Predicting Quantitative Features: Regression

36-350, Data Mining

6 October 2008

READING: sections 6.1–6.3 and 11.1 in *Principles of Data Mining.*

We've already looked at some examples of predictive modeling in the form of classification and factor analysis, but now we'll get into it more seriously, and it will largely occupy us for the rest of the course. The place we'll begin is with predictive quantitative features, i.e., regression. The math here is not so bad, and it connects more smoothly with your previous statistics courses; having learned some lessons here we'll come back to classification, and then go to more complicated kinds of prediction.

1 Guessing the Value of a Random Variable

We have a quantitative, numerical feature, which we'll imaginatively call Y. We'll suppose that it's a random variable, and try to predict it by guessing a single value for it. (Other kinds of predictions are possible — we might guess whether Y will fall within certain limits, or the probability that it does so, or even the whole probability distribution of Y. But some lessons we'll learn here will apply to these other kinds of predictions as well.) What is the best value to guess? Or, more formally, what is the **optimal point forecast** for Y?

To answer this question, we need to pick a function to be optimized, which should measure how good the guesses are — or equivalent how bad they are, how big an error is involved. A reasonable start point is the **mean squared error**:

$$\mathrm{MSE}(a) \equiv \mathbf{E}\left[(Y-a)^2\right] \tag{1}$$

So we'd like to find the value r where MSE(a) is smallest.

$$\begin{aligned}
\mathrm{MSE}(a) &= \mathbf{E}\left[(Y-a)^2\right] & (2)\\
&= \left(\mathbf{E}[Y-a]\right)^2 + \mathrm{Var}[Y-a] & (3)\\
&= \left(\mathbf{E}[Y-a]\right)^2 + \mathrm{Var}[Y] & (4)\\
&= \left(\mathbf{E}[Y]-a\right)^2 + \mathrm{Var}[Y] & (5)\\
\frac{d\mathrm{MSE}}{da} &= 2\left(\mathbf{E}[Y]-a\right) + 0 & (6)
\end{aligned}$$

$$2(\mathbf{E}[Y] - r) = 0 \quad (7)$$
$$r = \mathbf{E}[Y] \quad (8)$$

So, if we gauge the quality of our prediction by mean-squared error, the best prediction to make is the expected value.

> EXERCISE: Suppose we use the **mean absolute error** instead of the mean squared error:
> $$\mathrm{MAE}(a) = \mathbf{E}[|Y - a|]$$
> Is this also minimized by taking $a = \mathbf{E}[Y]$? If not, what value $\tilde{r}$ minimizes the MAE? Should we use MSE or MAE to measure error?

1.1 Estimating the Expected Value

Of course, to make the prediction $\mathbf{E}[Y]$ we would have to know the expected value of Y. Typically, we do not. However, if we have sampled values, $y_1, y_2, \ldots y_n$, we can estimate the expectation from the sample mean:

$$\widehat{r} \equiv \frac{1}{n}\sum_{i=1}^{n} y_i \quad (9)$$

If the samples are IID, then the law of large numbers tells us that

$$\widehat{r} \rightarrow \mathbf{E}[Y] = r \quad (10)$$

and the central limit theorem tells us something about how fast the convergence is (namely the squared error will typically be about $\mathrm{Var}[Y]/n$).

Of course the assumption that the y_i come from IID samples is a strong one, but we can assert pretty much the same thing if they're just uncorrelated with a common expected value. Even if they are correlated, but the correlations decay fast enough, all that changes is the rate of convergence. So "sit, wait, and average" is a pretty reliable way of estimating the expectation value.

2 The Regression Function

Of course, it's not very useful to predict *just one number* for a feature. Typically, we have lots of features in our data, and we believe that there is *some* relationship between them. For example, suppose that we have data on two featured, X and Y, which might look like Figure 1. The feature Y is what we are trying to predict, a.k.a. the **dependent variable** or **output** or **response**, and X is the **predictor** or **independent variable** or **covariate** or **input**. Y might be something like the profitability of a customer and X their credit rating, or, if you want a less mercenary example, Y could be some measure of improvement in blood cholesterol and X the dose taken of a drug. Typically we won't have just one input feature X but rather many of them, but that gets harder to draw and doesn't change the points of principle.

Figure 2 shows the same data as Figure 1, only with the sample mean added on. This clearly tells us something about the data, but also it seems like we should be able to do better — to reduce the average error — by using X, rather than by ignoring it.

Let's say that the we want our prediction to be a *function* of X, namely $f(X)$. What should that function be, if we still use mean squared error? We can work this out by using the law of total expectation, a.k.a. smoothing, a.k.a. the fact that $\mathbf{E}[U] = \mathbf{E}[\mathbf{E}[U|V]]$ for any random variables U and V.

$$\begin{aligned} \mathrm{MSE}(f(X)) &= \mathbf{E}\left[(Y - f(X))^2\right] & (11)\\ &= \mathbf{E}\left[\mathbf{E}\left[(Y - f(X))^2|X\right]\right] & (12)\\ &= \mathbf{E}\left[\mathrm{Var}\left[Y|X\right] + \left(\mathbf{E}\left[Y - f(X)|X\right]\right)^2\right] & (13) \end{aligned}$$

When we want to minimize this, the first term inside the expectation doesn't depend on our prediction, and the second term looks just like our previous optimization only with all expectations conditional on X, so for our optimal function $r(x)$ we get

$$r(x) = \mathbf{E}\left[Y|X = x\right] \tag{14}$$

In other words, the (mean-squared) optimal *conditional* prediction is just the conditional expected value. The function $r(x)$ is called the **regression function**. This is what we would like to know when we want to predict Y.

2.1 Some Disclaimers

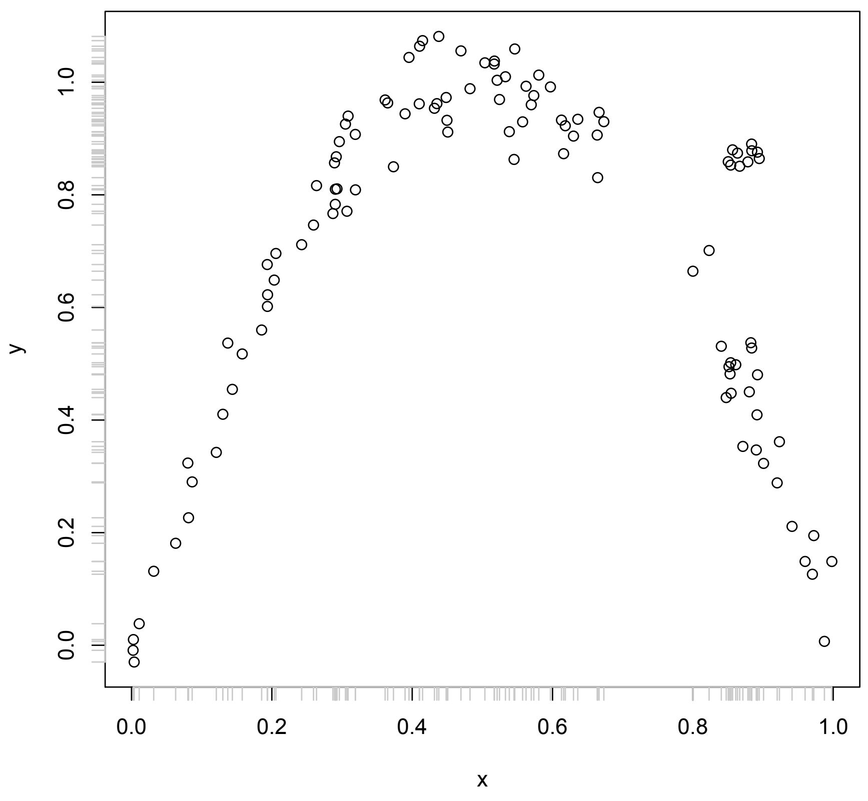
It's important to be clear on what is and is not being assumed here. Talking about X as the "independent variable" and Y as the "dependent" one suggests a causal model, which we might write

$$Y \leftarrow r(X) + \epsilon \tag{15}$$

where the direction of the arrow, $\leftarrow$, indicates the flow of cause and effect, and ϵ is some noise variable. If the gods of inference are very, very kind, then ϵ would have a fixed distribution, independent of X, and we could without loss of generality take it to have mean zero. ("Without loss of generality" because if it has a non-zero mean, we can incorporate that into $r(X)$ as an additive constant.) This is the kind of thing we saw with the factor model. However, *no* such assumption is required to get Eq. 14. It works when predicting effects from causes, or the other way around when predicting (or "retrodicting") causes from effects, or indeed when there is no causal relationship whatsoever between X and Y. It *is* always true that

$$Y|X = r(X) + \eta(X)$$

where $\eta(X)$ is a noise variable with mean zero, but as the notation indicates the distribution of the noise generally depends on X.



```
plot(all.x,all.y,xlab="x",ylab="y")
axis(1,at=all.x,labels=FALSE,col="grey")
axis(2,at=all.y,labels=FALSE,col="grey")
```

Figure 1: Scatterplot of the example data. (These are made up.) The `axis` commands add horizontal and vertical ticks to the axes to mark the location of the data (in grey so they're less strong than the main tick-marks). This isn't necessary but is often helpful. The data are in the `example.dat` file.

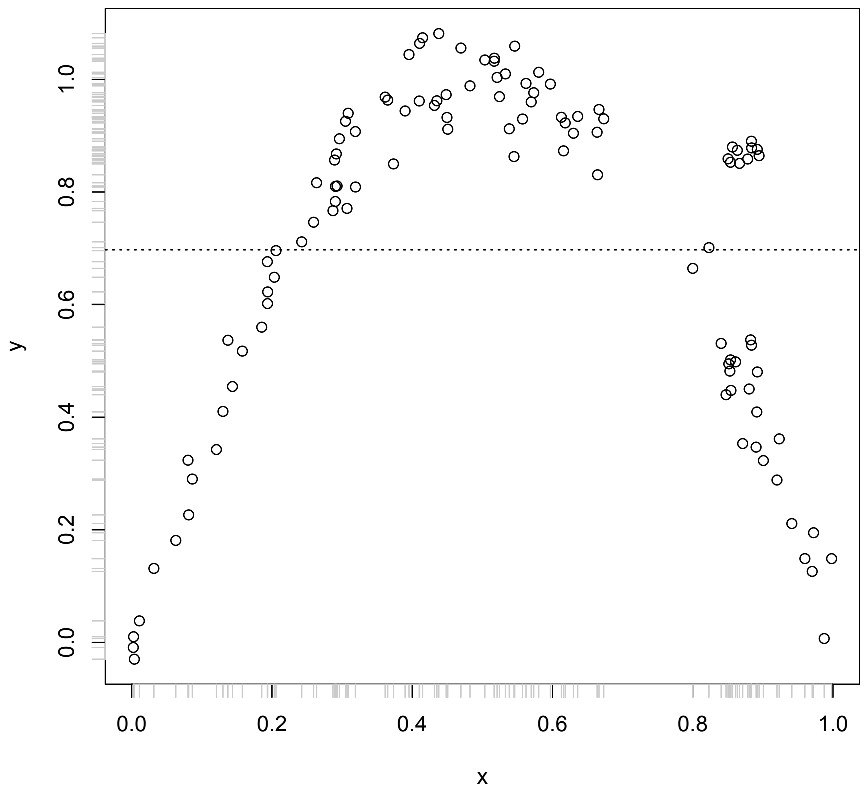


Figure 2: Data from Figure 1, with a horizontal line showing the sample mean of Y.

It's also important to be clear that when we find the regression function is a constant, $r(x) = r_0$ for all x, that this does not mean that X and Y are independent. If they are independent, then the regression function is a constant, but turning this around is the logical fallacy of "affirming the consequent".[1]

3 Estimating the Regression Function

We want to find the regression function $r(x) = \mathbf{E}[Y|X = x]$. suppose that we have a big set of pairs $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$. Then this is a *supervised* learning problem — we know the label (value of Y) for each of our n training points. How can we estimate the regression function from these training examples?

If X takes on only a finite set of values, then a simple strategy is to use the conditional sample means:

$$\widehat{r}(x) = \frac{1}{\#\{i : x_i = x\}} \sum_{i : x_i = x} y_i \tag{16}$$

By the same kind of law-of-large-numbers reasoning, we can be confident that $\widehat{r}(x) \rightarrow \mathbf{E}[Y|X = x]$.

Unfortunately, this *only* works if X has only a finite set of values. If X is continuous, then in general the probability of our getting a sample at any *particular* value is zero, is the probability of getting *multiple* samples at *exactly* the same value of x. This is a basic issue with estimating any kind of function from data — the function will always be **undersampled**, and we need to fill in between the values we see. We also need to somehow take into account the fact that each y_i is a *sample* from the conditional distribution of $Y|X = x_i$, and so is not generally equal to $\mathbf{E}[Y|X = x_i]$. So any kind of function estimation is going to involve interpolation, extrapolation, and smoothing.

Different methods of estimating the regression function — different regression methods, for short — involve different choices about how we interpolate, extrapolate and smooth. This involves our making a choice about how to approximate $r(x)$ by a limited class of functions which we know (or at least hope) we can estimate. There is no guarantee that our choice leads to a *good* approximation in the case at hand, though it is sometimes possible to say that the approximation error will shrink as we get more and more data. (Remember our discussion of bias and variance last lecture.)

For example, we could decide to approximate $r(x)$ by a constant r_0. The implicit smoothing here is very strong, but sometimes appropriate. For instance, it's appropriate when $r(x)$ really is a constant; then trying to estimate any additional structure in the regression function is just so much wasted effort. Alternate, if $r(x)$ is *nearly* constant, we may still be better off approximating it

[1] As in combining the fact that all human beings are featherless bipeds, and the observation that a cooked turkey is a featherless biped, to conclude that cooked turkeys are human beings. An econometrician stops there; an econometrician who wants to be famous writes a best-selling book about how this proves that Thanksgiving is really about cannibalism.

as one. For instance, suppose the true $r(x) = r_0 + asin(\nu x)$, where $a \ll 1$ and $\nu \gg 1$ (Figure 3 shows an example). With limited data, we can actually get better predictions by estimating a constant regression function than one with the correct functional form.

3.1 Ordinary Least Squares Linear Regression as Smoothing

Let's revisit ordinary least-squares linear regression from this point of view. Let's assume that the independent variable X is one-dimensional, and that both X and Y are centered (i.e. have mean zero) — neither of these assumptions is really necessary, but they reduce the book-keeping.

We are making the *choice* to approximate $r(x)$ by $\alpha + \beta x$, and ask for the best values a, b of those constants. These will be the ones which minimize the mean-squared error.

$$\begin{aligned} MSE(\alpha, \beta) &= \mathbf{E}\left[(Y - \alpha - \beta X)^2\right] && (17)\\ &= \mathbf{E}\left[(Y - \alpha - \beta X)^2 | X\right] && (18)\\ &= \mathbf{E}\left[\mathrm{Var}\left[Y|X\right] + \left(\mathbf{E}\left[Y - \alpha - \beta X|X\right]\right)^2\right] && (19)\\ &= \mathbf{E}\left[\mathrm{Var}\left[Y|X\right]\right] + \mathbf{E}\left[\left(\mathbf{E}\left[Y - \alpha - \beta X|X\right]\right)^2\right] && (20) \end{aligned}$$

The first term doesn't depend on α or β, so we can drop it for purposes of optimization. Taking derivatives, and then brining them inside the expectations,

$$\begin{aligned} \frac{\partial MSE}{\partial \alpha} &= \mathbf{E}\left[2(Y - \alpha - \beta X)(-1)\right] && (21)\\ \mathbf{E}\left[Y - a - bX\right] &= 0 && (22)\\ a &= \mathbf{E}\left[Y\right] - b\mathbf{E}\left[X\right] = 0 && (23) \end{aligned}$$

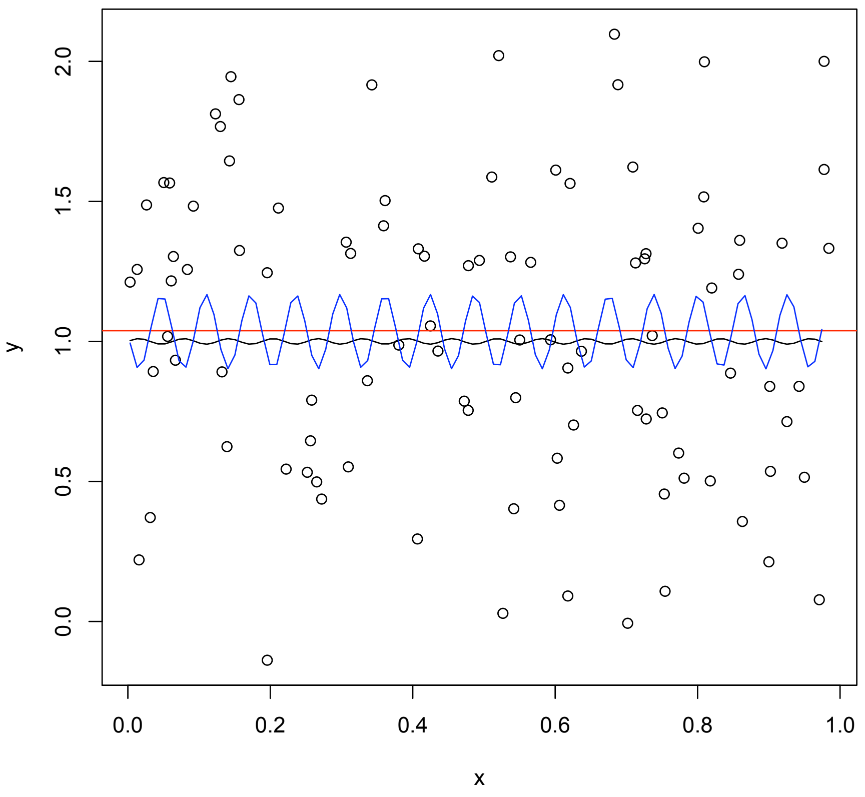
using the fact that X and Y are centered; and,

$$\begin{aligned} \frac{\partial MSE}{\partial \beta} &= \mathbf{E}\left[2(Y - \alpha - \beta X)(-X)\right] && (24)\\ \mathbf{E}\left[XY\right] - b\mathbf{E}\left[X^2\right] &= 0 && (25)\\ b &= \frac{\mathrm{Cov}\left[X, Y\right]}{\mathrm{Var}\left[X\right]} && (26) \end{aligned}$$

again using the centering of X and Y. That is, the mean-squared optimal linear prediction is

$$r(x) = x\frac{\mathrm{Cov}\left[X, Y\right]}{\mathrm{Var}\left[X\right]} \qquad (27)$$

Now, if we try to estimate this from data, there are two approaches. One is to replace the true population values of the covariance and the variance with



```
ugly.func = function(x) {1 + 0.01*sin(100*x)}
r = runif(100)
r.y = ugly.func(r) + rnorm(length(r),0,0.5)
plot(r,r.y,xlab="x",ylab="y")
curve(ugly.func,add=TRUE)
abline(h=mean(r.y),col="red")
sine.fit = lm(r.y ~ 1+ sin(100*r))
curve(sine.fit$coefficients[1]+sine.fit$coefficients[2]*sin(100*x),
      col="blue",add=TRUE)
```

Figure 3: A rapidly-varying but nearly-constant regression function; $Y = 1 + 0.01 \sin 100x + \epsilon$, with $\epsilon \sim \mathcal{N}(0, 0.1)$. (The x values are uniformly distributed between 0 and 1.) Red: constant line at the sample mean. Blue: estimated function of the same form as the true regression function, i.e., $a + b \sin 100x$. With enough data, it's better to estimate the true form of the function, but with limited data the constant may actually generalize better — the bias of using the wrong functional form is smaller than the additional variance from the extra degrees of freedom. In this case, the RMS error of the constant on new data is about 0.50, while that of the estimated sine function is 0.51 — in other words the extra predictive power of estimating the correct function is slightly negative!

their sample values, respectively

$$\frac{1}{n}\sum_i y_i x_i$$

and

$$\frac{1}{n}\sum_i x_i^2$$

(again, assuming centering). The other is to minimize the residual sum of squares,

$$RSS(\alpha,\beta) \equiv \sum_i (y_i - \alpha - \beta x_i)^2 \tag{28}$$

You may or may not find it surprising that both approaches lead to the same answer:

$$\begin{aligned} \widehat{a} &= 0 & (29)\\ \widehat{b} &= \frac{\sum_i y_i x_i}{\sum_i x_i^2} & (30) \end{aligned}$$

Provided that $\operatorname{Var}[X] > 0$, this will converge with IID samples, so we have a consistent estimator.[2]

> EXERCISE: We derived Eqs. 29 and 30 by minimizing Eq. 28 in class. Can you repeat the derivation without looking at your notes?

We are now in a position to see how the least-squares linear regression model is really a smoothing of the data. Let's write the estimated regression function explicitly in terms of the training data points.

$$\begin{aligned} \widehat{r}(x) &= \widehat{b}x & (31)\\ &= x\frac{\sum_i y_i x_i}{\sum_i x_i^2} & (32)\\ &= \sum_i y_i \frac{x_i}{\sum_j x_j^2} x & (33)\\ &= \sum_i y_i \frac{x_i}{n s_X^2} x & (34) \end{aligned}$$

where s_X^2 is the sample variance of X. In words, our prediction is a weighted average of the observed values y_i of the dependent variable, where the weights are proportional to how far x_i is from the center, relative to the variance, and proportional to the magnitude of x. If x_i is on the same side of the center as

[2]Eq. 29 may look funny, but remember that we're assuming X and Y have been centered. Centering doesn't change the slope of the least-squares line but does change the intercept; if we go back to the un-centered variables the intercept becomes $\overline{Y} - \widehat{b}\overline{X}$, where the bar denotes the sample mean.

x, it gets a positive weight, and if it's on the opposite side it gets a negative weight.

Figure 4 shows the data from Figure 1 with the least-squares regression line added. It will not escape your notice that this is very, very slightly different from the constant regression function; the coefficient on X is 0.004438. Visually, the problem is that there should be a positive slope in the left-hand half of the data, and a negative slope in the right, but the slopes are the densities are balanced so that the best *single* slope is zero.[3]

Mathematically, the problem arises from the somewhat peculiar way in which least-squares linear regression smoothes the data. As I said, the weight of a data point depends on how far it is from the *center* of the data, not how far it is from the *point at which we are trying to predict*. This works when $r(x)$ really is a straight line, but otherwise — e.g., here — it's a recipe for trouble. However, it does suggest that if we could somehow just tweak the way we smooth the data, we could do better than linear regression.

4 Linear Smoothers

The sample mean and the linear regression line are both special cases of **linear smoothers**, which are estimates of the regression function with the following form:

$$\widehat{r}(x) = \sum_i y_i \widehat{w}(x_i, x) \tag{35}$$

The sample mean is the special case where $widehatw(x_i, x) = 1/n$, regardless of what x_i and x are.

Ordinary linear regression is the special case where $\widehat{w}(x_i, x) = (x_i/ns_X^2)x$.

Both of these, as remarked, ignore how far x_i is from x.

4.1 k-Nearest-Neighbor Regression

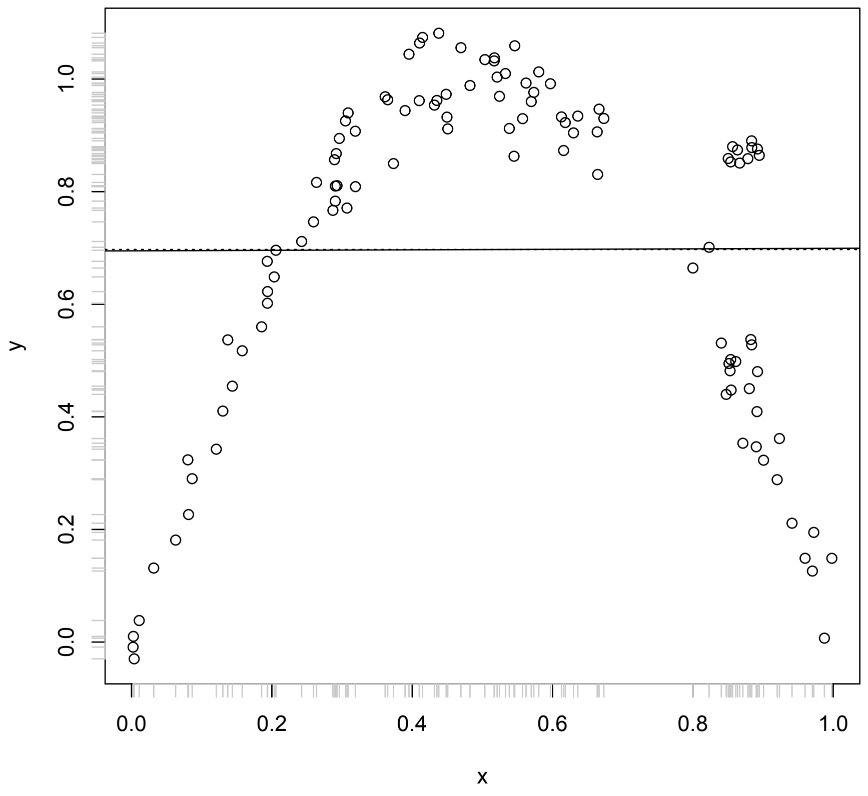
At the other extreme, we could do **nearest-neighbor regression**:

$$\widehat{w}(x_i, x) = \begin{cases} 1 & x_i \text{ nearest neighbor of } x \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

This is very sensitive to the distance between x_i and x. If $r(x)$ does not change too rapidly, and X is pretty thoroughly sampled, then the nearest neighbor of x among the x_i is probably close to x, so that $r(x_i)$ is probably close to $r(x)$. However, $y_i = r(x_i) + \text{noise}$, so nearest-neighbor regression will include the noise into its prediction. We might instead do k-nearest neighbor regression,

$$\widehat{w}(x_i, x) = \begin{cases} 1/k & x_i \text{ one of the } k \text{ nearest neighbors of } x \\ 0 & \text{otherwise} \end{cases} \tag{37}$$

[3]The standard test of whether this coefficient is zero is about as far from rejecting the null hypothesis as you will ever see, $p = 0.965$. You should remember this the next time you look at regression output.



```
abline(h=mean(all.y),lty=2)
fit.all = lm(all.y~all.x)
abline(a=fit.all$coefficients[1],b=fit.all$coefficients[2],col="blue")
```

Figure 4: Data from Figure 1, with a horizontal line at the mean (dotted) and the ordinary least squares regression line (solid). If you zoom in online you will see that there really are two lines there.

Again, with enough samples all the k nearest neighbors of x are probably close to x, so their regression functions there are going to be close to the regression function at x. But because we average their values of y_i, the noise terms should tend to cancel each other out. As we increase k, we get smoother functions — in the limit $k = n$ and we just get back the constant. Figure 5 illustrates this for our running example data.[4]

To use k-nearest-neighbors regression, we need to pick k somehow. This means we need to decide *how much* smoothing to do, and this is not trivial. We will return to this point.

Because k-nearest-neighbors averages over only a fixed number of neighbors, each of which is a noisy sample, it always has some noise in its prediction, and is generally not consistent. This may not matter very much with moderately-large data (especially once we have a good way of picking k). However, it is sometimes useful to let k systematically grow with n, but not too fast, so as to avoid just doing a global average; say $k \propto \sqrt{n}$. Such schemes *can* be consistent.

4.2 Kernel Smoothers

Changing k in a k-nearest-neighbors regression lets us change how much smoothing we're doing on our data, but it's a bit awkward to express this in terms of a number of data points. It feels like it would be more natural to talk about a range in the independent variable over which we smooth or average. Another problem with k-NN regression is that each testing point is predicted using information from only a few of the training data points, unlike linear regression or the sample mean, which always uses all the training data. If we could somehow use all the training data, but in a location-sensitive way, that would be nice.

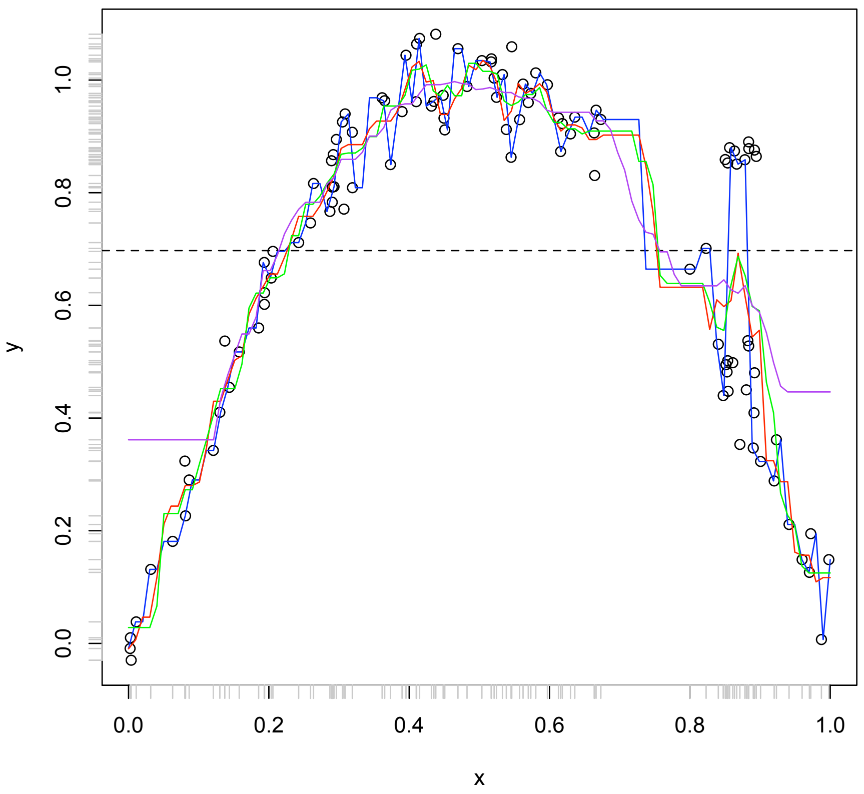
There are several ways to do this, as we'll see, but a particularly useful one is to use a **kernel smoother**, a.k.a. **kernel regression** or **Nadaraya-Watson regression**. To begin with, we need to pick a **kernel function**[5] $K(x_i, x)$ which satisfies the following properties:

1. $K(x_i, x) \geq 0$
2. $K(x_i, x)$ depends only on the distance $x_i - x$, not the individual arguments
3. $\int x K(0, x) dx = 0$
4. $0 < \int x^2 K(0, x) dx < \infty$

These conditions together (especially the last one) imply that $K(x_i, x) \to 0$ as $|x_i - x| \to \infty$. Two examples of such functions are the density of the

[4] The code uses the k-nearest neighbor function provided by the package `knnflex` (available from CRAN). This requires one to pre-compute a matrix of the distances between all the points of interest, i.e., training data and testing data (using `knn.dist`); the `knn.predict` function then needs to be told which rows of that matrix come from training data and which from testing data. See `help(knnflex.predict)` for more, including examples.

[5] There are many other mathematical objects which are *also* called "kernels". Some of these meanings are related, but not all of them. (Cf. "normal".)



```
library(knnflex)
all.dist = knn.dist(c(all.x,seq(from=0,to=1,length.out=100)))
all.nn1.predict = knn.predict(1:110,111:210,all.y,all.dist,k=1)
abline(h=mean(all.y),lty=2)
lines(seq(from=0,to=1,length.out=100),all.nn1.predict,col="blue")
all.nn3.predict = knn.predict(1:110,111:210,all.y,all.dist,k=3)
lines(seq(from=0,to=1,length.out=100),all.nn3.predict,col="red")
all.nn5.predict = knn.predict(1:110,111:210,all.y,all.dist,k=5)
lines(seq(from=0,to=1,length.out=100),all.nn5.predict,col="green")
all.nn20.predict = knn.predict(1:110,111:210,all.y,all.dist,k=20)
lines(seq(from=0,to=1,length.out=100),all.nn20.predict,col="purple")
```

Figure 5: Data points from Figure 1 with horizontal dashed line at the mean and the k-nearest-neighbor regression curves for $k = 1$ (blue), $k = 3$ (red), $k = 5$ (green) and $k = 20$ (purple). Note how increasing k smoothes out the regression line, and pulls it back towards the mean. ($k = 100$ would give us back the dashed horizontal line.)

Unif$(-h/2, h/2)$ distribution, and the density of the standard Gaussian $\mathcal{N}(0, \sqrt{h})$ distribution. Here h can be any positive number, and is called the **bandwidth**.

The Nadaraya-Watson estimate of the regression function is

$$\widehat{r}(x) = \sum_i y_i \frac{K(x_i, x)}{\sum_j K(x_j, x)} \tag{38}$$

i.e., in terms of Eq. 35,

$$\widehat{w}(x_i, x) = \frac{K(x_i, x)}{\sum_j K(x_j, x)} \tag{39}$$

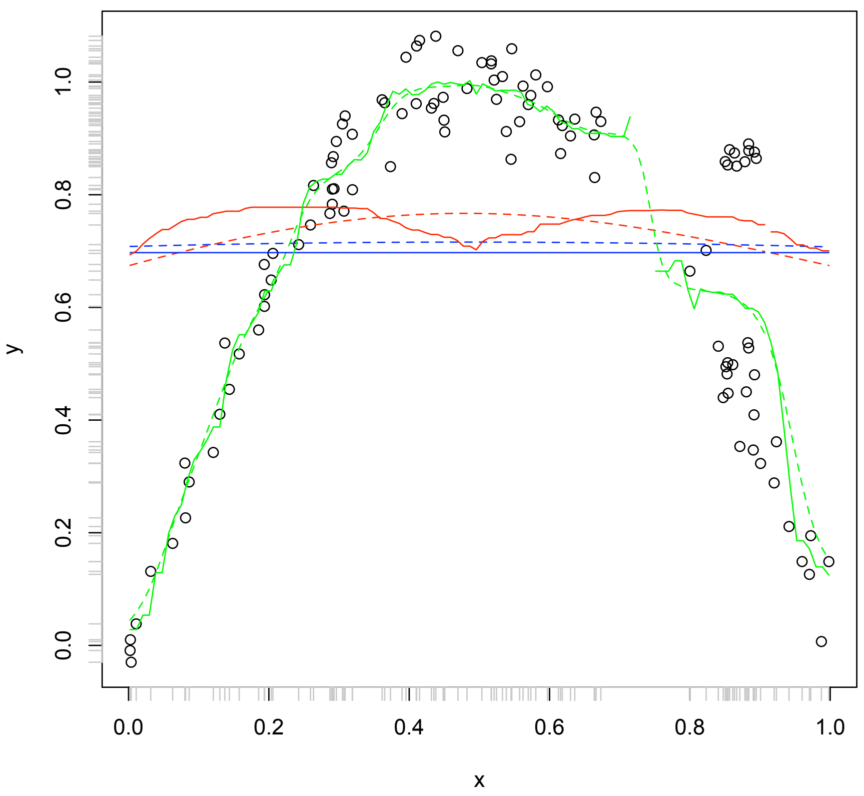
(Notice that here, as in k-NN regression, the sum of the weights is always 1. Why?)

What does this achieve? Well, $K(x_i, x)$ is large if x_i is close to x, so this will place a lot of weight on the training data points close to the point where we are trying to predict. More distant training points will have smaller weights, falling off towards zero. If we try to predict at a point x which is very far from any of the training data points, we will about the same very small number for all the $K(x_i, x)$, and so $\widehat{w}(x_i, x) \approx 1/n$. That is, far from the training data our prediction will tend towards the sample mean, rather than going off to $\pm\infty$, as linear regression's predictions do. (This assumes that we're using a kernel like the Gaussian, which never quite goes to zero, unlike the box kernel.) Whether this is good or bad of course depends on the true $r(x)$ — and how often we have to predict what will happen very far from the training data.

Figure 6 shows our running example data, together with kernel regression estimates formed by combining the uniform-density, or **box**, and Gaussian kernels with different bandwidths. The box kernel simply takes a region of width h around the point x and averages the training data points it finds there. The Gaussian kernel gives reasonably large weights to points within h of x, smaller ones to points within $2h$, tiny ones to points within $3h$, and so on, shrinking like $e^{-(x-x_i)^2/2h}$. As promised, the bandwidth h controls the degree of smoothing. As $h \to \infty$, we revert to taking the global mean. As $h \to 0$, we tend to get spikier functions — with the Gaussian kernel at least it tends towards the nearest-neighbor regression. (EXERCISE: Why is that?)

If we want to use kernel regression, we need to choose both which kernel to use, and the bandwidth to use with it. Experience, like Figure 6, suggests that the bandwidth usually matters a lot more than the kernel. This puts us back to roughly where we were with k-NN regression, needing to control the degree of smoothing, without knowing how smooth $r(x)$ really is. Similarly again, with a fixed bandwidth h, kernel regression is generally not consistent. However, if $h \to 0$ as $n \to \infty$, but doesn't shrink *too* fast, then we can get consistency.

Next time, we'll look more at linear regression and some extensions, and then come back to nearest-neighbor and kernel regression, and say something about how to handle things like the blob of data points around $(0.9, 0.9)$ in the scatter-plot.



```
plot(all.x,all.y,xlab="x",ylab="y")
axis(1,at=all.x,labels=FALSE)
axis(2,at=all.y,labels=FALSE)
lines(ksmooth(all.x, all.y, "normal", bandwidth=2),col="blue",lty=2)
lines(ksmooth(all.x, all.y, "normal", bandwidth=1),col="red",lty=2)
lines(ksmooth(all.x, all.y, "normal", bandwidth=0.1),col="green",lty=2)
lines(ksmooth(all.x, all.y, "box", bandwidth=2),col="blue")
lines(ksmooth(all.x, all.y, "box", bandwidth=1),col="red")
lines(ksmooth(all.x, all.y, "box", bandwidth=0.1),col="green")
```

Figure 6: Data from Figure 1 together with kernel regression lines. Solid colored lines are box-kernel estimates, dashed colored lines Gaussian-kernel estimates. Blue, $h = 2$; red, $h = 1$; green, $h = 0.5$; purple, $h = 0.1$ (per the definition of bandwidth in the `ksmooth` function). Note the abrupt jump around $x = 0.75$ in the box-kernel/$h = 0.1$ (solid purple) line — with a small bandwidth the box kernel is unable to interpolate smoothly across the break in the training data, while the Gaussian kernel can.