CHAPTER

11 Curve Fitting

OUTLINE

Thus the unfacts, did we possess them, are too imprecisely few to warrant our certitude...

–*James Joyce,* Finnegan's Wake

CHAPTER POINTS

- Given independent and dependent variables, we can fit a linear model to the data using least squares regression.
- Least squares can handle multiple independent variables and nonlinearity in the independent variables.
- Logarithmic transforms allow us to fit exponential and power law models.

In the previous chapter we investigated methods that exactly interpolated a set of data. That is we derived functions that passed through a set of data points. Sometimes it is better to find a function that does not exactly interpolate the data points. For example, if we did an experiment and we expected the measured value to be linear in some variable x we might write

$$f(x) = a + bx + \epsilon,$$

DOI: 10.1016/B978-0-12-812253-2.00013-3

where $f(x)$ is the measured value, and ϵ is an error term because the experimental data probably does not fall in exactly a straight line. In this case the ϵ contains the measurement error and inherent variability in the system. In such a case, rather than performing linear interpolation, we want to find the values of a and b that best match the measured data, but do not necessarily exactly match the data.

Another reason we might want to find this best match is that there might be another hidden variable we do not know about, but we want to find the best possible model given the variables we do know about. As an example say we want to know the radiation dose rate outside of a shield, but there is an unknown source variability, we could write

$$\log(\text{dose}) = a + b(\text{shield thickness}) + \epsilon,$$

where now ϵ is capturing the error in not including the source variation, as well as the measurement uncertainty.

In a less scientific example, a retail corporation has stores throughout the country and wants to know how the area around the store (sometimes called a trade area) and the size of the store affect the sales at a store. In this case we may write

$$\text{sales} = a + b(\text{Population within 5 minute drive of store}) + c(\text{Size of Store}) + \epsilon.$$

In this case there may be hundred of hidden variables such as the presence of competition in the market, the age of the store, or the time of year.

The question that we are faced with is that we have set of input variables and the value of an output variable at several points, and we want to fit a linear model to this data. By linear model we mean a function that is the sum of contributions from each input or some operation on the inputs. The form of the model could come from theory or we could just be looking for an explanation of the output using the data we have. In this case Lagrange polynomials or cubic splines will not work because we have a model form we want to approximate, and interpolation methods either prescribe a model form or have the form determined by the amount of data. Furthermore, if we do an experiment and measure at the same point multiple times, we will likely get several slightly different values of the outputs. Interpolation methods will not work because they expect the interpolating function to only take on a single value at each input.

11.1 FITTING A SIMPLE LINE

To proceed, we will take some data and try to fit a linear model to the data. Along the way we will see that the problem is not well-posed, and demonstrate a means to give it a unique solution.

Say we are given data for an input x and an output y:

x	y
1	11.94688
2	22.30126
3	32.56929
4	41.65564

and we want to fit a model

$$y = a + bx,$$

which makes sense because y does look roughly linear in x. In the parlance of curve fitting, x is an independent variable, and y is the dependent variable.

What we can do is write this as a linear system of equations, where the unknowns are $\mathbf{u} = (a, b)^{\mathrm{T}}$. That is, we want to solve for $\mathbf{u}$ in the equation

$$\mathbf{X}\mathbf{u} = \mathbf{y},$$

where $\mathbf{X}$ is the data matrix,

$$\mathbf{X} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix},$$

and $\mathbf{y}$ is the vector of dependent variables

$$\mathbf{y} = (11.94688, 22.30126, 32.56929, 41.65564)^{\mathrm{T}}.$$

Notice that the data matrix has a column of ones because for each equation in the system the constant a is the same. Putting in our data values our system gives

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 11.94688 \\ 22.30126 \\ 32.56929 \\ 41.65564 \end{pmatrix}.$$

I think that the problem here is obvious: we have 4 equations and 2 unknowns. This means that there is not expected to be a vector $\mathbf{u}$ that can satisfy every equation. To make the problem well-posed, we can multiply both sides of the equation by the transpose of the matrix:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 11.94688 \\ 22.30126 \\ 32.56929 \\ 41.65564 \end{pmatrix}.$$

We will perform this calculation using Python, rather than by hand:

```
In [1]: import numpy as np
        A = np.array([(1,1),(1,2),(1,3),(1,4)])
        RHS = np.array([11.94688, 22.30126, 32.56929, 41.65564])
        print("The system A x = b has A =\n",A)
        print("And b =",RHS)

The system A x = b has A =
 [[1 1]
 [1 2]
 [1 3]
 [1 4]]
And b = [ 11.94688  22.30126  32.56929  41.65564]

In [2]: AT_times_A = np.dot(A.transpose(),A)
        AT_times_RHS = np.dot(A.transpose(),RHS)
        print("The system after multiplying by A transpose is A^T A =\n",AT_times_A)
        print("A^T b =",AT_times_RHS)

The system after multiplying by A transpose is A^T A =
 [[ 4 10]
 [10 30]]
A^T b = [ 108.47307  320.87983]
```

Now we have a two-by-two system that we can solve:

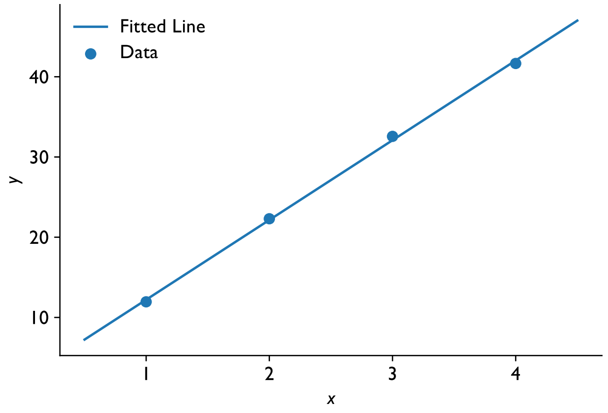
$$\begin{pmatrix} 4 & 10 \\ 10 & 30 \end{pmatrix} \mathbf{u} = \begin{pmatrix} 108.47307 \\ 320.87983 \end{pmatrix}.$$

This system is known as the "normal equations". We will solve this system with the Gaussian elimination code we wrote earlier, and that I have handily stored in GaussElim.py.

```
In [3]: from GaussElim import *
        ab = GaussElimPivotSolve(AT_times_A,AT_times_RHS)
        print("The constant a =",ab[0]," with a slope of b =",ab[1])

The constant a = 2.26969  with a slope of b = 9.939431
```

The data and the fitted line are shown next.



To get a measure of how this model fits the data we look at a quantity called R^2 (pronounced "R-squared"), defined as

$$R^2 = 1 - \frac{\sum_i (\hat{y}_i - y_i)^2}{\sum_i (\bar{y} - y_i)^2},$$

where $\hat{y}_i$ is the ith predicted data point by the model,

$$\hat{y}_i = \mathbf{x}_i \cdot \mathbf{u} = a + bx_i,$$

and $\bar{y}$ is the mean of the data. Notice that in the definition of $\mathbf{x}$ we had to include a 1 to handle the intercept term: $\mathbf{x}_i = (1, x_i)^{\mathrm{T}}$.

For our data we can compute R^2 in Python as

```
In [4]: yhat = ab[0] + ab[1]*A[:,1]
        print("yhat =\t",yhat)
        print("y =\t",RHS)
        residual = yhat - RHS
        print("error =\t",residual)
        r2Num = np.sum(residual**2)
        r2Denom = np.sum((RHS-RHS.mean())**2)
        print("R2 numerator =", r2Num)
        print("R2 denominator =", r2Denom)
        r2 = 1 - r2Num/r2Denom
        print("R2 =", r2)

yhat =    [ 12.209121  22.148552  32.087983  42.027414]
y =  [ 11.94688  22.30126  32.56929  41.65564]
error =  [ 0.262241 -0.152708 -0.481307  0.371774]
R2 numerator = 0.46196241067
R2 denominator = 494.423405429
R2 = 0.999065654244
```

A perfect R^2 is 1. Sometimes R^2 is called the fraction of variance explained by the model. Therefore, 1 or 100% implies that 100% of the variance in the data is explained by the model. A value of 0.999, as in this fit, is very high and typically only appears in contrived data sets used as examples.

11.1.1 Least-Squares Regression

So why did we multiply by the transpose of the matrix? It turns out that the above procedure, besides giving us a system that we can solve, minimizes the error

$$\bar{E} = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2.$$

That is why the above procedure is called least-squares regression because it minimizes the sum of the squares of the error at each point.

We can show that our solution gives the minimum, or least, squared error by differentiating the formula for $\bar{E}$ with respect to $\mathbf{u}$ and setting the result to 0. The derivative of $\bar{E}$ with respect to $\mathbf{u}$ is

$$\frac{d\bar{E}}{d\mathbf{u}} = 2\sum_{i=1}^{n} \mathbf{x}_i^{\mathrm{T}}(\mathbf{x}_i \cdot \mathbf{u} - y_i) = 0,$$

which upon rearranging and using the definition of a matrix vector product leads to

$$\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{u} = \mathbf{X}^{\mathrm{T}}\mathbf{y}.$$

That is, the value of **u** is found by multiplying the system by the transpose of **X**. This is the equation we solve to estimate the coefficients in the model.

Least-squares regression has the property that if we sum up the errors, we get zero. It works particularly well when the errors in the data (that is the deviation from the model) are independent random variables that do not depend on the value of the independent or dependent variables.

BOX 11.1 NUMERICAL PRINCIPLE

Least-squares regression is the most common type of model/curve fitting method: it will give errors with an average of zero, and it minimizes the sum of the squares of the error.

11.2 MULTIPLE LINEAR REGRESSION

We will now generalize the problem of fitting a linear function to data to allow for there to be multiple independent variables. Consider I observations of a dependent variable y and independent variables $\mathbf{x} = (1, x_1, x_2, \dots, x_J)^{\mathrm{T}}$. We desire to fit a linear model of the form

$$y(\mathbf{x}) = a_0 + a_1 x_1 + \cdots + a_J x_J + \epsilon.$$

The equations that govern the relationship between y and $\mathbf{x}$ are given by

$$\mathbf{X}\mathbf{u} = \mathbf{y},$$

where **X** is an $I \times (J+1)$ matrix of the form

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & x_{12} & \dots & x_{1J} \\ 1 & x_{21} & x_{22} & \dots & x_{2J} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{i1} & x_{i2} & \dots & x_{iJ} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{I1} & x_{I2} & \dots & x_{IJ} \end{pmatrix}, \tag{11.1}$$

where the notation x_{ij} is the ith observation for the jth independent variable. The vector **u** holds the coefficients of the model

$$\mathbf{u} = (a_0, a_1, \dots, a_J)^{\mathrm{T}}, \tag{11.2}$$

TABLE 11.1 Statistics from 2013 Texas A&M Football Team

Game	Opponent	Yards Gained	Points Scored	Off. T.O.[a]	Def. T.O.[a]
1	Rice	486	52	1	2
2	Sam Houston State	714	65	1	2
3	Alabama	628	42	2	1
4	SMU	581	42	1	3
5	Arkansas	523	45	0	2
6	Ole Miss	587	41	2	1
7	Auburn	602	41	2	1
8	Vanderbilt	558	56	5	3
9	UTEP	564	57	1	4
10	Mississippi State	537	51	3	1
11	LSU	299	10	2	0
12	Missouri	379	21	1	0
13	Duke	541	52	0	2

[a] *T.O. is an abbreviation for turnovers.*

and the vector **y** contains the observations of the dependent variable

$$\mathbf{y} = (y_1, \ldots, y_I)^{\mathrm{T}}. \tag{11.3}$$

As before, we will multiply both sides of the equation by $\mathbf{X}^{\mathrm{T}}$ to form the normal equations, and solve for **u**. We will now apply this more general technique to a real set of data.

11.2.1 Example From Outside of Engineering

Consider the following data in Table 11.1. This is data from the 2013 Texas A&M Aggies' football season. The numeric columns are

- Yards Gained on Offense
- Points Scored on Offense
- Turnovers by the Aggie Offense
- Turnovers received by the Aggie Defense.

For this example we will use the points scored as the dependent variable with a single independent variable: yards gained. The data in Table 11.1 has been stored in a csv file that is read in the next code block.

```
In [5]: import csv
        with open('FootballScores.csv', newline='') as csvfile:
            reader = csv.DictReader(csvfile)
            opponent = []
            Yards = np.array([])
            Points = np.array([])
            OffTurn = np.array([])
            DefTurn = np.array([])
            for row in reader:
                #print(row)
                opponent.append(row['Opponent'])
                Yards = np.append(Yards,float(row['Yards Gained']))
                Points = np.append(Points,float(row['Points Scored']))
                OffTurn = np.append(OffTurn,float(row['Off Turnovers']))
                DefTurn = np.append(DefTurn,float(row['Def Turnovers']))
        print(opponent)
        print(Yards)
        print(Points)
        print(OffTurn)
        print(DefTurn)

['Rice ', 'Sam Houston State ', 'Alabama ', 'SMU ', 'Arkansas ',
'Ole Miss ', 'Auburn ', 'Vanderbilt ', 'UTEP ', 'Mississippi State ',
'LSU ', 'Missouri ', 'Duke ']
[ 486.  714.  628.  581.  523.  587.  602.  558.  564.  537.  299.  379.
  541.]
[ 52.  65.  42.  42.  45.  41.  41.  56.  57.  51.  10.  21.  52.]
[ 1.  1.  2.  1.  0.  2.  2.  5.  1.  3.  2.  1.  0.]
[ 2.  2.  1.  3.  2.  1.  1.  3.  4.  1.  0.  0.  2.]
```

Now we want to perform our least squares procedure where the independent variable is yards gained and the dependent variable is points scored. First, we build the matrix as before. This is a good opportunity to use the `vstack` function in NumPy to stack two vectors on top of each other and then take the transpose to get a rectangular matrix with 13 rows and 2 columns:

```
In [6]: A = np.vstack([np.ones(Yards.size), Yards]).transpose()
        print("The A matrix is\n",A)

The A matrix is
 [[   1.  486.]
 [   1.  714.]
 [   1.  628.]
 [   1.  581.]
 [   1.  523.]
 [   1.  587.]
 [   1.  602.]
 [   1.  558.]
 [   1.  564.]
 [   1.  537.]
 [   1.  299.]
 [   1.  379.]
 [   1.  541.]]
```

NumPy also has a built-in least squares function `linalg.lstsq`. This function is used by passing it the data matrix and the righthand side. We use this function to determine the linear model for points scored as a function of yards gained:

```
In [7]: solution = np.linalg.lstsq(A,Points)[0]
        print("The function is Points =",solution[0],"+",solution[1],"* Yards")

The function is Points = -16.2577502382 + 0.112351872138 * Yards
```

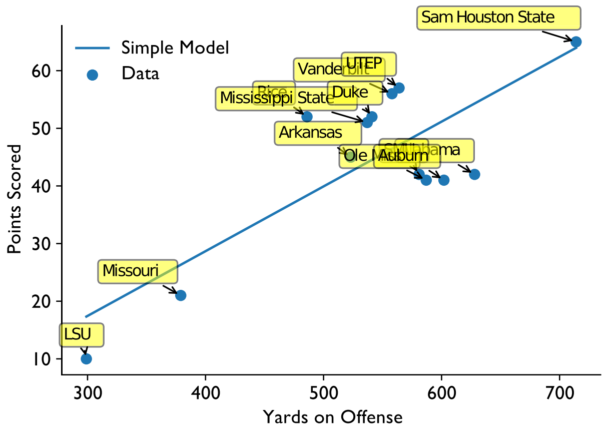
BOX 11.2 NUMPY PRINCIPLE

In the linear algebra section of NumPy there is a function that can solve the least squares problem called `lstsq`. It is called via the syntax

```
x = np.linalg.lstsq(A,b)[0]
```

where `x` is the vector containing the solution described in Eq. (11.2), `A` is the data matrix described above in Eq. (11.1), and `b` is the vector of dependent variables given in Eq. (11.3). The `[0]` selects the first of the many variables the function `lstsq` returns. This function returns several extra pieces of information that we do not need.

It is important to interpret what the coefficients in your model mean. In this case the model says that for every 10 yards gained, we expect to score 1.1 points. It also says that if zero yards are gained, then we expect -16 points to be scored. This seemingly anomalous result arises because we are trying to fit a simple model to this data. The original data and model are plotted next to help illustrate what the model is telling us:

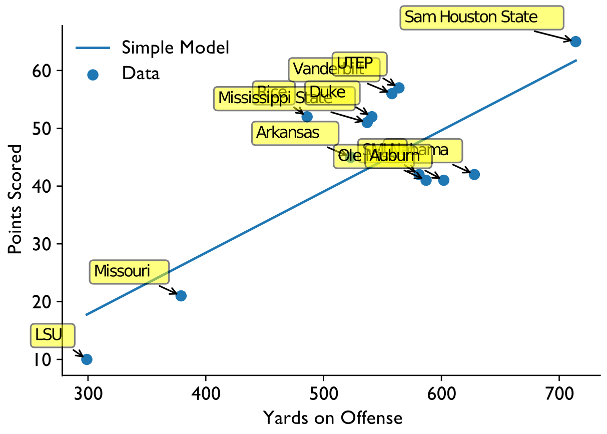


This is what real data typically look like: a lot messier than a simple line with a little noise. The model shows that several games with very high or very low yards set the trend for the line. Also we notice that half of the data are on one side of the line and the other half are on the other. This splitting of the data is typical of least-squares regression.

To adjust our model, we can remove two of the lower conference schools (Sam Houston State and UTEP); the scores in these games are expected to be different than the other games based on the strength of those teams. This removes two data points to the top right. Removing data points like this can help a model if we think that the data removed is not representative of the overall data set.

```
In [8]:  Asmall = A[(0,2,3,4,5,6,7,9,10,11,12),:]
         PointsSmall = Points[[0,2,3,4,5,6,7,9,10,11,12]]
         solution = np.linalg.lstsq(Asmall,PointsSmall)[0]
```

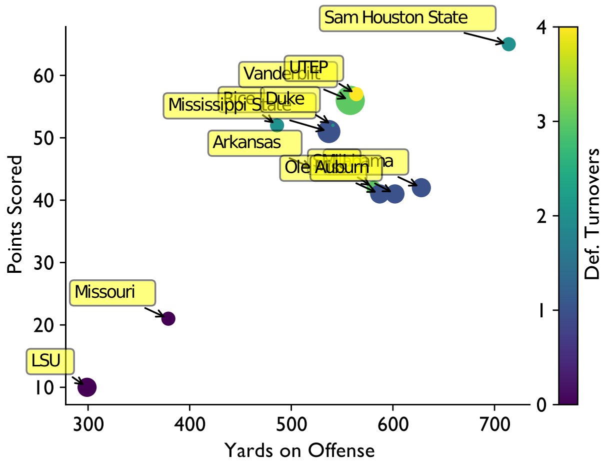
```
The function is Points = -13.9002357065 + 0.105908511234 * Yards
```



Notice that the slope of our line did not change much by removing some data (0.112 points per yard versus 0.106). This is a useful observation. It means that our model is robust to removing some games. Had the result changed drastically we should be worried that the model was driven by just a few data points.

11.2.2 Adding More Variables

To attempt to improve this model we could add more independent variables. For example, the number of turnovers produced by the defense (i.e., interceptions or fumbles that the opponent gives up) or given up by the offense (throwing an interception or losing a fumble by the offense) could also be important. The next figure colors the dots with the number of turnovers taken by the defense, and the size of the dot is the number of turnovers given up by the offense. Looking at this chart seems to indicate that the lower scoring games had fewer turnovers produced by the defense.



We will add these two variables to our model and compute the fit via least squares.

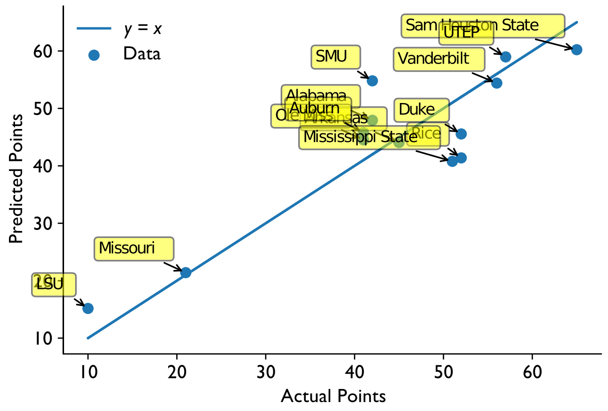
```
In [9]:  A = np.vstack([np.ones(Yards.size), Yards, OffTurn, DefTurn]).transpose()
          print("The A matrix is\n",A)
          solution = np.linalg.lstsq(A,Points)[0]
          print("The function is Points =",solution[0],
                "+",solution[1],"* Yards","+",solution[2],
                "* Off Turnovers","+",solution[3],"* Def Turnovers")
```

```
The A matrix is
 [[   1.  486.    1.    2.]
 [   1.  714.    1.    2.]
 [   1.  628.    2.    1.]
 [   1.  581.    1.    3.]
 [   1.  523.    0.    2.]
 [   1.  587.    2.    1.]
 [   1.  602.    2.    1.]
 [   1.  558.    5.    3.]
 [   1.  564.    1.    4.]
 [   1.  537.    3.    1.]
 [   1.  299.    2.    0.]
 [   1.  379.    1.    0.]
 [   1.  541.    0.    2.]]

The function is Points = -10.2946466989 + 0.0826356338026 * Yards +
     0.376961922593 * Off Turnovers + 5.57033662396 * Def Turnovers
```

Once again, we will try to interpret the coefficients in the model. Given the sign of the coefficient for offensive turnovers, the model indicates that turning over the ball on offense is actually good for scoring points. Even a basic understanding of football indicates that this is not likely to be a good strategy. When you get a model with an obviously wrong (or indefensible coefficient), it is best to remove that coefficient and refit the model. The reason we can get coefficients with strange values is that the model does not tell us what *causes* the dependent variable to change, it is simply telling us what is *correlated* with the dependent variable. To get at causation we would have to do a controlled experiment, not look at historical, observational data such as this data set.

Before we remove the offending independent variable from the mode, we will look at the values of the predictions versus the actual values. This is an important diagnostic for multivariate models because it is harder to visualize the trends like we did for a single variable model. We make a scatter plot where the x axis is the actual points scored and the y axis is the predicted points scored. If the model is perfect, then all the points will fall on the diagonal line $y = x$.



This figure indicates that our model does seem to predict the number of points scored based on the independent variables. To get a more quantitative measure of the model accuracy we will compute the average amount the model is off by. This quantity is the average absolute value of the error, which is simply called the mean absolute error. For this model the mean absolute error is

```
In [10]: yhat = np.dot(A,solution)
         np.mean(np.fabs(Points-yhat))
```

```
Out[10]: 5.3201734672523262
```

BOX 11.3 NUMERICAL PRINCIPLE

The mean absolute error can be a better measure of the model fit than R^2. This is especially true is the model is going to be used for predictions. In such a case you are often more concerned with how far off a prediction is going to be.

We have to compute the average absolute error because of error cancellation—if we computed the average error, the cancellation of positive and negative errors would give us an average error very close to zero. Indeed this is what we see:

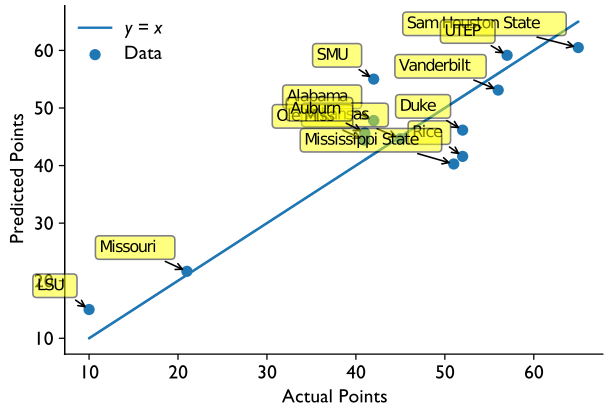
```
In [11]: np.mean(Points-yhat)
```

```
Out[11]: -1.1409676622301608e-13
```

As indicated above, we should remove offensive turnovers from the model because the coefficient for this variable did not make sense. We will remove this variable, refit the model, and plot the actual versus predicted values.

```
In [12]: A = np.vstack([np.ones(Yards.size), Yards, DefTurn]).transpose()
         solution = np.linalg.lstsq(A,Points)[0]
         print("The function is Points =",solution[0],
               "+",solution[1],"* Yards","+",solution[2],"* Def Turnovers")
```

```
The function is Points = -9.79027984448 + 0.0829036035041 * Yards +
                         5.54687804786 * Def Turnovers
```



The coefficients of this model indicate that an additional defensive turnover is worth about 5.5 points on offense and that each 10 yards gained is worth 0.8 points. The actual versus predicted values plot looks very similar to the previous model. Additionally, the mean absolute error is about the same.

```
In [12]: np.mean(np.fabs(Points-yhat))

Out[12]: 5.3393751264980898
```

The R^2 for this model is

```
In [13]: 1-np.sum((Points-yhat)**2)/np.sum((Points.mean() - Points)**2)

Out[13]: 0.79071317769713123
```

This is a respectable value for R^2 for a real data set: almost 80% of the variability in points scored can be explained by our model that contains only the number yards gained and the number of defensive turnovers. The mean absolute error also tells us that given the number of yards gained by the Aggies and the number of takeaways on defense, we can estimate how many points the team scored to within about ± 5 points.

It is a good thing to remove unnecessary variables in the model for several reasons. The first is that putting too many variables in the model can lead to overfitting: if we add enough variables eventually the model will have zero error (when we have the same number of variables as observations, the fit will be "perfect"). However, this model will perform poorly in making predictions.

The other reason is that strange values will call the model into question. Nobody will believe anything else you say if you make a patently false claim (like turn the ball over to get more points). It is important when performing curve fitting to think critically about what the model is indicating. Especially, if you want to use the model to explain what is occurring in the data set.

11.3 “NONLINEAR” MODELS

Least squares regression can be applied to models that are nonlinear in the independent variables, if we can map the nonlinearity to new independent variables. In this section we will demonstrate this procedure.

Suppose we wish to fit a model that has the form

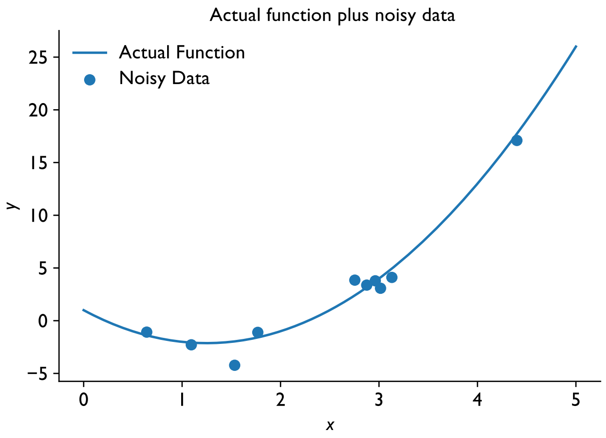
$$f(x) = a_0 + a_1 x_1 + a_2 x_1^2.$$

We can fit this model using least squares by defining a new variable $x_2 \equiv x_1^2$. This will make the model look like a multivariate regression model:

$$f(x) = a_0 + a_1 x_1 + a_2 x_2.$$

Therefore, to fit this quadratic model we set up the matrix with this additional variable x_2 in the appropriate column.

Consider the following data generated from the function $y = 1 - 5x + 2x^2$ with noise added:

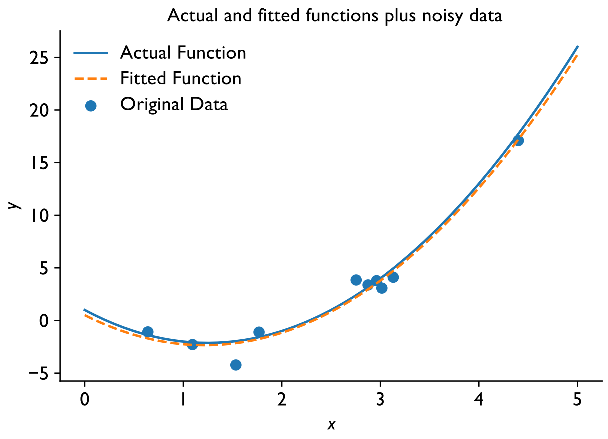


To fit the quadratic model need to build the matrix with x^2 in the appropriate column before calling the least squares function:

```
In [14]: #now build the matrix
         A = np.ones((N,3))
         for i in range(N):
             A[i,1] = x[i]
             A[i,2] = x[i]**2
         solution = np.linalg.lstsq(A,y)[0]
         print("The function is y =",solution[0],"+",
               solution[1],"* x","+",solution[2],"* x^2")
```

```
The function is y = 0.50907172038 + -4.69368434583 * x + 1.92891037106 * x^2
```

The fitted function is close to the actual function used to produce the data. The coefficients are slightly off. Graphically, the fitted and original function are close.

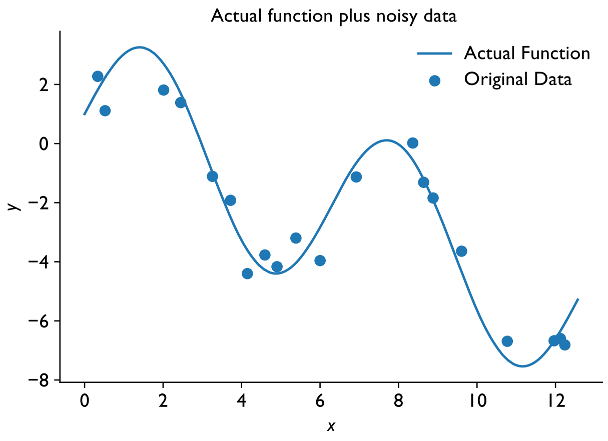


If we wanted to fit more complicated polynomial models, we can do this by placing the correct powers of the independent variables in the data matrix **X**. This makes the least squares curve fitting process very flexible.

We can also fit non-polynomial models such as

$$f(x) = a + bx + c\sin x.$$

In this case we have a column in the matrix that correspond to $\sin x$. We will demonstrate this with data produced from the function $y = 1 - 0.5x + 3\sin x$ with added noise:



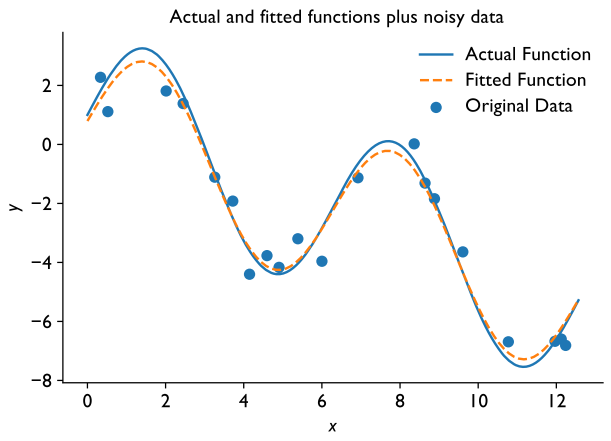
To fit this model we place the sine of x in the appropriate column of the data matrix and solve using least squares, as shown next:

```
In [15]: #now build the matrix
         A = np.ones((N,3))
         for i in range(N):
             A[i,1] = x[i]
```

```
        A[i,2] = np.sin(x[i])
    solution = np.linalg.lstsq(A,y)[0]
    print("The fitted function is y =",solution[0],
          "+",solution[1],"* x","+",solution[2],"* sin(x)")
```

```
The fitted function is y = 0.792955748456 + -0.482057534691 * x
                          + 2.7346948684 * sin(x)
```

The fitted function does deviate from the original function, primarily due to the noise in the data making some of the points used to fit the model being far away from the actual function. Despite this, the overall trend appears to be correct. This is apparent in a graphical comparison between the original function and the fitted function.



In this example with the sine function, the least squares regression model does a good job of finding the underlying model coefficients. Despite there being some data points far away from the actual function due to noise, the model fit is very close to the actual model used to generate the data.

BOX 11.4 NUMERICAL PRINCIPLE

Many different models can be fit with least-squares regression, even if at first glance the model does not look linear. If you can define new independent variables that are functional transformations of other independent variables, the resulting model can be fit with least-squares.

11.4 EXPONENTIAL MODELS: THE LOGARITHMIC TRANSFORM

Beyond nonlinear transformations to independent variables, we can use logarithms to fit models that are exponentials or power laws of independent variables. A particularly relevant example would be fitting a model to determine the half-life of a radioactive sample:

$$A(t) = A_0 e^{-\lambda t}$$

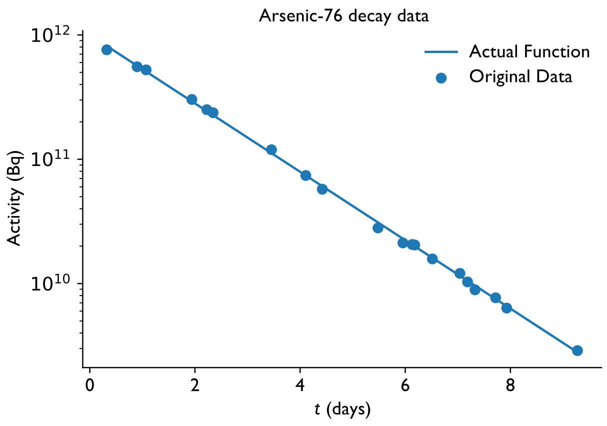
$$\lambda = \frac{\ln 2}{T_{1/2}},$$

where A is the activity. To fit this model we can take the natural logarithm of both sides and find

$$\ln A(t) = \ln A_0 - \lambda t.$$

Therefore, if we fit a model where the dependent variable (i.e., the left-hand side) is the natural logarithm of $f(x)$ then we can infer both the half-life and the initial activity.

As before we will generate data from a known exponential and add noise to the data points. In this case we generate decay data from a sample of 10^{12} atoms of arsenic-76, which has a half-life 1.09379 ± 0.00045 days.



To fit the exponential model, we need to make the righthand side in the least-squares equations equal to the natural logarithm. The data matrix will contain the number of days since the sample was obtained.

```
In [16]: #now build the matrix
         A = np.ones((N,2))
         for i in range(N):
             A[i,1] = t[i]
         print("The A matrix is\n",A)
         solution = np.linalg.lstsq(A,np.log(activity))[0]
         print("The inital activity is A_0 =",np.exp(solution[0]),
               "and the half-life is",-np.log(2)/solution[1],"days.")
```

```
The inital activity is A_0 = 1.00216154364e+12 and the half-life is
1.09039181369 days.
```

The fitted model gives a reasonable approximation of the half-life given the noise in the data.

11.4.1 Power Law Models

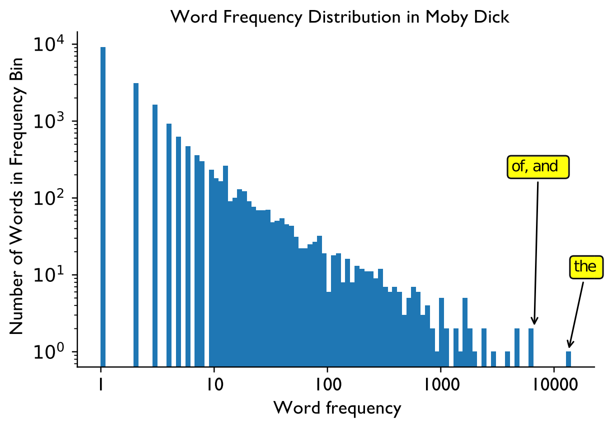
It is also possible fit power-law models using similar manipulations. The function

$$f(x) = ax^b,$$

can be transformed to a linear, additive model by writing a function

$$\ln f(x) = \ln a + b\ln(x),$$

that is we take the natural logarithm of x and $f(x)$ to get a linear function. Such power laws appear in all kinds of natural data. One, perhaps unexpected, place a power law appears is in the number of words used with a given frequency in language. In English it has been conjectured that the 100 most common words make up 50% of all writing. Another way to look at this, is that there are a small number of words that are used very frequently (e.g., the, a, and, etc.), and many words that are used very infrequently (e.g. consanguine or antiderivative). Therefore, if we look at any work of literature we expect there to be thousands of words used one or two times, and a few words used thousands of times. To demonstrate this we can look at the word frequency distribution for that venerable work of literature *Moby Dick*. The next figure is a histogram of word frequency in *Moby Dick*. For example, there are approximately 10^4 words that are only used once in the book out of the 17,227 unique words in the book.



The word "the" was used over 10,000 times.

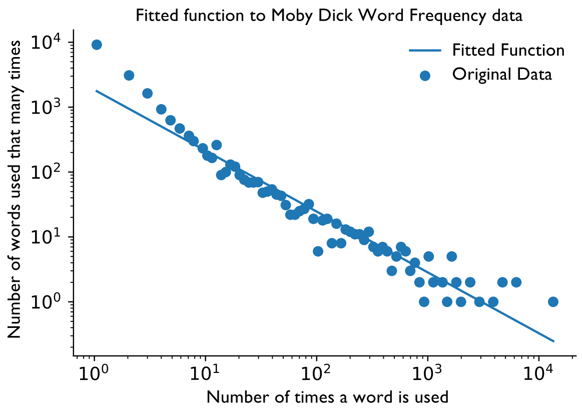
For this data, we want to fit a model as

$$\text{Number of words with a given frequency} = a(\text{Word Frequency})^b.$$

This will require us to make the righthand side of the least square equations equal to the logarithm of the dependent variable, and place the logarithm of the independent variable in the data matrix. The resulting model for *Moby Dick* is

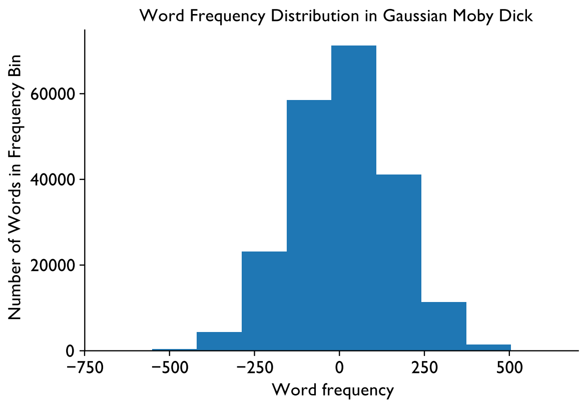
$$\text{Number of words with a given frequency} = 7.52(\text{Word Frequency})^{-0.94}.$$

We can then compare this model to the actual data, and see that a simple power law is a reasonable fit to the data:



Except for words used less than 3 times, the model predicts the number of words with a given frequency well. This natural behavior of language (and other systems such as income) are examples of power laws.

An important lesson that we can take from this data is that power law distributions, such as those for the word frequency in *Moby Dick* cannot be approximated by a normal or Gaussian distribution. For example, if we generate samples from a Gaussian distribution with a mean equal to the mean number of times a word is used in *Moby Dick*, and a variance equal to the observed variance in word frequencies the data looks nothing like a power law. The figure below shows 211,763 samples chosen from a Gaussian distribution with mean 11.137 (the average number of times a word in used in *Moby Dick*), and variance of 22,001 (the observed variance of word frequencies in *Moby Dick*).



This, you can agree, does not look anything like the distribution of word frequency in Moby Dick. Even if we ignore the negative frequencies, the distribution says there should be very few words used more frequently than 500 times. In actuality, there are about 100 words used more than 500 times. Gaussian Moby Dick is a very different book. “Call me Ishmael” may not have made the cut.

CODA

There are many times in engineering and quantitative analysis in general where you need to reconcile data with a model (as is the case in the half-life example above) or when you want to develop a model to explain data (similar to the football score example). The most basic, but still very useful, tool for these situations is least-squares regression. These regression models can be fit easily using the capabilities in NumPy.

FURTHER READING

The application of curve fitting to observed data is an important part of the study of statistics. As such, there are many works on applying regression techniques. For a deep dive into the field of machine learning (of which linear regression is an example) see the comprehensive monograph by Hastie, Tibshirani, and Friedman [13]. For the application of regression techniques to data collected from a variety of sources (including social science and public safety), Gelman and Hill’s book [14] is recommended.

N.N. Taleb is a crusader against using Gaussian ideas in a world of power law distributions. His various books on this subject are entertaining and informative reads [15–17].

PROBLEMS

Programming Projects

1. Power Law Fit

Fit a power law to the data below from the United States Internal Revenue Service for the number (in thousands) of tax returns filed in 2014 with particular values of incomes (in thousands). The incomes come from the midpoints of the brackets that the IRS uses to report data. Make the independent variable be the income, and the number of returns be the dependent variable. Compute R^2 and the mean absolute error for the model.

Income (Thousands $)	Returns (Thousands)
2.5	10263
7.5	11790
12.5	12290
17.5	11331
22.5	10062
27.5	8819
35	14600
45	11473
57.5	19395
87.5	12826

Income (Thousands $)	Returns (Thousands)
150	17501
350	4979
750	835
1250	180
1750	77
3500	109
7500	27
10000	17

2. *Inflating R^2*

Produce 10 uniform random samples in the range [0,1], call this the independent variable x_1. Produce another 10 uniform random samples in range [0,1], call this the independent variable x_2. Finally, produce 10 uniform random samples in the range [2,3] and call this the dependent variable y. Fit 5 linear models

$$y = a_0 + a_1 x_1,$$
$$y = b_0 + b_1 x_2,$$
$$y = c_0 + c_1 x_1 + c_2 x_2,$$
$$y = d_0 + d_1 x_1 + d_2 x_2 + d_3 x_1 x_2,$$
$$y = e_0 + e_1 x_1 + e_2 x_2 + e_3 x_1 x_2 + e_4 x_1^2 + e_5 x_2^2.$$

For each of these models compute R^2 and the mean absolute error. Discuss the results.

3. *k_∞ and Diffusion Length for a Water Balloon*

Consider a water balloon that you fill with a homogeneous mixture of heavy water and fissile material. You do not know the exact ratio of the water molecules to fissile atoms. However, you can fill the balloon with different amounts of the solution (thereby changing the radius of the sphere). You are able to measure the multiplication factor, k_{eff}, at several different balloon radii by pulsing the sphere with a neutron source and observing the subcritical multiplication. You also astutely remember that the 1-group diffusion theory relation between the multiplication factor, the infinite medium multiplication factor (k_∞), and the diffusion length, L is

$$k_{\text{eff}} = \frac{k_\infty}{1 + L^2 B_g^2},$$

where B_g^2 is the geometric buckling for the system. In this case we have

$$B_g^2 = \frac{\pi^2}{(R+d)^2},$$

where d is the extrapolation length. If we assume that $d \ll R$, we can do a linear Taylor expansion around $d = 0$ to write

$$B_g^2 = \frac{\pi^2}{R^2} - \frac{2\pi^2}{R^3} d.$$

Given the measurement data below, infer the values of k_∞, L, and d. Is the assumption on d correct? What radius will make the reactor critical? Report R^2 for your model.

Hint: make your regression model have the dependent variable be $1/k_{\text{eff}}$.

Experiment	R	k_{eff}
1	10.00	0.16
2	12.50	0.23
3	15.00	0.31
4	20.00	0.46
5	25.00	0.60
6	35.00	0.80
7	36.00	0.82
8	40.00	0.87
9	45.00	0.93
10	50.00	0.98

4. Model Without a Constant

It does not always make sense to have a constant in a linear model. What the constant indicates is the value of the dependent variable when all the independent variables are zero. One example of a model where this is the case is the temperature coefficient of reactivity for a nuclear system. This coefficient, denoted by α, given by

$$\Delta\rho = \alpha \Delta T,$$

where T is the temperature of the system and ΔT is the difference between the current temperature and the point where the reactivity ρ is 0. The reactivity is given by

$$\rho = \frac{k-1}{k}.$$

Your task is to find α using the data below by fitting a model without a constant. To do this you have to modify the data matrix in the least squares procedure. Report R^2 for your model.

k_{eff}	T **(K)**
1.000000	250
0.999901	300
0.998032	350
0.996076	400
0.994055	450
0.992329	500