AM 207 Learning a model

Keywords: empirical risk minimization | bias | sampling distribution | hypothesis space | law of large numbers | deterministic error | training error | out-of-sample error |
Download Notebook

Contents

Contents

The process of learning

There are challenges that occur in learning a model from data:

- small samples of data
- noise in the data
- issues related to the complexity of the models we use

Let us first ask the question: what is he process of learning from data in the absence of noise. This never really happens, but it is a way for us to understand the theory of **approximation**, and lets us build a base for understanding the learning from data with noise.

Lets say we are trying to predict is a human process such as an election. Here economic and sociological factors are important, such as poverty, race and religiousness. There are historical correlations between such factors and election outcomes which we might want to incorporate into our model. An example of such a model might be:

*The odds of Romney winning a county against Obama in 2012 are a function of population religiosity, race, poverty, education, and other social and economic indicators. *

Our **causal** argument motivating this model here might be that religious people are more socially conservative and thus more likely to vote republican. This might not be the correct causation, but thats not entirely important for the prediction.

As long as a **correlation** exists, our model is more structured than 50-50 randomness, and we can try and make a prediction. Remember of-course, our model may even be wrong (see Box's aphorism: https://en.wikipedia.org/wiki/All_models_are_wrong).

We'll represent the variable being predicted, such as the probability of voting for Romney, by the letter y, and the **features** or **co-variates** we use as an input in this probability by the letter x. This x could be multi-dimensional, with x_1 being poverty, x_2 being race, and so on.

We then write

and our jobs is to take x such as data from the census about race, religiousness, and so on, and y as previous elections and the results of polls that pollsters come up with, and to make a predictive model for the elections. That is, we wish to estimate $f(x)$.

A real simple model

To gently step feet in the modelling world, lets see consider very simple model, where the probability of voting for Romney is a function only of how religious the population in a county is. This is a model I've cooked up, and the data is fake.

Let x be the fraction of religious people in a county and y be the probability of voting for Romney as a function of x. In other words y_i is data that pollsters have taken which tells us their estimate of people voting for Romney and x_i is the fraction of religious people in county i. Because poll samples are finite, there is a margin of error on each data point or county i, but we will ignore that for now.

Let us assume that we have a "population" of 200 counties x:

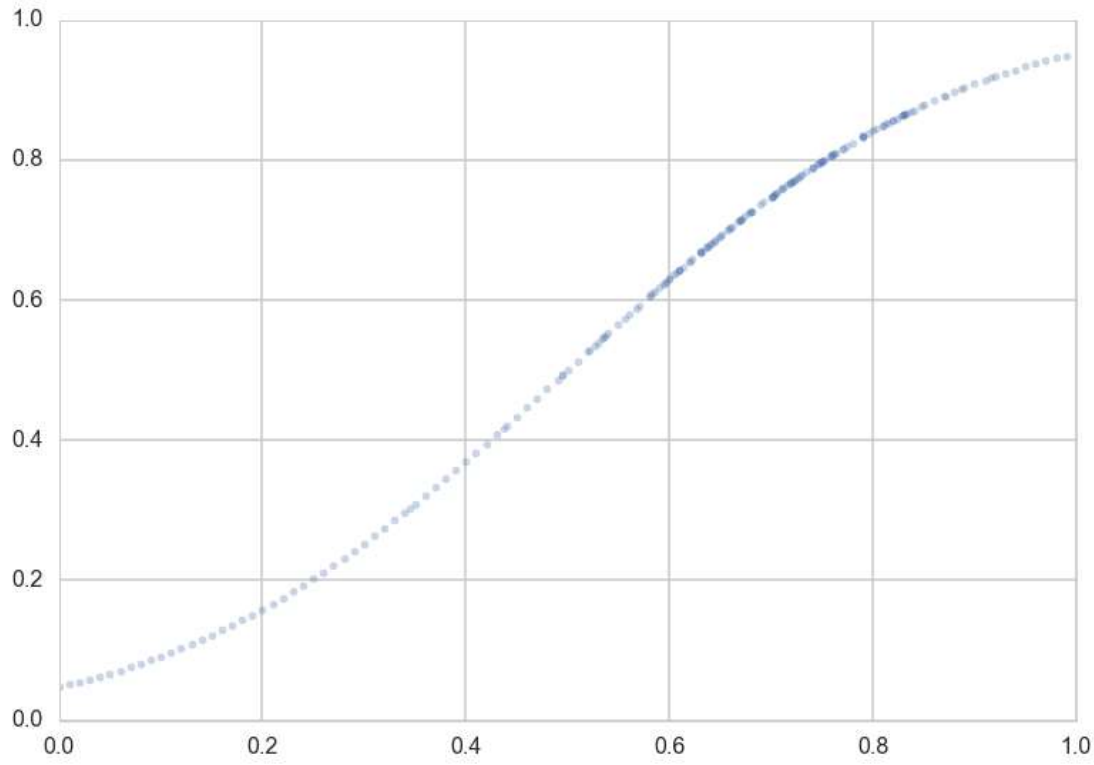
```
df=pd.read_csv("data/religion.csv")
df.head()
```

	promney	rfrac
0	0.047790	0.00
1	0.051199	0.01
2	0.054799	0.02
3	0.058596	0.03
4	0.062597	0.04

Lets suppose now that the Lord came by and told us that the points in the plot below captures $f(x)$ exactly. In other words, there is no specification error, and God knows the generating process exactly.

```
x=df.rfrac.values
f=df.promney.values
plt.plot(x,f,'.', alpha=0.3)
```

```
[<matplotlib.lines.Line2D at 0x116353cf8>]
```



Notice that our sampling of x is not quite uniform: there are more points around x of 0.7.

Now, in real life we are only given a sample of points. Lets assume that out of this population of 200 points we are given a sample $\cal{D}$ of 30 data points. Such data is called **in-sample data**. Contrastingly, the entire population of data points is also called **out-of-sample data**.

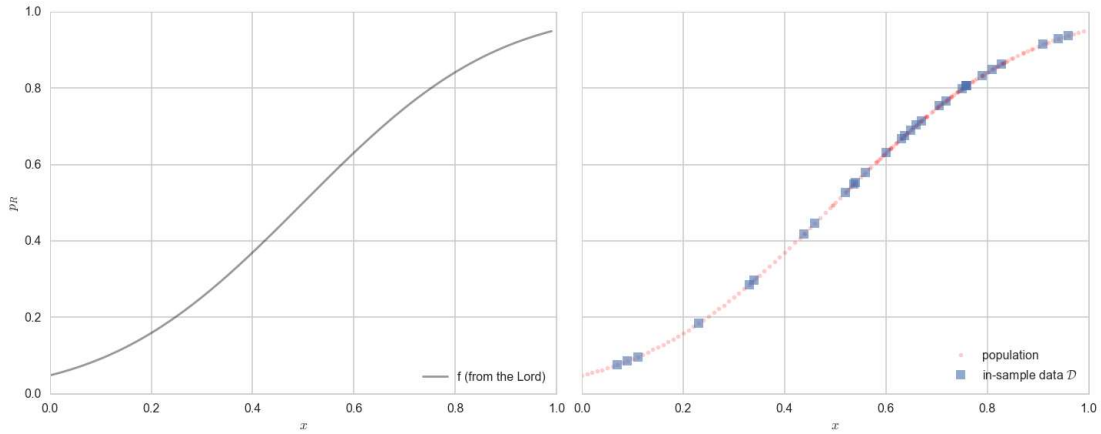
```
#indexes=np.sort(np.random.choice(x.shape[0], size=30, replace=Fal
dfsample = pd.read_csv("data/noisysample.csv")
dfsample.head()
```

	f	i	x	y
0	0.075881	7	0.07	0.138973
1	0.085865	9	0.09	0.050510
2	0.096800	11	0.11	0.183821
3	0.184060	23	0.23	0.057621
4	0.285470	33	0.33	0.358174

```
indexes = dfsample.i.values

samplex = x[indexes]
samplef = f[indexes]
```

```
axes=make_plot()
axes[0].plot(x,f, 'k-', alpha=0.4, label="f (from the Lord)");
axes[1].plot(x,f, 'r.', alpha=0.2, label="population");
axes[1].plot(samplex,samplef, 's', alpha=0.6, label="in-sample dat
axes[0].legend(loc=4);
axes[1].legend(loc=4);
```



The lightly shaded squares in the right panel plot are the in-sample $\cal{D}$ of 30 points given to us. Let us then pretend that we have forgotten the curve that the Lord gave us. Thus, all we know is the blue points on the plot on the right, and we have no clue about what the original curve was, nor do we remember the original "population".

That is, imagine the Lord gave us f but then also gave us amnesia. Remember that such amnesia is the general case in learning, where we *do not know* the target function, but rather just have some data. Thus what we will be doing is *trying to find functions that might have generated the 30 points of data that we can see* in the hope that one of these functions might approximate f well, and provide us a **predictive model** for future data. This is known as **fitting** the data.

The Hypothesis or Model Space

Such a function, one that we use to fit the data, is called a **hypothesis**. We'll use the notation h to denote a hypothesis. Lets consider as hypotheses for the data above, a particular class of functions called polynomials.

A polynomial is a function that combines multiple powers of x linearly. You've probably seen these in school, when working with quadratic or cubic equations and functions:

In general, a polynomial can be written thus:

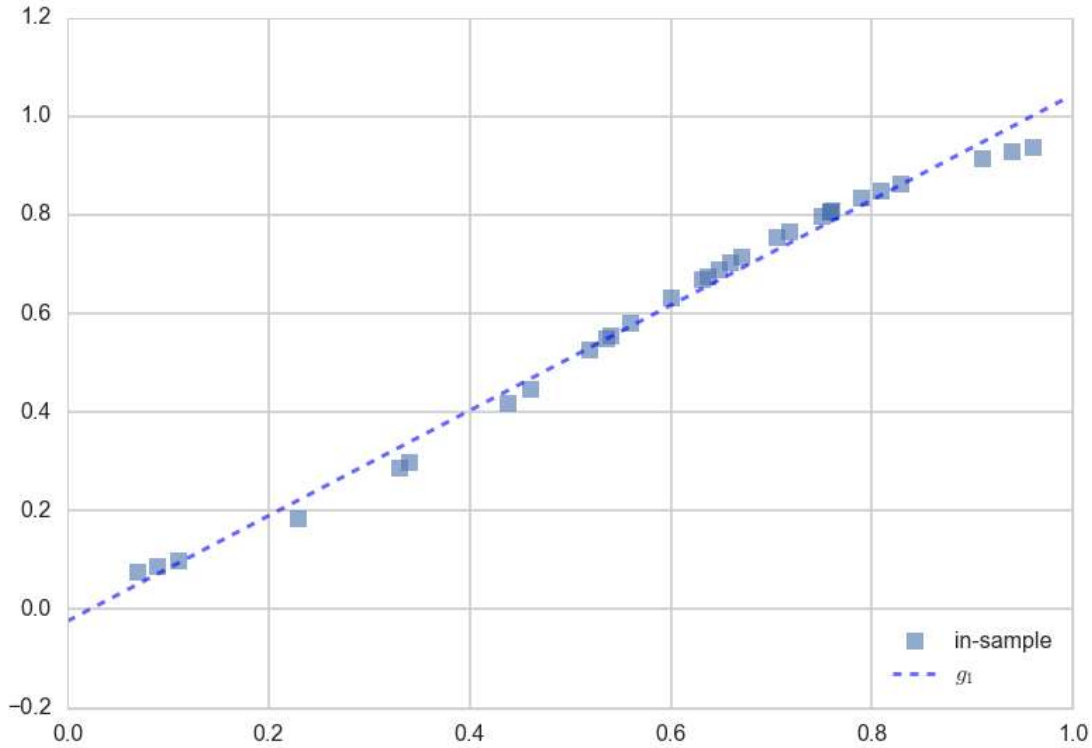
Thus, by linearly we mean a sum of coefficients a_i times powers of x, x^i. In other words, the polynomial is **linear in its coefficients**.

Let us consider as the function we used to fit the data, a hypothesis h that is a straight line. We put the subscript 1 on the h to indicate that we are fitting the data with a polynomial of order 1, or a straight line. This looks like:

We'll call the **best fit** straight line the function $g_1(x)$. The "best fit" idea is this: amongst the set of all lines (i.e., all possible choices of $h_1(x)$), what is the best line $g_1(x)$ that represents the in-sample data we have? (The subscript 1 on g is chosen to indicate the best fit polynomial of degree 1, ie the line amongst lines that fits the data best).

The best fit $g_1(x)$ is calculated and shown in the figure below:

```
g1 = np.poly1d(np.polyfit(x[indexes],f[indexes],1))
plt.plot(x[indexes],f[indexes], 's', alpha=0.6, label="in-sample")
plt.plot(x,g1(x), 'b--', alpha=0.6, label="$g_1$");
plt.legend(loc=4);
```



How did we calculate the best fit? We'll come to that in a bit, but in the meanwhile, lets formalize and generalize the notion of "best fit line amongst lines" a bit.

The set of all functions of a particular kind that we could have used to fit the data is called a **Hypothesis Space**. The words "particular kind" are deliberately vague: its our choice as to what we might want to put into a hypothesis space. A hypothesis space is denoted by the notation $\cal{H}$.

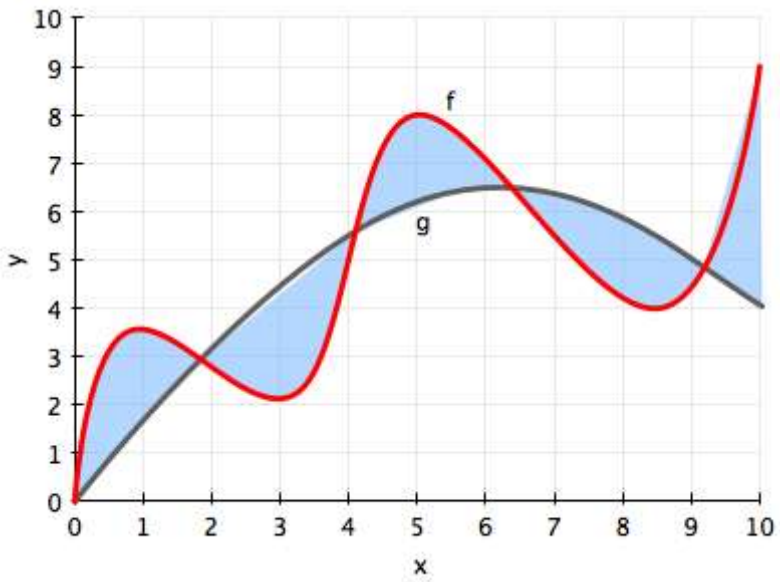
Lets consider the hypothesis space of all straight lines $h_1(x)$. We'll denote it as $\cal{H}_1$, with the subscript being used to mark the order of the polynomial. Another such space might be $\cal{H}_2$, the hypothesis space of all quadratic functions. A third such space might combine both of these together. We get to choose what we want to put into our hypothesis space.

In this set-up, what we have done in the code and plot above is this: we have found the best g_1 to the data $\cal{D}$ from the functions in the hypothesis space $\cal{H}_1$. This is not the best fit from all possible functions, but rather, the best fit from the set of all the straight lines.

The hypothesis space is a concept we can use if we want to capture the **complexity** of a model you use to fit data. For example, since quadratics are more complex functions than straight lines (they curve more), $\cal{H}_2$ is more complex than $\cal{H}_1$.

Deterministic Error or Bias

Notice from the figure above that models in $\cal{H}_1$, i.e., straight lines, and the best-fit straight line g_1 in particular, do not do a very good job of capturing the curve of the data (and thus the underlying function f that we are trying to approximate. Consider the more general case in the figure below, where a curvy f is approximated by a function g which just does not have the wiggling that f has.



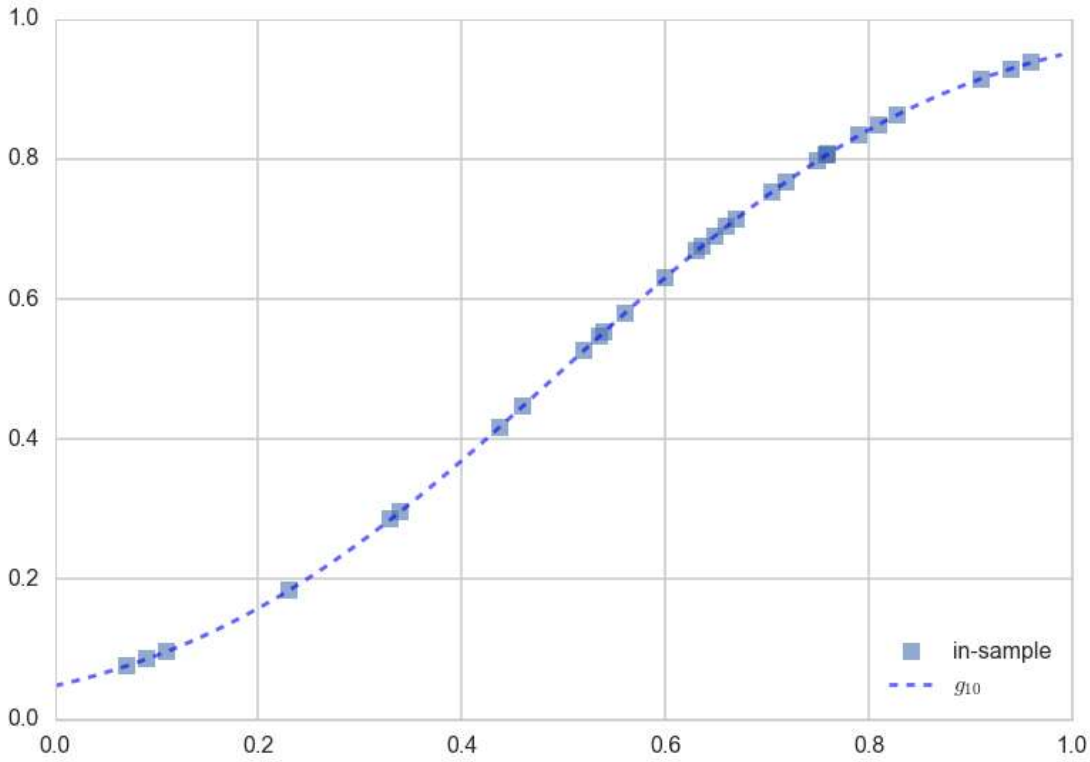
There is always going to be an error then, in approximating f by g. This *approximation error* is shown in the figure by the blue shaded region, and its called **bias**, or **deterministic error**. The former name comes from the fact that g just does not wiggle the way f does (nothing will make a straight line curve). The latter name (which I first saw used in http://www.amlbook.com/) comes from the notion that if you did not know the target function f, which is the case in most learning situations, you would have a hard time distinguishing this error from any other errors such as measurement and noise…

Going back to our model at hand, it is clear that the space of straight lines $\cal{H_1}$ does not capture the curving in the data. So let us consider the more complex hypothesis space $\cal{H_{20}}$, the set of all 20th order polynomials $h_{20}(x)$:

To see how a more complex hypothesis space does, lets find the best fit 20th order polynomial $g_{20}(x)$.

```
g20 = np.poly1d(np.polyfit(x[indexes],f[indexes],20))

plt.plot(x[indexes],f[indexes], 's', alpha=0.6, label="in-sample")
plt.plot(x,g20(x), 'b--', alpha=0.6, label="$g_{10}$");
plt.legend(loc=4);
```

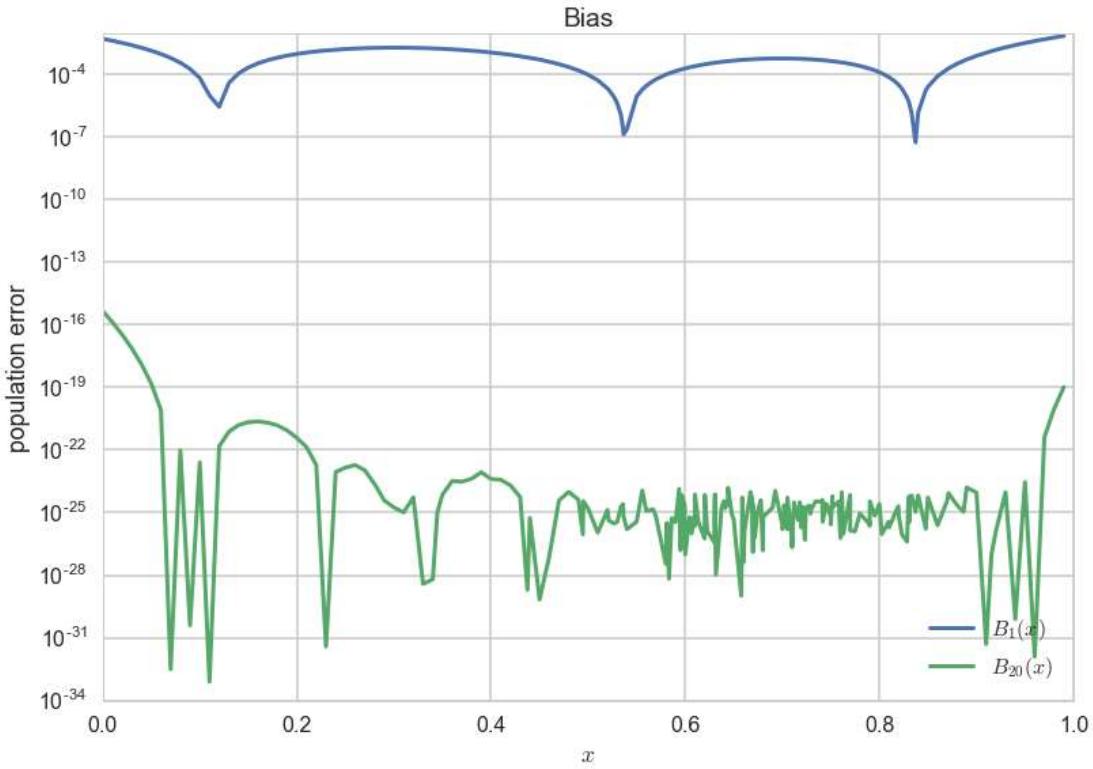


Voila! You can see the 20th order polynomial does a much better job of tracking the points, because of the wiggle room it has in making a curve "go near or through" all the points as opposed to a straight line, which well, cant curve. Thus it would seem that $\cal{H}_{20}$ might be a better candidate hypothesis set from which to choose a best fit model.

We can quantify this by calculating some notion of the bias for both g_1 and g_{20}. To do this we calculate the square of the difference between f and the g's on the population of 200 points i.e.:

Squaring makes sure that we are calculating a positive quantity.

```
plt.plot(x, (g1(x)-f)**2, lw=3, label="$B_1(x)$")
plt.plot(x, (g20(x)-f)**2, lw=3,label="$B_{20}(x)$");
plt.xlabel("$x$")
plt.ylabel("population error")
plt.yscale("log")
plt.legend(loc=4);
plt.title("Bias");
```



As you can see the **bias or approximation error** is much smaller for g_{20}.

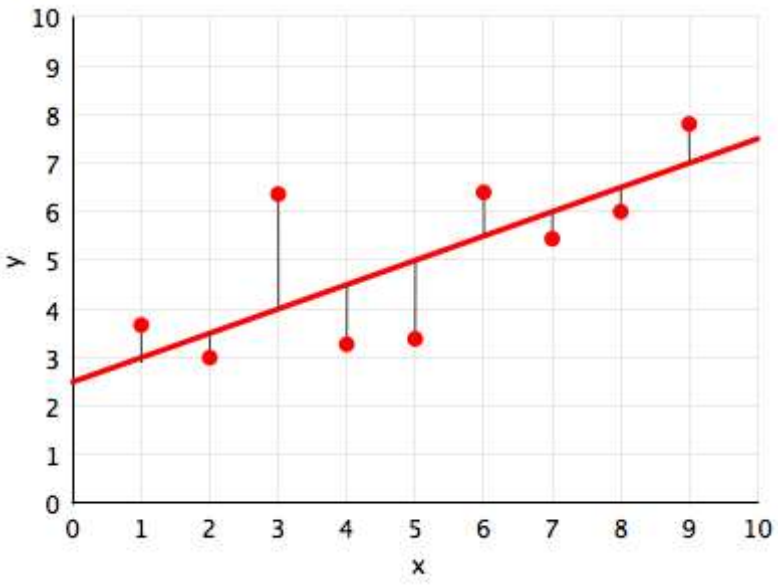
Is g_{20} the best model for this data from all possible models? Indeed, how do we find the best fit model from the best hypothesis space? This is what **learning** is all about.

We have used the python function `np.polyfit` to find g_{1} the best fit model in $\cal{H_1}$ and g_{20} the best fit model in $\cal{H_{20}}$, but how did we arrive at that conclusion? This is the subject of the next section.

How to learn the best fit model in a hypothesis space

Let's understand in an intuitive sense, what it means for a function to be a good fit to the data. Lets consider, for now, only the hypothesis space $\cal{H}_{1}$, the set of all straight lines. In the figure below, we draw against the data points (in red) one such line $h_1(x)$ (in red).

You might think you want to do this statistically, using ML Estimation or similar, but note that at this point there is no statistical notion of a generating process. We're just trying to approximate a function by another, with the latter being chosen amongst many in a hypothesis space.



The natural way of thinking about a "best fit" would be to minimize the distance from the line to the points, for some notion of distance. In the diagram we depict one such notion of distance: the vertical distance from the points to the line. These distances are represented as thin black lines.

The next question that then arises is this: how exactly we define the measure of this vertical distance? We cant take the measure of distance to be the y-value of the point minus the y value of the line at the same x, ie $y_i - h_1(x_i)$. Why? If we did this, then we could have points very far from the line, and as long as the total distance above was equal to the total distance below the line, we'd get a net distance of 0 even when the line is very far from the points.

Thus we must use a positive estimate of the distance as our measure. We could take either the absolute value of the distance, $\vert y_i - h_1(x_i) \vert$, or the square of the distance as our measure, $(y_i - h_1(x_i))^2$. Both are reasonable choices, and we shall use the squared distance for now. (Now its probably clear to you why we defined bias in the last section as the pointwise square of the distance).

We sum this measure up over all our data points, to create whats known as the **error functional** or **risk functional** (also just called **error**, **cost**, or **risk**) of using line $h_1(x)$ to fit our points $y_i \in \cal{D}$ (this notation is to be read as "y_i in $\cal{D}$") :

where N is the number of points in $\cal{D}$.

What this formula says is: the cost or risk is just the total squared distance to the line from the observation points. Here we use the word **functional** to denote that, just as in functional programming, the risk is a *function of the function* $h_1(x)$.

We also make explicit the in-sample data $\cal{D}$, because the value of the risk depends upon the points at which we made our observation. If we had made these observations y_i at a different set of x_i, the value of the risk would be somewhat different. The hope in learning is that the risk will not be too different, as we shall see in the next section

Now, given these observations, and the hypothesis space $\cal{H}_1$, we minimize the risk over all possible functions in the hypothesis space to find the **best fit** function $g_1(x)$:

Here the notation

$"\arg\min_{x} F(x)"$

means: give me the argument of the functional x at which $F(x)$ is minmized. So, for us: give me the function $g_1(x) = h_1$ at which the risk $R_{\cal{D}}(h_1)$ is minimized; i.e. the minimization is over *functions* h_1.

And this is exactly what the python function `np.polyfit(x,h,n)` does for us. It minimizes this squared-error with respect to the coefficients of the polynomial.

Thus we can in general write:

where $\cal{H}$ is a general hypothesis space of functions.

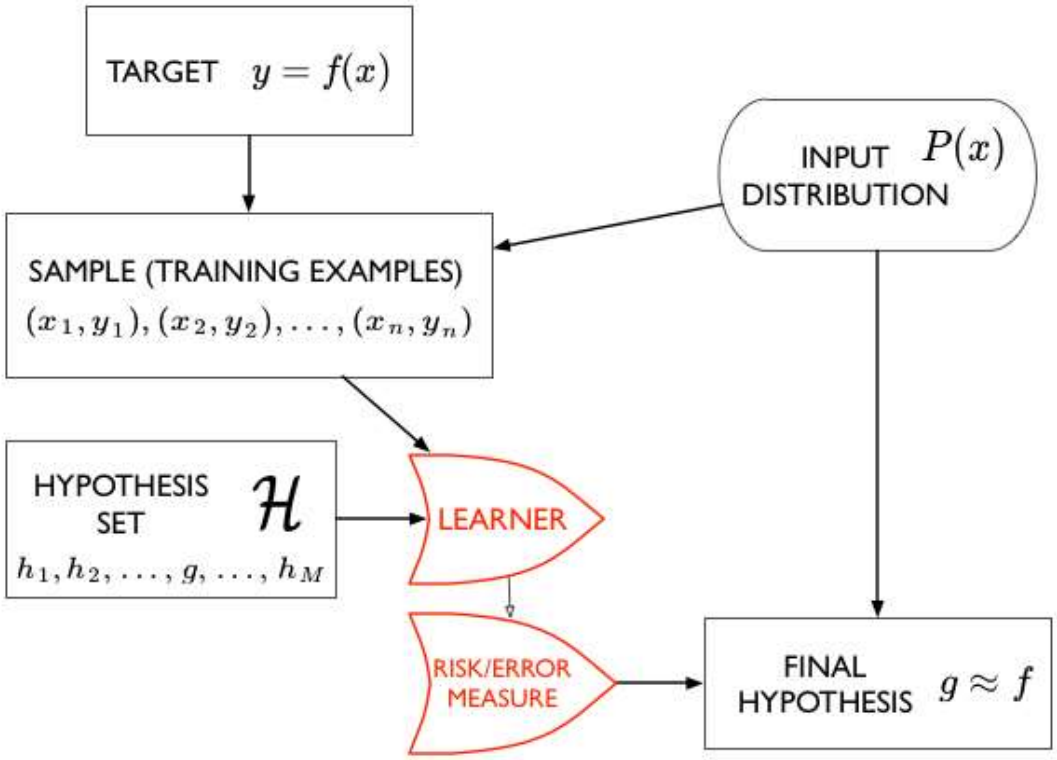
The Structure of Learning

We have a target function $f(x)$ that we do not know. But we do have a sample of data points from it, $(x_1,y_1), (x_2,y_2), …, (x_n,y_n)$. We call this the **sample** or **training examples** $\cal{D}$. We are interested in using this sample to estimate a function g to approximate the function f, and which can be used for prediction at new data points, or on the entire population, also called **out-of-sample prediction**.

Notice the way that statistics comes into this approximation problem is from the notion that we are trying to reconstruct the original function from a small-ish sample rather than a large-ish population.

To do this, we use an algorithm, called the **learner**, which chooses functions from a hypothesis set $\cal{H}$ and computes a cost measure or risk functional R (like the sum of the squared distance over all points in the data set) for each of these functions. It then chooses the function g which **minimizes** this cost measure amonst all the functions in $\cal{H}$, and thus gives us a final hypothesis g which we then use to approximate or estimate f **everywhere**, not just at the points in our data set.

Here our learner is called **Polynomial Regression**, and it takes a hypothesis space $\cal{H}_d$ of degree d polynomials, minimizes the "squared-error" risk measure, and spits out a best-fit hypothesis g_d.



Out-of-Sample and in-sample

We write $g \approx f$, or g is the **estimand** of f.In statistics books you will see g written as $\hat{f}$.

Why do we think that this might be a good idea? What are we really after?

What we'd like to do is **make good predictions**. In the language of cost, what we are really after is to minimize the cost **out-of-sample**, on the **population** at large. But this presents us with a conundrum: *how can we minimize the risk on points we havent yet seen*?

This is why we (a) minimize the risk on the set of points that we have to find g and then (b) hope that once we have found our best model g, our risk does not particularly change out-of-sample, or when using a different set of points

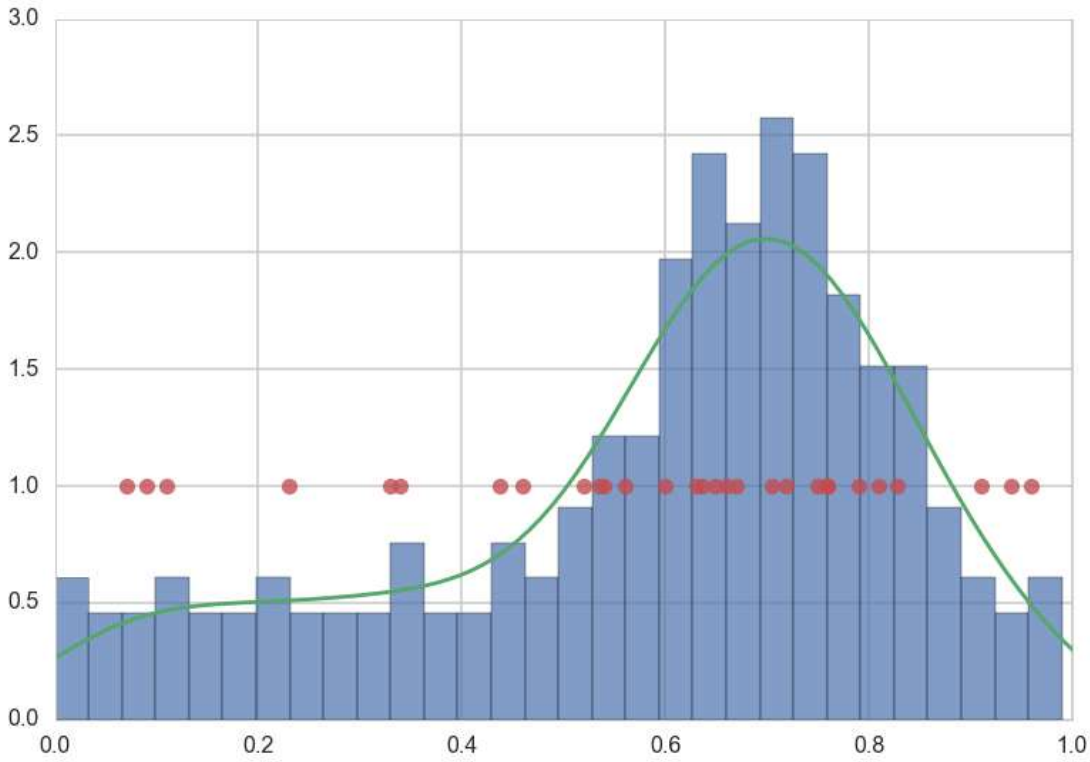
We are, as is usual in statistics, **drawing conclusions about a population from a sample**.

Intuitively, to do this, we need to ask ourselves, how representative is our sample? Or more precisely, how representative is our sample of our training points of the population (or for that matter the new x that we want to predict for)?

We illustrate this below for our population of 200 data points and our sample of 30 data points (in red).

```
plt.hist(x, normed=True, bins=30, alpha=0.7)
sns.kdeplot(x)
plt.plot(x[indexes], [1.0]*len(indexes),'o', alpha=0.8)
plt.xlim([0,1]);
```

```
//anaconda/envs/py35/lib/python3.5/site-packages/statsmodels/nonpa
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```



In our example, if we only want to use g, our estimand of f to predict for large x, or more religious counties, we would need a good sampling of points x closer to 1. And, similarly, the new x we are using to make predictions would also need to be representative of those counties. We wont do well if we try and predict low-religiousness counties from a sample of high-religiousness ones. Or, if we do want to predict over the entire range of religiousness, our training sample better cover all x well.

Our red points seem to follow our (god given) histogram well.

The relation to the Law of Large Numbers.

The process of minimization we do is called **Empirical Risk Minimization** (ERM) as we minimize the cost measure over the "empirically observed" training examples or points. But, on the assumption that we were given a training set representative of the population, ERM is just an attempt use of the law of large numbers.

What we really want to calculate is:

As usual we do not have access to the population but just some samples and thus we want to do something like:

We do not have an infinitely large training "sample". On the assumption that its representative (i.e. drawn from $p(x)$) we calculate

We could calculate the usual mean of sample means and all that, and shall see later that it is these quantities that are related to bias and variance.

Statement of the learning problem.

Once we have done that, we can then intuitively say that, if we find a hypothesis g that minimizes the cost or risk over the training set; this hypothesis *might* do a good job over the population that the training set was representative of, since the risk on the population ought to be similar to that on the training set, and thus small.

Mathematically, we are saying that:

In other words, we hope the **empirical risk estimates the out of sample risk well, and thus the out of sample risk is also small**.

Indeed, as we can see below, g_{20} does an excellent job on the population, not just on the sample.

```
#plt.plot(x[indexes],f[indexes], 's', alpha=0.6, label="in-sample'
plt.plot(x,g20(x), 'b--', alpha=0.9, lw=2, label="$g_{20}$");
plt.plot(x,f, 'o', alpha=0.2, label="population");
plt.legend(loc=4);
```