Linear Statistical Models

Basic Concepts with Implementations in Python and Java

Nicholas Bennett
nickbenn@g-r-c.com

November 5, 2011

Copyright and License Information

Copyright © 2011 by Nicholas Bennett, with the exception of the first section, “Things Change” (see Acknowledgements for more information). All rights reserved.

“Linear Statistical Models: Basic Concepts with Implementations in Python and Java” by Nicholas Bennett is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. Permissions beyond the scope of this license may be available; for more information, contact nickbenn@g-r-c.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.

Things Change

(Adapted with permission from “Things Change” by the Maryland Collaborative for Teacher Preparation [1].)

Introduction

The only thing certain in life is change. From birth we each grow taller, heavier (and sometimes lighter), older, wiser, richer (and sometimes poorer). We live in human communities with populations that are changing minute by minute through births and deaths and immigration, and at any instant of time many of those people are in motion on foot, bicycles, cars, buses, trains, and airplanes. The physical features of the world around us are in constant motion – pushed by forces of wind and water and gravity – and our planet Earth is racing around the sun at nearly 1,700,000 miles per day. In our economic lives the prices we pay for food, clothes, shelter, transportation, education, and entertainment go up and down in response to consumer demand and producer supply.

Many of the most important problems in mathematics beyond arithmetics require description and prediction of changes in related quantitative variables – in other words, construction and use of models of change. In some cases those problems involve analysis of changes in variables as time passes; in other cases the problem is to understand the ways that changes in some variables cause changes in other variables. Algebra and calculus are at the heart of this study of change.

Patterns of Change

When we have information about a relationship between two or more variables, one of the best ways to present that information is with coordinate graphs of (x, y) data pairs. Such graphs may display specific pairs of related numerical values, a line or curve representing the general relationship between the x and y values, or both.

The following statements describe nine different situations in which two variables are (or at least seem to be) related to each other. Match each situation to the graph that you believe is most likely to represent the relation between those variables. Then explain as carefully as you can what the shape of the graph tells about the ways the variables change in relation to each other.

1. When a tennis player hits a high lob shot, its height changes as time passes. What pattern seems likely to relate time and height?
2. The senior class officers at Lincoln High School decided to order and sell souvenir baseball caps with the school insignia, name, and "Class of '95" on them. One supplier said it would charge $100 to make the design and then an additional $4 for each cap made. How would the total cost of the order be related to the number of caps in the order?
3. The population of the world has been increasing for as long as data or estimates have been available. What pattern of population growth has occurred over that time?

4. In planning a bus trip to Florida for spring break, a travel agent worked on the assumption that each bus would hold at most 40 students. How would the number of buses be related to the number of student customers?

5. The depth of water under the U. S. Constellation in Baltimore Harbor changes due to tides as time passes in a day. What pattern would that (time, depth) data fit?

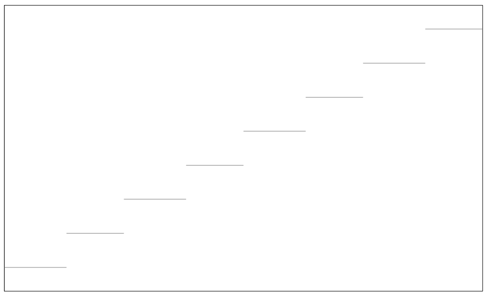
6. When the Lincoln High School class officers decided to order and sell t-shirts with names of everyone in the Class of `95, they checked with a sample of students to see how many would buy at various proposed prices. How would sales be related to price charged?

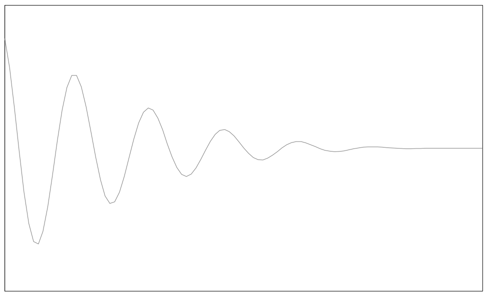
7. How does the height of a bungee jumper vary as time passes in the jump?

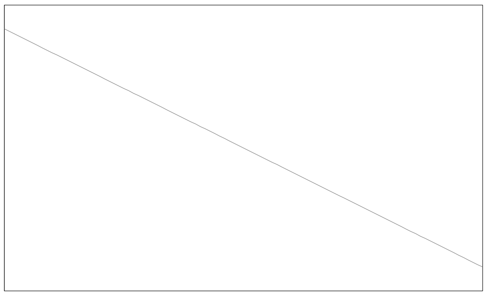
8. In a wildlife experiment, all fish were removed from a lake and the lake was restocked with 1000 new fish. The population of fish then increased over the years as time passed. What pattern would likely describe change in fish population over time?

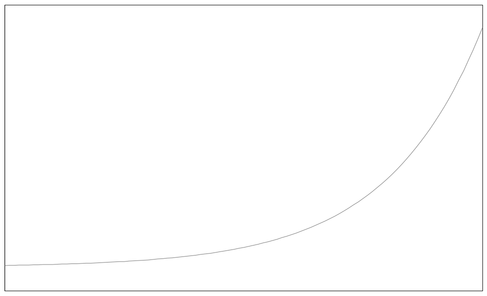
9. According to *The Old Farmer's Almanac*, if you live near crickets, you can estimate the nighttime outdoor temperature in degrees Fahrenheit by counting the number of cricket chirps in 14 seconds and adding 40 to that number. If you tested that by gathering data and plotting chirps vs. temperature, and the data seemed to support the rule of thumb, what might the graph of observations look like?

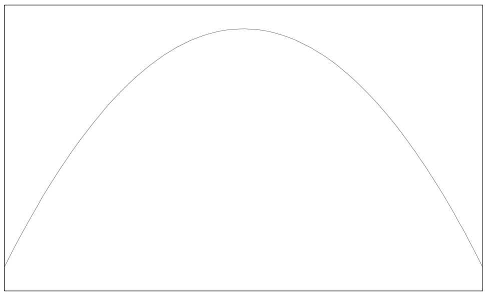
For each of the scenarios above, select the graph from the following page that you think most closely matches the pattern or relationship described. Try not to focus on the units or scales used on the axes of each graph (in fact, none of the graphs have such notations), but on the general nature of the relationships between the quantities described.

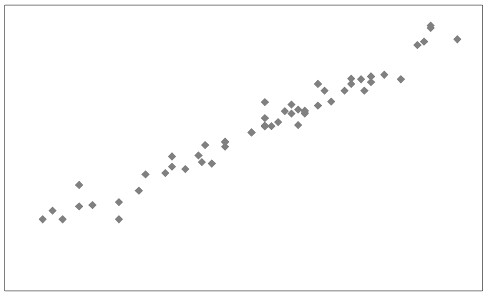

A

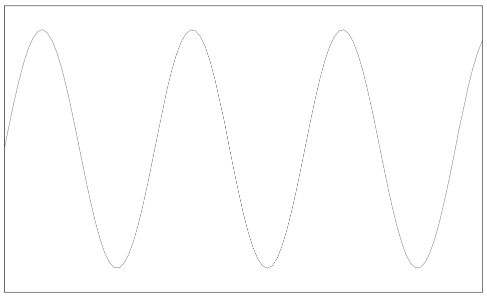

B

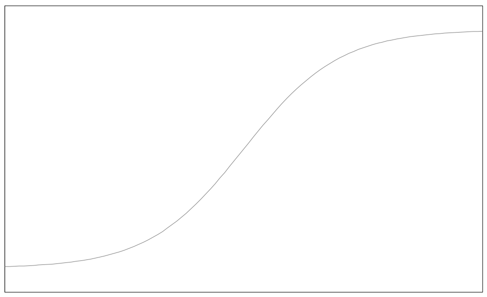

C

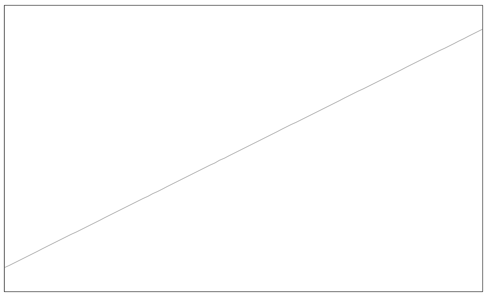

D


E


F


G


H


I

Mathematical Models

Introduction

When the relationships between the quantities of a real-world problem can be expressed in mathematical formulas, we refer to those formulas – along with the “key” that maps the variables and constants in the formulas to real-world quantities and units of measure – as a *mathematical model* of the problem.

Example

While exiting the Lunar Module, Apollo 14 mission commander Alan Shepard rolls a golf ball off the platform at the top of the ladder.[1] The golf ball leaves the platform moving horizontally and falls to the Moon's surface from a height of 3 meters. The Moon's gravitational acceleration close to the surface is 1.63 meters per second per second. (In other words, after 1 second, a body will be falling at 1.63 meters per second; after 2 seconds, it will be falling at 3.26 meters per second; and so on.) Since the Moon's atmosphere is so thin, we can ignore atmospheric drag.

If we want to know how the height above the surface of the golf ball at some time *t* after it leaves the platform, up to the moment it hits the ground, we can start with the general formulas for a body falling in a vacuum, and adapt them to our problem.

Given

$$\begin{aligned} g &= \text{gravitational acceleration} \\ t &= \text{time} \\ v_0 &= \text{initial vertical speed} \\ h_0 &= \text{initial height} \\ v_t &= \text{vertical speed at time } t \\ h_t &= \text{height at time } t \end{aligned} \tag{1}$$

Then

$$\begin{aligned} v_t &= v_0 + g\,t \\ h_t &= h_0 + v_0 t + \frac{g}{2} t^2 \end{aligned} \tag{2}$$

For our scenario,

$$\begin{aligned} g &= -1.63 \text{ meters/sec}^2 \\ v_0 &= \quad 0 \text{ meters/sec} \\ h_0 &= \quad 3 \text{ meters} \end{aligned} \tag{3}$$

1 In reality, while Alan Shepard famously used a makeshift golf club to hit two golf balls on the Moon, he didn't roll any golf balls off the LM platform, as far as we know.

Taken together, (1), (2), and (3) make up a mathematical model that can be used to answer a number of questions about our hypothetical golf ball. We can even set $h_t = 0$ and use the quadratic formula to solve the second equation of (2) for *t*, to find out how long it will take the ball to reach the ground.

Mathematical Formulas as Patterns of Change

By itself, (2) might not be a very useful model, since without (1) it might not be clear what the different variables refer to, and since it doesn't include the information in (3) that's specific to our scenario. But general formulas like those in (2) are the essential core of a mathematical model: just as a graph visually conveys the relationship between variables, a formula does the same thing symbolically.

See if you can match the equations shown below to the corresponding scenarios or graphs in "Patterns of Change". While none of the formulas is a complete model, some are tied very specifically to the corresponding scenarios. Others fit the scenarios to some degree (at least in the general shape), but might not express the underlying mathematics accurately. Some are written in a general form, where c_0, c_1, etc. represent constant values in the equations.

This is intended to be a challenging exercise, especially when it comes to the last few formulas. Don't worry if you don't understand all of the symbols and notation used; see if you can guess their meaning from their appearance and use. If you can't match all of the formulas to scenarios or graphs, keep in mind that it's possible that some of the formulas don't correspond to any of the scenarios or graphs, and vice versa. On the other hand, it's also possible that one (or more) of the general formulas matches more than one scenario, or that more than one formula can be matched with a single scenario.

i. $y = 4\mathrm{x} + 100$

ii. $y = -16(x-2)^2 + 70$

iii. $y = \left\lceil \frac{x}{40} \right\rceil$

iv. $y = x - 40 + \varepsilon$

v. $y = c_0 + c_1 x$

vi. $y = c_0 + c_1 \sin x$

vii. $y = c_0 + c_1 \frac{\cos x}{x+1}$

viii. $y = \frac{c_0}{1 + e^{c_1 - x}}$

ix. $y = c_0 e^{c_1(x + c_2)}$

Statistical Models

Introduction

The final “Patterns of Change” scenario (page 4) mentions a relationship claimed to exist between temperature and the rate of cricket chirps. Of course, even if such a relationship exists, there could be many other factors besides temperature that affect the rate of crickets' chirps; because of this, creating a formal model for this scenario presents a challenge: Even if we analyze the data very carefully, and quantity all the factors we can, we won't be able to predict the outcome entirely; some portion will remain unpredictable. This unpredictable part might be due to known factors that are beyond our practical ability to measure or control them, or it might be due to factors we're not even aware of. Whatever the reason, when we build mathematical models that acknowledge or include some randomness, uncertainty, or unpredictability in the *dependent* (output) variables, given known *independent* (input) variable values, we call them *statistical models*.

Statistical Inference

Statistical models are usually constructed by *inference*. There's not a formal definition of the statistical inference process, but we can think of it generally as:

1. Collect, explore, and analyze data.
2. Quantify relationships in the data.
3. Test the model elements – i.e. the relationships quantified in step 2.
4. If the relationships aren't sufficiently supported by the test results, use those results to revise assumptions and search for other relationships.
5. Repeat steps 1-4 as appropriate.

This might sound a bit like what you've learned as the scientific method. In fact, statistical inference is often an important part of scientific inquiry.

Value of Statistical Models

Can a statistical model still be useful, or does the inherent uncertainty make such a model too inaccurate? That's not an easy question to answer. In part, it depends on how well the model fits the data. So we not only need to quantify the relationships – we also need to quantify how well we can quantify those relationships!

Ultimately, however, the answer boils down to our needs. A model that leaves a majority of the variation in the dependent variable unexplained may be sufficient for some purposes, while others may dictate that only a model that explains almost all of that change is acceptable.

Types of Statistical Models

Just as the set of even the most commonly used types of mathematical models is far too broad to show in "Patterns of Change", there's a wide variety of commonly used statistical model types, and we can only hope to touch on a few here. For example, in a *polynomial* model, the dependent variable is expressed as a polynomial function of one or more independent variables (a simple form of the polynomial model is the *linear* model, where there are no terms with degree higher than 1); in a *cyclical* time series model, the dependent variable cycles in value over time (the independent variable), either in a relatively smooth sinusoidal fashion, or as the sum of multiple sinusoidal cycles of different lengths; in an *autoregressive* time series model, the value of the dependent variable at one moment in time is used as an input value at a later moment, following a lag period. (Of course, there are also hybrid models that use combinations of these and other types.)

Discussion

1. Besides the cricket chirps scenario, which of the "Patterns of Change" scenarios (if any) could be described by statistical models?
2. Assume that we're developing statistical models for each of the following groups of variables. What are the independent and dependent variables in each? What type (or combination of types) of statistical model, from those mentioned briefly above, might best fit real data in each case?
    a) Daily low temperature in Socorro, New Mexico vs. day of the year.
    b) Populations of rabbits in the wild in New Mexico, measured or estimated monthly, based on the previous month's rabbit and coyote populations.
    c) Total monthly precipitation, measured at Denver International Airport, vs. month of the year.
    d) Annual number of hurricanes in the Atlantic ocean over the past 75 years.
    e) End-of-day value of the Dow Jones Industrial Average (a set of representative stocks listed on the New York Stock Exchange) over the past 50 years, taking into account long-term trends, seasonal variations throughout the year, and the fact that the ending value on one day strongly influences the early prices for those stocks in the following day.

Linear Models

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:[2]

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \varepsilon, \tag{4}$$

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 (4) becomes

$$Y = \beta_0 + \beta_1 X + \varepsilon. \tag{5}$$

This is often written as

$$Y = \alpha + \beta X + \varepsilon. \tag{6}$$

As you've probably figured out already, a 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

How do we find the straight line that best fits the data? To illustrate the problem, let's look at actual data for cricket chirps and temperature, collected by Dr. Peggy LeMone [3]. We'll begin by expanding graph F (page 5), adding some details on the scale and units of measure (Figure 1, data from Appendix A).

2 In this usage, *linear model* is synonymous with *linear regression model*. In other contexts, the same term can refer to other model types – e.g. the *general linear model*, which allows for multiple dependent variables, linear combinations of functions (possibly non-linear) of the independent variables, and categorical values.

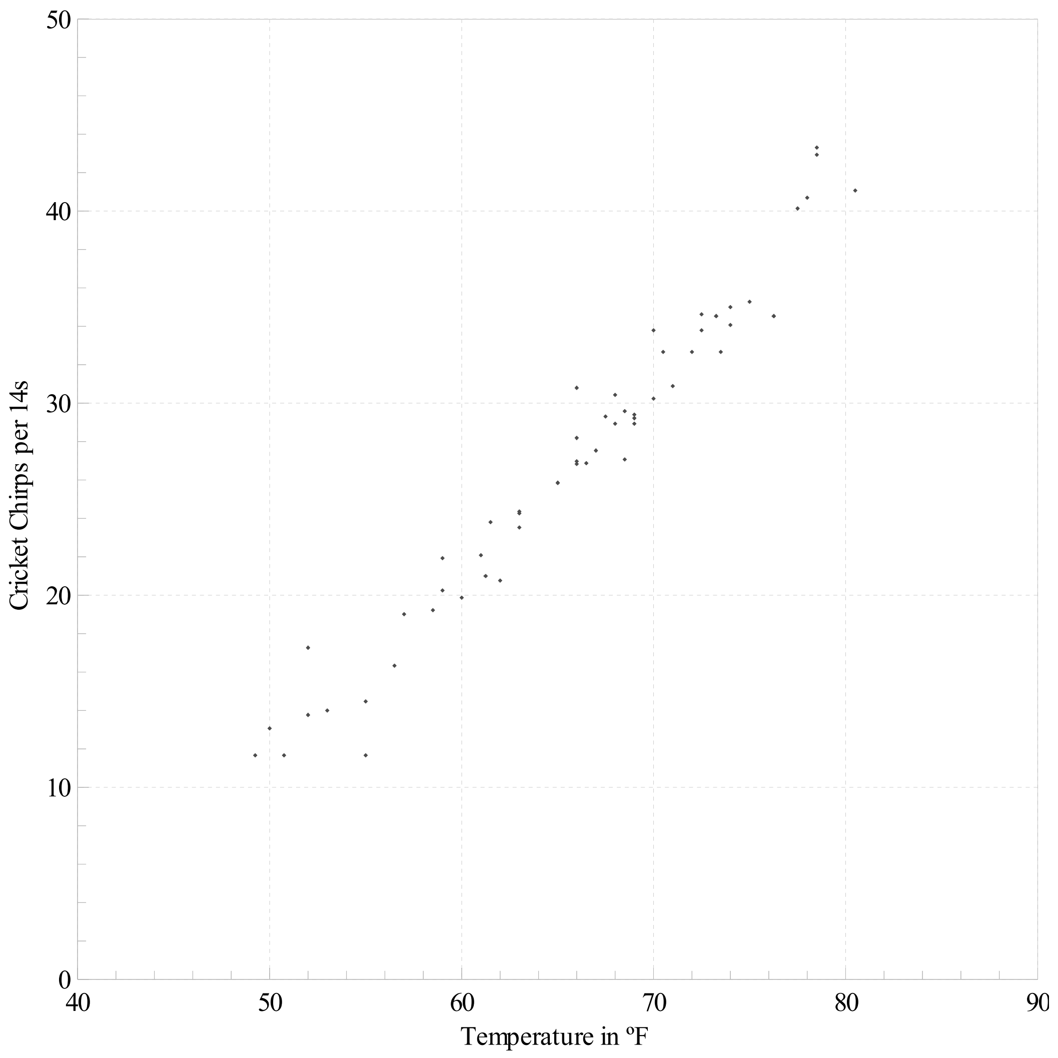


Figure 1: Cricket Chirps vs. Temperature

Arguably, the most important question to ask at this point is whether a linear model makes sense for the relationship between temperature and cricket chirp rate. Even if the answer is yes, we shouldn't assume that any apparent relationship will hold under conditions far outside the bounds of the observed data. For instance, it seems quite likely that very high or very low temperatures would have a disastrous effect on the crickets themselves – and consequently on their chirps.

A linear model seems appropriate in this case, for the range of temperatures observed. But two different people, fitting a straight line to the data by sight, will probably draw slightly different lines. How can we measure how well each line fits, so that we can select the best fitting one?

It might seem reasonable to evaluate the fit by adding up the differences between the fitted *Y* values (denoted by $\hat{y}_i$) on the line and the actual *Y* values (these differences are called *residuals* or *errors*) for each line under consideration, and use the line with the sum closest to zero. However, for any non-vertical straight line that passes through $(\bar{x}, \bar{y})$, the positive and negative residuals cancel each other out, and the sum is zero. (For a proof of this, see Appendix B.)

Linear Least-Squares Regression

If we want to use the residuals to assess the quality of fit, we need to keep the negative values from canceling out the positive values. Fortunately, we can avoid the problem if we treat the residuals as positive values; to do that, we can take the absolute values or the squared values of the residuals. Both approaches are used, but taking the squared values makes an analytical solution easier. The method of finding the best fitting line by minimizing the sum of the squared residuals is called *linear* (or *ordinary*) *least-squares regression*.[3]

If we denote our model estimates for α and β by a and b, respectively,[4] and the fitted line by

$$\hat{Y} = a + bX, \tag{7}$$

then the sum of the squared residuals (*sum of squared errors*, or SSE) for the model is given by

$$\text{SSE} = \sum_{i=1}^{n} \left(y_i - \hat{y} \right)^2. \tag{8}$$

We could calculate SSE for multiple lines, and choose the line with the lowest value as the best fit among them. However, instead of comparing specific alternatives lines, we can use calculus to find the values of a and b that give the smallest possible SSE. Even better, we can derive general formulas for a and b that can be used with any appropriate data set. This is the aim of least-squares regression.

Minimizing SSE

We know that if a differentiable function of a single variable has a finite minimum or maximum value, it occurs at a *stationary point* – i.e. a point where the first derivative of the function is equal to zero. For example, the minimum y value of an upward-opening parabola occurs at its vertex, where the slope is zero.

The same rule applies to differentiable functions of N variables, as well: if a finite minimum or maximum value of the function exists, it must be located at a point where all N of the *partial derivatives* of the function are equal to zero.[5] Therefore, to find a and b that give the minimum value of SSE, we need to find the combination where the partial derivatives of SSE with respect to a and b are equal to zero.[6] The first step is thus to compute these partial derivatives.

3 The linear least-squares method is most useful when the residuals aren't correlated with each other or with the independent variable, and when they follow a normal distribution. However, even when these conditions are known to be not strictly satisfied, this method is still often used, at least as an exploratory tool.

4 The fact that b is typically used for the estimated slope in the linear regression model can cause some confusion, since the standard slope-intercept form of the equation of a line is $y = mx + b$, where b is the intercept.

5 In a function of two or more variables, the partial derivative of the function with respect to one of the variables is obtained by taking the derivative with respect to that variable, while treating all other variables as constants.

6 Just as we can use the second derivative of a function of a single variable to determine whether a stationary point is a minimum, maximum, or inflection point, we can use the matrix of second partial derivatives (called the *Hessian* matrix) to do this with a function of multiple variables. Though it's outside the scope of this document, it can be shown from its Hessian that the sole stationary point of SSE is in fact a minimum, for any data set with at least two distinct values of the independent variable.

$$\begin{aligned}
\frac{\partial(\mathrm{SSE})}{\partial a} &= \sum_{i=1}^{n}\left[-2\left(y_i - a - b\,x_i\right)\right] \\
&= -2\left(\sum_{i=1}^{n} y_i - a\,n - b\sum_{i=1}^{n} x_i\right) \\
\frac{\partial(\mathrm{SSE})}{\partial b} &= \sum_{i=1}^{n}\left[-2\,x_i\left(y_i - a - b\,x_i\right)\right] \\
&= -2\left(\sum_{i=1}^{n} x_i\,y_i - a\sum_{i=1}^{n} x_i - b\sum_{i=1}^{n} x_i^2\right)
\end{aligned} \tag{9}$$

By setting the value of both partial derivatives to 0, we get 2 equations in 2 unknowns. (The *x* and *y* values aren't unknowns in this case; they're assumed to come from actual data.) Since all of our summations are over the same range, we'll leave out the limits and indices from here on.

$$\begin{aligned}
0 &= -2\left(\sum y - a\,n - b\sum x\right) \\
0 &= -2\left(\sum x\,y - a\sum x - b\sum x^2\right)
\end{aligned}$$

$$\begin{aligned}
a\,n \quad + b\sum x &= \sum y \\
a\sum x + b\sum x^2 &= \sum x\,y
\end{aligned} \tag{10}$$

We can now solve the equations in (10) simultaneously to find *a* and *b*.

$$\begin{array}{l}
a\,n\sum x^2 + \qquad b\sum x\sum x^2 = \sum x^2\sum y \\
a\left(\sum x\right)^2 + \qquad b\sum x\sum x^2 = \sum x\sum x\,y \\
\hline
a\left[n\sum x^2 - \left(\sum x\right)^2\right] \qquad = \sum x^2\,y - \sum x\sum x\,y
\end{array}$$

$$\begin{array}{l}
a\,n\sum x + b\left(\sum x\right)^2 \qquad = \sum x\sum y \\
a\,n\sum x + b\,n\sum x^2 \qquad = n\sum x\,y \\
\hline
\qquad b\left[n\sum x^2 - \left(\sum x\right)^2\right] = n\sum x\,y - \sum x\sum y
\end{array}$$

$$\begin{aligned}
a &= \frac{\sum x^2\sum y - \sum x\sum x\,y}{n\sum x^2 - \left(\sum x\right)^2} \\
b &= \frac{n\sum x\,y - \sum x\sum y}{n\sum x^2 - \left(\sum x\right)^2}
\end{aligned} \tag{11}$$

A Simple Example

Let's use (11) to find *a* and *b* for a simple data set. Given the *x* and *y* values provided in Table 1, we begin by calculating and filling in the values in the *xy* and x^2 columns. Then, we sum the values in each column, and write the totals in the bottom row.

x	***y***	***xy***	x^2
0	-0.5	0	0
1	0	0	1
2	1.5	3	4
3	2	6	9
$\sum x = 6$	$\sum y = 3$	$\sum xy = 9$	$\sum x^2 = 14$

Table 1: Data and calculations for simple regression example

Finally, we can substitute the values from the bottom row of Table 1, along with the number of data points ($n = 4$), in place of the corresponding sums in (11).

$$a = \frac{\sum x^2 \sum y - \sum x \sum xy}{n\sum x^2 - \left(\sum x\right)^2}$$

$$= \frac{14\cdot 3 - 6\cdot 9}{4\cdot 14 - 6^2} = -\frac{12}{20} = -\frac{3}{5}$$

$$b = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - \left(\sum x\right)^2}$$

$$= \frac{4\cdot 9 - 6\cdot 3}{4\cdot 14 - 6^2} = \frac{18}{20} = \frac{9}{10}$$

We now have our fitted line, found by linear least-squares regression.

$$\hat{Y} = -\frac{3}{5} + \frac{9}{10}X \tag{12}$$

To visualize the fit, we can plot the data points along with the fitted regression line (Figure 2). It's easy to see that the fit is pretty good – and because we used least-squares regression, we know that this fitted line minimizes SSE. But we're still not sure how to quantify how good the fit is.

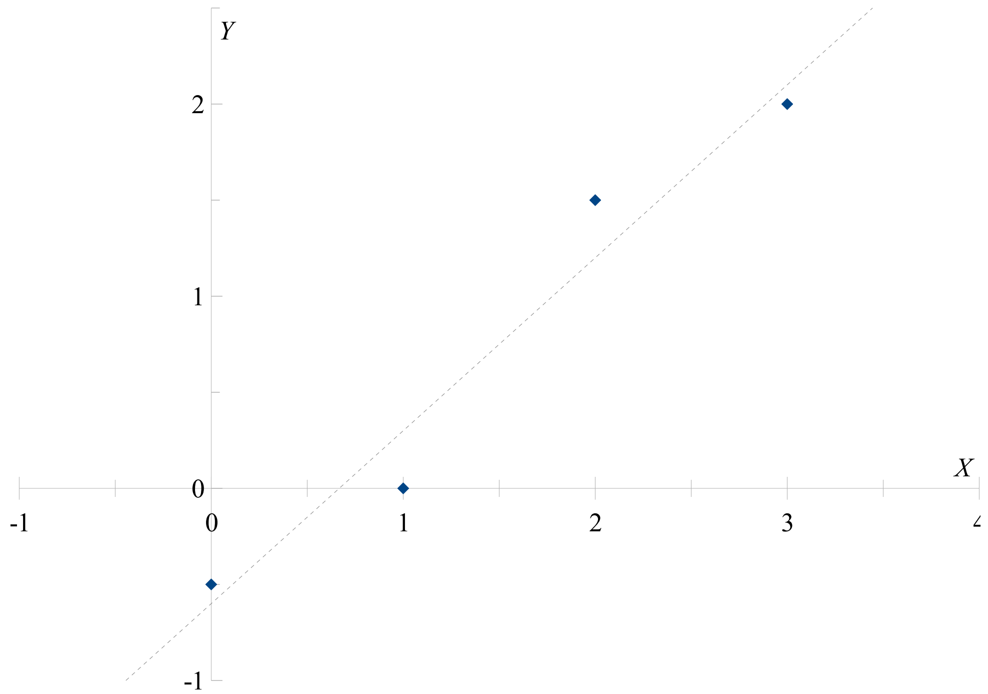


Figure 2: Simple example data points and fitted regression line

The Coefficient of Determination

To assess the quality of a model, or to compare alternative models, we usually need some way to measure how well a model fits the data – i.e. to measure its *goodness-of-fit*. (This isn't the only type of measure of interest when evaluating a statistical model, but it's one of the most important.)

One way to measure goodness-of-fit is to measure how much of the change in the dependent variable is accounted for by the model. First, we need to quantify the total change in the dependent variable; we can do this by summing the squared deviations of its observed values from its sample mean. We call this the *total sum of squares*, or SST.

$$\begin{aligned} \text{SST} &= \sum (y - \bar{y})^2 \\ &= \sum y^2 - \frac{\left(\sum y\right)^2}{n} \end{aligned} \tag{13}$$

SST can also be expressed as the sum of SSE and the *sum of squares of regression* (SSR).[7]

$$\text{SST} = \text{SSE} + \text{SSR}, \tag{14}$$

7 Unfortunately, while these abbreviations are used in many textbooks, some others define SSE and SSR with meanings opposite to these. In those texts, SSR is the *sum of squared residuals* (which we're calling the sum of squared errors), while SSE is the *sum of squares explained* (i.e. the regression sum of squares). Adding more confusion, still other texts use ESS for *explained sum of squares*, RSS for *residual sum of squares*, and TSS for *total sum of squares*. The best way to avoid confusion is to define these terms explicitly when using them.

where SSE is given by (8), and

$$SSR = \sum \left(\hat{y} - \bar{y}\right)^2. \tag{15}$$

The larger that SSR is in relation to SST, the more that the change in the dependent variable is explained by the model. This leads us to a useful goodness-of-fit measure: the *coefficient of determination*, or R^2.

$$R^2 = \frac{\text{SSR}}{\text{SST}} \tag{16}$$

We can interpret R^2 as the fraction of the variation in the dependent variable that's determined or explained by the model.[8]

Simple Example Revisited: Computing the Coefficient of Determination

Let's calculate R^2 for the simple data set in Table 1. To begin, let's add a few more columns to the table, for y^2, $\hat{y}$, and $\left(y - \hat{y}\right)^2$. Fill in those columns using the original data and the fitted line equation given by (12); then, compute the sums for y^2 and $\left(y - \hat{y}\right)^2$ (Table 2).

x	y	xy	x^2	y^2	$\hat{y}$	$\left(y - \hat{y}\right)^2$
0	-0.5	0	0	0.25	-0.6	0.01
1	0	0	1	0	0.3	0.09
2	1.5	3	4	2.25	1.2	0.09
3	2	6	9	4	2.1	0.01
$\sum x = 6$	$\sum y = 3$	$\sum xy = 9$	$\sum x^2 = 14$	$\sum y^2 = 6.5$		$\sum \left(y - \hat{y}\right)^2 = 0.2$

Table 2: Calculations for coefficient of determination in simple regression example

Finally, we can use (8), (13), (14), and (16) to find R^2.

$$\begin{aligned}
\text{SSE} &= \sum \left(y - \hat{y}\right)^2 \\
&= 0.2 \\
\text{SST} &= \sum y^2 - \frac{\left(\sum y\right)^2}{n} \\
&= 6.5 - \frac{3^2}{4} = 4.25 \\
\text{SSR} &= \text{SST} - \text{SSE} \\
&= 4.25 - 0.2 = 4.05
\end{aligned}$$

8 While R^2 is useful, it's often misused. It's easy to fall into the trap of adding more independent variables or polynomial terms to a model to increase R^2, at the expense of a loss of general predictive power. For this reason, an *adjusted* R^2, which discounts the increase from additional terms, is often used when building multivariate or polynomial models.

$$\begin{aligned} R^2 &= \frac{\text{SSR}}{\text{SST}} \\ &= \frac{4.05}{4.25} \approx 0.95 \end{aligned}$$

From this, we conclude that approximately 95% of the total change in the dependent variable is determined or explained by the independent variable.

Discussion

1. Are there any other curves besides a straight line that fit the data in Table 1?
2. Should we consider an R^2 value of 0.95 to be high enough for a usable model?
3. Do you think a non-linear model could have a higher R^2 value?
4. What other factors, besides a difference in R^2, might lead us to prefer a linear model over a non-linear model, or vice versa?

Implementation of Simple Least-Squares Linear Regression

General Features and Components

To tackle the cricket chirp problem described in “Patterns of Change” scenario 9, and to see how the core techniques of linear regression can be implemented, we'll use code written in two different programming languages: Python and Java. The programs provided are functional but incomplete: they'll compile and run as-is, but to perform the simple linear least-squares regression analysis in each will require some additions 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. To that end, the Python and Java programs use the Matplotlib and JFreeChart open source libraries (respectively) to display the input data and the regression results [4], [5]. The capabilities and *application programing interfaces* (*APIs*) of the two libraries are extensive, but very different from each other; because of that, we'll only touch briefly on the code used to plot the data and the regression results.

Apart from the fundamental differences between the two languages, and the differences in the plotting libraries, the overall structure of the two implementations is nearly identical by design.

Development Tools

The provided Java code is packaged as a DrJava project, while the Python code is packaged as a PyScripter project [6], [7]. However, the programs can be edited and run used with virtually any Python v2.5+ or Java v5+ development and runtime environments, provided that Matplotlib or JFreeChart is installed.

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 (11).

$$
\begin{aligned}
b &= \frac{\sum (x - \bar{x})(y - \bar{y})}{\sum (x - \bar{x})^2} \\
a &= \bar{y} - b\bar{x}
\end{aligned}
\tag{17}
$$

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 (17), 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 (11) 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, it's used in the programs we'll look at now.

Preparation

Python

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

Java

Locate and open the `LinearRegressionJava` project, and open the files `Chirps.java` and `SimpleLinear.java`, both in the `org.nm.challenge.kickoff.statmodel` package (i.e. the `org/nm/challenge/kickoff/statmodel` directory within the project). `Chirps.java` contains the Java program that you'll run to execute the linear regression, while `SimpleLinear.java` contains the code that performs the mathematical calculations for the regression.

Concepts and Comparisons

Intrinsic data structures

Both Java and Python have certain types of data that are *intrinsic* – that is, they're recognized and supported natively by the compilers/interpreters of the languages. Some of these types are *scalar*, holding a single value, and some hold sequences of multiple values. In Java, the intrinsic type that holds multiple values is an *array*; it can be thought of as a linear, contiguous sequence of data, with a fixed number of elements.[9] 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.[10] 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. On the other hand, the fact that Java arrays are stored contiguously sometimes allows the Java virtual machine to optimize performance dynamically, in ways that are difficult for Python to match.

These differences in structures and supporting operations contribute to fundamental differences between the two languages in the approaches to many problems.

Line and statement structure

In Java, a statement may be a *simple statement*, or a *statement block* (below). A simple statement always ends with a semicolon; even if the statement extends over several lines, it doesn't end until a semicolon is encountered. Line breaks, tabs, and space characters are all treated as whitespace by the Java compiler. Among other things, this allows a great deal of flexibility (which can be misused) in formatting code.

In Python, a line break ends a statement, unless a line continuation character (the backslash) is the last non-whitespace character before a line break, or a line break occurs inside a set of parentheses or brackets. 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; this is usually done only when including multiple statements on the same line.)

In Java, a *statement block* can be used wherever a simple statement is allowed. A statement block consists of a zero or more statements, enclosed within a pair of curly braces. So when the formal syntax of flow control statements like `if`, `for`, and `while` (see below) allows for conditional or iterated execution of a statement, that statement can actually be a statement block. Note that the reverse isn't necessarily the case: in some contexts, a statement block must be used; a simple statement doesn't suffice in these cases. For example, the definition of a Java *method* includes a method *body* (the code that implements the method), which must be a statement block.

9 Java also has a `String` type, which is defined in the standard library. Some of the basic `String` operations are handled as special cases by the compiler, so it is treated in some contexts as an intrinsic type.

10 Python also has the intrinsic `buffer` and `xrange` types, which are special-purpose sequences.

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. (Unlike statement blocks in Java, suites can't be used arbitrarily in place of simple statements; they can only be used with a controlling clause.) 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.

We'll see examples of these differences when we look at our Java and Python programs.

Conditional execution of statements

Like virtually all programming languages, Java and Python include statements for testing a condition, then following one path of execution if the condition is true, and either simply skipping that path or following another path if the condition is false. The syntax is very similar for the two languages, with some basic differences due to the different statement structures, and some additional differences in the details:

- In Python, any number of **`elif`** (*else-if*) clauses may follow the **`if`** clause (and accompanying suite). In Java, there's no special *else-if* construct: **`else if`** is simply the **`else`** of the preceding **`if`** statement, followed by another **`if`**. However, in most cases, these two syntaxes are roughly equivalent.

- In Python, **`if`**, **`else`**, and **`elif`** are all clauses; each is followed by a statement suite. In Java, **`if`** and **`else`** are followed by statements, which may be simple statements or statement blocks.

- In Java, the condition to be tested must be enclosed in parentheses; in Python, such parentheses are optional.

Before we look at conditionals in our implementations, let's look at a few simple code fragments in Java and Python (Listing 1, Listing 2). Note the difference in comment syntax between the two languages; both languages also support a special format for documentation comments, but that's not shown here. Also, note the different conventions used for multi-word identifiers (variable or function names). Finally, note that ***`doSomething()`***, ***`doSomethingElse()`***, 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.

```java
// Execute a statement block if the value of some variable x is less than or
// equal to the value of another variable y; otherwise, skip over the
// statement block.
if (x <= y) {
    doSomething();
}

// Execute a statement block if the value of some variable x is less than or
// equal to the value of another variable y; otherwise, execute a second
// statement block.
if (x <= y) {
    doSomething();
}
else {
    doSomethingElse();
}

// Execute a statement block if the value of some variable x is less than or
// equal to the value of another variable y; otherwise, execute a second
// statement block if x is greater than z; otherwise, execute a third block.
if (x <= y) {
    doSomething();
}
else if (x > z) {
    doAnotherThing();
}
else {
    doSomethingElse();
}
```

Listing 1: Conditionals in Java

```python
# 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 2: Conditionals in Python

Explicit and implicit iteration

Java has two mechanisms for iterating over a range of values, over the elements of an array or sequence, or over the members of an unordered collection: the **for** statement and the **while** statement. With a few different variations, these repeat a statement (which may be a statement block) as long as a specified condition is true. The repeated statement may be simple statement, or (as in Listing 3) a statement block.

```
// Repeat a statement block as long as i is less than the number of items in
// someArray, incrementing i after each iteration.
for (int i = 0; i < len(someArray); i++) {
    doSomething();
}

// Repeat a statement block for each element of someIntArray (assumes
// someIntArray is an array of int). In each iteration, j assumes the value
// of the current element.
for (int j : someIntArray) {
    doSomething();
}

// Repeat a statement block as long as an already declared variable k is less
// than limit.
while (k < limit) {
    doSomething();
}

// Repeat a statement block as long as an already declared variable k is less
// than limit, testing the condition at the end of the loop (thus, the loop
// is executed at least once).
do {
    doSomething();
}
while (k < limit);
```

Listing 3: Iteration in Java

(It's a fairly common practice, in Java and other C-lineage languages, to use a statement block even when just one simple statement is included in the block. Among other benefits, this reduces potential logical errors that often occur unintentionally when adding statements to a simple statement that's executed conditionally or iteratively.)

Python has **for** and **while** statements for explicit iteration as well. However, it also has implicit iteration statements that are used for applying an operation to all the items in a sequence, and returning a new sequence. The most direct of these is *list comprehension*, which uses the elements of one sequence to construct a new list.

See Listing 4 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) as long as i is less than or equal to the number of items in
    # someArray, 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 4: Explicit and implicit iteration in Python

References to members of an object

Both Java and Python support the definition of *classes* for *object-oriented programming*. A class is simply a mechanism for packaging data with the behaviors that act on that data, and 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 modeled or embodied in the program. A presentation of even just the key concepts of classes and object-oriented programming is beyond the scope of this document, so we'll focus on just those aspects that are most relevant to our implementations.

Java was designed from the start as an object-oriented language, and all Java programs consist of one or more class definitions. Python doesn't insist on this approach; this often allows for simpler code in cases where classes aren't needed – but when an object-oriented approach is employed, Python code in class definitions can be more verbose than the equivalent Java code. This is particularly evident in the way that methods (functions that are part of a class definition) refer to an *instance* of that class (an object based on the class definition), and to data and other methods of such an instance.

In Java, **`this`** refers to the current object instance, but it's usually optional; the Java compiler generally recognizes when the code in a method is referring to data and other methods of the same object. In Python, **`self`** refers to the current object instance, but its use is not optional: 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.[11]

11 Both Python and Java also support the definition and use of *static* methods and data, which are associated with the class as a whole, not with instances of a class; **`this`** and **`self`** aren't used in static methods.

Running the Initial Programs

Python program (`chirps.py`)

Before we make any additions to the code, let's first see what it does already. Let's begin by running the `chirps.py` script in your Python development environment. In PyScripter, we can do this by opening the `chirps.py` script from the **Project Explorer**; then selecting the **Run/Run** menu command, or typing *Ctrl-F9*, or pressing the **Run** button (with the green triangular icon) on the button bar near the top of the window. (Because `chirps.py` is set up as the main script in the PyScripter project, we can also run it by right-clicking on the **Default** run configuration in the **Project Explorer**, and selecting **Run** from the context menu.)

The first time you run the program, it may take several seconds (even a minute or longer) to interpret the code in the three files of the project. But you should eventually see a window displaying the cricket chirps vs. temperature data (Figure 3).

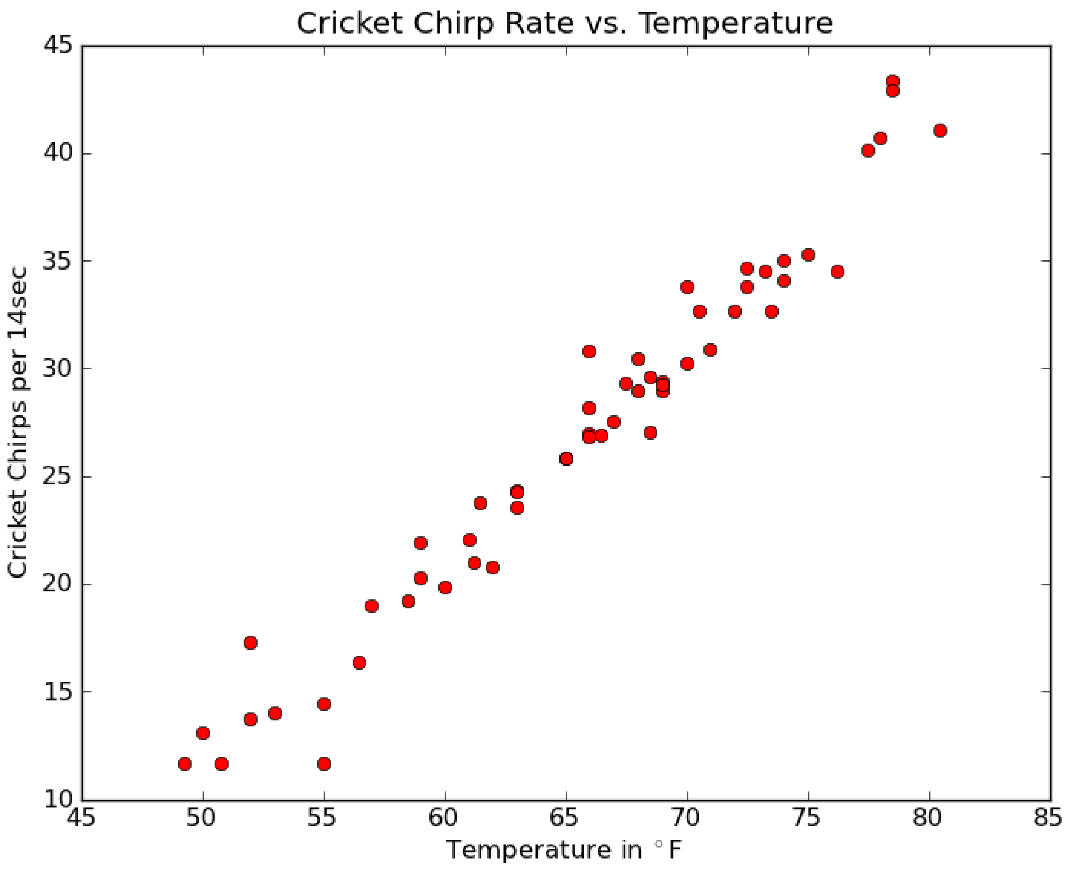


Figure 3: Initial Python program display

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

1. `__main__` script – `chirps.py`

   This is the main program script, which calls functions in `chirps.py` and methods (directly and indirectly) in the other files.

   (a) `load` function – `chirps.py`

      i. `TableFileParser` 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` 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 $\sum x$, $\sum y$, $\sum x\,y$, $\sum x^2$, 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 calls various methods in the Matplotlib library, which render the scatterplot. Notice that this function has an `if` clause; that the suite for that clause (which displays the regression line and equation) will only be executed if `model.r2 >= 0`; and that `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? Was the suite for this `if` statement executed when you first ran the program?

Outline 1: Python program structure

Java program (`Chirps.java`)

Now let's run the Java program. Java programs require a separate compile step before running, but many development environments do that automatically every time a file is saved. DrJava will compile automatically if the source files haven't already been compiled, but it doesn't always detect when a recompile is needed. So we'll try to make a habit of compiling manually whenever we save some changes.

Compile the project and run the Chirps class. In DrJava, do this by first selecting the **Project/Compile Project** menu option, or by clicking the **Compile Project** button near the top of the window; then, select the **Project/Run Main Class of Project** menu option, or click the **Run Project** button. Each of these steps will probably take a few seconds, after which you'll see another scatterplot display (Figure 4).

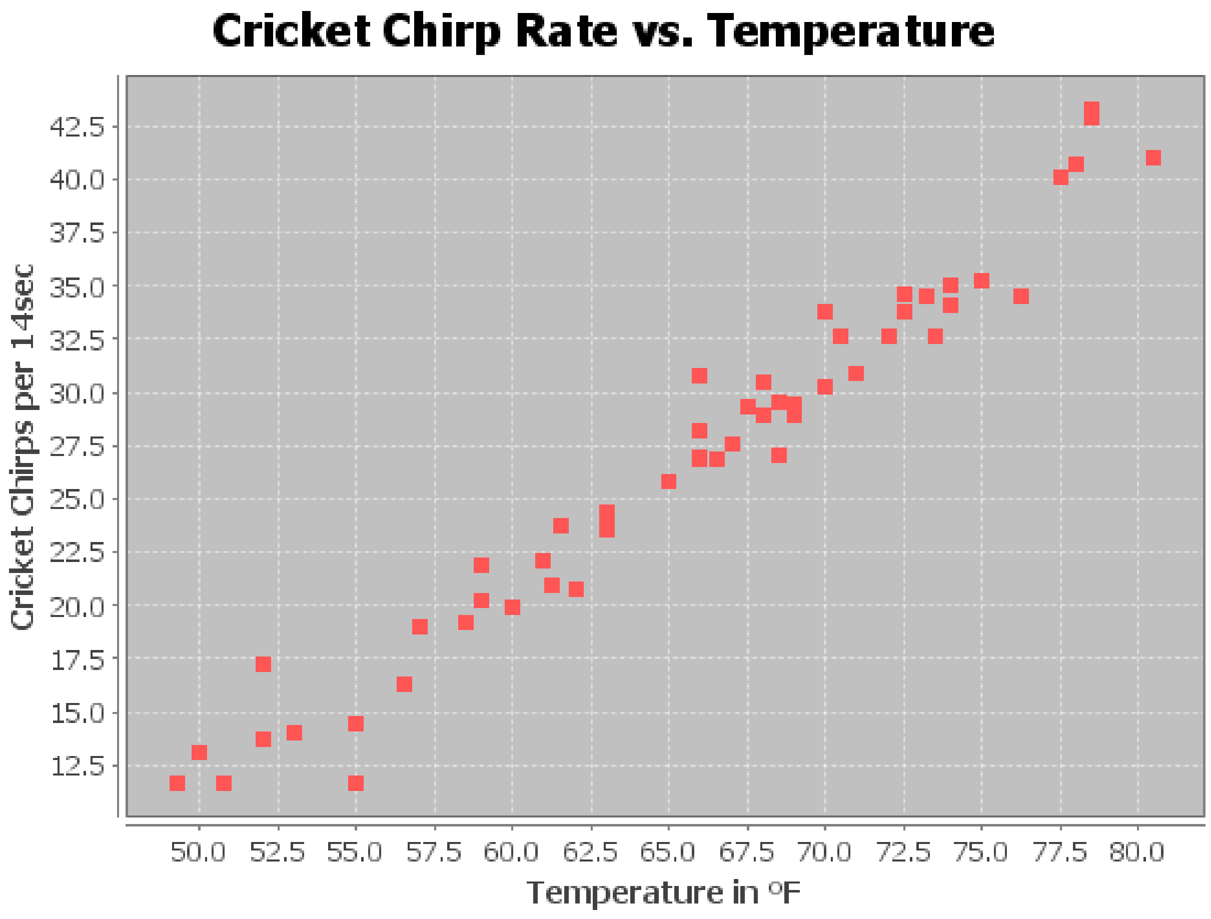


Figure 4: Initial Java program display

Again, let's examine the high-level structure of the program, and the corresponding sections of the code. This time, try to notice those aspects which are significantly different from the analogous portions of the Python program.

1. `main` method – `Chirps.jav`

   This is the main program method, which calls other methods in `Chirps.jav` and (directly and indirectly) in the other files.

   (a) `load` method – `Chirps.jav`

      i. `TableFileParser` constructor – `TableFileParser.jav`

         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.toDoubleArray` method – `TableFileParser.jav`

         Returns the previously read data values as double-precision numbers.

   (b) `SimpleLinear` constructor – `SimpleLinear.java`

      Copies the provided data to `this.x` and `this.y`, and stores the number of data points in `this.n`, in preparation for linear regression.

   (c) `SimpleLinear.regress` method – `SimpleLinear.java`

      i. `SimpleLinear.computeSums` method – `SimpleLinear.java`

         Currently, this method doesn't do anything (notice the empty statement block) Shortly, we'll add the code to calculate $\sum x$, $\sum y$, $\sum x\,y$, $\sum x^2$, and $\sum y^2$, and store those values in `this.sumX`, `this.sumY`, `this.sumXY`, `this.sumX2`, and `this.sumY2`, respectively.

      ii. `SimpleLinear.estimateParameters` method – `SimpleLinear.java`

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

      iii. `SimpleLinear.measureFit` method – `SimpleLinear.java`

         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 `this.r2`.

   (d) `plot` function – `Chirps.jav`

      This function calls various methods in the jFreeChart library, which render the scatterplot. Notice that this function has an `if` statement; based on what you've already seen in the Python program, and what you now see in the Java program, was the associated statement block executed when you first ran the program?

Outline 2: Python program structure

Computing the Sums

Python (`simple_linear.py`)

References

[1] Maryland Collaborative for Teacher Preparation. “Unit III: Things Change (Part A)”. Internet: http://www.towson.edu/csme/mctp/Courses/Mathematics/UnitIIIA.html [Sep. 21, 2011].

[2] “Cricket Chirps: Nature's Thermometer”. Internet: http://www.almanac.com/cricket-chirps-temperature-thermometer [Oct. 12, 2011].

[3] M. A. LeMone. “Measuring temperature using crickets”. Internet: http://blog.globe.gov/sciblog/2007/10/05/measuring-temperature-using-crickets/, Oct. 5, 2007 [Oct. 12, 2011].

[4] John Hunter, Darren Dale, Michael Droettboom. Matplotlib v1.1.0. Internet: http://matplotlib.sourceforge.net, Oct. 11, 2011 [Oct. 12, 2011].

[5] Object Refinery Limited. JFreeChart v1.0.13. Internet: http://www.jfree.org/jfreechart, Sep. 28, 2011 [Oct. 12, 2011].

[6] JavaPLT Group, Rice University. DrJava 20100913-r5387. Internet: http://www.drjava.org, Sep. 13, 2010 [Nov. 3, 2011].

[7] PyScripter Internet Group. PyScripter v2.4.3. Internet: http://code.google.com/p/pyscripter/, Sep. 20, 2011 [Nov. 3, 2011].

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 [3]. According to Dr. LeMone, the measurements were originally in chirps per 30 seconds (averaged over multiple successive observations, and halved for chirps per 15 seconds) and degrees Fahrenheit (taking the average reading from multiple thermometers); the column for chirps per 14 seconds was derived from the original data.

Date	Time	Chirps/15s	Chirps/14s	Temp (ºF)
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

Date	Time	Chirps/15s	Chirps/14s	Temp (ºF)
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
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

Appendix B: Proof that the Sum of Residuals is Zero

For any straight line that passes through the point $(\bar{x}, \bar{y})$, the sum of errors (residuals) is zero. We can prove this using $\hat{y} = \bar{y} + m(x - \bar{x})$ as a general equation for any such line.

Given

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

$$\hat{y} = \bar{y} + m(x - \bar{x})$$

Then

$$\begin{aligned}
\text{SE} &= \text{sum of errors (residuals)} \\
&= \sum_{i=1}^{n} (y_i - \hat{y}_i) \\
&= \sum_{i=1}^{n} \left[y_i - \bar{y} - m(x_i - \bar{x}) \right] \\
&= \sum_{i=1}^{n} (y_i - \bar{y}) - m \sum_{i=1}^{n} (x_i - \bar{x}) \\
&= \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} \bar{y} - m \left(\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \bar{x} \right) \\
&= n\bar{y} - n\bar{y} - m(n\bar{x} - n\bar{x}) \\
&= 0
\end{aligned}$$

Appendix C: Python Implementation of Simple Linear Regression

```python
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[1:])
    model = SimpleLinear(x, y)
    model.regress()
    plot(model)
```

Listing 5: `chirps.py`

```python
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):
        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 6: `simple_linear.py`

```python
from __future__ import with_statement   # Not required in Python v2.6+

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 7: `parser_util.py`

Appendix D: Java Implementation of Simple Linear Regression

```java
package org.nm.challenge.kickoff.statmodel;

import java.awt.Font;
import java.awt.BasicStroke;
import java.awt.Color;
import java.io.FileNotFoundException;
import java.io.IOException;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.annotations.XYLineAnnotation;
import org.jfree.chart.annotations.XYTitleAnnotation;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.title.TextTitle;
import org.jfree.data.xy.DefaultXYDataset;
import org.jfree.ui.HorizontalAlignment;
import org.jfree.ui.RectangleAnchor;
import org.nm.challenge.kickoff.util.TableFileParser;

public class Chirps {

    public static final String DEFAULT_DATA_FILE = "chirps.csv";
    public static final String DEFAULT_DELIMITER = ",";
    public static final int DEFAULT_SKIP_LINES = 1;

    private static final String WINDOW_TITLE = "Simple Linear Regression";
    private static final String CHART_TITLE =
            "Cricket Chirp Rate vs. Temperature";
    private static final String X_AXIS_TITLE = "Temperature in \u00B0F";
    private static final String Y_AXIS_TITLE = "Cricket Chirps per 14sec";
    private static final String MODEL_SPEC =
            "Y\u0302 = %8.6f + %8.6fX\nR\u00B2 = %8.6f";

    public static void main(String[] args) {
        try {
            double[][] data = load(args);
            SimpleLinear model = new SimpleLinear(data[0], data[1]);
            model.regress();
            plot(model);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static double[][] load(String[] args)
            throws FileNotFoundException, IOException {
        double[][] observations;
        double[][] transposed;
        int n;
        String dataFile = DEFAULT_DATA_FILE;
        String delimiter = DEFAULT_DELIMITER;
        int skipLines = DEFAULT_SKIP_LINES;
        if (args.length > 0) {
            dataFile = args[0];
            if (args.length > 1) {
                delimiter = args[1];
                if (args.length > 2) {
                    skipLines = Integer.parseInt(args[2]);
                }
            }
        }
        observations = new TableFileParser(dataFile, delimiter, skipLines)
                .toDoubleArray();
        n = observations.length;
        transposed = new double[2][n];
        for (int i = 0; i < n; i++) {
            transposed[0][i] = observations[i][0];
            transposed[1][i] = observations[i][1];
        }
        return transposed;
    }

```

```java
    private static void plot(SimpleLinear model) {
        JFreeChart chart;
        DefaultXYDataset data = new DefaultXYDataset();
        double[] xValues = model.getX();
        double[] yValues = model.getY();
        data.addSeries("", new double[][] {xValues, yValues});
        chart = ChartFactory.createScatterPlot(CHART_TITLE,
                X_AXIS_TITLE, Y_AXIS_TITLE, data, PlotOrientation.VERTICAL,
                false, false, false);
        if (model.getR2() >= 0) {
            XYTitleAnnotation annotation;
            XYLineAnnotation line;
            XYPlot plot;
            String modelSpec = String.format(MODEL_SPEC,
                    model.getIntercept(), model.getSlope(), model.getR2());
            TextTitle modelTitle = new TextTitle(modelSpec,
                    new Font(Font.SERIF, Font.PLAIN, 12));
            double minX = Double.POSITIVE_INFINITY;
            double maxX = Double.NEGATIVE_INFINITY;
            double yMinX;
            double yMaxX;
            for (double x : xValues) {
                if (x < minX) {
                    minX = x;
                }
                if (x > maxX) {
                    maxX = x;
                }
            }
            plot = chart.getXYPlot();
            yMinX = model.getIntercept() + model.getSlope() * minX;
            yMaxX = model.getIntercept() + model.getSlope() * maxX;
            line = new XYLineAnnotation(minX, yMinX, maxX, yMaxX,
                    new BasicStroke(1f), Color.BLUE);
            modelTitle.setTextAlignment(HorizontalAlignment.LEFT);
            annotation = new XYTitleAnnotation(0.001, 0.999, modelTitle,
                    RectangleAnchor.TOP_LEFT);
            plot.addAnnotation(line);
            plot.addAnnotation(annotation);
        }
        ChartFrame frame = new ChartFrame(WINDOW_TITLE, chart);
        frame.pack();
        frame.setVisible(true);
    }

}
```

Listing 8: `Chirps.java`

```java
package org.nm.challenge.kickoff.statmodel;

public class SimpleLinear {

    private int n = 0;
    private double[] x;
    private double[] y;
    private double sumX = 0;
    private double sumY = 0;
    private double sumXY = 0;
    private double sumX2 = 0;
    private double sumY2 = 0;
    private double intercept = 0;
    private double slope = 0;
    private double r2 = -1;

    public SimpleLinear(double[] x, double[] y) {
        n = x.length;
        this.x = (double[]) x.clone();
        this.y = (double[]) y.clone();
    }

    public void regress() {
        computeSums();
        estimateParameters();
        measureFit();
    }

    public int getN() {
        return n;
    }

    public double[] getX() {
        return x;
    }

    public double[] getY() {
        return y;
    }

    public double getIntercept() {
        return intercept;
    }

    public double getSlope() {
        return slope;
    }

    public double getR2() {
        return r2;
    }

    private void computeSums() {
        for (int i = 0; i < n; i++) {
            sumX += x[i];
            sumY += y[i];
            sumXY += x[i] * y[i];
            sumX2 += x[i] * x[i];
            sumY2 += y[i] * y[i];
        }
    }

    private void estimateParameters() {
        intercept = (sumX2 * sumY - sumX * sumXY) / (n * sumX2 - sumX * sumX);
        slope = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX * sumX);
    }

    private void measureFit() {
        double sst = sumY2 - sumY * sumY / n;
        double sse = 0;
        for (int i = 0; i < n; i++) {
            double residual = y[i] - (intercept + slope * x[i]);
            sse += (residual * residual);
        }
        r2 = 1 - sse / sst;
    }

}
```

Listing 9: `SimpleLinear.java`

```java
package org.nm.challenge.kickoff.util;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import java.util.LinkedList;

public class TableFileParser {

    private String[][] table;

    public TableFileParser(String fileName, String delimiterRegex, int skipLines)
            throws FileNotFoundException, IOException {
        this(new File(fileName), delimiterRegex, skipLines);
    }

    public TableFileParser(File file, String delimiterRegex, int skipLines)
            throws FileNotFoundException, IOException {
        FileReader reader = null;
        BufferedReader buffer = null;
        LinkedList<String[]> work = new LinkedList<String[]>();
        String line;
        try {
            reader = new FileReader(file);
            buffer = new BufferedReader(reader);
            while (null != (line = buffer.readLine())) {
                if ((0 >= skipLines--) && (0 < line.trim().length())) {
                    String[] values = line.trim().split(delimiterRegex);
                    work.add(values);
                }
            }
        }
        finally {
            if (null != buffer) {
                buffer.close();
            }
            if (null != reader) {
                reader.close();
            }
            table = work.toArray(new String[0][]);
        }
    }

    public String[][] toStringArray() {
        String[][] result = new String[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = Arrays.copyOf(table[i], table[i].length);
        }
        return result;
    }

    public int[][] toIntArray() {
        int[][] result = new int[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = new int[table[i].length];
            for (int j = 0; j < table[i].length; j++) {
                result[i][j] = Integer.parseInt(table[i][j]);
            }
        }
        return result;
    }

    public double[][] toDoubleArray() {
        double[][] result = new double[table.length][];
        for (int i = 0; i < table.length; i++) {
            result[i] = new double[table[i].length];
            for (int j = 0; j < table[i].length; j++) {
                result[i][j] = Double.parseDouble(table[i][j]);
            }
        }
        return result;
    }

}
```

Listing 10: `TableFileParser.java`

Acknowledgements

Original text for “Things Change” is from course materials for “Introductory Mathematics at College Park” (Math 110), developed by the Maryland Collaborative for Teacher Preparation, funded by the National Science Foundation, Division of Undergraduate Education, Collaboratives for Excellence in Teacher Preparation (NSF Cooperative Agreement No. DUE 9814650) [1]. Used with permission.

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