Concepts & Computations" (using data in Appendix A, illustrated in Figure 3), we'll use the Python programming language. The code that accompanies this document is functional but incomplete: it runs as-is, but it doesn't perform the simple linear least-squares regression analysis; for that, we'll have to add to the code.

An important feature that's already provided in the code, but that we won't be examining, is the capability to read a simple table of values from a comma-separated-values (CSV) text file.

While graphical output isn't strictly necessary for regression analysis, it can be very useful in exploratory analysis and in presenting regression results. Thus, the provided Python code uses the Matplotlib open source library to display the input data and the regression results [3].

Development Tools

The provided Python code requires Python 2.7+ or Python 3.4+ with Matplotlib 2.x, or Python 3.5+ with Matplotlib 3.x. It can be edited and run with virtually any Python development and runtime environment that satisfies these version constraints.

Computational Methods

There are a number of different ways to compute measures and estimates used in descriptive and inferential statistics, including those for least-squares regression. For example, the following formulas for a and b are mathematically equivalent to (6).

$$b = \frac{\sum (x - \bar{x})(y - \bar{y})}{\sum (x - \bar{x})^2} \qquad (11)$$
$$a = \bar{y} - b\,\bar{x}$$

It's easy to think of computers as perfect calculators, but the reality is different—especially when we put them to the task of numerical analysis with floating-point values:

- Some methods are efficient, but potentially *unstable*—i.e. in some cases, the calculations magnify the inaccuracy inherent in the standard computer representations of most floating-point values.
- Some methods, like (11), are more stable, but less efficient.
- Still other methods are relatively stable and efficient, but not as easy to understand or implement.

The method for computing a and b used in (6) has potential scale and stability problems; on the

other hand, it's easily understood and implemented, and it has generally good performance. For that reason, we'll use this method in the first set of additions to our Python code.

Programming Concepts

Intrinsic data structures

Like virtually all programming languages, Python has certain types of data that are *intrinsic*—that is, they're recognized and supported natively by compilers or interpreters of the Python language. Some of these types are *scalar*, holding a single value, and some hold sequences of multiple values. The basic Python intrinsic sequence types (which are in fact called *sequences*) include *strings* of characters (**`str`** and **`unicode`**), *arrays* of single bytes (**`bytearray`**), and *lists* and *tuples*, which are sequences of objects. A **`list`** is *mutable*—that is, items can be added to and removed from a list—while a **`tuple`** is *immutable*. In Python, a programmer doesn't really know or care how the members of a tuple or list are stored in memory, and whether they're contiguous or not; the focus tends to be less on scanning across the elements in a step-by-step fashion, and more on the high-level operations that are applied to the entire list at once, or to a subset satisfying some condition.

Line and statement structure

In Python, a program statement is ended by a line break, unless that line break occurs inside a set of parentheses or brackets, or the line ends with the continuation character (the backslash). This means that care must be taken when breaking a line for readability purposes. (A semicolon may also be used to end a Python statement, but this is usually done only when including multiple statements on the same line.)

In Python, flow control statements (other than **`return`** and **`break`**), as well as **`class`** and **`def`** statements (**`def`** marks the start of a function or method definition) are called *clauses*; a clause is terminated by a colon, and must be followed by a *suite* of statements. A suite consists of one or more statements that are controlled by the flow control or definition clause that precedes it. If the suite contains more than one statement, those statements must be indented below the controlling clause; a suite with one statement may follow the clause on the same line. If no statements at all are wanted in a suite (this often happens when first writing the code for a clause), the **`pass`** statement is used.

The rules for indentation of suites result in one of the most distinctive characteristics of Python code: rather than simply being a matter of style, indentation is syntactically significant. Further, inconsistent indentation (including mixing tab characters and space characters) can produce syntax errors that prevent a Python program from running. For programmers who aren't used to Python syntax, this can be inconvenient, at least at the start. But there's a benefit to these rules: in syntactically correct Python code, the visual structure of the code usually matches the logical structure quite closely.

Conditional execution of statements

Like virtually all programming languages, Python includes statements for testing a condition, then following one path of execution if the condition is true, and (optionally) following another path if the condition is false. The keyword that introduces these conditional statements in Python is **`if`**; zero or more **`elif`** (*else-if*) clauses may follow the **`if`** clause. After any **`elif`** clauses, there may be an **`else`** clause.

Let's look at a few simple code fragments (Listing 1) showing conditional statements. Notice that the **`#`** character starts a comment that continues until the end of the line. Python also supports a special comment format for code documentation (which we'll see in the code we modify), but that's not shown here. Also, note the "lower snake case" convention used for multi-word identifiers (variable or function names). Finally, note that ***`do_something()`***, ***`do_something_else()`***, etc. are simply placeholders for actual functions—they don't actually correspond to anything in the example listing, or to built-in functions in either language.

```
# Execute a suite if the value of some variable x is less than or equal to
# the value of another variable y; otherwise, skip over the suite.
if x <= y:
    do_something()

# Execute a suite if the value of some variable x is less than or equal to
# the value of another variable y; otherwise, execute a second suite.
if x <= y:
    do_something()
else:
    do_something_else()

# Execute a suite if the value of some variable x is less than or equal to
# the value of another variable y; otherwise, execute a second suite if x is
# x is greater than z; otherwise, execute a third block.
if x <= y:
    do_something()
elif x > z:
    do_another_thing()
else:
    do_something_else()
```

Listing 1: Conditionals in Python

Explicit and implicit iteration

Python has multiple mechanisms for iterating over a sequence. The **`for`** and **`while`** statements are used for explicit iteration; additionally, there are implicit iteration constructs, such as the **`map`** function and *comprehensions*, that apply an operation to every item in a sequence, and return a new sequence. See Listing 2 for simple examples of both explicit and implicit iteration.

```
# Repeat a suite for each value of i from 0 to len(some_list)−1
# (inclusive), incrementing i after each iteration.
for i in range(len(some_list)):
    do_something()

# Repeat a suite for each element of another_list.
for j in another_list:
    do_something()

# Repeat a suite as long as an already declared variable k is less than
# limit.
while k < limit:
    do_something()

# Use list comprehension to construct dest_list based on the elements of
# source_list, where each element of dest_list is the square of the
# corresponding value from source_list.
dest_list = [x ** 2 for x in source_list]

# Use list comprehension to construct dest_list based on a subset of the
# elements of source_list: each element of dest_list is the doubled value
# of the corresponding value from source_list, but only for the positive
# values in source_list.
dest_list = [2 * x for x in source_list if x > 0]
```

Listing 2: Explicit and implicit iteration in Python

References to members of an object

Python supports the definition of *classes* for *object-oriented programming*. A class is simply a data structure definition, *along with definitions of the behaviors associated with that data*; an object is a variable based on a class definition. Classes often correspond to the types of physical or logical real-world objects that are represented in the program. A general introduction to object-oriented programming is beyond the scope of this document, so we'll focus just on those aspects that are most relevant to our immediate needs.

Note that Python doesn't insist on an object-oriented approach; this often allows for simpler code in cases where new classes aren't needed—but when an object-oriented approach is employed, Python code in class definitions can be a bit unintuitive. In particular, the use of `self` to refer to the current object instance is not optional (as it is in some languages): it is required within methods when referring to data and other methods of the object, and `self` must be the first parameter of each method defined in the class.[1]

1 Python also supports the definition and use of *class* methods and *static* methods and data. These are associated with the class as a whole, and not with instances of a class; `self` isn't a parameter for class or static methods.

Running the Initial Program

Python program (`chirps.py`)

Using your chosen Python development tools, locate and open the **`chirps.py`** and **`simple_linear.py`** files. The first of these is the main Python script that you'll run to execute the linear regression, and the second will perform the mathematical calculations.

Before we make any additions to the code, let's see what it does so far. Begin by running the **`chirps.py`** script in your Python development environment (or from the command line); you should see a window displaying the cricket chirps vs. temperature data (Figure 1).

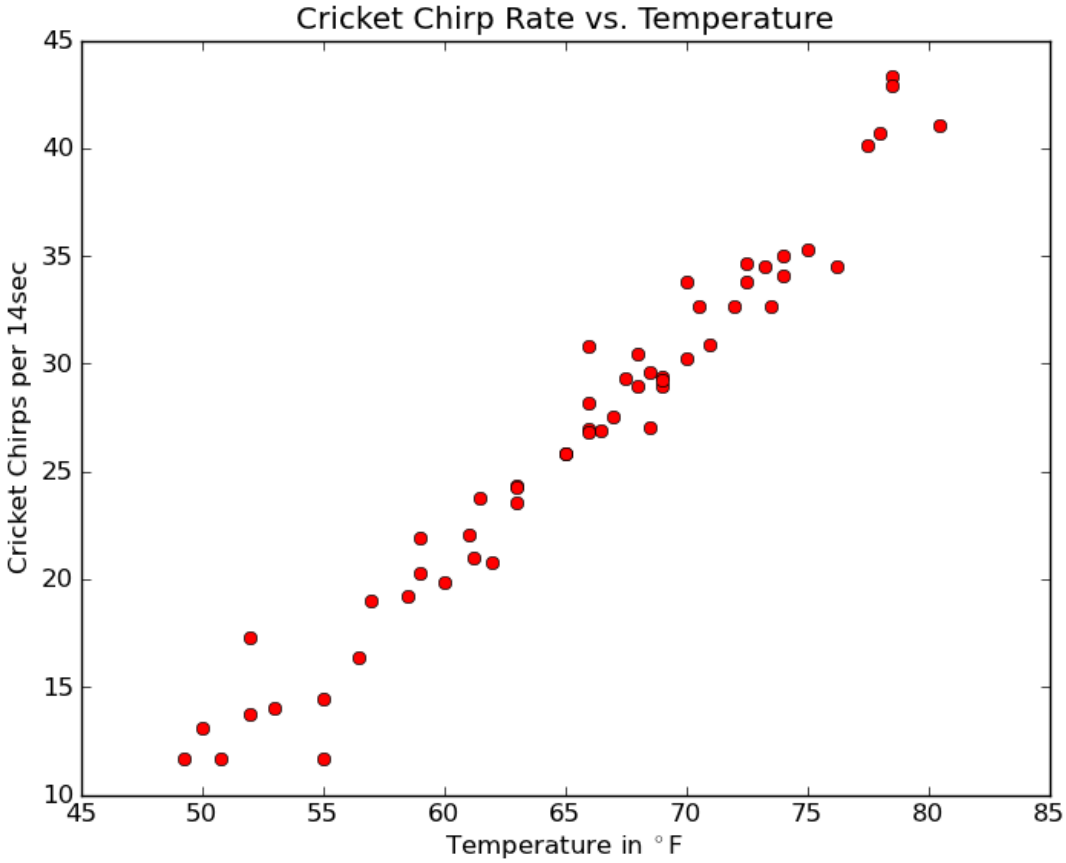

Figure 1: Initial Python program display

Let's examine what's happening at a high level, and follow along with the code itself (Outline 1).

1. **`__main__`** script [**`chirps.py`**]

   Main program script, which invokes functions in **`chirps.py`** and the other files.

   (a) **`load`** function [**`chirps.py`**]

      i. **`TableFileParser.__init__`** constructor [**`parser_util.py`**]

         Opens the specified data file, and reads it into memory as a list of lines, where each line is a list of data values.

      ii. **`TableFileParser.floats`** method [**`parser_util.py`**]

         Returns the previously read data values as floating-point numbers.

   (b) **`SimpleLinear.__init__`** constructor [**`simple_linear.py`**]

      Copies the provided data to **`self._x`** and **`self._y`**, and stores the number of data points in **`self._n`**, in preparation for linear regression.

   (c) **`SimpleLinear.regress`** method [**`simple_linear.py`**]

      i. **`SimpleLinear._compute_sums`** method [**`simple_linear.py`**]

         Currently, this method doesn't do anything (notice the **`pass`** statement) Shortly, we'll add the code to calculate $\sum x$, $\sum y$, $\sum xy$, $\sum x^2$, and $\sum y^2$, and store those values in **`self._sum_x`**, **`self._sum_y`**, **`self._sum_xy`**, **`self._sum_x2`**, and **`self._sum_y2`**, respectively.

      ii. **`SimpleLinear._estimate_parameters`** method [**`simple_linear.py`**]

         We'll add code to this method to use the values computed by **`SimpleLinear._compute_sums`** to calculate the estimated intercept and slope of the regression line, and store those in **`self._intercept`** and **`self._slope`** (respectively).

      iii. **`SimpleLinear._measure_fit`** method [**`simple_linear.py`**]

         Here, we'll add the code to compute SST, the fitted points on the line, and SSE, and use those values to calculate R^2, then store that value in **`self._r2`**.

   (d) **`plot`** function [**`chirps.py`**]

      This function invokes methods in the Matplotlib library to render the scatterplot. Notice that this function has an **`if`** clause: the suite for that clause (which displays the regression line and equation) will only be executed if **`model.r2 >= 0`** (**`model`** is a variable based on the **`SimpleLinear`** class). Before we add the code for the **`SimpleLinear._compute_sums`**, **`SimpleLinear._estimate_parameters`**, and **`SimpleLinear._measure_fit`** methods, what is the **`self._r2`** value of a **`SimpleLinear`** object?

Outline 1: Python program structure

Implementing Linear Regression

Computing the sums

In computing $\sum x$, $\sum y$, $\sum xy$, $\sum x^2$, and $\sum y^2$, the fact that the *X* and *Y* values are stored in Python sequences (specifically, in tuples) makes our job relatively easy. For one thing, Python includes a **`sum`** function that computes the sum of a sequence, without us having to write code to iterate over the sequence. Another useful function is **`map`**, which transforms 1 or more input sequences into an output sequence; this is ideal for constructing a sequence containing the products of corresponding terms in 2 different sequences, so that we can then use **`sum`** on the resulting sequence. Finally, for computing the sum of squared items in a sequence, we can use either the **`map`** function or list comprehension to create a sequence of squared values, and then apply the **`sum`** function to that.

The code below shows the **`_compute_sums`** method of the **`SimpleLinear`** class. The code that we need to keep from the initial version is in gray, normal-weight type; the code we need to add is in dark bold type. One line of code that we should delete is shown in strike-through type.

```
    def _compute_sums(self):
        """Computes intermediate sums for regression."""
        pass
        self._sum_x = sum(self._x)
        self._sum_y = sum(self._y)
        self._sum_xy = sum(map(lambda x, y: x * y, self._x, self._y))
        self._sum_x2 = sum(x * x for x in self._x)
        self._sum_y2 = sum(y * y for y in self._y)
```

Listing 3: **`SimpleLinear._compute_sums`** method

(Be sure you still have a blank line after the added code, before the **`_estimate_parameters`** method that follows it. Also, watch out for underscore characters, which can be hard to see.)

The first 2 lines we added are pretty straightforward: in each, we're simply using the **`sum`** function to add up all of the items in a sequence. The next line, however, is a little less obvious: the map function expects at least 2 arguments: the first is a *lambda*, or *anonymous function*, which states how the items in the sequences specified in the remaining arguments should be transformed to produce new values. In this case, **`lambda x, y: x * y`** specifies that there will be 2 sequences processed as inputs (based on the number of parameters declared for the lambda, between the **`lambda`** keyword and the colon character), and that each value in the resulting sequence will be produced by multiplying the corresponding values from the input sequences together.

The last 2 lines we added use *comprehension*, in which we specify an expression (e.g. **`x * x`**) to be applied and evaluated for every item in an existing sequence, to produce a new sequence. This may sound very similar to the operation of the **`map`** function, and it is: In many cases, comprehension can be seen as a simplified way of expressing an operation that we might otherwise express with the **`map`** and/or **`filter`** functions.

Finally, note that we're assigning each of the computed sums to the relevant field of the current

SimpleLinear object (**self**).

Save your changes and run **chirps.py** again. Fix any syntax errors reported by Python.

Estimating the Regression Parameters

Now that we have the component sums, we can use them to compute *a* and *b*. To do this, let's add a few lines of code to the **SimpleLinear._estimate_parameters** method:

```
    def _estimate_parameters(self):
        """Computes intercept and slope from sums."""
        pass
        self._intercept = (
                (self._sum_x2 * self._sum_y - self._sum_x * self._sum_xy)
                / (self._n * self._sum_x2 - self._sum_x * self._sum_x))
        self._slope = (
                (self._n * self._sum_xy - self._sum_x * self._sum_y)
                / (self._n * self._sum_x2 - self._sum_x * self._sum_x))
```

Listing 4: **SimpleLinear._estimate_parameters** method

We're only adding 2 statements this time—but because each is fairly long, and includes several terms in its calculations, together they extend over 6 lines. Be careful to keep your parentheses balanced (also, notice that Python allows a statement to extend over multiple lines, even without a line continuation character, if there's an open set of parentheses, brackets, or braces).

The expressions for the values computed and assigned to **_intercept** and **_slope** are taken directly from the expressions for *a* and *b* (respectively) in (6).

Once again, you can check for some errors in the code by saving your changes and running **chirps.py**. There won't be any visible change to the plotted output (since we haven't yet computed R^2), but the new code will be parsed and executed, allowing Python to catch syntax errors.

Computing $\hat{Y}$ and R^2

Now that we have our regression coefficients, *a* and *b*, we can compute the fitted values, $\hat{Y}$; from the latter, we can then compute the coefficient of determination, R^2. This will be performed by code in the **_measure_fit** method of of the **SimpleLinear** class:

```
    def _measure_fit(self):
        """Computes fitted values and goodness-of-fit statistic(s)."""
        pass
        sst = self._sum_y2 - self._sum_y * self._sum_y / self._n;
        fitted = [self._intercept + self._slope * x for x in self._x]
        sse = sum(map(lambda y, y_hat: (y - y_hat) ** 2, self._y, fitted))
        self._r2 = 1 - sse / sst
```

Listing 5: **SimpleLinear._measure_fit** method

In computing R^2, we're taking advantage of the relationship between SST, SSR, and SSE stated in (10). Specifically, we have

$$R^2 = \frac{\text{SSR}}{\text{SST}}$$
$$= \frac{\text{SST} - \text{SSE}}{\text{SST}},$$
$$= 1 - \frac{\text{SSE}}{\text{SST}} \qquad (12)$$

Displaying the Output of the Linear Regression Analysis

If we review the code of the plot function in **`chirps.py`**, we recall that when R^2 is not equal to zero, the fitted line is plotted, and the equation of the fitted line (along with the value of R^2) is added to the plot as a legend. Given that, and assuming our code is written correctly, our program should now compute and display the linear model fitted (using linear regression) to the chirps & temperature data. Save your code changes and execute the **`chirps.py`** script once more, and you should see something like Figure 2.

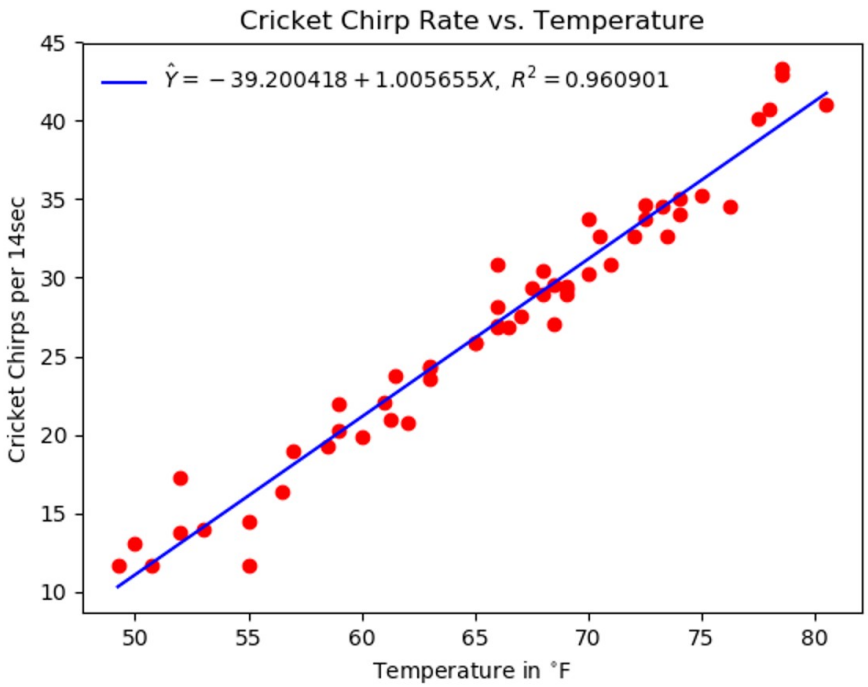

Figure 2: Cricket chirps vs. temperature, with fitted model

Using SciPy for Linear Regression

Given that fitting statistical models (linear or otherwise) to data is a common task in scientific computing, decision sciences, etc., we would expect to find this functionality in some 3rd-party Python libraries. In fact, SciPy—one of the most widely used Python libraries for scientific and engineering applications—includes functions for performing regression analysis.

We can easily replace our regression code with in invocation of the **`scipy.stats.linregress`** function, with a few changes to the **`simple_linear.py`** file.

Import `linregress`

First, let's tell Python that we want to use the `linregress` function. To do this, add the following line near the top of the file. (In general, `import` statements are found in the first non-blank, non-comment lines of a Python file.)

```
from scipy.stats import linregress
```

Listing 6: Importing `scipy.stats.linregress` in `simple_linear.py`

Estimate parameters, compute R^2 and fitted values

The `linregress` function returns the values of *a* and *b*, as well as the correlation coefficient (which can be squared to get the coefficient of determination)—along with some other values that we won't be using (for now). Essentially, we can replace our `_compute_sums`, `_estimate_parameters`, and `_measure_fit` methods with an invocation of `linregress`, plus a couple more lines of code. So we might as well put this code in the `SimpleLinear.regress` method:

```
    def regress(self):
        """Perform least squares regression on data."""
        self._slope, self._intercept, r, p, std_err = linregress(self._x, self._y)
        self._r2 = r * r
        fitted = [self._intercept + self._slope * x for x in self._x]
        self._compute_sums()
        self._estimate_parameters()
        self._measure_fit()
```

Listing 7: Invoking `linregress` in `SimpleLinear.regress` method

Remove unused methods

Since we're no longer invoking the `_compute_sums`, `_estimate_parameters`, and `_measure_fit` methods, we should remove them from the `SimpleLinear` class; however, this is left as an exercise for you.

Test the SciPy version

Save your changes and run `chirps.py`. Do you notice any changes? Should we expect any?

References

[1] “Cricket Chirps: Nature's Thermometer”. Retrieved Oct. 7, 2018 from http://www.almanac.com/cricket-chirps-temperature-thermometer

[2] M. A. LeMone. (Oct. 5, 2007). “Measuring temperature using crickets”. Retrieved Oct. 7, 2018 from https://www.globe.gov/explore-science/scientists-blog/archived-posts/sciblog/index.html_p=45.html

[3] John Hunter, Darren Dale, Michael Droettboom. (Sep. 21 2018). Matplotlib v3.0.0. Retrieved Oct. 9, 2018 from https://matplotlib.org

Appendix A: Cricket Chirps vs. Temperature

The following observations were recorded by Dr. Margaret LeMone in Boulder, Colorado, over a 30 day period in August and September, 2007 [2]. The measurements were originally in chirps per 30 seconds; the column for chirps per 14 seconds was derived from the original data.

Date	Time	Chirps/15s	Chirps/14s	Temp (°F)
21 Aug	2030	44	41.067	80.5
21 Aug	2100	46.4	43.307	78.5
21 Aug	2200	43.6	40.693	78
24 Aug	1945	35	32.667	73.5
24 Aug	2015	35	32.667	70.5
24 Aug	2100	32.6	30.427	68
24 Aug	2200	28.9	26.973	66
24 Aug	2230	27.7	25.853	65
25 Aug	0030	25.5	23.8	61.5
25 Aug	0330	20.375	19.017	57
25 Aug	0500	12.5	11.667	55
25 Aug	2000	37	34.533	76.25
25 Aug	2030	37.5	35.0	74
25 Aug	2100	36.5	34.067	74
25 Aug	2200	36.2	33.787	72.5
26 Aug	0530	33	30.8	66
26 Aug	2030	43	40.133	77.5
26 Aug	2200	46	42.933	78.5
27 Aug	2000	29	27.067	68.5
27 Aug	2030	31.7	29.587	68.5
27 Aug	2100	31	28.933	68
27 Aug	2200	28.75	26.833	66
28 Aug	0240	23.5	21.933	59
28 Aug	2010	32.4	30.24	70
28 Aug	2050	31	28.933	69
28 Aug	2200	29.5	27.533	67
29 Aug	0240	22.5	21.0	61.25
29 Aug	0440	20.6	19.227	58.5
29 Aug	2000	35	32.667	72
29 Aug	2050	33.1	30.893	71
29 Aug	2200	31.5	29.4	69
29 Aug	2330	28.8	26.88	66.5
30 Aug	0330	21.3	19.88	60
30 Aug	2000	37.8	35.28	75
30 Aug	2055	37	34.533	73.25
30 Aug	2200	37.1	34.627	72.5
1 Sep	2200	36.2	33.787	70
2 Sep	0330	31.4	29.307	67.5

Date	Time	Chirps/15s	Chirps/14s	Temp (ºF)
2 Sep	0600	30.2	28.187	66
4 Sep	0240	31.3	29.213	69
4 Sep	0505	26.1	24.36	63
5 Sep	0500	25.2	23.52	63
6 Sep	0600	23.66	22.083	61
7 Sep	0215	22.25	20.767	62
7 Sep	0525	17.5	16.333	56.5
9 Sep	2010	15.5	14.467	55
9 Sep	2110	14.75	13.767	52
10 Sep	2115	15	14.0	53
10 Sep	2210	14	13.067	50
11 Sep	0315	18.5	17.267	52
16 Sep	2100	27.7	25.853	65
17 Sep	2200	26	24.267	63
18 Sep	0130	21.7	20.253	59
19 Sep	0415	12.5	11.667	50.75
19 Sep	0435	12.5	11.667	49.25

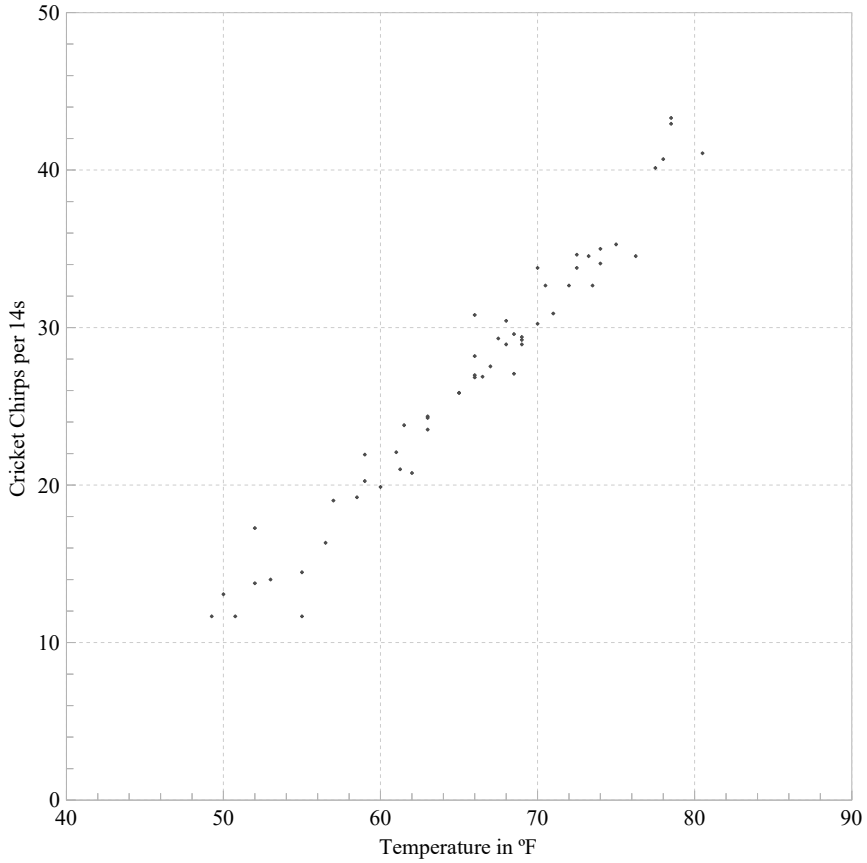

Figure 3: Cricket Chirps vs. Temperature

Appendix B: Python Implementation of Simple Linear Regression

All comments and docstrings have been omitted from these code listings, and any occurrence of multiple consecutive blank lines has been collapsed into a single line.

```
import sys

import matplotlib.pyplot as pyplot
from parser_util import TableFileParser
from simple_linear import SimpleLinear

DEFAULT_DATA_FILE = "chirps.csv"
DEFAULT_DELIMITER = ","
DEFAULT_SKIP_LINES = 1

CHART_TITLE = "Cricket Chirp Rate vs. Temperature"
X_AXIS_TITLE = "Temperature in $^{\circ}$F"
Y_AXIS_TITLE = "Cricket Chirps per 14sec"
MODEL_SPEC = "$\hat{Y} = %8.6f + %8.6fX, \; R^2 = %8.6f$"

def load(args):
    default_params = [DEFAULT_DATA_FILE, DEFAULT_DELIMITER, DEFAULT_SKIP_LINES]
    data_file, delimiter, skip = map(lambda default, actual:
            actual if actual is not None else default,
            default_params, args)[:3]
    observations = TableFileParser(data_file, delimiter, skip).floats()
    x, y = zip(*observations)
    return (x, y)

def plot(model):
    pyplot.title(CHART_TITLE)
    pyplot.xlabel(X_AXIS_TITLE)
    pyplot.ylabel(Y_AXIS_TITLE)
    pyplot.plot(model.x, model.y, marker='o', linestyle='None', color='red')
    if model.r2 >= 0:
        x_bounds = (min(model.x), max(model.x))
        y_fit = [model.intercept + model.slope * x for x in x_bounds]
        fit, = pyplot.plot(x_bounds, y_fit, marker='None', linestyle='-',
                color='blue')
        pyplot.legend([fit],
                [MODEL_SPEC % (model.intercept, model.slope, model.r2)],
                loc='upper left', frameon=False)
    pyplot.show()

if __name__ == "__main__":
    x, y = load((sys.argv + [None] * 3)[1:4])
    model = SimpleLinear(x, y)
    model.regress()
    plot(model)
```

Listing 8: `chirps.py`

```
class SimpleLinear(object):

    def __init__(self, x, y):
        self._n = len(x)
        self._x = tuple(x)
        self._y = tuple(y)
        self._intercept = 0
        self._slope = 0
        self._r2 = -1

    def regress(self):
        self._compute_sums()
        self._estimate_parameters()
        self._measure_fit()

    @property
    def n(self):
        return self._n

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    @property
    def intercept(self):
        return self._intercept

    @property
    def slope(self):
        return self._slope

    @property
    def r2(self):
        return self._r2

    def _compute_sums(self):
        self._sum_x = sum(self._x)
        self._sum_y = sum(self._y)
        self._sum_xy = sum(map(lambda x, y: x * y, self._x, self._y))
        self._sum_x2 = sum(x * x for x in self._x)
        self._sum_y2 = sum(y * y for y in self._y)

    def _estimate_parameters(self):
        self._intercept = (
                (self._sum_x2 * self._sum_y - self._sum_x * self._sum_xy)
                / (self._n * self._sum_x2 - self._sum_x * self._sum_x))
        self._slope = (
                (self._n * self._sum_xy - self._sum_x * self._sum_y)
                / (self._n * self._sum_x2 - self._sum_x * self._sum_x))

    def _measure_fit(self):
        fitted = [self._intercept + self._slope * x for x in self._x]
        sst = self._sum_y2 - self._sum_y * self._sum_y / self._n;
        sse = sum(map(lambda y, y_hat: (y - y_hat) ** 2, self._y, fitted))
        self._r2 = 1 - sse / sst
```

Listing 9: `simple_linear.py`

```
from scipy.stats import linregress

class SimpleLinear(object):

    def __init__(self, x, y):
        self._n = len(x)
        self._x = tuple(x)
        self._y = tuple(y)
        self._intercept = 0
        self._slope = 0
        self._r2 = -1

    def regress(self):
        self._slope, self._intercept, r, p, std_err = linregress(
                self._x, self._y)
        self._r2 = r * r
        fitted = [self._intercept + self._slope * x for x in self._x]

    @property
    def n(self):
        return self._n

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    @property
    def intercept(self):
        return self._intercept

    @property
    def slope(self):
        return self._slope

    @property
    def r2(self):
        return self._r2
```

Listing 10: **simple_linear.py** (using SciPy)

```
class TableFileParser(object):

    def __init__(self, file_name, delimiter, skip=0):
        specified file and parsing them into a list of string lists."""
        with open(file_name) as file:
            table = [line.strip().split(delimiter) for line in file
                if 0 != len(line.strip())]
            self._table = table[skip:]

    def strings(self):
        return [row[:] for row in self._table]

    def ints(self):
        return [[int(col) for col in row] for row in self._table]

    def floats(self):
        return [[float(col) for col in row] for row in self._table]
```

Listing 11: `parser_util.py`

Acknowledgements

These materials, and the accompanying Python code, were written with the invaluable editorial and creative assistance of L.D. Landis, Drew Einhorn, and Joel Castellanos.