k-Nearest Neighbors

36-462/662, Spring 2022

22 February 2022 (Lecture 11)

Contents

1 Setting: Prediction, including both classification and regression

Let's fix our setting. We have a data set of n **data-points**, **items**, **cases** or **units**. Each item is **represented** by a vector of **variables** or **features**, which we have somehow decided are the important information we want to keep track of about the item, generally in numerical form. There are p of these features we want to use as **predictors**. Following the usual notation for regression courses, we'll write this as an $n \times p$ matrix $\mathbf{x}$; the vector for data-point or item i will be $\vec{x}_i$. Beyond these features, we have an additional variable for each item that we want to **predict**, based on the features. We'll write it y_i for data-point i, compiled into the $n \times 1$ matrix $\mathbf{y}$ (again, this is regression notation). This variable is called the **label**, **outcome**, **target**, **output** or (oddly) **dependent variable** (sometimes the **predictand** ["thing to be predicted" in Latin] or **regressand**).

A prediction here is going to be a function of the features which outputs a guess ("point prediction") about the outcome or label.

- **Regression**: Y is a continuous numerical variable, so the **regression function** should map $\vec{x}_0$ to a number.
- **Classification**: Y is binary, so the **classification rule** should map $\vec{x}_0$ to 0 or 1.
  - Multi-class classification works similarly but with more notation, which I don't feel like getting into.
  - You can always reduce binary classification to regression: use your favorite regression method, and then threshold the prediction for Y, saying "1" if the prediction is ≥ 0.5 and saying "0" otherwise. (This is basically what k-nearest-neighbor classification will do for us.) This may be less efficient than using some method directly built for classification, especially if your regression method doesn't realize that probabilities should be between 0 and 1, but it's always an *option*.

1.1 Prediction quality, risk, optimal risk and optimal predictors

How good is the guess?[1]

- Regression: we usually pick the squared error loss function, so we usually measure the quality of a regression with **expected squared error**. That is, $\mathbb{E}\left[(Y - m(\vec{X}))^2\right]$ indicates how bad the function m is.
- Classification: we often pick the 0-1 loss function, so we usually measure the quality of a classifier with its **inaccuracy** or **error rate**. That is, $\Pr\left(Y \neq m(\vec{X})\right)$ indicates how bad the function m is.

This expected loss on new data is called the **risk**. In predictive modeling, we want to learn functions with low risk.

There's an optimal function, i.e., one which has a lower risk than any other function.

- Regression: The optimal function is $\mu(\vec{x}) = \mathbb{E}\left[Y|\vec{X} = \vec{x}\right]$. This is called the **true regression function**, or sometimes the **optimal regression function**.
- Classification: The optimal function depends on the conditional probability $\Pr\left(Y = 1|\vec{X} = \vec{x}\right) \equiv p(\vec{x})$. The function is $c(\vec{x}) = 1$ if $p(\vec{x}) \geq 0.5$ and $c(\vec{x}) = 0$ otherwise. (Or, as we know from the homework, $c(\vec{x}) = 1$ if $p(\vec{x} > 0.5$ and $c(\vec{x}) = 0$ otherwise.) This is called the **optimal classifier**[2].

Even the optimal function will not, in general, have zero risk.

[1] We have gone over a lot of this in lecture 3 and homework 2, but it doesn't hurt to have reminders, especially in this particular context.

[2] Some people call the optimum prediction function or prediction rule the **Bayes rule**, even though it has nothing to do with Bayesian inference, or with Thomas Bayes, and only the most tenuous connection to Bayes's rule (the fact that $\Pr(A|B) = \Pr(B|A)\Pr(A)/\Pr(B)$). (The name goes back to the early days of statistical decision theory, where some of the pioneers were *also* partisans of "Bayesian statistics", the idea that all statistical inference should be done by using Bayes's rule.)

- Regression: Suppose $Y = \mu(\vec{X}) + \epsilon$, where the noise ϵ has $\mathbb{E}\left[\epsilon|\vec{X}\right] = 0$, $\text{Var}\left[\epsilon|\vec{X} = \vec{x}\right] = \sigma^2(\vec{x})$. (In your linear-regression class, you would have assumed this, *plus* that σ^2 is constant, and much else besides.) Then the risk of the true regression function is $\mathbb{E}\left[\sigma^2(\vec{X})\right] > 0$. (Why?)
- Classification: Suppose $p(\vec{x})$ isn't either 0 or 1 everywhere. Then the probability of mis-classifying at $\vec{x}$ is $p(\vec{x})$ if $p(\vec{x}) < 0.5$ (because there $c(\vec{x}) = 0$), and $1 - p(\vec{x})$ if $p(\vec{x}) \geq 0.5$. A little thought shows that we can write the conditional probability in a unified way[3] as $\min\{p(\vec{x}), 1 - p(\vec{x})\}$. The risk of the optimal classifier is then $\mathbb{E}\left[\min\left\{p(\vec{X}), 1 - p(\vec{X})\right\}\right]$. This minimal risk will be > 0 unless $p(\vec{x}) = 0$ or $= 1$ almost everywhere. (We saw this in homework 2.)

Unfortunately, the optimal function depends on the true distribution generating the data. So does the risk of the optimal function. What we want, then, is some way of estimating a function from the data which can learn what the true function is, or at least learn to predict almost as well as the true function, without having to know in advance too much about that function.

2 Nearest neighbors as a predictor

This is where nearest neighbors comes in.

In this context, "distance" always refers to distances between the p-dimensional feature vectors. The **nearest neighbor** of a vector $\vec{x}_0$ is the $\vec{x}_i$ closest to it. The k nearest neighbors are the k vectors $\vec{x}_i$ closest to $\vec{x}_0$. (Notice that these definitions make sense whether or not $\vec{x}_0$ is also one of the $\vec{x}_i$.) We will often need a way of keeping track of the indices of the neighbors, so we'll write $NN(\vec{x}_0, j)$ for the index of the j^{th} nearest neighbor of $\vec{x}_0$. Thus $NN(\vec{x}_0, 1)$ is a number between 1 and n, but $\vec{x}_{NN(\vec{x}_0,1)}$ is a p-dimensional vector.

The k-nearest-neighbor estimate of the regression function is then the average value of the response over the k nearest neighbors:

$$\hat{\mu}(\vec{x}_0) = \frac{1}{k}\sum_{j=1}^{k} y_{NN(\vec{x}_0,j)} \tag{1}$$

For classification, we similarly average the labels of neighbors to estimate $p(\vec{x}_0)$,

$$\hat{p}(\vec{x}_0) = \frac{1}{k}\sum_{j=1}^{k} y_{NN(\vec{x}_0,j)} \tag{2}$$

and then threshold it:

$$\hat{c}(\vec{x}_0) = \mathbf{1}(\hat{p}(\vec{x}_0) \geq 0.5) \tag{3}$$

[3] An alternative expression would be $\frac{1}{2} - \left|p(\vec{x}) - \frac{1}{2}\right|$, but this will be less useful later on.

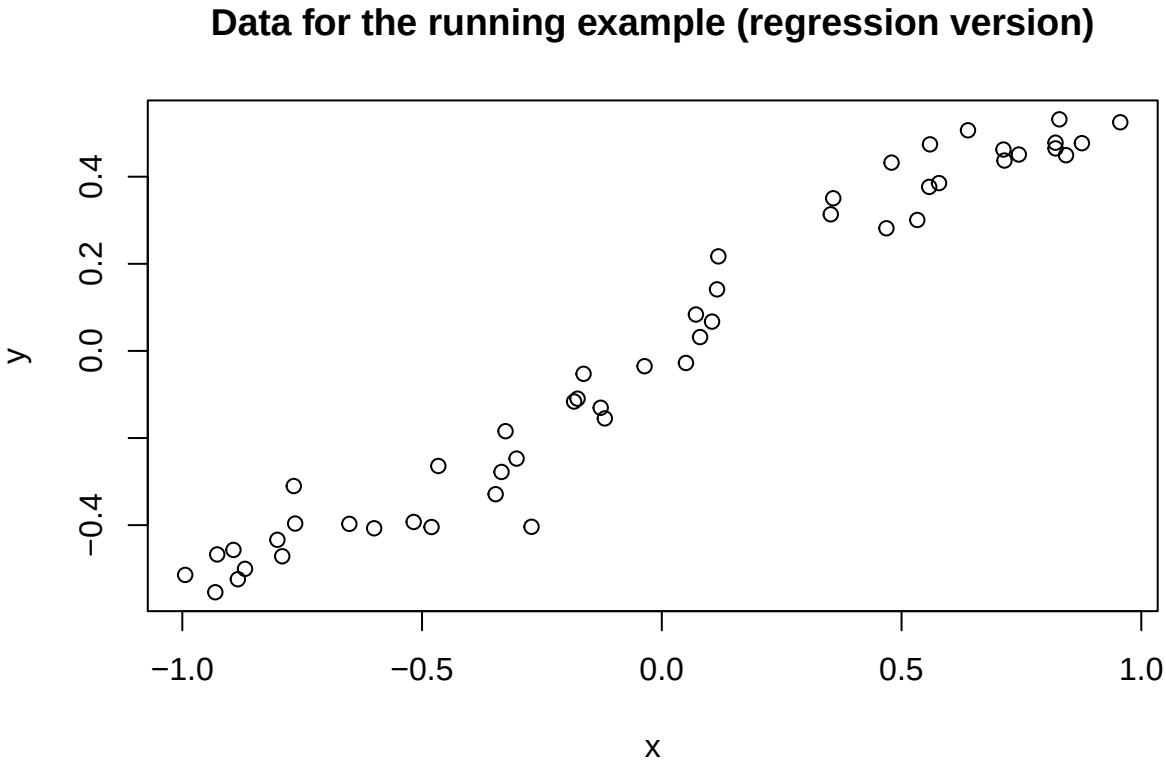


Figure 1: Simulation data for the regression version of our running example. Can you guess the true regression function, without looking at the code?

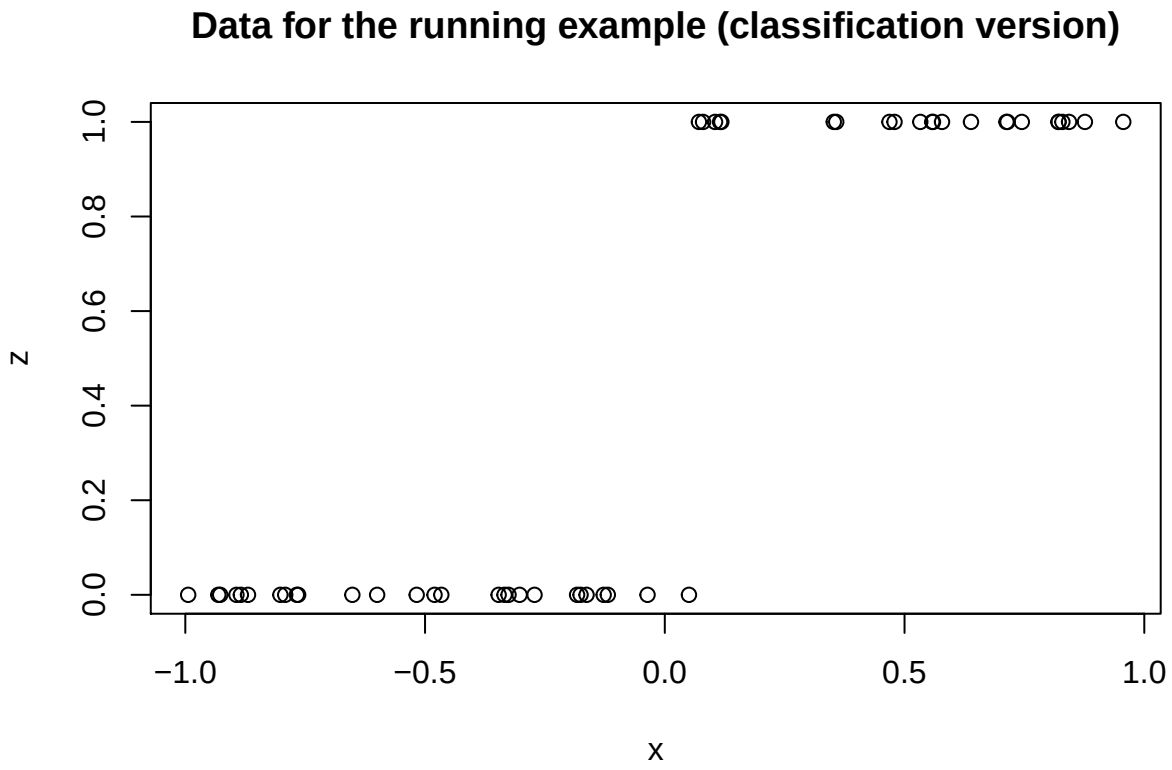


Figure 2: Simulation data for the classification version of our running example. Can you guess the rule assigning the class labels, without looking at the code?

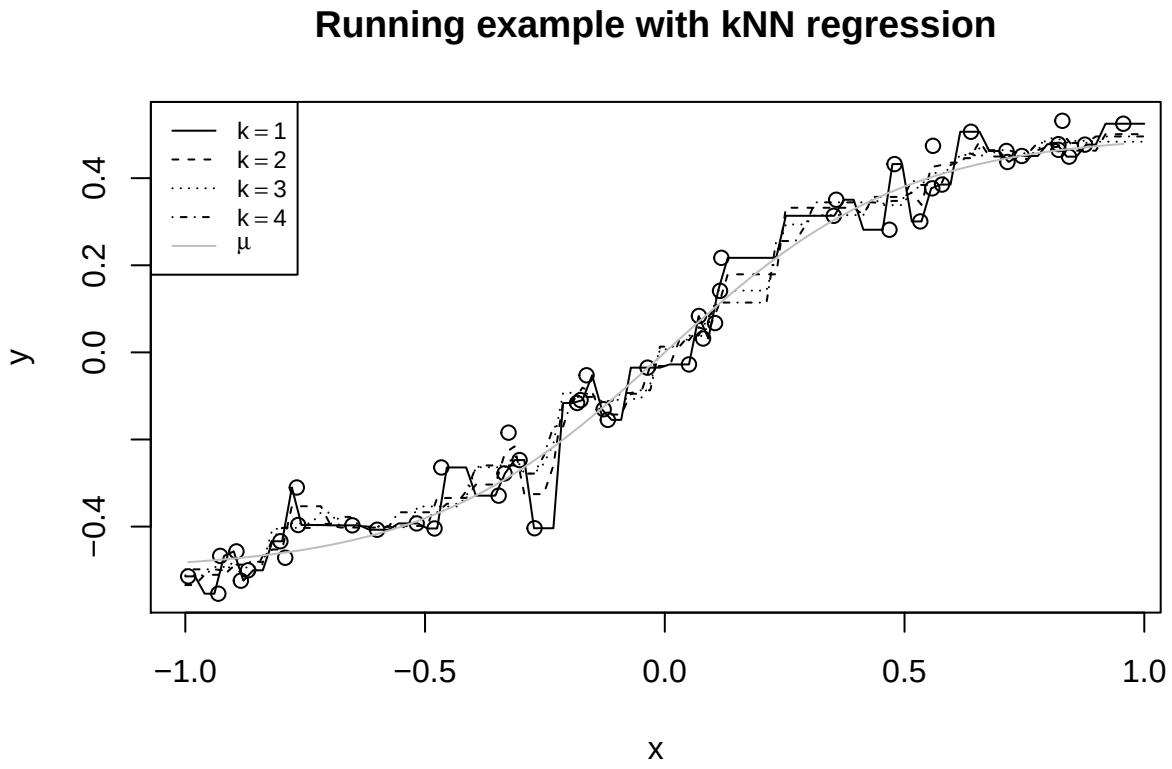


Figure 3: Our running example data, together with four different kNN estimates of the regression function and the true regression function (μ).

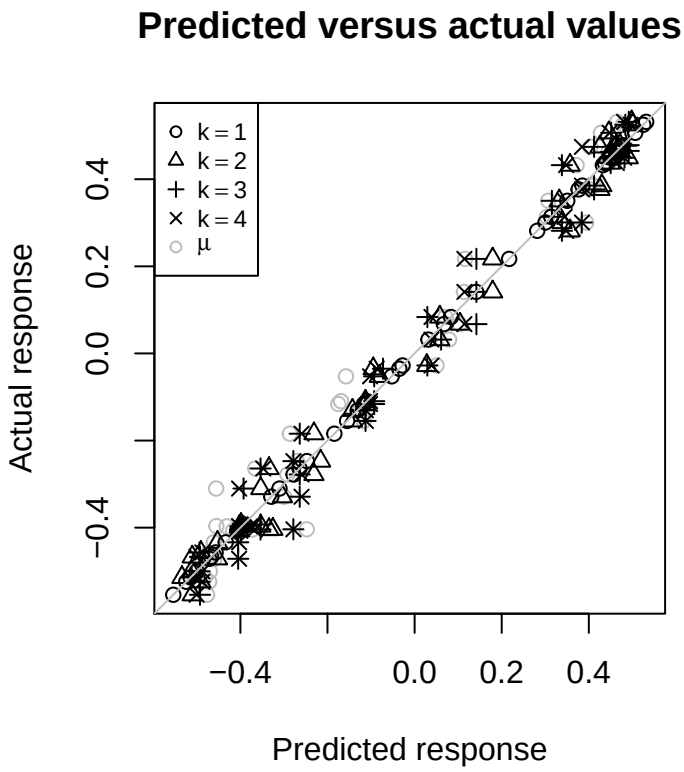


Figure 4: Predicted versus actual responses for kNN regression, $k \in 1:4$, plus the true regression function (μ), and a diagonal line as a guide to the eye.

3 Analysis of 1-Nearest-Neighbors for Learning Noise-Free Functions

There are lots of estimation methods we *could* use. To decide on using this one, nearest neighbors, we should have some reason to think it will predict well. This is where theory comes in.

Start with the simplest, most extreme setting, to build ideas. We'll assume that there is *no noise* in the outcomes (responses, labels). This means for regression that $y_i = \mu(\vec{x}_i)$, and for classification that $y_i = c(\vec{x}_i)$. If nearest neighbors can't learn to predict here, it's got to be toast; if it can, we'll add noise back in. Let's also make our lives simple by only looking at 1-nearest-neighbors, $k = 1$. To simplify notation, I'll write NN as the index for the nearest neighbor of $\vec{x}_0$, leaving the dependence on $\vec{x}_0$ implicit.

In this setting — no noise, $k = 1$ — the error nearest neighbors will make for regression at $\vec{x}_0$ will be

$$\mu(\vec{x}_0) - y_N N = \mu(\vec{x}_0) - \mu(x_{NN}) \tag{4}$$

so the risk will be

$$\mathbb{E}\left[(\mu(\vec{X}) - \mu(\vec{X}_{NN}))^2\right] \tag{5}$$

Similarly, for classification, the risk will be

$$\Pr\left(c(\vec{X}) \neq c(\vec{X}_{NN})\right) \tag{6}$$

We'd like these risks to go to 0 as $n \to \infty$ (because here the optimal risk *is* zero). The equations above suggests that for regression we want the true function to be continuous, but for classification we want it to be piecewise constant. (In fact, piecewise continuity is usually enough for regression.) But even with this, we need to see that $\vec{x}_{NN} \to \vec{x}_0$, otherwise continuity won't help.

3.1 Convergence of the nearest neighbor

Requiring $\vec{x}_{NN} \to \vec{x}_0$ is the same as requiring that $\|\vec{x}_{NN} - \vec{x}_0\| \to 0$. When will this happen?

Well, pick some positive distance $d > 0$. What is the probability that $\|\vec{x}_{NN} - \vec{x}_0\| > d$? Ideally, we'd like this to go to zero as $n \to \infty$, no matter how small that d might be; that would indicate that the nearest neighbor is approaching the point of interest.

A little thought should convince you that the nearest neighbor is more than d away from $\vec{x}_0$ if and only if every $\vec{x}_i$ is more than d away. So

$$\Pr\left(\|\vec{x}_{NN} - \vec{x}_0\| > d\right) = \Pr\left(\forall i, \|\vec{x}_i - \vec{x}_0\| > d\right) \tag{7}$$

At this point, we need to make an assumption about the feature vectors. We'll assume they're IID. Then the probability of *all* the feature vectors doing the same thing (being far from $\vec{x}_0$) turns into the product of *each* of them doing that thing:

$$\begin{aligned}
\Pr\left(\|\vec{X}_{NN} - \vec{x}_0\| > d\right) &= \Pr\left(\forall i, \|\vec{X}_i - \vec{x}_0\| > d\right) && (8)\\
&= \prod_{i=1}^{n} \Pr\left(\|\vec{X}_i - \vec{x}_0\| > d\right) && (9)\\
&= \left(\Pr\left(\|\vec{X} - \vec{x}_0\| > d\right)\right)^n && (10)
\end{aligned}$$

This has got to go to zero as $n \to \infty$ (which is what we want), unless the probability we're raising to the power n is exactly 1. To get a handle on that, let's re-write it a bit more:

$$\begin{aligned}
\Pr\left(\|\vec{X}_{NN} - \vec{x}_0\| > d\right) &= \left(\Pr\left(\|\vec{X} - \vec{x}_0\| > d\right)\right)^n && (11)\\
&= \left(1 - \Pr\left(\|\vec{X} - \vec{x}_0\| \leq d\right)\right)^n && (12)
\end{aligned}$$

So all we need is for there to be *some* probability of being within d of $\vec{x}_0$. If we're asking for a prediction at a point in the middle of a region of zero probability, nearest neighbors is not a great idea, but otherwise, we're set.

We can be a little bit more detailed by approximating the probability in question. Assume $\vec{X}$ follows a pdf $f(\vec{u})$. Then we get the probability by integrating the pdf f over the radius-d ball centered on $\vec{x}_0$,

$$\Pr\left(\|\vec{X} - \vec{x}_0\| \leq d\right) = \int_{\vec{u}:\|\vec{u}-\vec{x}_0\|\leq d} f(\vec{u}) d\vec{u} \tag{13}$$

Let's assume d is small. (Ultimately we want it to shrink towards zero, after all.) Over a small enough ball, $f(\vec{u})$ will be nearly constant, and equal to $f(\vec{x}_0)$. So

$$\Pr\left(\|\vec{X} - \vec{x}_0\| \leq d\right) \approx c_p d^p f(\vec{x}_0) \tag{14}$$

where c_p is a constant, geometrical factor ($c_2 = \pi$, $c_3 = \frac{4}{3}\pi$, etc.). That is, the probability of a *small* ball centered around $\vec{x}_0$ is about $f(\vec{x}_0)$ times the volume of the ball.

Putting all this together,

$$\Pr\left(\|\vec{X}_{NN} - \vec{x}_0\| > d\right) \approx (1 - c_p d^p f(\vec{x}_0))^n \tag{15}$$

Let's make use of one last approximation, that $(1+h)^b \approx 1 + bh$ when $|h| \ll 1$. (Use the binomial theorem if you don't believe me.) Then we get

$$\Pr\left(\|\vec{X}_{NN} - \vec{x}_0\| > d\right) \approx 1 - n c_p d^p f(\vec{x}_0) \tag{16}$$

This is going to zero for each fixed d. If we want this to be constant — say, if we want to find an d which bounds the distance to the nearest neighbor with 50% confidence, the median nearest-neighbor distance — we'd need to say

$$1 - n c_p d^p f(\vec{x}_0) = \delta \tag{17}$$

or

$$d = n^{-1/p} \left(\frac{(1-\delta)}{c_p f(\vec{x}_0)}\right)^{1/p} \tag{18}$$

So the typical distance to the nearest neighbor is shrinking to 0, at rate $n^{-1/p}$. This $\to 0$ as $n \to \infty$, as desired.

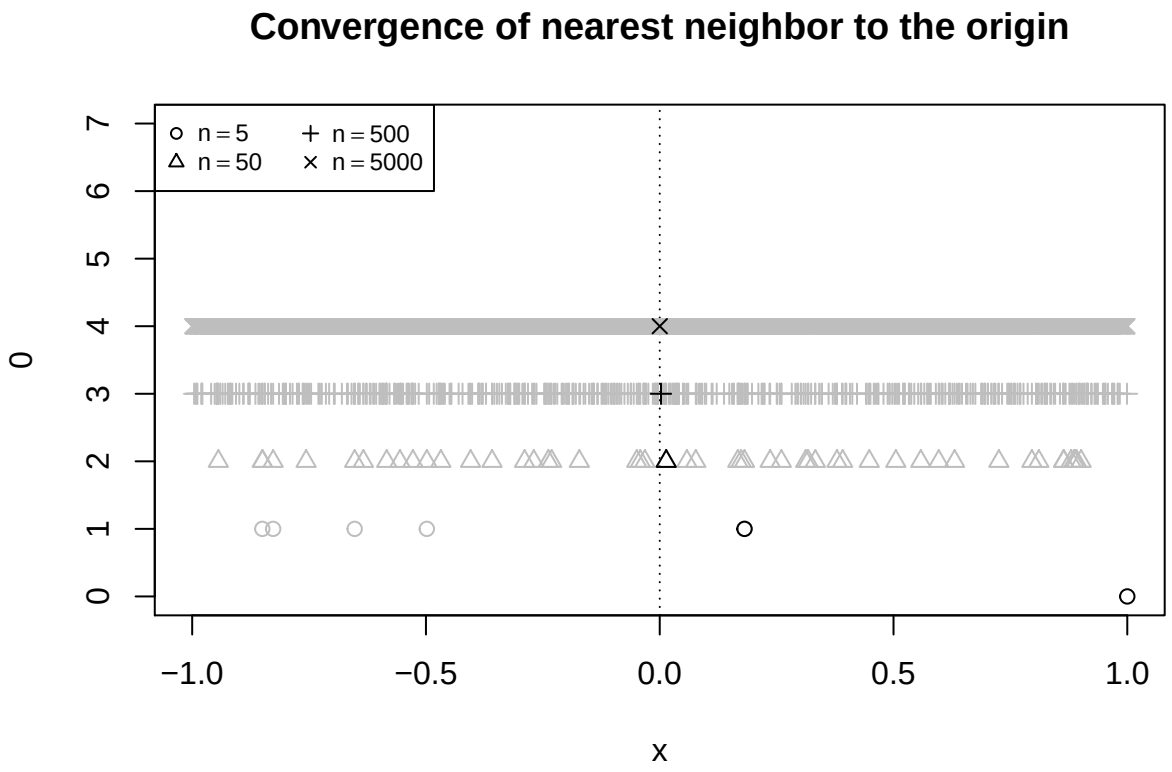


Figure 5: Visualizing the convergence of the nearest neighbor to the origin with increasing sample size

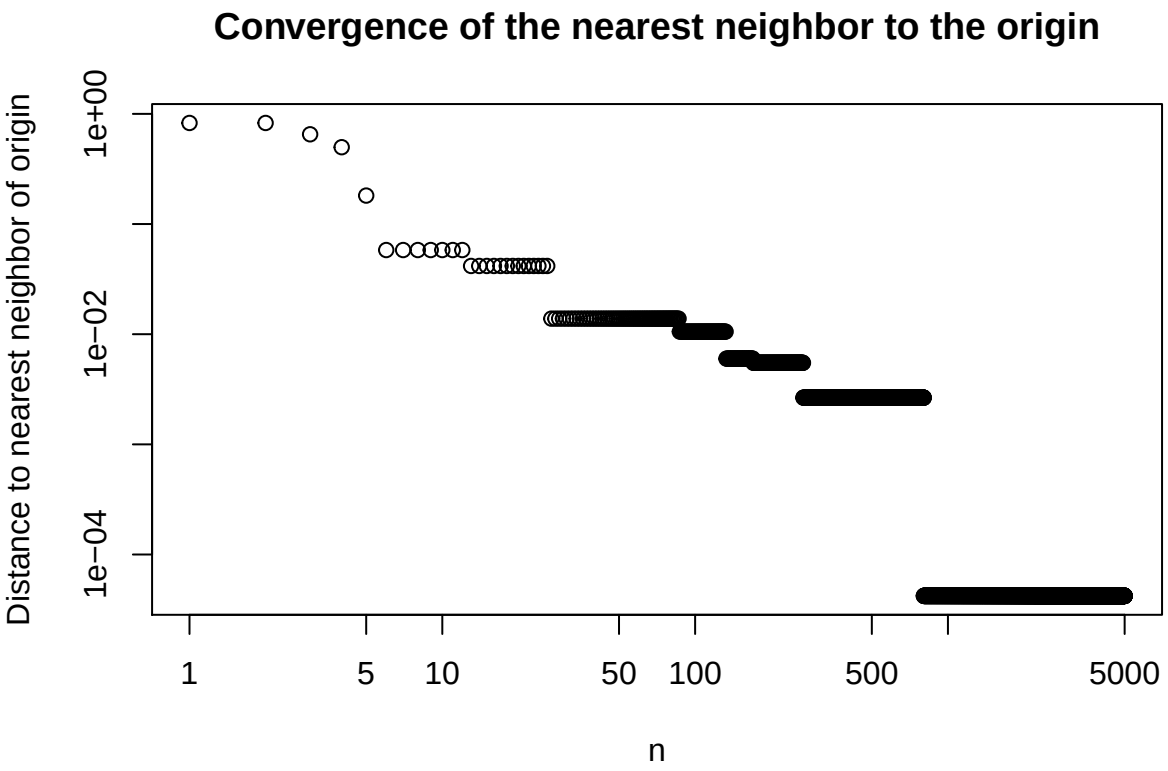


Figure 6: Plotting the distance from the origin to its nearest neighbor as we increase the sample size. If we re-ran the simulation, the exact distances and jumps would change, but the general pattern would not. Notice the log scale for both the horizontal axis (sample size) and the vertical axis (distance to the nearest neighbor).

3.2 1NN is consistent for noise-free functions

To recap, because $\|\vec{x}_N N - \vec{x}_0\| \to 0$ as $n \to \infty$, if the true function is (piecewise) continuous, then 1NN will approximate it arbitrarily well given enough data. When an estimator converges on the truth as $n \to \infty$, it's called "consistent", so we've just shown that nearest neighbors is consistent for learning noise-free functions.

4 Putting the noise back in for 1NN

What happens if we add in noise, but still use 1NN? In the regression case, $Y = \mu(X) + \epsilon$, so

$$\begin{aligned} \hat{\mu}(\vec{x}_0) &= y_{NN} & (19)\\ &= \mu(\vec{X}_{NN}) + \epsilon_{NN} & (20) \end{aligned}$$

The error in predicting a new response at $\vec{x}_0$, Y_{new}, is thus

$$\begin{aligned} Y_{new} - \hat{\mu}(\vec{x}_0) &= \mu(\vec{x}_0) + \epsilon_{new} - \mu(\vec{X}_{NN}) - \epsilon_{NN} & (21)\\ &= (\mu(\vec{x}_0) - \mu(\vec{X}_{NN})) + \epsilon_{new} - \epsilon_{NN} & (22) \end{aligned}$$

As $n \to \infty$, the μ term in parentheses $\to 0$, since μ is continuous (by assumption) and the nearest neighbor converges on the point. So the error approaches $\epsilon_{new} - \epsilon_{NN}$. Squaring, taking expectations, and remembering that the noises are uncorrelated, we get that the risk of 1NN regression at $\vec{x}_0$ approaches

$$\mathrm{Var}\left[\epsilon | \vec{X} = \vec{x}_0\right] + (-1)^2 \mathrm{Var}\left[-\epsilon | \vec{X} = \vec{x}_0\right] = 2\sigma^2(\vec{x}_0) \tag{23}$$

The over-all risk of 1NN regression thus approaches

$$2\mathbb{E}\left[\sigma^2(\vec{X})\right] \tag{24}$$

as $n \to \infty$. But the risk of the true regression function is already $\mathbb{E}\left[\sigma^2(\vec{X})\right]$, so we've come within a factor of two of the optimum risk.[4]

For classification, the risk at a particular point $\vec{x}_0$ is

$$\begin{aligned} \Pr\left(Y_{new} \neq \hat{c}(\vec{x}_0)\right) &= \Pr\left(Y_{new} \neq Y_{NN}\right) & (25)\\ &= \Pr\left(Y_{new} = 1, Y_{NN} = 0\right) + \Pr\left(Y_{new} = 0, Y_{NN} = 1\right) & (26)\\ &= p(\vec{x}_0)(1 - p(\vec{x}_{NN})) + (1 - p(\vec{x}_0))p(\vec{x}_{NN}) & (27) \end{aligned}$$

As $\vec{x}_{NN} \to \vec{x}_0$, this approaches

$$2p(\vec{x}_0)(1 - p(\vec{x}_0)) \tag{28}$$

provided p is a continuous function (or at least piecewise continuous). Recall from earlier that the conditional risk of the optimal classification function is $\min\{p(\vec{x}_0), 1 - p(\vec{x}_0)\}$, say $q(\vec{x}_0)$. So the conditional risk of 1NN approaches $2q(\vec{x}_0)(1 - q(\vec{x}_0)) \leq 2q(\vec{x}_0)$. The over-all risk will thus approach

$$2\mathbb{E}\left[p(\vec{X})(1 - p(\vec{X}))\right] \tag{29}$$

which is at most twice the risk of the optimal classifier.

[4] If the noise variance is constant, $\sigma^2(\vec{x}) = \sigma^2$, this simplifes: the risk of 1NN regression approach $2\sigma^2$, while the risk of the true regression function is just σ^2.

5 Two, three, many nearest neighbors

Recall how we defined the predictions for k-nearest-neighbor regression[5]:

$$\hat{\mu}(\vec{x}_0) = \frac{1}{k}\sum_{j=1}^{k} Y_{NN(\vec{x}_0,j)} \tag{30}$$

For every data point, $Y = \mu(\vec{X}) + \epsilon$, where quite generally $\mathbb{E}\left[\epsilon|\vec{X} = \vec{x}\right] = 0$. So we can write

$$\hat{\mu}(\vec{x}_0) = \frac{1}{k}\sum_{j=1}^{k} \mu(\vec{x}_{NN(\vec{x}_0,j)} + \frac{1}{k}\sum_{j=1}^{k} \epsilon_{NN(\vec{x}_0,j)} \tag{31}$$

What we'd like the prediction to be is of course $\mu(\vec{x}_0)$, as before.

The last equation makes it clear that the error in kNN-regression has two sources:

1. Evaluating the true regression function at the nearest neighbors. That is, we're *approximating* the quantity we want ($\mu(\vec{x}_0)$) by something else, namely $\frac{1}{k}\sum_{j=1}^{k}\mu(\vec{x}_{NN(\vec{x}_0,j)})$. We'll call this the approximation error[6].
2. The noise in the response values for the nearest neighbors $\left(\frac{1}{k}\sum_{j=1}^{k}\epsilon_{NN(\vec{x}_0,j)}\right)$. This is pure noise.

For 1NN, we controlled the approximation error by realizing that it goes to zero as $\vec{x}_0$'s nearest neighbor converges on $\vec{x}_0$. You[7] can extend the argument to show that the k^{th} nearest neighbor does too, for any fixed k. If the k^{th} nearest neighbor is within ϵ of $\vec{x}_0$, then *all* of k nearest neighbors must be too. And then continuity of μ says that the approximation error $\to 0$ as $n \to \infty$.

As for the noise, it's the average of k noise terms. If we assume the ϵs are uncorrelated across data points, we can say that

$$\text{Var}\left[\frac{1}{k}\sum_{j=1}^{k}\epsilon_{NN(\vec{x}_0,j)}\right] = \frac{1}{k}\sum_{j=1}^{k^2}\text{Var}\left[\epsilon_{NN(\vec{x}_0,j)}\right] \tag{32}$$

If $\text{Var}\left[\epsilon|\vec{X} = \vec{u}\right] = \sigma^2(\vec{u})$, then all of those variances are converging on $\sigma^2(\vec{x}_0)$, and we get

$$\frac{\sigma^2(\vec{x}_0)}{k} \tag{33}$$

for the variance of the noise.

The over-all risk of kNN-regression at $\vec{x}_0$ will thus tend, as $n \to \infty$, to

$$(\text{system noise}) + (\text{approximation error}) + (\text{estimation noise}) \to \sigma^2(\vec{x}_0) + 0 + \frac{\sigma^2(\vec{x}_0)}{k} = \left(1 + \frac{1}{k}\right)\sigma^2(\vec{x}_0) \tag{34}$$

That is, rather than having twice the optimum risk with $k = 1$, kNN regression gets only $1 + 1/k$ of the optimum risk — at least as $n \to \infty$.

[5] The analysis for kNN-classification is very similar and comes to the same conclusion, but I don't feel like writing everything out twice.

[6] If we think of the locations of the nearest neighbors as fixed, and only the responses Y as random, then we can call this "bias" in the technical sense, as the expected difference between the estimate $\hat{\mu}(\vec{x}_0)$ and the truth $\mu(\vec{x}_0)$. If we treat the locations of the nearest neighbors as random, then the bias would be $\mathbb{E}\left[\frac{1}{k}\sum_{j=1}^{k}\mu(\vec{X}_{NN(\vec{x}_0,j)}) - \mu(\vec{x}_0)\right]$, which is a bit of a mess, though fortunately not something we'll need to know in detail, as the next paragraph will explain.

[7] "You", meaning "not me, at least not now". The trick is however to realize that if the kth neighbor is more than ϵ away from $\vec{x}_0$, *at least* $n - k + 1$ of the data points must be more than ϵ away. (Said differently, if the kth neighbor is within ϵ, then at least $k - 1$ other data points must also be within ϵ.) The probability of this happening is something we can calculate from a binomial distribution, with n trials and a success probability depending on the probability of a random point being in the ϵ-ball around $\vec{x}_0$.

That last phrase, "as $n \to \infty$", is of course why we don't just automatically set k to be as large as possible. At any *finite* n, we face a trade-off:

- Increasing k means averaging over more data points for each prediction, which reduces the variance by averaging together more noise terms (i.e., big k means less variance);
- Decreasing k means averaging over fewer data points for each prediction, which reduces the approximation error by averaging over points closer to where we want a prediction (i.e., small k means less bias).

This is a manifestation of one of the fundamental issues in statistics, the **bias-variance tradeoff**. When we are doing prediction, we don't (usually) *care* about whether our errors come from bias or from variance, just about the over-all magnitude of the error. We will usually find that we want methods with *some* bias, because the error added by the bias is more than compensated for by the reduction in variance. We need some practical way of deciding *how much* bias we want to trade for less variance. It is important to recognize though that the trade-off will depend on n. For fixed k, the approximation error / bias contribution is going to *shrink* as n grows, because the k nearest neighbors will get closer and closer to $\vec{x}_0$. But the noise / variance contribution will have the same (expected) size, because we'll only be averaging k terms. So what would be ideal is if k could somehow *grow* as n grows, but not so fast that we fall back in to doing a constant or nearly-constant prediction.

6 Selecting k

To recap: When we predict with k-nearest-neighbors, our prediction is always an average (or vote, which is a kind of average) over k data points. We must however pick k. If we increase k, we average over more data points, which is good, because averaging reduces noise and improves precision. But as we decrease k, we average over fewer data points, which is good, because we're focusing our attention only on points close to where we want to make a prediction, which reduces approximation error and improves accuracy. This is a fundamental tradeoff which has many names: bias vs. variance (low k has small bias but large variance), approximation vs. estimation error, accuracy vs. precision. How do we actually make this trade-off?

6.1 Risk minimization

What we *want* to do is to pick the k which will do best on new data. For regression[8], this means we want to pick the k which will minimize

$$r(k) = \mathbb{E}\left[(Y - \widehat{\mu}_k(\vec{X}))^2\right] \tag{35}$$

where $\widehat{\mu}_k$ is our estimate of the true regression function μ obtained by using k nearest neighbors. Unfortunately, $r(k)$ involves the true joint distribution of the features $\vec{X}$ and the outcomes Y, so we can't calculate it.

6.2 Empirical risk minimization

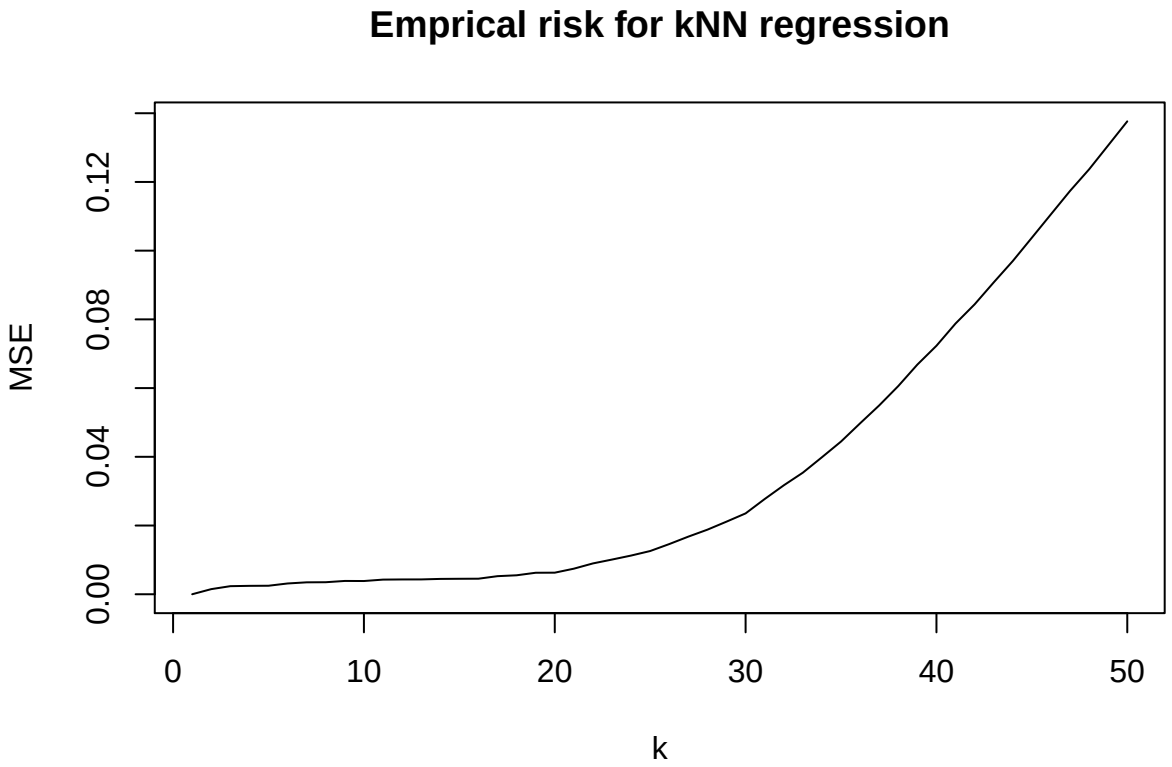
The obvious approach is to replace the true risk $r(k)$ with a "plug-in" estimate,

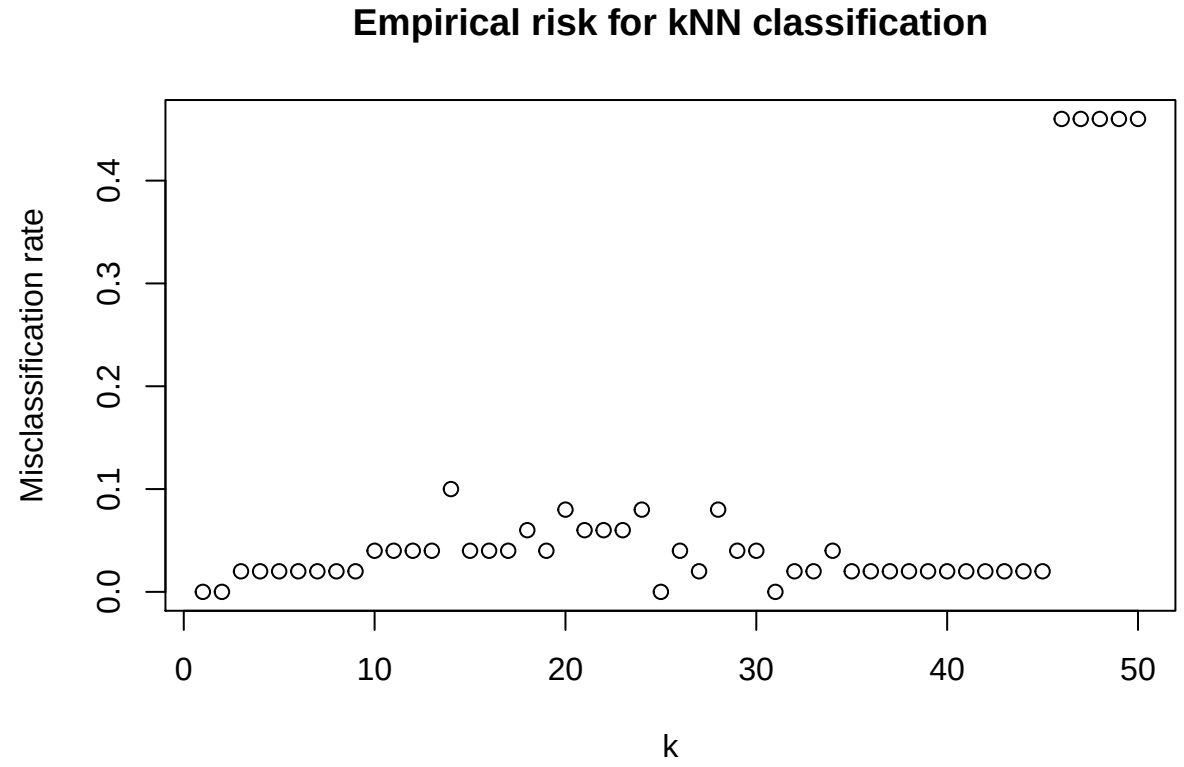
$$\hat{R}(k) \equiv \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{\mu}_k(\vec{x}_i))^2 \tag{36}$$

and select the k with the smallest $\hat{R}_k$. The quantity on the RHS of this equation is called the **empirical risk** of k-NN regression. Picking the model which makes the empirical risk as small as possible is called **empirical risk minimzation**, or ERM. For regression, it's equivalent to minimizing the sum of squares, or the mean squared error, or (under one definition) maximizing R^2.

This is a perfectly reasonable way of picking the parameters within a parametric model. (It doesn't *always* work, but it's a reasonable starting point.) Unfortunately, it's a horrible way of *comparing* model specifications. To see that it's awful for kNN, think about what it will tell us to do for nearest-neighbor regression. The nearest neighbor of $\vec{x}_i$ is, obviously, $\vec{x}_i$. Hence the mean-squared-error of 1-nearest-neighbor regression is always exactly 0. Thus, ERM tells us that the optimal value of $k = 1$, no matter what the data look like.

[8] Everything will apply equally to classification, but I don't want to introduce the extra notation that would go along with being that general. You can make the obvious substitutions for yourself.





To sum up, ERM can be a good way of setting parameters within a model, but it's a horrible way of comparing models.

6.3 True validation set

Since what we'd really like is to minimize $\mathbb{E}\left[(Y - \hat{\mu}_k(X))^2\right]$, in many ways our best option would be to have a large, separate, independent **validation set** of data points (x'_i, y'_i), $i \in 1 : m$, drawn independent from the same distribution as the data we use to develop our models. Then, by the law of large numbers,

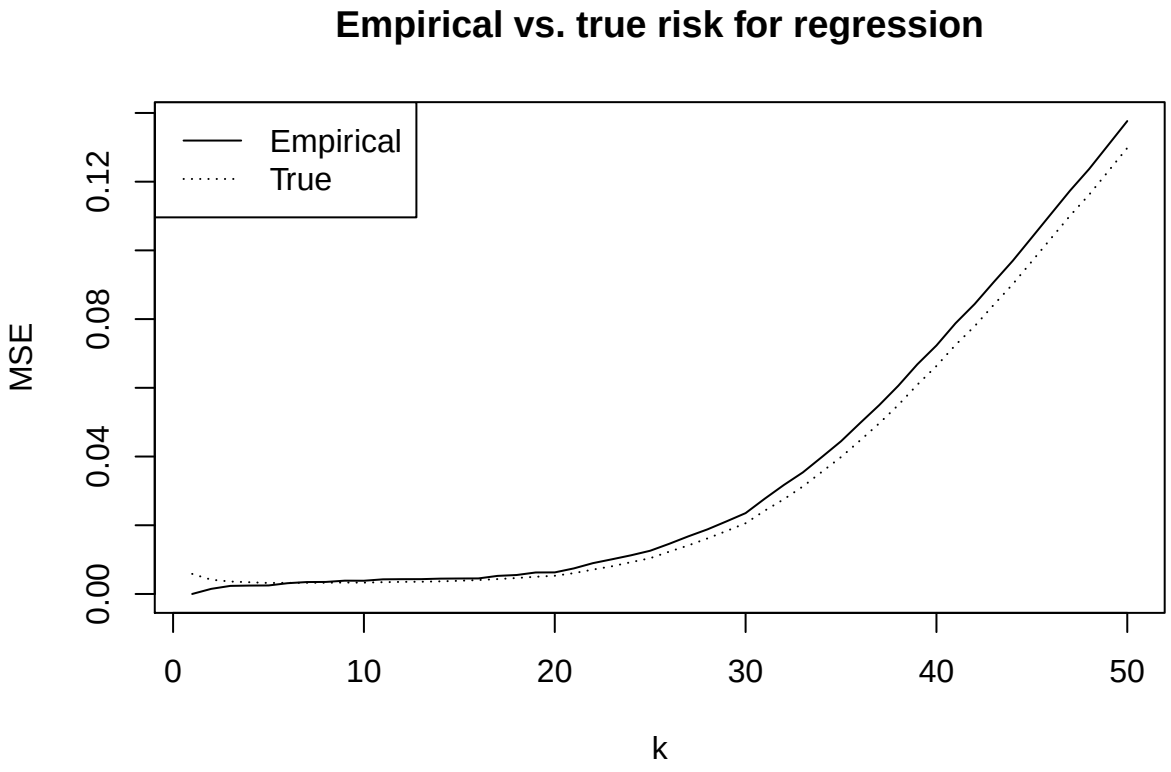
$$\frac{1}{m}\sum_{i=1}^{m}(y'_i - \hat{\mu}_k(x'_i))^2 \approx \mathbb{E}\left[(Y - \hat{\mu}_k(X))^2\right] \tag{37}$$

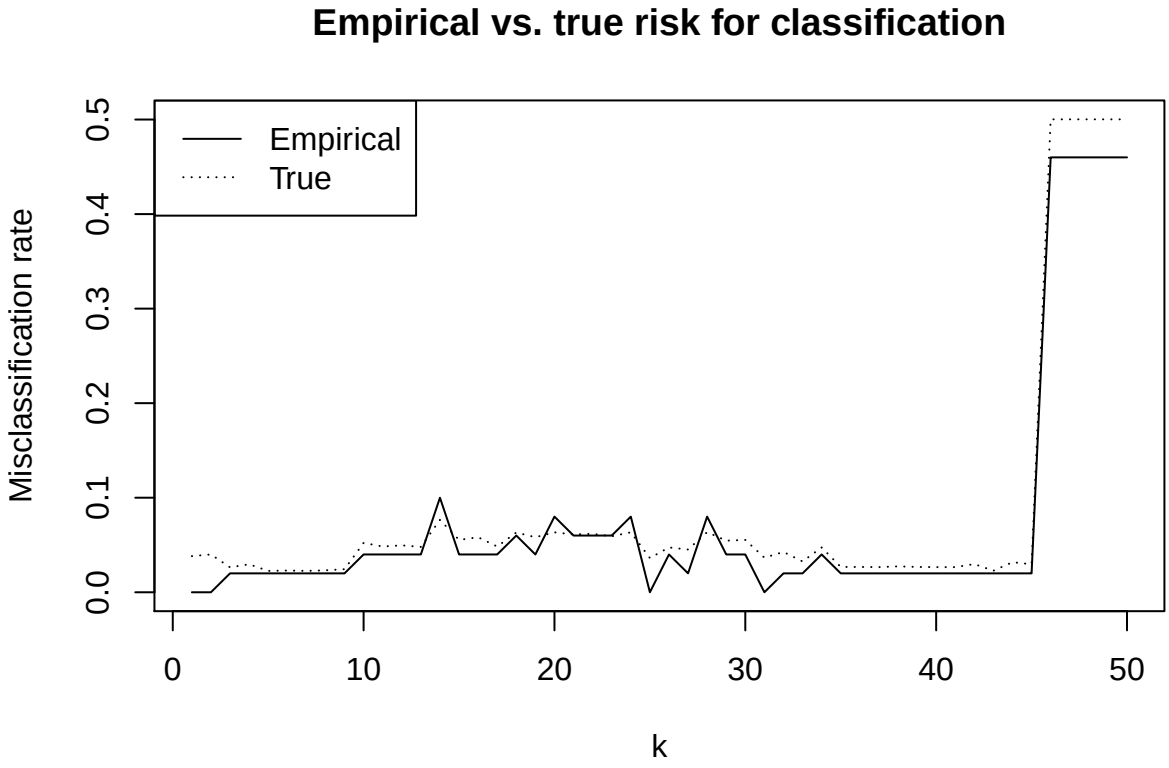
and similarly for the misclassification rate.

It's very important here that the validation data (with primes) be totally independent of the data used to fit the models, since that ensures that $\hat{\mu}$ is independent of the validation data. That in turn guarantees that error on the validation set is an unbiased and consistent estimate of the generalization error on new data.

This is a great approach when you can make it happen; the problem is that waiting for genuinely new, independent data from the relevant distribution to appear is often very slow and/or expensive, and that makes it extremely tempting to include it in the data you use to fit and develop the model.

— For our running simulation example, we can just simulate *much* more data from the same process, and see how well models trained on the limited initial data generalize to this validation set.





6.4 Data splitting ("roll your own validation set")

Rather than waiting for a genuinely-independent validation set to accumulate, a common alternative tactic is to *split* your data into two parts, conventionally called the "training set" and the "testing set" (or "test set"). Their sizes are n_{train} and n_{test} respectively, with $n_{test} = n - n - train$. We typically insist that the division into two parts be totally random; e.g., we might decide that $n_{train} = n/2$, randomly select *exactly* that number of rows to belong to the training set, and declare everything else to be in the testing set.

The fundamental rule in data splitting is that all models can *only* use the training set to generate predictions. For parametric models, this means that parameters are estimate entirely on the basis of training points. For kNN, it means that nearest neighbors are sought only among the training points. Whether the place where we are making a prediction is one of the training points or not is irrelevant; only training points go in to making the prediction.

Contrarily, we only *care* about how well we can predict points in the testing set. For each i in the testing set, we try to predict y_i on the basis of $\vec{x}_i$, where each model gets estimated using *only* the training set. We average the (squared) prediction errors, and pick the k with the smallest error when generalizing to unseen data. So: fit on the training set, evaluate on the testing set.

Because we've *randomly* split the data into two parts, the training and the testing set follow the same distribution, and are independent. Because the models are estimated using *only* the training set, their predictions on the testing set aren't influenced by what we're trying to predict. (More exactly, of course we want $\hat{\mu}(\mathbf{X}_{test})$ to be correlated with $\mathbf{Y}_{test}$, but we want $\hat{\mu}(\mathbf{X}_{test})$ to be *independent* of $\mathbf{Y}_{test}$, **conditional on** $\mathbf{X}_{test}$. This is why it's important that the model is only estimated using the *training* data.) This lack of influence means we can appeal to the law of large numbers again, as in the genuine-validation-set idea, to say that average performance on the testing set gives us a good estimate of how well we can generalize to future data.

Because you've done this a couple of times in the homework already, I won't include illustrative code.

6.4.1 Drawbacks of data splitting

There are two big drawbacks to simple data splitting, as I've presented it.

1. We need to divide the data into training and testing sets completely at random, but that means that which model (e.g., which value of k) we select is *also* somewhat random. We can hope that it's not *very* random, if both n_{train} and n_{test} are big, but we're still just making one random split.
2. We have reason to think that the *right* (best-predicting) model can change with n. Specifically for nearest neighbors: We *know* that bigger values of k are *ultimately* better as n grows. But we're seeing which value of k predicts best at size n_{train}, which might be different from the best one to use on the full data, of size n.

We could ameliorate the second issue by making n_{train} close to n, and so much bigger than n_{test}. But now we're picking only a small number of points out of the whole data set, and using them to decide on the model (or on k), and so the first problem, of randomness, is amplified. Can we do something about *both* issues at once?

6.5 Cross-validation

This is where **cross-validation** comes in. The basic idea is to repeat data splitting multiple times, averaging across different splits into training and testing.

6.5.1 V-Fold Cross-Validation

The most common form is what's called "v-fold cross-validation"[9]. This goes as follows:

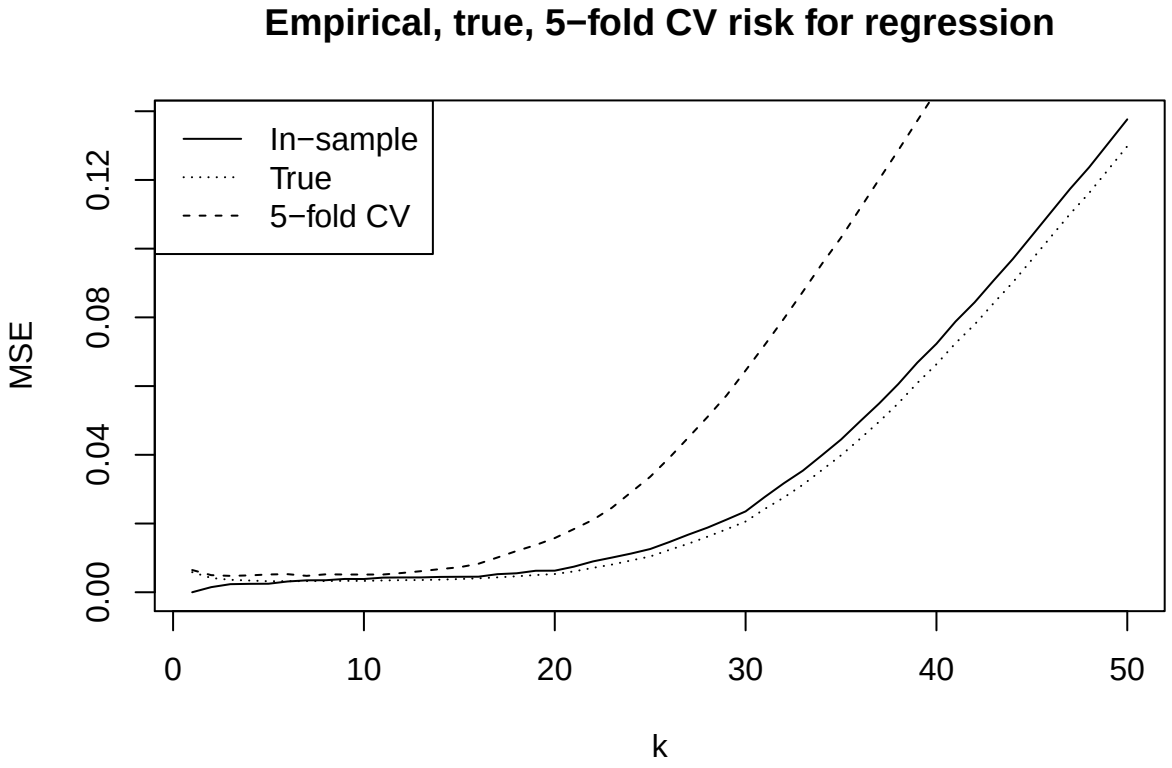
1. Randomly divide the n data points into v different, non-overlapping sets, the **folds**.
2. For each fold $f \in 1:v$
    a. The points in fold f are, temporarily, the testing set. The points in the other $v-1$ folds, taken together, are the training set.
    b. For each model m (e.g., possible value of k)
        i. Estimate m using the temporary training set.
        ii. Compute the prediction, and the prediction error, for each point in the temporary testing set (i.e., each point in fold f)
        iii. Record the average prediction error.
3. Average prediction errors for each model across folds. This is the VFCV estimate of the generalization error.
4. Report the VFCV scores, and pick the model with the best score.

Because we don't just make *one* random split into training and testing sets, but many (or v splits at any rate), we reduce the noise in our estimate, compared to simple data splitting. On the other hand, each training set contains a fraction $\frac{v-1}{v}$ of the complete data, so we're seeing what generalizes best at a sample size close to that of the full data.

(In passing, notice that we've ensured that each data point appears once, and only once, in a testing set, and gets used in exactly $v-1$ training sets. You could imagine just doing v random training/testing splits, but this leads to [slightly] more variance in our estimate of generalization performance, because it randomly over-emphasizes some data points compared to others.)

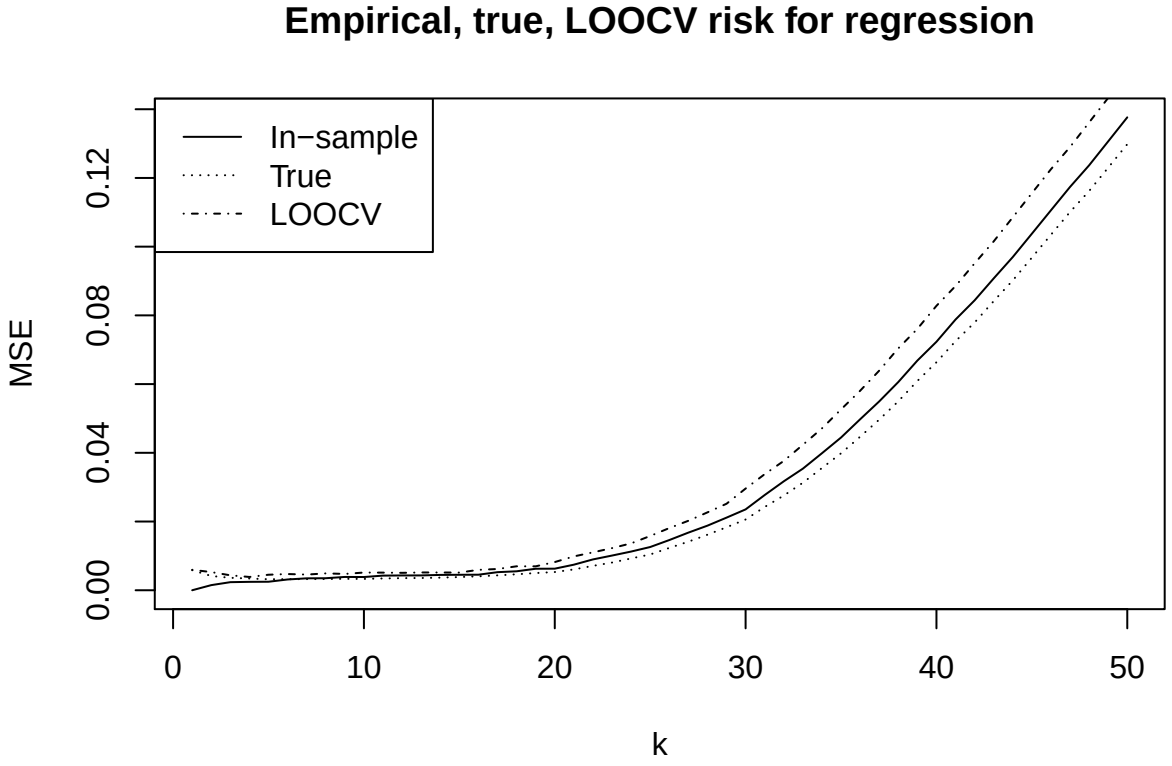
(Also in passing, observe that the estimates of generalization error we get from each fold are correlated. This is because when we consider the models we test against fold 1 to those we test against fold 2, they still had $n\frac{v-2}{v}$ *training* data points in common. This means that the variance of the estimate of generalization error isn't reduced quite as much as if we were averaging v *uncorrelated* random variables.)

[9] Actually, people usually call it "k-fold", but here k is pre-empted by the number of nearest neighbors, and it is *sometimes* called "v-fold" (Arlot and Celisse 2010).



6.5.2 Leave-one-out cross-validation (LOOCV)

The extreme of v-fold cross-validation is to set $v = n$, so that each point is put in its own "fold", with a testing set of size of 1. Each training set thus contains $n - 1$ data points. This is called "leave one out" cross-validation, because each data point is, in turn, "left out" (made the testing set), and we try to predict it from a model trained with all the other points.



6.5.3 Which cross-validation?

V-fold CV can be much faster computationally than LOOCV; we re-fit each model v times, instead of n times. Moreover, if we're using CV to select among models of different complexity, LOOCV tends to "overfit", in the sense that it tends to pick a model with all of the right terms *and* extra, superfluous ones as well, even as $n \to \infty$. For this reason, 5- or 10- fold CV is pretty much the "industry standard". There are two issues which complicate this, however.

1. The strictly *predictive* performance of LOOCV can be better than that of v-fold cross-validation. In particular, if there's no true model, but we just want to predict as well as possible (which is the situation with k-nearest-neighbors), LOOCV will tend (as $n \to \infty$) to select the best-predicting model among those we compare (Azadkia 2019).
2. If we're doing regression and using a linear smoother, there is a short-cut formula which lets us calculate the LOOCV score from the fit to the full data and its weight matrix (Wahba 1990, Theorem 4.2.1, p. 51). This short-cut applies to kNN regression, but not to classification.

7 Computational aspects

There are two (main) costs to any computational procedure: the time it takes to run (measured in elementary operations of the computer), and the memory it needs to run (sometimes referred to as "space").

The memory cost of kNN is pretty demanding. We need to keep around the entire data set, which means p features plus 1 label per data point, for a total memory cost of $O(n(p+1)) = O(np)$. Admittedly, we don't need all of this in memory at once. But there's no extra memory needed for the model itself.

The time complexity is more interesting. The most straightforward way to implement kNN is to compute the distance between the new point $\vec{x}_0$ where we're making a prediction, and *every* data point with a label. The time to compute one distance with p features is going to be $O(p)$, so computing all the distances will take $O(np)$ time. The time to find the k smallest distances, from a list of n distances, will be $O(n)$ (regardless of p). Once we find those k nearest neighbors, averaging their labels will take $O(k)$ time (regardless of n and p). So the total time will be $O(np + n + k) = O(np + k)$. Now, a *theoretical* computer scientist will tell you that anything which is sub-exponential in n is "tractable", but if $n = 10^9$ or 10^12, that's a bit delusional...

There are *two* parts of the straightforward implementation which take $O(n)$ time: computing all the distances, and finding the k smallest distances. Since $k \ll n$, most of the distances we compute end up being useless. This suggests some lines of inquiry for speeding up kNN:

1. Do we need *all* n data points?
2. Can we be faster about computing *each* distance? Can we compute *approximate* distances quickly?
3. Can we rule out some points as potential nearest neighbors, *without* examining them?

7.1 Using fewer than n data points

If our algorithm slows down with n, one obvious approach is to try to make n smaller. One reason this may not be hopeless is that there are **diminishing returns**, predictively, to increasing n: the risk shrinks as n grows, but more and more slowly. To understand this, remember that the distance to the nearest neighbor is $O(n^{-1/p})$. The bias will be on the order of the distance (Taylor expand μ), and it's the *squared* bias that matters for regression risk, so the bias's contribution to the risk is $O(n^{-2/p})$. Against this, for fixed k, the variance's contribution to the risk is $O(1/k)$, which is constant in n. Thus if $p = 10$, doubling n doubles computing time, but the bias is stil ≈ 0.87 of what it was before, and the variance hasn't gone down. Past some size n, the extra risk reduction just isn't worth the extra computing time.

This idea leads to **sampling** strategies: we pick a random subset of $n' \ll n$ data points, and ignore the others. This can keep the computing time small, at an acceptable level of extra prediction risk. Picking n' in a principled way needs a price at which we can trade risk against computing time. (Alternately, we could impose a hard constraint on either time or risk, but then a Lagrange multiplier will give us the implied price, the "shadow price", at which these trade off against each other.) Sampling leads to an extra element of randomness, but, by design, we're no worse off in terms of risk than if we really did just have n' data points.

Beyond random sampling, there are some strategies for trying to *select* data points to keep in the training sample. Details get complicated, but the basic idea is usually to identify points whose removal will do the least to change the predictions, and drop them. (For instance, if some data point is *never* the nearest neighbor of any other point, it's a good candidate for deletion.) Relatedly, if there are multiple points which are near each other with the same or very similar labels, we might try "condensing" them into one point. But random sampling is often competitive with these more complicated procedures.

The drawback of using fewer points is that it seems wasteful to collect the data, and then ignore most of it!

7.2 Faster distance computation: the random projection trick

Our naive implementation of nearest neighbors involves calculating the distances between our test point and n different p-dimensional vectors; this is the $O(np)$ part of the over-all time. One way to speed this up is to use the following remarkable mathematical fact about vectors[10]:

> Take any n different p-dimensional vectors, and consider projecting them on to q different *random* directions. If $q = O(\frac{\log n}{\epsilon^2})$, then, with high probability, the distances between the *projected* vectors are within a factor of $1 \pm \epsilon$ of the distances between the *original* vectors.

In other words, if we take our p-dimensional data, and *randomly* project it down to $O(\log n)$ dimensions, we nonetheless preserve distances (to within a factor of $1 \pm \epsilon$). The time to do one of these projections is $O(p)$, so the time to do *all* of the projections for *all* the data points is $O(np \log n)$, but we only have to do this *once*, regardless of how many predictions we make. We can then find (approximate) nearest neighbors in time $O(n \log n + p \log n + k)$:

- $O(p \log n)$ to project the new vector on to the $O(\log n)$ random vectors;
- $O(n \log n)$ to find distances between the projected vectors;
- $O(n)$ to find the smallest projected distance (absorbed in the $O(n \log n)$;
- $O(k)$ to average the nearest neighbors' labels

This random projection trick will help when $p \gg \log n$, which can easily be the case in modern data (say $p = 10^5$ while $n = 10^6$). But it doesn't get us away from scaling badly with n.

7.3 Pre-selecting possible neighbors

There are two ways to avoid having to look at every data point. One is to use clever, deterministic data structures. The other is to use random summaries of the data.

7.3.1 Data structures: $k-d$ trees

Algorithm designers have spent decades devising clever data structures for finding nearest neighbors, and I won't pretend to survey the state of the art. But it is worth understanding one of the classic approaches to this, $k-d$ **trees** (or k**-dimensional trees**). Partly this is because it's a very neat approach, and partly this is because it's the default in the `FNN` package.

$k-d$ trees are sorting or search trees, where each of the n original data points is located at a leaf. Each internal node is labeled by a feature, and a possible value of that feature[11]. Generally, all the internal nodes at the same level use the same feature[12]. One searches through the tree for neighbors to a new point by "dropping the point down the tree": starting at the root, go to either the left child node or the right node depending on whether the new point's value for the current node's feature is above or below the current node's threshold. We continue going down the tree until there are only k leaf nodes below us: those are the neighbors.

To see why $k-d$ trees find neighbors *quickly*, assume that the number of points we *could* be matched to gets cut in half at each node, so there are n leaf nodes under the root, $n/2$ leaf nodes under each child of the root, etc. (We'll see how to make sure this assumption holds in a moment.) How many levels d do we need to go down to reach k candidate neighbors? Set $ 2^{-d}$ to k and solve:

$$n2^{-d} = k \tag{38}$$
$$\log_2 n - d = \log_2 k \tag{39}$$
$$d = \log_2 n/k \tag{40}$$

[10] This fact is called the **Johnson-Lindenstrauss lemma**, and we'll revisit it in the context of dimension reduction.
[11] This is like the CART trees which we'll soon learn to use for classification and regression.
[12] This is unlike CART.

So the time needed to find k approximate nearest neighbors, using a $k-d$ tree, is $O(\log n)$, not $O(n)$!

Now, this might not work at finding the *nearest* neighbors. (There might be nearer neighbors on the other side of one of the splits we've taken.) There are more sophisticated tricks which will guarantee finding the nearest neighbors with the $k-d$ tree, which you can find in standard references on algorithms and data structures (like Cormen et al. (2001)). Using those tricks, finding the nearest neighbors take $O(\log n)$ steps *on average*, though the worst-case time is $O(n)$.

I still haven't said how to actually build the $k-d$ tree. Here's the procedure:

- Put the features in some fixed order from 1 to p
- At step i, we'll be dividing on feature $i \mod p$
- Initially, all points sit under the root node; divide at the median on feature 1
  - Finding a median takes $O(n)$ time
  - sometimes randomly select a fixed small set of $n' \ll n$ points and take *their* median
- Within each child node, split on the median of the associated points
- Recurse until there is only one data point within each node; those are the leaves

Because we've used the median, we've ensured that each child contains 1/2 of the points of its parents, which is what we wanted to ensure that finding (approximate) neighbors would be fast.

7.3.2 Clustering

Here is a very different approach to finding (approximate) nearest neighbors quickly (Paulevé, Jégou, and Amsaleg 2010):

- Randomly pick q data points, $q \ll n$, and label them with the numbers from 1 to q.
- Until nothing changes:
  - Set the **cluster center** vector $\vec{c}_j$ to be the average of all data points labeled j
  - Label each of the n data points according to the cluster center $\vec{c}_i$ which it's closest to, i.e., $L_i = \text{argmin}_{j \in 1:q} \|\vec{x}_i - \vec{c}_j\|$
- To find neighbors for a new point $\vec{x}_0$, assign it to the cluster j whose center $\vec{c}_j$ is closest to $\vec{x}_0$, and only look for neighbors among other points assigned to that cluster

If $q = \sqrt{n}$, then it takes $O(p\sqrt{n})$ to find the right cluster in which to look for neighbors, and $O(pn/q) = O(p\sqrt{n})$ to search for neighbors within that cluster. So the over-all time to run kNN, using clusters to search for *approximate* neighbors, is $O(p\sqrt{n} + k)$, which will be a lot faster than the naive implementation. (You could also combine this with, say, the random projection trick to speed up calculating distances.)

This division of the data points in to clusters is an example of what we will see later as **k-means clustering** (because the number of clusters is usually called k; I wrote it as q here to avoid confusion with the number of nearest neighbors averaged for predictions.) When we look at it in more detail, we'll see that it tends to produce compact clusters of near-by points, with (roughly) similar numbers of points in each cluster. This is precisely what we want for finding (approximate) nearest neighbors, *whether or not* the data really fall into well-separated clumps or clusters.

This use of clustering is an example of a broader family of methods for quickly and randomly dividing the data points into **bins** or **buckets**, with a guarantee that points placed in the same bucket are *probably* similar in some respect. This means that once we find which bucket our test point belongs to, the other points in that bucket are good candidates for being its nearest neighbors. These methods are known, for convoluted historical reasons, as **locality sensitive hashing** (LSH)[13] techniques, and have been extensively developed

[13] In computer science, a "hash function" is one which takes inputs in some domain, say strings of text characters, and maps them to numbers, thought of as "bins" or "buckets", or more generally as categories. Good hash functions have two properties: (i) the distribution of the output is *uniform* over the categories, and (ii) changing the input puts the output in a different category, at least with high probability. If we want to *approximately* compare two inputs, without actually looking at them, we can compare their hashed values: if those are different, the inputs were probably different. So hash functions are used to check for data-entry errors, for tampering in files, etc. A "locality sensitive hash function" is one with the extra property (iii) if two inputs do get hashed to the same bin, then they are probably similar (in some specified respect). As for the origin of the

in the literature on database systems; see further reading for more details.

name “hash function” itself, that seems to come from the fact that the symbol `#`, sometimes called the number sign or the pound sign, is also called the “hash sign” or “hash mark”, and a hash function is thought of as assigning inputs to cells in a grid. The word “hash” itself meant something chopped up with an axe or cleaver (as in “hash browned potatoes”, or “make a hash of something”), and the hash mark looks like what you’d see looking down on some hashed-up vegetables. The English “hash” comes from the old French *hache*, an axe (also the root of “hatchet”) and the corresponding verb *hacher*, to cut up with an axe. So locality sensitive hash functions are, at the root, ways of cutting the data up into small bits with an axe, while keeping near-by things together.

8 R aspects: `FNN`

There are many R packages which implement kNN, sometimes just for classification, sometimes for classification and regression, sometimes even more flexibly. Any list I gave here would quickly be obsolete, so I won't try. What I will do is recommend the `FNN` package (Beygelzimer et al. 2013), which I have found robust, well-documented, and (as the first letter suggests) *fast*, because it contains good implementations of several of the standard tricks for speeding up nearest neighbors described in the previous section. All the places where I've actually computed nearest-neighbor predictions in these notes have used `FNN`, so when you look at the code in the .Rmd file, you'll see more examples of it in use.

8.1 No model objects

Many (most) pieces of R code for making predictions from models return a model **object**, which stores the results of the fitting procedure and can then be passed to other functions like `predict()`, `coefficients()`, `summary()`, etc. This is how `lm()` works, and `glm()` and `glmnet()`. It's also how other packages we'll look at do it. One peculiarity of `FNN` is that it *doesn't* do this. The reason the designers wrote it that way is that there really isn't much to store, beyond the data set!

8.2 Making predictions

Instead, the basic functions in `FNN` are all about making predictions. Here, for instance, is how to make regression predictions with $k = 5$:

```
library(FNN)
knn.reg(train = training.x.matrix, test = testing.x.matrix, y = training.y.matrix,
    k = 5)
```

Some notes:

- The return value of `knn.reg()` is a list. The most important part of that list is the `$pred` component, which is the vector of predicted values.
  - All of the predictions are averages of values in `y`.
- `train` and `y` should have the same number of rows; `test` can have as many rows as it likes.
- `train` and `test` should have the same number of columns.
  - Also, the columns should be in the same order.
- When we use `lm()` or `glm()`, with named variables in the formula, the `predict()` function will look for the corresponding variables in `newdata` by those names, so the column order doesn't actually matter. `knn.reg()` is more finicky this way (but faster).
- As I've indicated with the names I've given to the `train` and `test` arguments, `knn.reg()` likes those arguments to be matrices; if you've stored the appropriate values as something else, like a data frame (or selected columns of a data frame), the `as.matrix()` function is handy for making the connection.
- We do not need to give values of Y on the testing set. (If we're genuinely making predictions, we don't know those values yet!)
  - Do not include a column in `train` which corresponds to the target variable Y.

If we omit the `test` argument, and do something like this

then `knn.reg()` will do leave-one-out cross-validation. The return value will then include components `$residuals` (the vector of residuals) and `$PRESS`, the "PREdictive Sum of Squares", i.e., the sum of the squared residuals[14]. The leave-one-out CV estimate of the risk would thus be `$PRESS` divided by the number of rows of `train` (which the output of `knn.reg()` stores as `$n`).

If we want to do classification instead of regression, we do this:

[14]The acronym goes back to Geisser and Eddy (1979).

```
knn(train = training.x.matrix, test = testing.x.matrix, cl = training.class.labels,
    k = 137)
```

This would do 137-nearest-neighbor classification. If we want to do leave-one-out CV, that's

```
knn.cv(train = training.x.matrix, cl = training, class.labels, k = 137)
```

There is no equivalent to `$residuals` or `$PRESS` for `knn.cv()`.

8.2.0.1 How not to do cross-validation

If I do

```
knn.reg(train = train.x.matrix, test = train.x.matrix, y = y.vector, k = 1)
```

I will *not* get the correct results for leave-one-out cross-validation with $k = 1$. Instead, I will get a perfect (though meaningless) fit. Similarly,

```
knn(train = train.x.matrix, test = train.x.matrix, cl = the.classes, k = 1)
```

will give me perfect classification every time.

You *could* write code to do leave-one-out cross-validation in either case, using just `knn.reg()` or `knn()`. It would look something like this:

```
my.knn.reg.cv <- function(train, y, k, ...) {
    n <- nrow(train)
    loo <- vector(length = n)
    for (leave.out in 1:n) {
        prediction <- knn.reg(train = train[-leave.out, ], test = train[leave, out,
            ], y = y[-leave.out, ], k = k, ...)$pred
        loo[leave.out] <- y[leave.out] - prediction
    }
    return(c(residuals = loo, PRESS = sum(loo^2), loocv = mean(loo^2)))
}
```

(I'll leave writing the corresponding `my.knn.cv()` as an exercise.) But because this is a really common thing to want to do, the `FNN` package comes with tools to do it. v-fold CV is less common for nearest-neighbor methods, so you'll find my code for doing it above (in the .Rmd file).

8.2.1 Conditional probabilities

In addition to predicted class labels, the `knn()` function can return some information about the *probability* of the predicted class. This is stored in the return value of `knn()` as what R calls an **attribute**, which can be accessed using the `attr()` function. To do so, it's generally convenient to save the output:

```
class.out <- knn(train = train.x.matrix, test = test.x.matrix, cl = training.class.labels,
    k = 137, prob = TRUE)
```

(The default with `knn()` is `prob=FALSE`.)

Now

```
attr(class.out, "prob")
```

will be a vector, with one row for each row in `test`, giving the conditional probability of the "winning" class for each test point.

Notes:

- This is always a vector, even if there are more than 2 classes, because it's always the estimated conditional probability for the winning class.

- It's always the estimated conditional probability for the *winning* class, not the *positive* class, or the *actual* class. So if we want to use this to (for instance) calculate the average log loss, we'll need to do a little bit of coding.

8.2.2 Just finding the nearest neighbors and/or the distances

If we *just* want to find the nearest neighbors, and/or the distances to the nearest neighbors, the best approach is to use the underlying functions in the `FNN` package[15].

If I make `a.matrix` a matrix with n rows, then

```
get.knn(data = a.matrix, k = 5)
```

will return a list. `$nn.index` will be $n \times 5$ matrix, giving the indices (numbers from 1 to n) for the 5 nearest neighbors, in `data`, of each row of `data`. Similarly, `$nn.dist` will be the $n \times 5$ matrix of distances to the nearest neighbors. (If I change `k`, the number of columns of these matrices will of course change.) Notice that the argument is called `data` and not, say, `train`.

```
get.knnx(data = a.matrix, query = another.matrix, k = 5)
```

will tell us the same things about the nearest neighbors, among the rows of `data`, for each row of `query`. (Again, notice, `data` and `query`, not `train` and `test`.)

8.2.3 Changing how nearest neighbors are searched for

All four of `knn()`, `knn.reg()`, `get.knn()` and `get.knnx` take an optional argument, `algorithm`, which controls the procedure used to search for nearest neighbors. The default is to use `kd_tree`, i.e., the k-d tree approach described above. Other options are described in `help(get.knn)`. I strongly advise against using `algorithm=brute`, which calculates all n distances and sorts the list to find the the k smallest. This is guaranteed to find the nearest neighbors, but is needlessly slow for all but very small datasets.

[15] An alternative would be to use the fact that the output of `knn()`, and `knn.cv()`, always has attributes (in the sense of the previous sub-sub-section) named `nn.index` and `nn.dist`, which do what I'm about to describe. So we could always run `knn()` with `cl` set to whatever we like (because `cl` doesn't change which points are neighbors), and then examine the attributes. While I've seen people code things up this way, that's just needless overhead, compared to the approach I'm about to describe.

9 Extensions and complements

You can skip these without hurting your ability to do the homework, but they're interesting.

9.1 Nearest neighbors for other decision problems

In general, when the real state of the world is y and we take action a, we incur a loss $\ell(y, a)$. We'd like to choose a, based on information X, to minimize this, but since the state of world is a random variable Y, we can't *always* minimize. Instead, we try to minimize the risk, $r(s) = \mathbb{E}[\ell(Y, s(X))]$.

We can ask what action would minimize the risk conditional on $X = x$. The risk of the action a is

$$\mathbb{E}[\ell(Y, a)|X = x] = \int \ell(y, a)p(y|X = x)dx \tag{41}$$

$$\text{or} = \sum_y \ell(y, a)p(y|X = x) \tag{42}$$

depending on whether Y is discrete or continuous. The risk of any particular action a is clearly going to change with x, so the risk-minimizing action will be a *function* of x:

$$\sigma(x) \equiv \operatorname*{argmin}_a \mathbb{E}[\ell(Y, a)|X = x] \tag{43}$$

The definitions of the optimal regression function and of the optimal classifier are special cases of this, when the loss functions are squared error and 0-1 loss, respectively. If we used the log loss, then (as you saw in HW 3) the possible "actions" are really possible probability distributions for Y, and the optimal prediction is the conditional distribution of Y, $p(y|X = x)$.

So far this is just the general decision theory of Lecture 4. What is the kNN approach? When we are interested in making a decision, with the information that $X = \vec{x}_0$, we look at the k distinct (X, Y) pairs whose X values are closest to $\vec{x}_0$. We then ask what action would minimize the average loss for those k data points. We take that action. In symbols,

$$\hat{s}_k(\vec{x}_0) \equiv \operatorname*{argmin}_a \frac{1}{k}\sum_{j=1}^{k} \ell(Y_{NN(\vec{x}_0,j)}, a) \tag{44}$$

Note that:

- The X coordinates only matter for finding the nearest neighbors and so the y_i values. The average loss we're trying to minimize doesn't involve them.
- The pattern is perfectly general and can be applied to any loss function.
  - For some loss functions, the minimizer might not be unique; pick one at random.
  - kNN regression, and kNN classification, as defined above, are special cases of this general pattern, for the squared error loss and for the 0-1 loss, respectively.
  - If we wanted to make a distributional prediction for Y, and use the log loss, what we should do is predict the sample distribution of the y_is among the k nearest neighbors. This is also the maximum likelihood estimate from those observations. (Cf. Hand, Mannila, and Smyth (2001), p.348.)

9.1.1 Asymptotic risk of nearest neighbors

$\hat{s}_k$ is random, and changes with n. We can ask what will happen as the sample size $n \to \infty$. Long ago, Cover (1968a), sec. III, p. 53, made some weak continuity assumptions about ℓ and $p(y|x)$, and proves that

$$\lim_{n\to\infty} \mathbb{E}[\ell(Y, \hat{s}_1(s))|X = x] \leq 2\mathbb{E}[\ell(Y, \sigma(x))|X = x] \tag{45}$$

That is, the conditional risk of 1-nearest-neighbor is within a factor of 2 of the best possible risk. In fact[16],

$$\lim_{n\to\infty} r(\hat{s}_1) \leq 2r(\sigma) \tag{46}$$

9.2 Additional optional exercises

1. Using leave-one-out CV, create a plot of estimated risk versus k for classification, using the running example data.
2. Re-write the v-fold CV code to work with classification. Create a plot of the risk for different k for classification, using the running example data.
3. Prove that for fixed $k > 1$, the distance to the k^{th} nearest neighbor of x_0 goes to 0 as $n \to \infty$, provided that $f(x_0) > 0$. Specifically, prove that the probability that this distance is $\geq \epsilon$ is going to zero. Can you say how much slower this convergence gets as k grows?
4. Using `get.knnx()`, write a function which will use k nearest neighbors to get the conditional distribution of Y given $X = x$, assuming Y is categorical. (How would you handle the possibility that not every class is represented among the k nearest neighbors of every point?)

[16] This is immediate if the loss is bounded (Cover 1968a, Corollary 1, p. 52), and still true under an additional weak assumption if the loss is unbounded (Cover 1968a, Corollary 2, p. 52).

10 Further reading and historical notes

The pioneering theoretical analysis of nearest neighbors, covering both regression and classification as special cases of prediction-in-general, was done by Cover in the 1960s (Cover and Hart 1967; Cover 1968a, 1968b). What I've done above is basically "Cover made (even) simpler". For more refined analyses of kNN classification and regression, see the appropriate chapters of Devroye, Györfi, and Lugosi (1996) and Györfi et al. (2002), respectively.

Historical note on nearest neighbors: "Find the most similar case with a known outcome, and guess that a new case will be similar" is such a natural idea that it's almost impossible to trace its earliest history. The recognition that this idea could be a general, explicit statistical method, along with the name "nearest neighbors", seems to go back to the 1950s (see Cover (1968a) for references). But *because* it's such a natural idea that it keeps getting re-invented in different subjects: in nonlinear dynamics and the physics of chaotic systems, for instance, it was introduced in the 1980s as the "method of analogs" (see Kantz and Schreiber (1997) for references).

k-d trees were introduced by Bentley (1975) (a very clear paper). Gershenfeld (1999), sec. 14.1, gives a brief introduction, and explains how to use them to do density estimation. Cormen et al. (2001) is a deservedly-standard textbook on algorithms and data structures, including efficiently working with search trees.

Combining clustering with kNN, and locality-sensitive hashing: Locality-sensitive hashing is due to Gionis, Indyk, and Motwani (1999). There are good explanations of the idea, and its uses in data mining (beyond just fast nearest neighbors) in Leskovec, Rajaraman, and Ullman (2014), chapter 3. The specific procedure for using k-means clustering as a locality-sensitive hash I sketched above comes from Paulevé, Jégou, and Amsaleg (2010). The general idea of using clustering to speed up finding approximate nearest neighbors in large datasets is however much older (Hand, Mannila, and Smyth 2001, sec. 10.6, p. 352, with references given on p. 365).

References

Arlot, Sylvain, and Alain Celisse. 2010. "A Survey of Cross-Validation Procedures for Model Selection." *Statistics Surveys* 4:40–79. https://doi.org/10.1214/09-SS054.

Azadkia, Mona. 2019. "Optimal Choice of k for k-Nearest Neighbor Regression." E-print, arxiv:1909.05495. http://arxiv.org/abs/1909.05495.

Bentley, Jon Louis. 1975. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM* 18:508–17. https://doi.org/10.1145/361002.361007.

Beygelzimer, Alina, Sham Kakade, John Langford, Sunil Arya, David Mount, and Shengqiao Li. 2013. *FNN: Fast Nearest Neighbor Search Algorithms and Applications*. http://CRAN.R-project.org/package=FNN.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*. Second. Cambridge, Massachusetts: MIT Press.

Cover, Thomas M. 1968a. "Estimation by the Nearest Neighbor Rule." *IEEE Transactions on Information Theory* 14:50–55. http://www-isl.stanford.edu/~cover/papers/transIT/0050cove.pdf.

———. 1968b. "Rates of Convergence for Nearest Neighbor Procedures." In *Proceedings of the Hawaii International Conference on Systems Sciences*, edited by B. K. Kinariwala and F. F. Kuo, 413–15. Honolulu: University of Hawaii Press. http://www-isl.stanford.edu/~cover/papers/paper009.pdf.

Cover, Thomas M., and P. E. Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory* 13:21–27. http://www-isl.stanford.edu/~cover/papers/transIT/0021cove.pdf.

Devroye, Luc, László Györfi, and Gábor Lugosi. 1996. *A Probabilistic Theory of Pattern Recognition*. Berlin: Springer-Verlag.

Geisser, Seymour, and William F. Eddy. 1979. "A Predictive Approach to Model Selection." *Journal of the American Statistical Association* 74:153–60. https://doi.org/10.1080/01621459.1979.10481632.

Gershenfeld, Neil. 1999. *The Nature of Mathematical Modeling*. Cambridge, England: Cambridge University Press.

Gionis, Aristides, Piotr Indyk, and Rajeev Motwani. 1999. "Similarity Search in High Dimensions via Hashing." In *Proceedings of the 25th International Conference on Very Large Data Bases [Vldb '99]*, edited by Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, 518–29. San Francisco: Morgan Kaufmann.

Györfi, László, Michael Kohler, Adam Krzyżak, and Harro Walk. 2002. *A Distribution-Free Theory of Nonparametric Regression*. New York: Springer-Verlag.

Hand, David, Heikki Mannila, and Padhraic Smyth. 2001. *Principles of Data Mining*. Cambridge, Massachusetts: MIT Press.

Kantz, Holger, and Thomas Schreiber. 1997. *Nonlinear Time Series Analysis*. Cambridge, England: Cambridge University Press.

Leskovec, Jure, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets*. Second. Cambridge, England: Cambridge University Press. http://www.mmds.org.

Paulevé, Loïc, Hervé Jégou, and Laurent Amsaleg. 2010. "Locality Sensitive Hashing: A Comparison of Hash Function Types and Querying Mechanisms." *Pattern Recognition Letters* 31:1348–58. https://doi.org/10.1016/j.patrec.2010.04.004.

Wahba, Grace. 1990. *Spline Models for Observational Data*. Philadelphia: Society for Industrial; Applied Mathematics.