5 Linear Regression

We introduce the basic model of linear regression. It builds a linear model to predict one variable from one other variable or from a set of other variables. We will demonstrate how this simple technique can extend to building potentially much more complex polynomial models. Then we will introduce the central and extremely powerful idea of cross-validation. This method fundamentally changes the statistical goal of validating a model, to characterizing the data.

5.1 Simple Linear Regression

We will begin with the simplest form of linear regression. The input is a set of n 2-dimensional data points $P = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. The ultimate goal will be to predict the y values using only the x-values. In this case x is the *explanatory variable* and y is the *dependent variable*.

In order to do this, we will "fit" a line through the data of the form

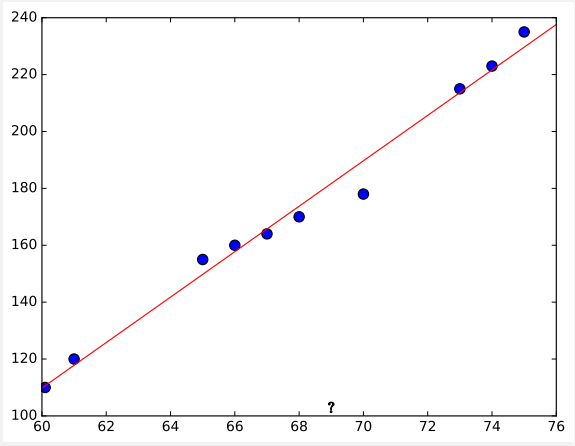
$$y = \ell(x) = ax + b,$$

where a (the slope) and b (the intercept) are parameters of this line. The line ℓ is our "model" for this input data.

Example: Fitting a line to height and weight

Consider the following data set that describes a set of heights and weights.

height (in)	weight (lbs)
66	160
68	170
60	110
70	178
65	155
61	120
74	223
73	215
75	235
67	164
69	?



Note that in the last entry, we have a height of 69, but we do not have a weight. If we were to guess the weight in the last column, how should we do this?

We can draw a line (the red one) through the data points. Then we can guess the weight for a data point with height 69, by the value of the line at height 69 inches: about 182 pounds.

Measuring error. The purpose of this line is not just to be close to all of the data (for this we will have to wait for PCA and dimensionality reduction). Rather, its goal is prediction; specifically, using the explanatory variable x to predict the dependent variable y.

In particular, for every value $x \in \mathbb{R}$, we can predict a value $\hat{y} = \ell(x)$. Then on our dataset, we can examine for each x_i how close $\hat{y}_i$ is to y_i. This difference is called a *residual*:

$$r_i = |y_i - \hat{y}_i| = |y_i - \ell(x_i)|.$$

Note that this residual is not the distance from y_i to the line ℓ, but the distance from y_i to the corresponding point with the same x-value. Again, this is because our only goal is prediction of y. And this will be important as it allows all of the techniques to be immune to the choice of units (e.g., inches or feet, pounds or kilograms)

So the residual measures the error of a single data point, but how should we measure the overall error of the entire data set? The common approach is the sum of squared errors:

$$\mathsf{SSE}(P, \ell) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \ell(x_i))^2.$$

Why is this the most common measure? Here are 3 explanations?

- The sum of squared errors was the optimal result for a single point estimator under Gaussian noise using Bayesian reasoning, when there was assumed Gaussian noise (See T2). In that case the answer was simply the mean of the data.
- If you treat the residuals as a vector $r = (r_1, r_2, \ldots, r_n)$, then the standard way to measure total size of a vector r is through its norm $\|r\|$, which is most commonly its 2-norm $\|r\| = \|r\|_2 = \sqrt{\sum_{i=1}^{n} r_i^2}$. The square root part is not so important (it does not change which line ℓ minimizes this error), so removing this square root, we are left with SSE.
- For this specific formulation, there is a simple closed form solution (which we will see next) for ℓ. And in fact, this solution will generalize to many more complex scenarios.

There are many other formulations of how best to measure error for the fit of a line (and other models), but we will not cover them in this class.

Solving for ℓ. To solve for the line which minimizes $\mathsf{SSE}(P, \ell)$ there is a very simply solution, in two steps. Calculate averages $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, and create centered n-dimension vectors $\bar{P}_x = (x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_n - \bar{x})$ for all x-coordinates and $P_y = (y_1 - \bar{y}, y_2 - \bar{y}, \ldots, y_n - \bar{y})$ for all y-coordinates.

1. Set $a = \langle \bar{P}_y, \bar{P}_x \rangle / \|\bar{P}_x\|^2$
2. Set $b = \bar{y} - a\bar{x}$

This defines $\ell(x) = ax + b$.

We will not give the full proof of why this is optimal solution (it can be shown by expanding out the SSE expression, taking the derivative, and solving for 0), but we will give a couple points of intuition.

First lets examine the intercept

$$b = \frac{1}{n}\sum_{i=1}^{n} (y_i - ax_i) = \bar{y} - a\bar{x}$$

This setting of b ensures that the line $y = \ell(x) = ax + b$ goes through the point $(\bar{x}, \bar{y})$ at the center of the data set since $\bar{y} = \ell(\bar{x}) = a\bar{x} + b$.

Second, to understand how the slope a is chosen, it is illustrative to reexamine the dot product as

$$a = \frac{\langle \bar{P}_y, \bar{P}_x \rangle}{\|x\|^2} = \frac{\|\bar{P}_y\| \cdot \|\bar{P}_x\| \cdot \cos\theta}{\|\bar{P}_x\|^2} = \frac{\|\bar{P}_y\|}{\|\bar{P}_x\|} \cos\theta,$$

where θ is the angle between the n-dimensional vectors y and x. Now in this expression, the $\|\bar{P}_y\|/\|\bar{P}_x\|$ captures how much on (root-squared) average y increases as x does (the rise-over-run interpretation of slope). However, we may want this to be negative if there is a negative correlation between $\bar{P}_x$ and $\bar{P}_y$, or really this does not matter much if there is no correlation. So the $\cos\theta$ term captures the correlation after normalizing the units of x and y. Again, this is not a formal proof, but hopefully provides some insight.

In python. This is quite easy to demonstrate in python.

```
import numpy as np

x = np.array([66, 68, 65, 70, 65, 62, 74, 70, 71, 67])
y = np.array([160, 170, 159, 188, 150, 120, 233, 198, 201, 164])

ave_x = np.average(x)
ave_y = np.average(y)

#first center the data points
xc = x - ave_x
yc = y - ave_x

a = xc.dot(yc)/xc.dot(xc)
b = ave_y - a*ave_x
print a, b

#or with scipy
from scipy import polyfit
(a,b)=polyfit(x,y,1)
print a, b

#predict weight at x=69
w=a*69+b
```

5.2 Linear Regression with Multiple Explanatory Variables

Magically, using linear algebra, everything extends gracefully to using more than one explanatory variables. Now consider a set of d explanatory variables $x_1, x_2, \ldots, x_d$, and one dependent variable y. We would now like to use all of these variables at once to make a single (linear) prediction about the variable y. That is, we would like to create a model

$$\begin{aligned} y = M(x_1, x_2, \ldots, x_d) &= \alpha_0 + \sum_{i=1}^{d} \alpha_i x_i \\ &= \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_d x_d. \end{aligned}$$

In this notation α_0 serves the purpose of the intercept b, and all of the α_is replace the single coefficient a in the simple linear regression. Now we have described a more complex linear model M.

Example: Predicting customer value
A website specializing in dongles (dongles-r-us.com) wants to predict the total dollar amount that visitors will spend on their site. It has installed some software that can track three variables:

- time (the amount of time on the page in seconds): x_1,
- jiggle (the amount of mouse movement in cm): x_2, and
- scroll (how far they scroll the page down in cm): x_3.

Also, for a set of past customers they have recorded the

- sales (how much they spend on dongles in cents): y.

We see a portion of their data set here with $n = 11$ customers:

time: x_1	jiggle: x_2	scroll: x_3	sales: y
232	33	402	2201
10	22	160	0
6437	343	231	7650
512	101	17	5599
441	212	55	8900
453	53	99	1742
2	2	10	0
332	79	154	1215
182	20	89	699
123	223	12	2101
424	32	15	8789

To build a model, we recast the data as an 11×4 matrix $X = [\mathbf{1}, x_1, x_2, x_3]$. We let y be the 11-dimensional column vector.

$$X = \begin{bmatrix} 1 & 232 & 33 & 402 \\ 1 & 10 & 22 & 160 \\ 1 & 6437 & 343 & 231 \\ 1 & 512 & 101 & 17 \\ 1 & 441 & 212 & 55 \\ 1 & 453 & 53 & 99 \\ 1 & 2 & 2 & 10 \\ 1 & 332 & 79 & 154 \\ 1 & 182 & 20 & 89 \\ 1 & 123 & 223 & 12 \\ 1 & 424 & 32 & 15 \end{bmatrix} \quad y = \begin{bmatrix} 2201 \\ 0 \\ 7650 \\ 5599 \\ 8900 \\ 1742 \\ 0 \\ 1215 \\ 699 \\ 2101 \\ 8789 \end{bmatrix}$$

The goal is to learn the 4-dimensional column vector $\alpha = [\alpha_0; \alpha_1; \alpha_2, \alpha_3]$ so

$$y \approx X\alpha.$$

Setting $\alpha = (X^T X)^{-1} X^T y$ obtains (roughly) $\alpha_0 = 262$, $\alpha_1 = 0.42$, $\alpha_2 = 12.72$, and $\alpha_3 = -6.50$. This implies an average customer with no interaction on the site generates $\alpha_0 = \$2.62$. That time does not have a strong effect here (only a coefficient α_1 at only 0.42), but jiggle has a strong correlation (with coefficient $\alpha_2 = 12.72$, this indicates 12 cents for every centimeter of mouse movement). Meanwhile scroll has a negative effect (with coefficient $\alpha_3 = -6.5$); this means that the more they scroll, the less likely they are to spend (just browsing dongles!).

Given a data point $x = (x_1, x_2, \ldots, x_d)$, we can again evaluate our prediction $\hat{y} = M(x)$ using the residual value $r_i = |y_i - \hat{y}_i| = |y_i = M(x_i)|$. And to evaluate a set of n data points, it is standard to consider the sum of squared error as

$$\mathsf{SSE}(X, y, M) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - M(x_i))^2.$$

To obtain the coefficients which minimize this error, we can now do so with very simple linear algebra.

First we construct a $n \times (d+1)$ data matrix $X = [\mathbf{1}, x_1, x_2, \ldots, x_d]$, where the first column $\mathbf{1}$ is the all ones column vector $[1; 1; \ldots; 1]$. The next d columns describe for the ith row, the data values of the explanatory variables x_1 through x_d for the ith data point. Then we let y be a n-dimensional column vector with all data for the dependent variable. Now we can simply calculate the $(d+1)$-dimensional column vector $\alpha = [\alpha_0; \alpha_1; \ldots; \alpha_d]$ as

$$\alpha = (X^T X)^{-1} X^T y.$$

Let us compare to the simple case where we have 1 explanatory variable. The $(X^T X)^{-1}$ term replaces the $\frac{1}{\|\bar{P}_x\|^2}$ term. The $X^T y$ replaces the dot product $\langle \bar{P}_y, \bar{P}_x \rangle$. And we do not need to separately solve for the intercept b, since we have created a new column in X of all 1s. Now for any dependent data values, we multiply the found coefficient α_0 by an imaginary 1 data value.

In python: We can directly write this in python as

```
import numpy as np
from numpy import linalg as LA

# directly
alpha = np.dot(np.dot(LA.inv(np.dot(X.T,X)),X.T),y.T)

# or with LA.lstsq
alpha = LA.lstsq(X,y)[0]
```

Or in many other ways.

5.3 Polynomial Regression

Sometimes linear relations are not sufficient to capture the true pattern going on in the data with even a single dependent variable x. Instead we would like to build a model of the form:

$$\hat{y} = M_2(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2$$

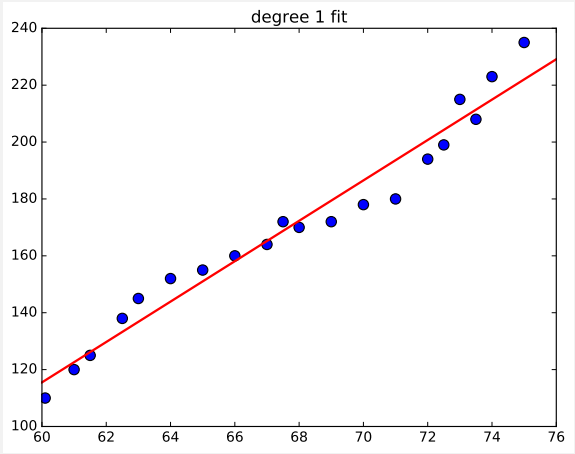
or more generally for some polynomial of degree p

$$\begin{aligned} \hat{y} = M_p(x) &= \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_p x^p \\ &= \alpha_0 + \sum_{i=1}^{p} \alpha_i x^i. \end{aligned}$$

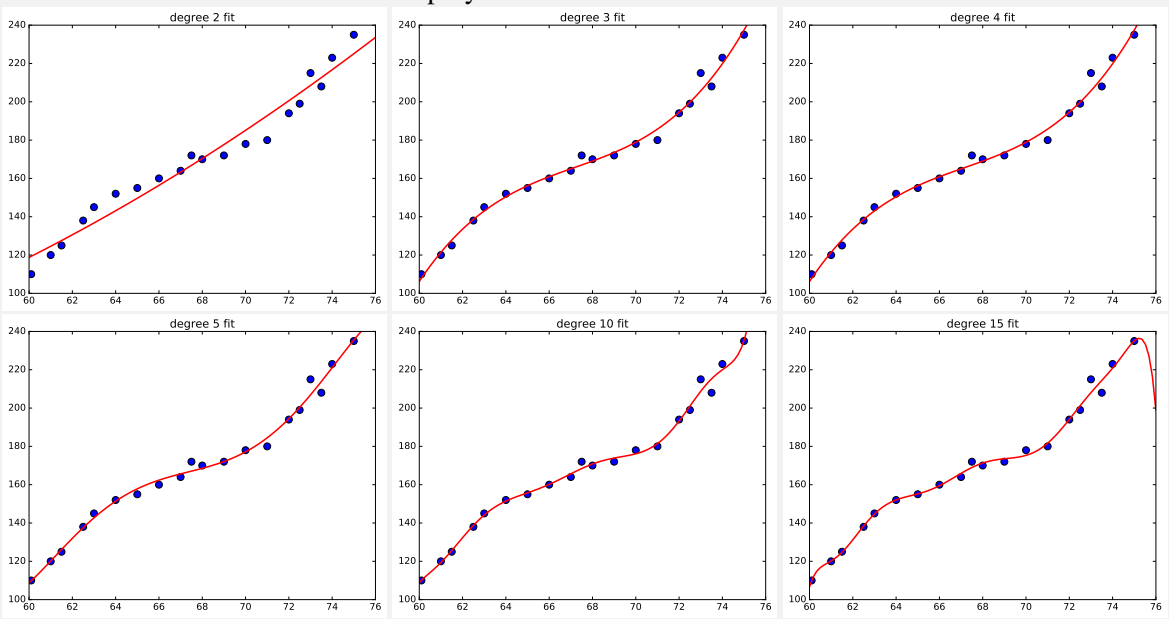
Example: Predicting Height and Weight with Polynomials

We found more height and weight data, in addition to the ones in the height-weight example above.

height (in)	weight (lbs)
61.5	125
73.5	208
62.5	138
63	145
64	152
71	180
69	172
72.5	199
72	194
67.5	172



But can we do better if we fit with a polynomial?



Again we can measure error for a single data point $p_i = (x_i, y_i)$ as a residual as $r_i = |\hat{y} - y_i| = |M_p(x_i) - y_i|$ and the error on n data points as the sum of squared residuals

$$\mathsf{SSE}(P, M_p) = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (M_p(x_i) - y_i)^2.$$

Under this error measure, it turns out we can again find a simple solution for the residuals $\alpha = [\alpha_0, \alpha_1, \ldots, \alpha_p]$. For each dependent variable data value x we create a $(p+1)$-dimensional vector

$$v = (1, x, x^2, \ldots, x^p).$$

And then for n data points $(x_1, y_1), \ldots, (x_n, y_n)$ we can create an $n \times (p+1)$ data matrix

$$X_p = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^p \\ 1 & x_2 & x_2^2 & \ldots & x_2^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^p \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ v_n \end{bmatrix}.$$

Then we can solve the same way as if each data value raised to a different power was a different dependent variable. That is we can solve for the coefficients $\alpha = [\alpha_0; \alpha_1; \alpha_2; \ldots; \alpha_n]$ as

$$\alpha = (X_p^T X_p)^{-1} X_p^T y.$$

5.4 Cross Validation

So how do we choose the correct value of p, the degree of the polynomial fit?

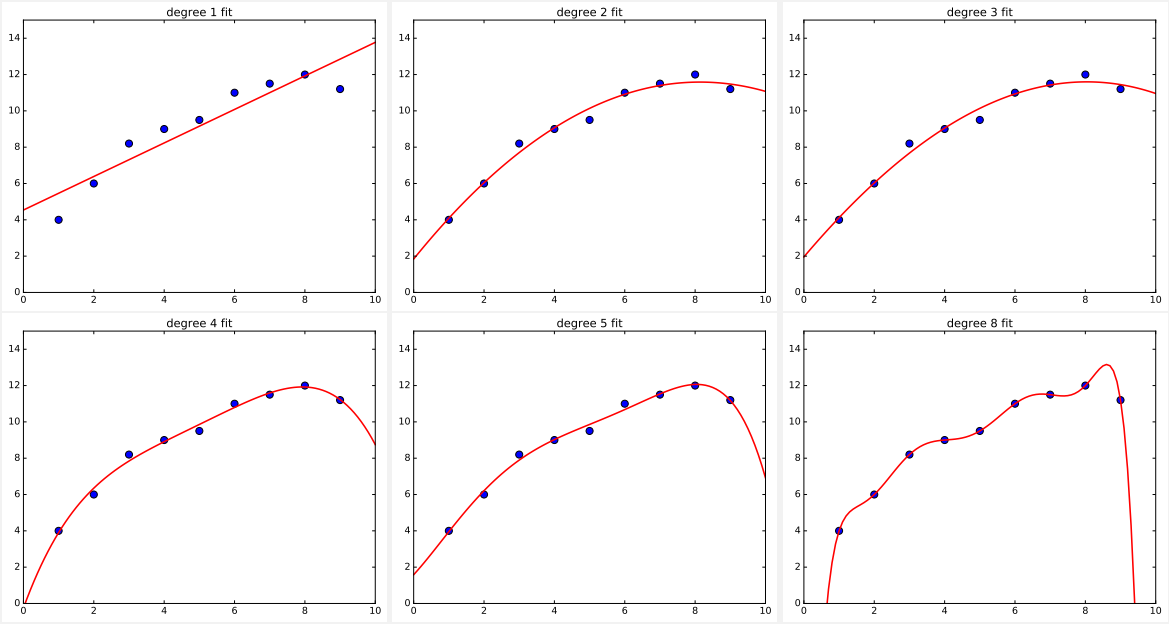
A (very basic) statistical (hypothesis testing) approach may be choose a model of the data (the best fit curve for some polynomial degree p, and assume Gaussian noise), then calculate the probability that the data fell outside the error bounds of that model. But maybe many different polynomials are a good fit?

In fact, if we choose p as $n-1$ or greater, then the curve will *polynomially interpolate* all of the points. That is, it will pass through all points, so all points have a residual of exactly 0 (up to numerical precision). This is the basis of a lot of geometric modeling (e.g., for CAD), but bad for data modeling.

Example: Simple polynomial example
Consider the simple data set of 9 points

x	1	2	3	4	5	6	7	8	9
y	4	6	8.2	9	9.5	11	11.5	12	11.2

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$. Believe your eyes, for $p = 8$, the curve actually passes through each and every point exactly.



Recall, our goal was for a new data point with only an x value to predict its y-value. Which do you think does the best job?

Generalization and Cross-Validation. Our ultimate goal in regression is *generalization* (how well do we do on new data), not SSE! Using some error measure (SSE) to fit a line or curve, is a good proxy for what we want, but in many cases (as with polynomial regression), it can be abused. We want to know how our model will generalize to new data. How would we measure this without new data?

The solution is *cross-validation*. In the simplest form, we **randomly** split our data into training data (on which we build a model) and testing data (on which we evaluate our model). The testing serves to estimate how well we would do on future data which we do not have.

- *Why randomly?:* Because you do not want to bias the model to do better on some parts than other in how you choose the split. Also, since we assume the data elements come iid from some underlying distribution, then the test data is also iid if you chose it randomly.

- *How large should the test data be?:* It depends on the data set. Both 10% and 33% are common.

Let (X, y) be the full data set (with n rows of data), and we split it into data sets $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$ with n_{train} and n_{test} rows, respectively. With $n = n_{\text{train}} + n_{\text{test}}$. Next we build a model with the training data, e.g.,

$$\alpha = (X_{\text{train}}^T X_{\text{train}})^{-1} X_{\text{train}}^T y_{\text{train}}.$$

Then we evaluate the model M_α on the test data X_{test}, often using $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)$ as

$$\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha) = \sum_{(x_i, y_i) \in (X_{\text{test}}, y_{\text{test}})} (y_i - M_\alpha(x_i))^2 = \sum_{(x_i, y_i) \in (X_{\text{test}}, y_{\text{test}})} (y_i - \langle (x_i; 1), \alpha \rangle)^2.$$

We can use the testing data for two purposes:

- To estimate how well our model would perform on new data, yet unseen. That is the predicted residual of a new data point is precisely $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_\alpha)/n_{\text{test}}$.

- To choose the correct parameter for a model (which p to use)?

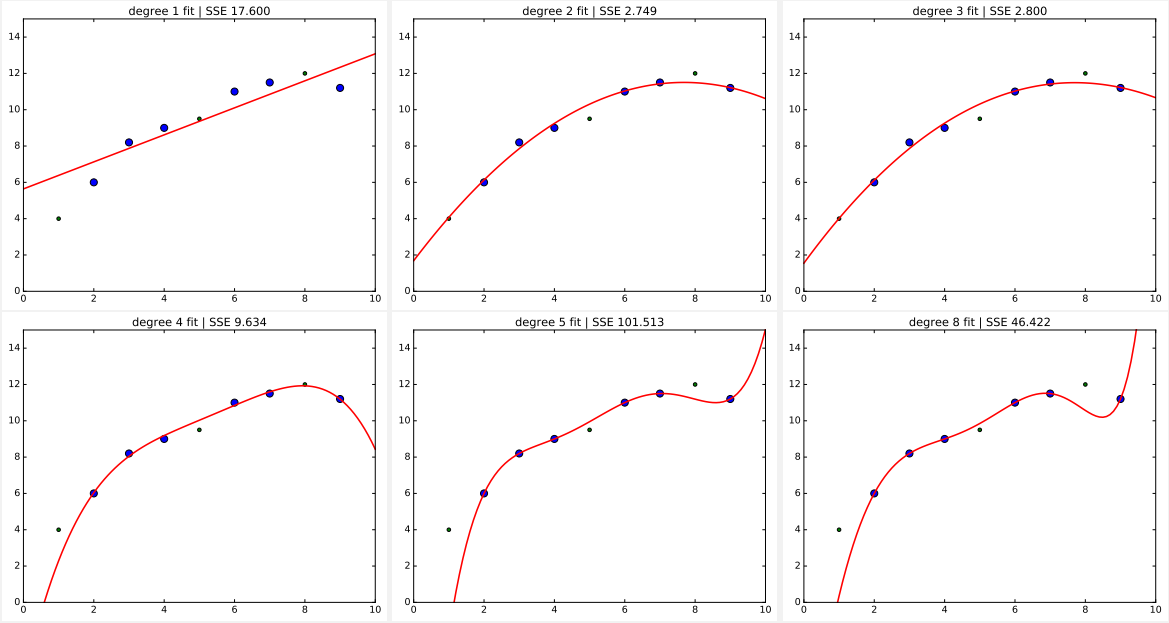
Its important to keep in mind that we should not use the same $(X_{\text{test}}, y_{\text{test}})$ to do both tasks. If we choose a model with $(X_{\text{test}}, y_{\text{test}})$, then we should reserve even more data for predicting the generalization error. When using the test data to choose a model parameter, the it is being used to build the model; thus evaluating generalization with this same data can suffer the same fate as testing and training with the same data.

So how should we choose the best p? We calculate models M_{α_p} for each value p on the same training data. Then calculate the model error $\mathsf{SSE}(X_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ for each p, and see which has the smallest value. That is we train on $(X_{\text{train}}, y_{\text{train}})$ and test(evaluate) on $(X_{\text{test}}, y_{\text{test}})$.

Example: Simple polynomial example with Cross Validation
Now split our data sets into a train set and a test set:

$$\text{train: } \begin{array}{c|cccccc} x & 2 & 3 & 4 & 6 & 7 & 8 \\ y & 6 & 8.2 & 9 & 11 & 11.5 & 12 \end{array} \qquad \text{test: } \begin{array}{c|ccc} x & 1 & 5 & 9 \\ y & 4 & 9.5 & 11.2 \end{array}$$

With the following polynomial fits for $p = \{1, 2, 3, 4, 5, 8\}$ generating model M_{α_p} on the test data. We then calculate the $\mathsf{SSE}(x_{\text{test}}, y_{\text{test}}, M_{\alpha_p})$ score for each (as shown):



And the polynomial model with degree $p = 2$ has the lowest SSE score of 2.749. It is also the simplest model that does a very good job by the "eye-ball" test. So we would choose this as our model.

Leave-one-out Cross Validation. But, not training on the test data means that you use less data, and your model is worse! You don't want to waste this data!

If your data is very large, then leaving out 10% is not a big deal. But if you only have 9 data points it can be. The smallest the test set could be is 1 point. But then it is not a very good representation of the full data set.

The alternative is to create n different training sets, each of size $n-1$ $(X_{1,\text{train}}, X_{2,\text{train}}, \ldots, X_{n,\text{train}})$, where $X_{i,\text{train}}$ contains all points except for x_i, which is a one-point test set. Then we build n different models $M_1, M_2, \ldots, M_n$, evaluate each model M_i on the one test point x_i to get an error $E_i = (y_i - M_i(x_i))^2$, and average their errors $E = \frac{1}{n}\sum_{i=1}^{n} E_i$. Again, the parameter with the smallest associated average error E is deemed the best. This allows you to build a model on as much data as possible, while still using all of the data to test.

However, this requires roughly n times as long to compute as the other techniques, so is often too slow for really big data sets.

```
import matplotlib as mpl
mpl.use('PDF')
import matplotlib.pyplot as plt
import scipy as sp
import numpy as np
import math
from numpy import linalg as LA

def plot_poly(x,y,xE,yE,p):
  plt.scatter(x,y, s=80, c="blue")
  plt.scatter(xE,yE, s=20, c="green")
  plt.axis([0,10,0,15])

  s=sp.linspace(0,10,101)

  coefs=sp.polyfit(x,y,p)
  ffit = np.poly1d(coefs)
  plt.plot(s,ffit(s),'r-',linewidth=2.0)

  #evaluate on xE, yE
  resid = ffit(xE)
  RMSE = LA.norm(resid-yE)
  SSE = xE.size * RMSE * RMSE

  title = "degree %s fit | SSE %0.3f" % (p, SSE)
  plt.title(title)
  file = "CVpolyReg%s.pdf" % p
  plt.savefig(file, bbox_inches='tight')
  plt.clf()
  plt.cla()



# train data
xT = np.array([2, 3,   4,  6,  7,     9])
yT = np.array([6, 8.2, 9,  11, 11.5,  11.2])

#test data
xE = np.array([1, 5,   8])
yE = np.array([4, 9.5, 12])


p_vals = [1,2,3,4,5,8]
for i in p_vals:
  plot_poly(xT,yT,xE,yE,i)
```