Statistical Machine Learning Notes 5

Template Methods

Instructor: Justin Domke

1 Learning by Memorization

Do we really need these complex learning methods we have been talking about? One philosophy is to forget all that and "just use the data". Let's recall our example from the empirical risk minimization notes. We take an input $\mathbf{x}$, which is either CLEAR, CLOUDY, or MIXED, and want to predict and output y, either RAIN, or NOPE.

Our training data would be a set of cloud conditions, with the actual outcome on each day, for example,

$$\{(\text{CLEAR}, \text{NOPE}), (\text{MIXED}, \text{NOPE}), (\text{MIXED}, \text{RAIN}), ...\}.$$

A very simplistic approach for this problem would be the following: on a new input $\mathbf{x}$, look up all the training elements with the same conditions. Then, predict the most common outcome on that set. Suppose we need to predict for the conditions MIXED. If it rained 67% of the time in our training data when we had mixed conditions, then we would predict RAIN.

We might have a more complex input. In addition to the conditions, we might have the temperature, LOW or HIGH. Now, if we want to predict if it will rain on input $(\text{MIXED}, \text{LOW})$ we restrict ourselves to the days matching that input.

How well will this work? If we have a lot of data, and only a few possible inputs like above, it will work very well. The approach makes no assumptions about the relationship between inputs and outputs, and so will converge to the optimal predictor as the dataset gets larger.

However, we run into trouble if there are many inputs. If $\mathbf{x}$ has d binary attributes, there are a total of 2^d possible input vectors. This means that, for large d, we will need an *exponentially increasing* amount of data to accurately predict if it will rain in all possible situations. Thus, we pay a huge variance price for the low bias of simple memorization. This is our first glimpse of a problem called the "Curse of Dimensionality", which plagues attempts to fit high-dimensional predictors with out making simplifying assumptions– (that is, with out increasing bias).

Another problem is what to do if the input is not discrete. What if we measure the temperature in Kelvins, instead of LOW/HIGH? If $\mathbf{x}$ is a general real vector, we can't assume that the exact same input should ever occur more than once. One approach would be to discretize $\mathbf{x}$ somehow, and then apply learning by memorization. This can work reasonably well, but introduces several issues. For example, how big should the grid be? Below, we explore a more elegant solution working directly in the real space.

2 K-Nearest Neighbors Methods

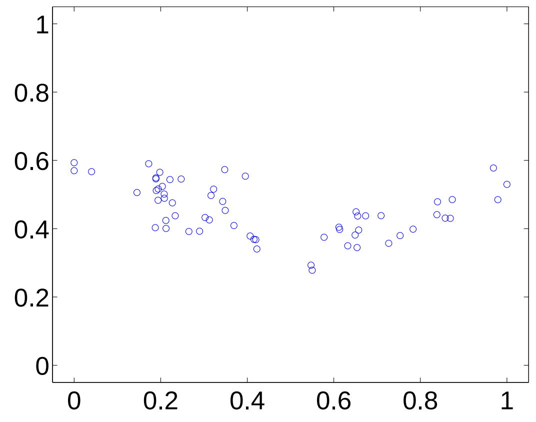
Nearest-neighbor methods can be used for either regression or classification. Given some dataset $D = \{(\hat{\mathbf{x}}, \hat{y})\}$ let $N_k(\mathbf{x})$ be the set of k closest "neighbors" of $\mathbf{x}$, as measured by Euclidean distance. (Originally, these notes introduced a bunch of equations to try to define this formally, but this appears to create more problems than it solves.)

For regression, the standard k-nearest-neighbors predictor is

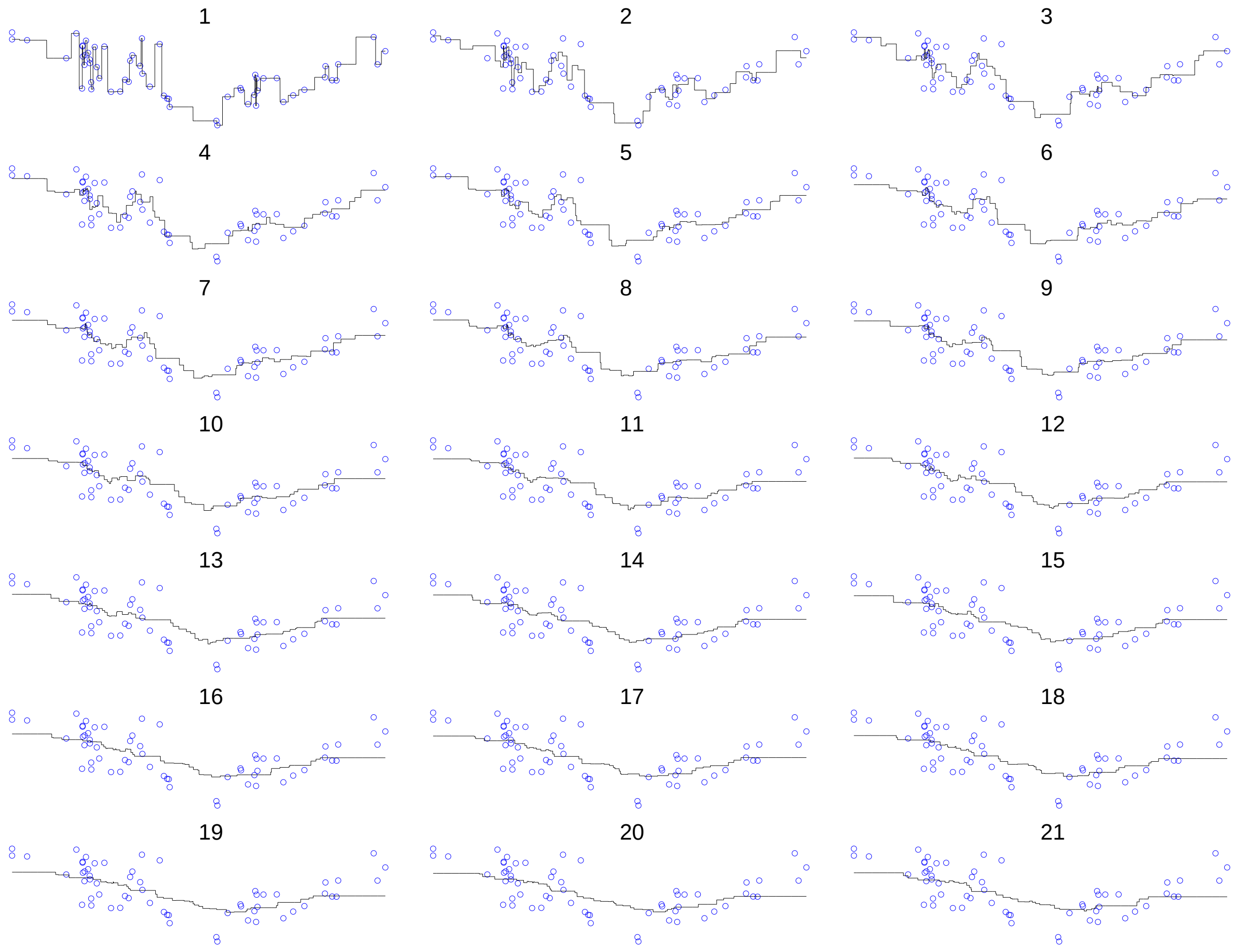
$$f(\mathbf{x}) = \frac{1}{k} \sum_{\{(\hat{\mathbf{x}}, \hat{y})\} \in N_k(\mathbf{x})} \hat{y}.$$

Operationally, we input the new point $\mathbf{x}$, find the k closest points to $\mathbf{x}$ in the dataset, and output the mean of these.

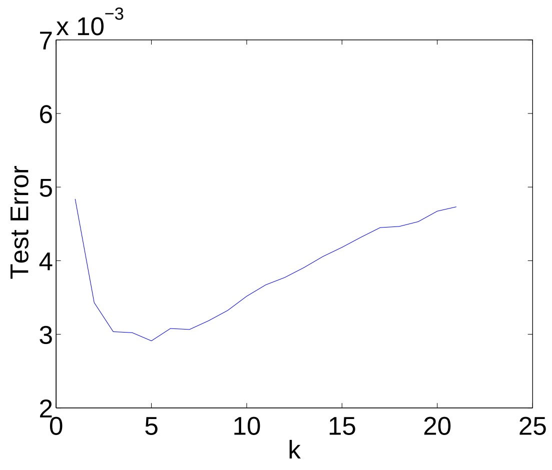
Recall our one-dimensional dataset from the cross-validation notes.



Here, we plot $f(\mathbf{x})$ for $k = 1, 2, ..., 21$.



We can see that the graphs gets progressively smoother for larger k. If we plot the error on a large test set, we see that the minimum is at $k = 5$. This looks intuitively plausible from the graphs above.



For classification, k-NN generally uses the rule

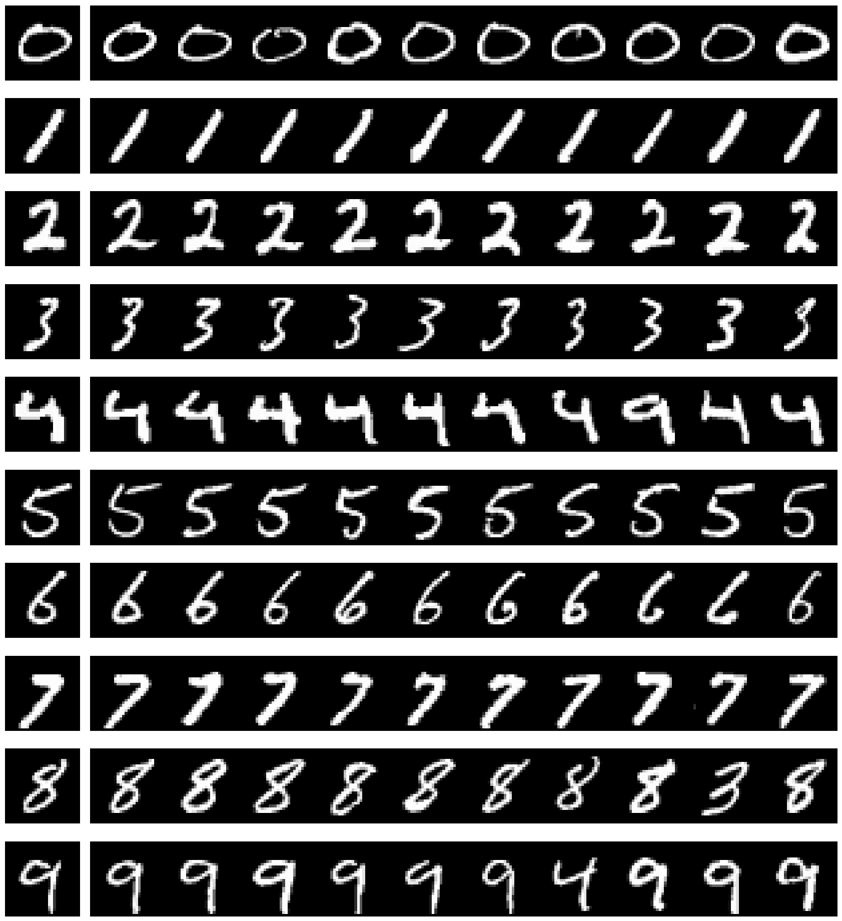
$$f(\mathbf{x}) = \arg\max_y \sum_{\{(\hat{\mathbf{x}},\hat{y})\} \in N_k(\mathbf{x})} I[y = \hat{y}].$$

Again, here, the notation seems much more complicated than the recipe it describes. Operationally, we find the k closest points N_k, and then output the most common class in that set.

Notice that this applies easily to *multiclass* classification. We can still use this algorithm if there are, say, 10 classes.

A bothersome detail is that we can encounter "ties". For binary classification, this can be avoided by setting k to be an odd number, but for multiclass classification this is unavoidable. In general, we can break ties randomly.

3 MNIST



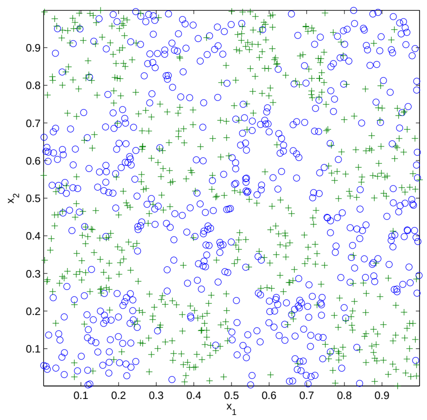
The MNIST dataset is a set of 60,000 28×28 images of handwritten digits, along with 10,000 test images. A huge number of different algorithms have been applied to it[1]. Here, we treat these simply as 728 dimensional vectors and apply 10 nearest neighbors. The figures above show the ten nearest neighbors found for one test image from each of the 10 digits. In most cases, there are several training images quite close the the test image. A 10-NN classifier on this dataset has an error rate around 5%. This can be improved significantly by various pre-processing heuristics, such as centering or blurring the data.

4 Synthetic Example

To try to understand the properties of linear methods, we will consider a simple "grid" dataset[2]. Each component of $\mathbf{x}$ is distributed uniformly from 0 to 1. The class labels y are binary, given by

$$\hat{y} = \text{mod}(\sum_i \lfloor 4\hat{x}_i \rfloor, 2),$$

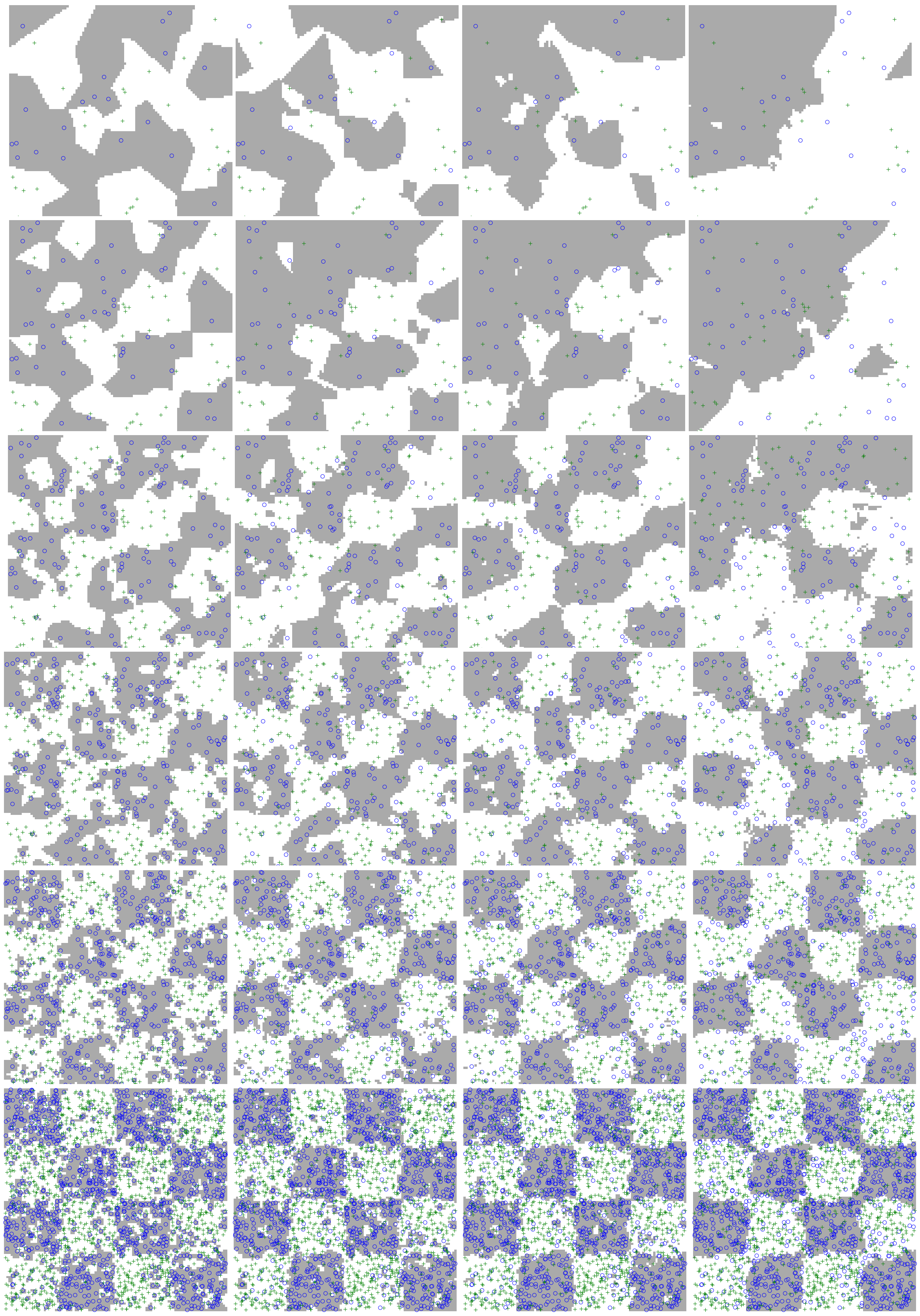
with noise consisting of 15% of labels flipped randomly. This looks complicated, but we can understand it pretty easily by just looking at a 2-D plot. (Here, circles shows points with $y = 0$, and pluses show points with $y = 1$.)



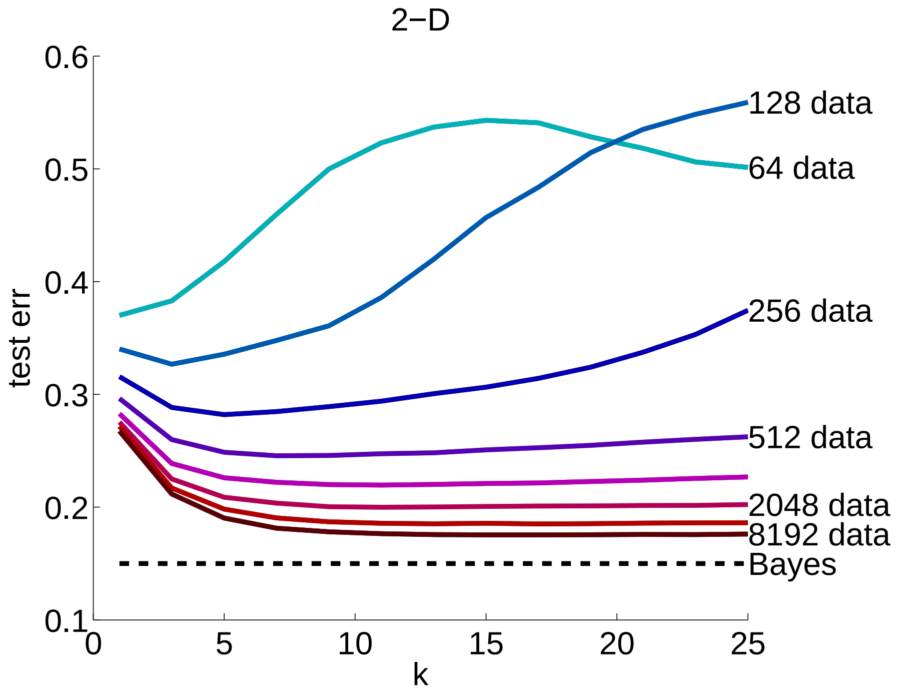
The next set of figures show examples with datasets of size 50, 100, 250, 500, 1000 and 2500 with $k = 1, 3, 5$, and 25. The gray regions show the part of the plane with $f(\mathbf{x}) = 0$, while the white regions show the part of the plane with $f(\mathbf{x}) = 1$.

[1]See `http://yann.lecun.com/exdb/mnist/` for details.

[2]This is inspired by the synthetic data in section 13.3.1 of The Elements of Statistical Learning.


k=1
k=3
k=5
k=25

We will explore the average test error as a function of N (the training size), k (the number of neighbors) and d (the number of dimensions for $\mathbf{x}$). First, we freeze $d = 2$. The following plots are averaged over many randomly generated training and test sets.



Since there are 15% randomly flipped labels, even a perfect classifier will suffer from at least 15% misclassification error. By tradition, the best possible classifier is called the **Bayes optimal classifier**, and the corresponding error rate the **Bayes error rate**. This name appears to descend from from Bayes' rule of probability $p(y|x) = p(x|y)p(y)/p(x)$, not from anything having to do with Bayesian statistics. Thus, the Bayes error is a totally uncontroversial concept in both the frequentist and Bayesian worldviews.

There are several trends to observe here. Most obviously, more data reduces test error. However, even with 8192 data, we are still noticeable above the Bayes error. Notice that even though the curves represent exponentially increasing amounts of data they still appear to get closer together. Thus, as we get closer to the Bayes error, more and more data is required for improvement.

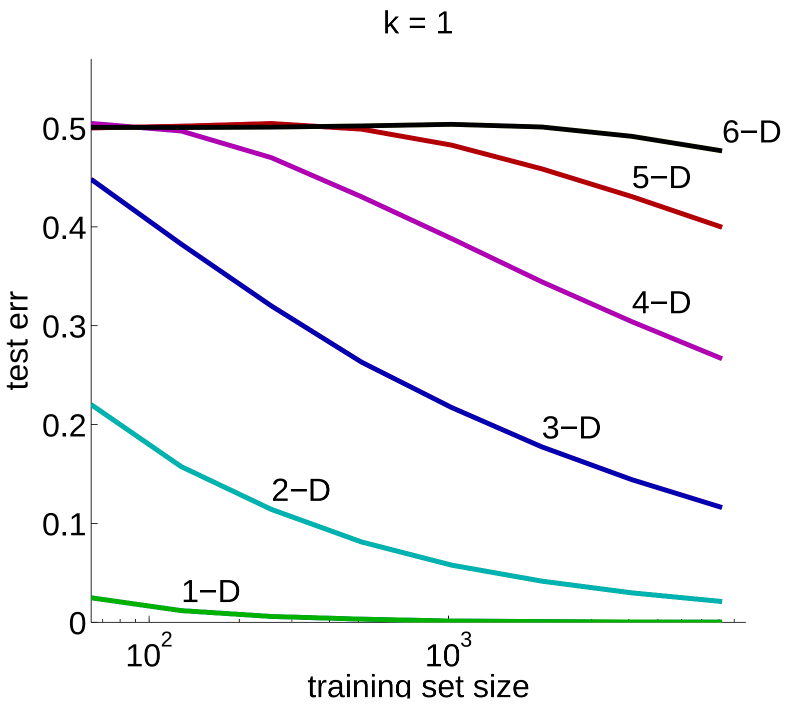
Secondly, we notice that for small amounts of data, the best results are given for small k, while for larger amounts of data we do better with larger k. This is caused by a tradeoff between two issues, which can be understood intuitively from the above plots.

1. With small k, the classifier that results is more "noisy". Even with an *infinite* amount of data, the 1-NN algorithm will not achieve the Bayes error. Consider the plots above

with the largest dataset. The optimal classifier is a perfect "checkerboard". However, a test point somewhere in a predominantly + region has a 15% chance of being closest to an o. We can calculate that there is a $.15 \cdot .85 + .85 \cdot .15 = .255$ probability of error. (15% chance of getting "wrong" label, in which case we have 85% probability of error, plus 85% chance of getting the "right" label, in which case we have a 15% probability of error.) As k gets bigger, we will have a near 100% probability of getting the "right" label, with 15% probability of error.

2. With larger k, the classifier is more "smoothed". Clearly, in the limit $k \to \infty$, $f(\mathbf{x})$ will just be a constant, of whatever class happened to be more common in the dataset. For finite k, we have a similar problem– the regions of a single class become larger and larger.

Now, we switch gears, and compare error rates for varying dimensions. For these experiments, there is no noise, and so the Bayes error is zero.



We can see that, somehow, dimensionality is extremely harmful to the performance of the algorithm. We will investigate the cause of this in the next section.

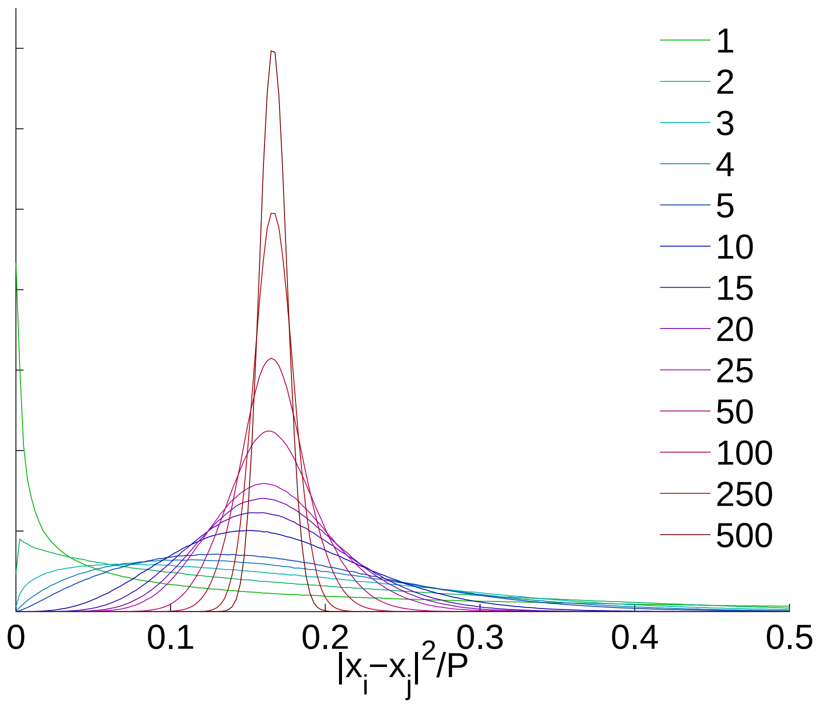
5 The Curse of Dimensionality

With learning by memorization, as we saw in the first section, if we have d binary variables, there are 2^d joint configurations. Thus, to maintain a constant number of examples to average for each configuration, we will need an amount of data exponential in d.

For non-discrete data, we face fundamentally the same problem, though understanding it is not quite so straightforward as it requires a bit of understanding of the geometry of high-dimensional spaces.

Consider the following experiment. For data of varying dimensionality P, generate a large number of vectors $\mathbf{x}_i$, with each component uniformly distributed between 0 and 1. Then, create histograms of the squared distances between all the points, divided by the number of dimensions $|\mathbf{x}_i - \mathbf{x}_j|^2/d$. For high d, we see an increasingly clustering around $\frac{1}{6}$.

(**Why** $\frac{1}{6}$? For two variables drawn uniformly from $[0,1]$, the average squared distance between them is $\int_{x=0}^{1}\int_{y=0}^{1}(x-y)^2 dxdy = \frac{1}{6}$. For large P, we are averaging many such values.)



Though it may not be obvious at first glance, this is terrible news. With high d, for a particular query point, the distances to all points in the dataset tend to be about the same. The nearest neighbor will have a distance of nearly $\frac{1}{6}$, just like the *furthest* neighbor. There is so much "space" in high dimensions that if distribute points equally, it is hugely improbable that any two of them will be close together. If all points are nearly equidistant, there is no neighborhood structure to exploit, and so nearest neighbors breaks down.

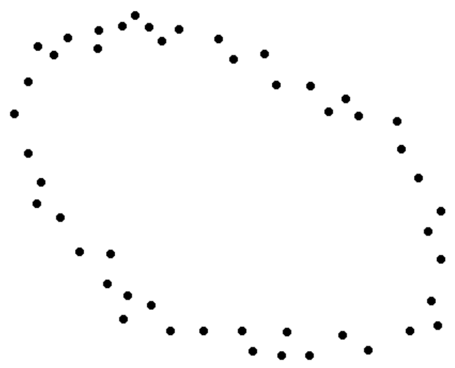
Now, a word of caution. The above experiments are for data *uniformly distributed* in high dimensions. For non-uniform data, nearest neighbors can and do sometimes continue to work well. To understand why, suppose we have 100 dimensional data with x_1 uniform on $[0,1]$ and $x_2 = x_3 = ... = x_{100} = .5$. Since 99 of the 100 dimensions are constant, they will contribute nothing to distances, and so a nearest neighbors method will behave exactly as if we had data consisting of x_1 alone.

More generally, suppose that

$$\mathbf{x}_i = A\mathbf{z}_i$$

where $\mathbf{z}$ is a q dimensional vector and A is a $d \times q$ matrix. Here again the data $\mathbf{x}_i$ will only lie in a q-dimensional subspace of $\Re^d$, and so nearest neighbors is likely to work well if q is small.

In practice, we usually don't expect the data to exactly lie in a linear subspace, but only "close to" some sort of low-dimensional "manifold". For example, the following data in 2-D is "almost" one-dimensional in nature. Nearest neighbors will "automatically" take advantage of structure like this.



6 Relationship to Loss Functions

In our discussion so far, we have not mentioned loss functions. Do these have a place in an understanding of nearest-neighbor methods? To get started, suppose we are doing regression. Instead of predicting

$$f(\mathbf{x}) = \frac{1}{k} \sum_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} \hat{y} = \operatorname*{mean}_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} \hat{y},$$

couldn't we do some variation, like, say,

$$f(\mathbf{x}) = \underset{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})}{\text{median}} \hat{y}?$$

What does it mean to take the median instead of the mean? Are there other choices? How does this reflect our priorities? As we will see, taking the mean can be seen as implicitly reflecting a *least-squares* loss function, while the median implicitly reflects a *least-absolute deviation* loss function.

The way to understand this is that nearest neighbor methods make a quick and dirty estimate of the conditional probability of y given $\mathbf{x}$. Then, these methods minimize the loss "on the fly" for that particular point.

For example, if the neighbors are $N_3(\mathbf{x}) = \{(\hat{\mathbf{x}}, .2), (\hat{\mathbf{x}}, .5), (\hat{\mathbf{x}}, .9)\}$, 3-NN puts $1/3$ probability at .2, .5, and .9, and zero everywhere else. More generally, methods put $1/k$ probability at each of the nearest neighbors. Of course, this isn't a *good* estimate of the conditional probability, but as long as we don't do anything too fancy with it, we can hopefully get away with this estimate.

(**Aside**: Readers who took mathematical analysis– yes, yes, what I *really* mean is that there are delta functions centered at .2, .5 and .9. Here we made a tactical decision not to bother with that level of rigour since the basic idea can be made pretty clear with out it. Thank you for noticing, though!)

Now, suppose we have to pick a single y to minimize the risk, using this rough estimate of the conditional distribution. That is, we want to find

$$c^* = \arg\min_c E_{\hat{p}}[L(c, y)],$$

where $\hat{p}$ is our rough estimate of the conditional probability. This is equivalent to

$$c^* = \arg\min_c \frac{1}{k} \sum_{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})} L(c, \hat{y}).$$

Let's think about what is happening here. Back before, when fitting linear methods, we would fit a single relatively complex model to try to fit all the data well. Here, we fit a very simple model (a constant!) to try to predict just nearest neighbors of $\mathbf{x}$. Thus, we trade a powerful model for a simple one, but compensate for this by only fitting the model *locally.*

Let's see how this works out with various loss functions. If we use the least-squares loss $L(c, y) = (c - y)^2$ for regression, we can set the derivative of

$$\frac{1}{k}\sum_{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})}(c-\hat{y})^2$$

with respect to c to zero to find

$$c^* = \frac{1}{k}\sum_{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})}\hat{y}.$$

If we use the least-absolute deviation loss, the non-differentiability of the loss prevents a derivative analysis. However, it isn't too hard to show (assuming k is odd) that

$$c^* = \underset{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})}{\text{median}}\ \hat{y}.$$

What about classification? If we need to minimize the $0-1$ loss, we would do

$$c^* = \arg\min_c \frac{1}{k}\sum_{(\hat{\mathbf{x}},\hat{y})\in N_k(\mathbf{x})} I[c\neq\hat{y}],$$

which is clearly minimized by setting c^* to be the class that is most common in $N_k(\mathbf{x})$. We could also consider fancy loss functions that penalize different types of mistakes differently. (For example predicting A when the true class is B has a higher loss then predicting B when the true class is A). These can yield different decisions than always just picking the most common class.

7 Extensions

We can extend K-NN in a bunch of ways.

- **Pre-processing.** As mentioned above with the MNIST data, one can often use domain knowledge to pre-process data and improve results. This can be very useful to enforce "invariants" that you may know about. For example, with the MNIST data, we know that if we shift one of the images a few pixels in a given direction, it will still be the same digit. Similarly, we can rotate an image a few degrees, and still represent the same digit. One practical way to use this knowledge is to take each training image, and rotate and shift it randomly a bunch of times. If we use the results as a new "synthetic" dataset, we are likely to reduce errors.

- **Distance Measures**. Obviously, we don't have to measure "nearest" by the Euclidean distance. In some cases, another metric might be suggested by the problem. Alternatively, one could pick a metric by cross-validation.

- **Weighting**. With regression above, we would take the mean of all the nearest neighbors. Intuitively, however, it is natural to give more influence to the *closest* neighbors than the furthest ones. One can assign weights r to each of the neighbors, giving more weight to the closest ones. Then, the *weighted* mean is taken instead of the mean.
- **Local fitting.** Rather than just picking the mean of the nearest neighbors, why not *fit a local model* to them, and then predict from that? We can write this as

$$f(\mathbf{x}) = g(\mathbf{x}), \;\; g = \arg\min_{g \in F} \sum_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} r((\hat{\mathbf{x}}))\, L(g(\hat{\mathbf{x}}), \hat{y})$$

Let's consider a few examples of this.

1. Uniform weights $r = 1$, the set of constant functions $F = \{g(\mathbf{x}) : g(\mathbf{x}) = c\}$, and the squared loss. Then, we have

$$f(\mathbf{x}) = c, \;\; c = \arg\min_{c} \sum_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} (c - \hat{y})^2$$

   which is equivalent to

$$f(\mathbf{x}) = \operatorname*{mean}_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} \hat{y},$$

   i.e. good old fashioned nearest neighbors.

2. Uniform weights $r = 1$, the set of linear functions, $F = \{g(\mathbf{x}) : g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}\}$, and the squared loss, but use all the data, with $k = |D|$. Then, we have

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}, \;\; \mathbf{w} = \arg\min_{\mathbf{w}} \sum_{(\hat{\mathbf{x}},\hat{y})} (\mathbf{w} \cdot \mathbf{x} - \hat{y})^2.$$

   This is just regular least-squares regression. Thus, we can see that this framework generalizes both nearest-neighbors, and linear methods.

3. Uniform weights $r = 1$, the set of linear functions, and the squared loss. Then, we have

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}, \;\; \mathbf{w} = \arg\min_{\mathbf{w}} \sum_{(\hat{\mathbf{x}},\hat{y}) \in N_k(\mathbf{x})} (\mathbf{w} \cdot \hat{\mathbf{x}} - \hat{y})^2$$

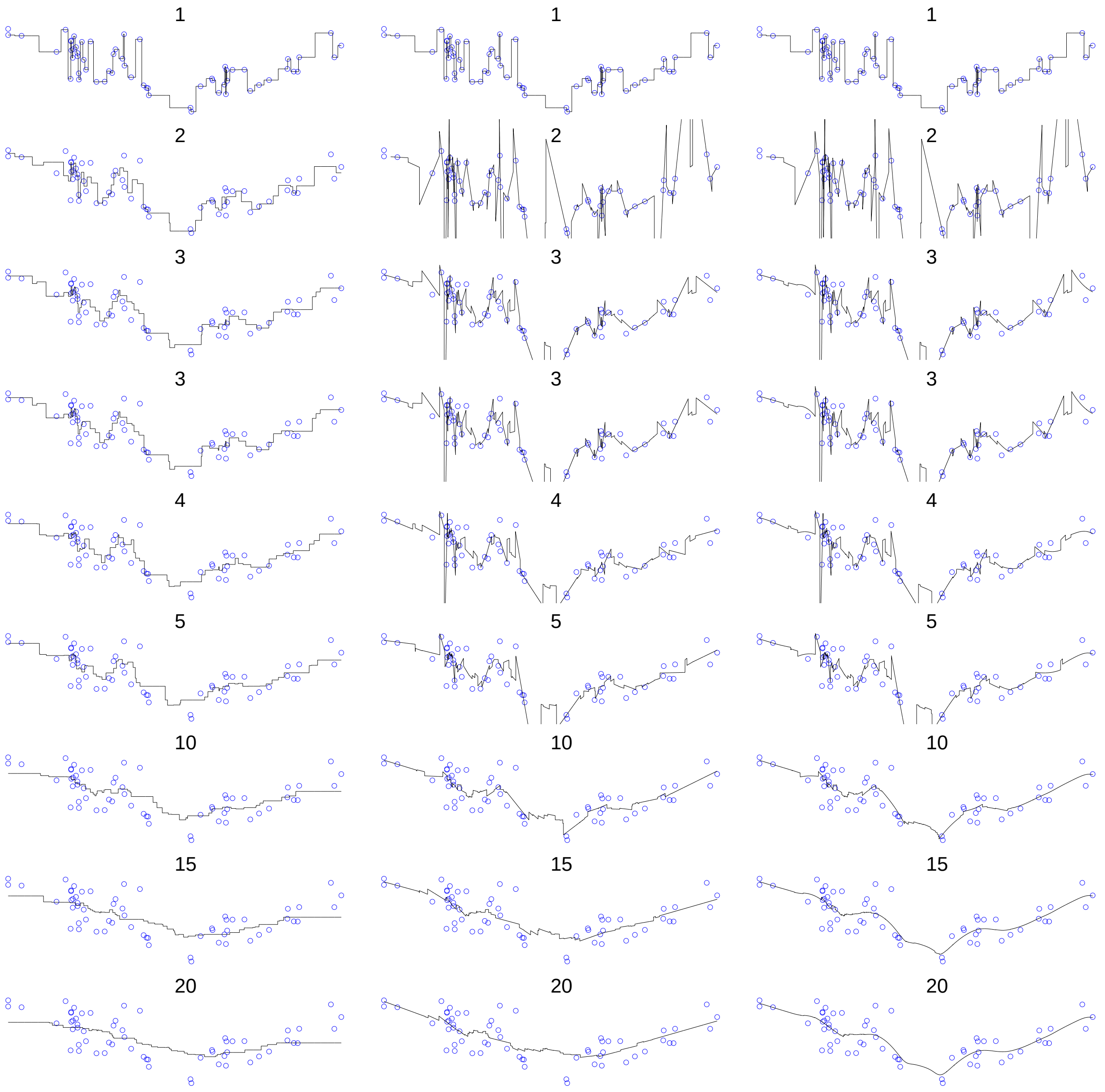
   i.e. we just fit a linear function to the nearest neighbors, then use this to predict the new point.

4. The set of linear functions F again. Use all the data, $k = |D|$, but give more weight to neighbors via Gaussian weights like

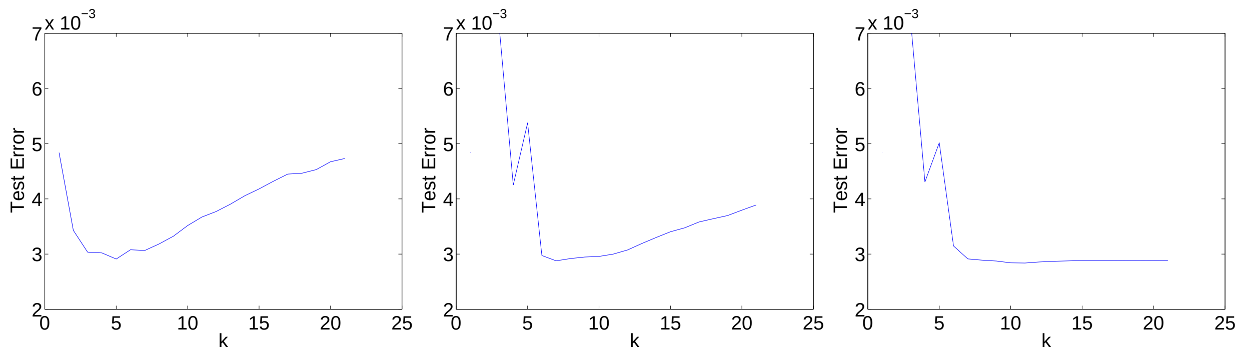
$$r(\hat{\mathbf{x}}) = \exp\big(\frac{1}{2\sigma^2}||\hat{\mathbf{x}} - \mathbf{x}||^2\big).$$

Locally weighted regression is sometimes called LOESS or LOWESS. Of course, all these things could be combined with different loss functions, different weights, etc.

The following plots compare traditional K-NN (left) with unweighted local regression (middle), and weighted local regression with $\sigma^2 = \frac{1}{250}$ (right).



Here we show the test errors for the three methods. Unweighted local regression does best for a larger k than standard K-NN. Further, weighted local regression does best if we allow an essentially unlimited number of neighbors. This has the effect of considering more neighbors in dense areas, since the number of neighbors *with significant weight* depends on the sampling density. It is important, however, to set σ^2 appropriately.



8 Nearest Neighbors vs. Linear Methods

The great advantage of nearest neighbors is that it makes few assumptions about how $\mathbf{x}$ is related to y beyond smoothness. However, we pay a price in variance for this. Thus, linear methods tend to perform better with high dimensions and few data, while nearest neighbors commonly performs better with low dimensions and many data. (Note that these are heuristic guidelines only.)

In training time, nearest neighbors has an unbeatable time complexity of $O(0)$. (If you assume the data has already been written down somewhere, there is literally nothing to do.)

At test time, however, linear methods can be much faster. With d dimensions, we can compute $\mathbf{w} \cdot \mathbf{x}$ in time $O(d)$. With nearest neighbors methods, however, we must compute the distance of a query point to each point in the dataset. Done naively, this has a complexity of $O(Nd)$. In large-scale applications, Nd might represent gigabytes of data, and so this can be prohibitive. In low dimensions, algorithms exist to do this faster (e.g. in time logarithmic in N). In high dimensions there have been very exciting recent results in the theoretical algorithms community to produce randomized approximation algorithms for nearest neighbors–algorithms that *usually* find a point *close* to the nearest. For one such method, search for "Locality Sensitive Hashing".

As we will see when we get to SVMs, there are hybrid methods that blur the line between template methods and linear methods.

9 A Theoretical Claim

If you read about nearest neighbor methods almost anywhere, you will soon encounter some variant of the following claim:

> Asymptotically, the 1-NN algorithm has an error rate no worse than twice the best possible of any algorithm.

This is true. However, without an understanding of *why* this is true, one can get the impression that nearest-neighbor methods posses magical powers. In particular, the origin of the factor of two is rather mysterious.

To try to give an explanation for why this is true, let's first consider a much simpler situation, namely a game in which we want to predict rolls of a fair but biased N-sided die. The true distribution gives a weight of p_i to side i of the die.

What is the best way to guess rolls of the die? Clearly, this is to just repeatedly guess the side of the die with highest probability.

> **Guess Best**
>
> Always guess $i = \arg\max_i p_i$.

An alternative (less good) procedure, would be to guess randomly. If you have your own die, with the same probabilities, you just roll that die, and predict whatever comes up.

> **Guess Randomly**
>
> Guess side i with probability p_i.

Now, what will the errors of these two procedures be? Let p^* be the probability of the most likely face. It is pretty easy to see that the Guess Best procedure will have an error of

$$1 - p^*$$

since it will be wrong whenever the most likely side doesn't come up. Now, we can also calculate that Guess Randomly will have an error rate of

$$\sum_i p_i(1 - p_i).$$

Through some algebra, we can show that Guess Randomly's error rate is at most twice Guess Best's. The key idea is to consider the "worst case": for a given p^*, what is the worst possible way to set all the other p_i in order to maximize Guess Randomly's error rate. It turns out that the worst case is to just set all the other p_i *equally.* This is pretty intuitive. You can see that Guess Randomly will do much better with a die with probabilities $(.5, .5, .0)^T$ (error rate .5) than one with probabilities $(.5, .25, .25)^T$ (error rate .875). Allocating equally means setting $p_i = \frac{1}{N-1}(1-p^*)$. Intuitively, take the remaining $1-p^*$ probability after giving p^* to the best face, and allocate equal chunks for the other $N-1$ faces of the die. Then, we have

$$\begin{aligned}
\sum_i p_i(1-p_i) &= p^*(1-p^*) + (N-1)\frac{1}{N-1}(1-p^*)\big(1-\frac{1}{N-1}(1-p^*)\big) \\
&= p^*(1-p^*) + (1-p^*)\big(\frac{N-1}{N-1} - \frac{1-p^*}{N-1}\big) \\
&= p^*(1-p^*) + (1-p^*)\frac{N-2+p^*}{N-1} \\
&= (1-p^*)\big(p^* + \frac{N-2+p^*}{N-1}\big) \\
&= (1-p^*)\big(\frac{p^*N-p^*}{N-1} + \frac{N-2+p^*}{N-1}\big) \\
&= (1-p^*)\big(\frac{p^*N-1}{N-1} + \frac{N-1}{N-1}\big) \\
&= (1-p^*)\big(\frac{p^*N-1}{N-1} + 1\big) \\
&\leq 2(1-p^*)
\end{aligned}$$

The details are presented here for your enjoyment, but you don't really need to follow everything in full. From the first line to the second to last line is just algebra. In the last step, we exploit the fact that $p^* \leq 1$, and so $p^*N \leq N$.

The bottom line is that Guess Randomly makes at most twice as many errors as Guess Best.

So! That was fun, but what does this all have to do with nearest neighbors? Imagine trying to guess the class y corresponding to an input $\mathbf{x}$. If we had the true conditional distribution $p(y|\mathbf{x})$, the best bet would be to pick $y^* = \arg\max_y p(y|\mathbf{x})$, which will be wrong with probability $1 - \max p(y|\mathbf{x})$. You can easily see that this is the best possible procedure. Now, a less optimal strategy would be to draw y' randomly from $p(y|\mathbf{x})$ and then guess y'. This will be wrong with probability $\sum_y p(y|\mathbf{x})(1-p(y|\mathbf{x}))$. By our reasoning above with the dice, the sampling approach will make at most twice as many errors as the optimal approach.

How do we connect this to nearest neighbors? Well, in the infinite data limit, picking a 1-NN is the same thing as sampling from $p(y|\mathbf{x})$! (Under some very mild technical conditions.)

Let's think about this in practice. Suppose the inputs $\mathbf{x}$ are 1000x1000 grayscale images of human faces, while y is the sex of the person in the image. The above theory basically claims that if we have enough data, we can predict the sex of a new person by simply finding a near-identical image in the database, and outputting the sex of that person. This is true, but wow will we need a lot of data to do it!

Thus, we can think of this claim in two parts:

1. In the infinite-data limit, 1-NN is equivalent to sampling from $p(y|\mathbf{x})$.
2. Sampling from $p(y|\mathbf{x})$ makes at most twice as many errors as optimal guessing.

The first is true, but of unknown value in practice where we have finite data. The second is a relatively mundane observation about probability.