CHAPTER

2 Nearest Neighbor Methods

A very simple class of learners are nearest neighbor models. In essence, nearest neighbor methods reflect the most basic concept of prediction: if we wish to predict the target value associated with a feature measurement x, we should mainly use the data located near x to estimate it. “Neighbor-based” methods store a set of training examples, and make their predictions by finding the closest examples to x. This is an example of “exemplar-based” modeling, in which specific data examples are stored and used to define the model.

2.1 Nearest neighbor prediction

Recall that we are given a set of training examples, which we denote as $D = \{x^{(i)}, y^{(i)}\}$. Then, the *nearest neighbor* (or 1-nearest neighbor) predictor $f(x)$ takes the form

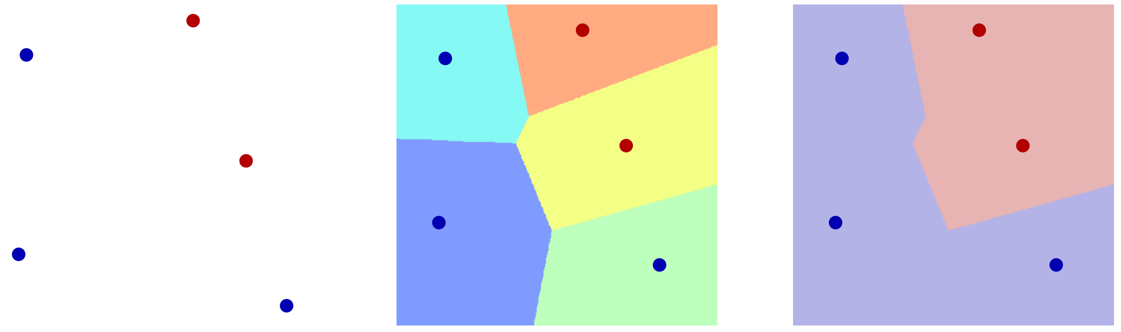
$$f(x\,;\,D) = y^{(i^*)} \qquad \text{where} \qquad i^* = \arg\min_i d(x, x^{(i)}),$$

where $d(x, x')$ is a *distance* or *dissimilarity* function that measures how far the vectors x is from x'. The $\arg\min$ operator then finds the data point i^* whose feature vector $x^{(i^*)}$ is closest to the test point x, and our predictor returns that data point’s value $y^{(i^*)}$. This procedure trivially works for both classification and regression, since we are returning some data point’s target value.

In practice, most implementations of nearest neighbor methods use the standard (squared) Euclidean distance,

$$d(x, x') = \|x - x'\|^2 = \sum_j (x_j - x'_j)^2.$$

Note that, since we are only interested in *which* data point is closest, we do not need to bother taking the square root; the data point i^* that has the smallest distance will also have the smallest squared distance. However, a major strength of nearest-neighbor methods is that they can easily use arbitrary, potentially complex distance functions as well. For example, if our data points correspond to text documents, we may be able to define a distance function (or, conversely, a similarity function) that measures how close two documents are, and use this to select our nearest neighbor. For sequence data, such as in genetics, we could define similarity by aligning two sequences, or computing their “edit distance”, the number of changes that need to be made to transform one into the other. For simplicity and for easy visualization, here we confine our attention to using Euclidean distance.



(a) Data (b) Voronoi regions (c) Decision function

Figure 2.1: (a) A small set of 5 training data for classification. Using a nearest-neighbor classifier, the learner always predicts the value of the closest data point. (b) We can view the data points as partitioning the space into polygons, each point "claiming" all locations x that are closer to it than any other point. (c) For a nearest-neighbor classifier, the decision boundary is then piecewise linear, composed of the subset of boundaries between data of different classes.

"Learning" such a predictor is essentially trivial – we simply store the training data in a database, and at test time, we find the data point i^* which is closest to the test point x. However, this means that the memory requirements of the learner are $O(mn)$, i.e., increase linearly with both number of data (m) and their size (the number of features, n). Similarly, each prediction at test time must find the nearest data point, which is nominally also $O(mn)$, to search through m points and calculate a sum of n squared values for each. (Both storage and prediction time can sometimes be improved using computational tricks; see Section 2.5.)

Characterizing the learner

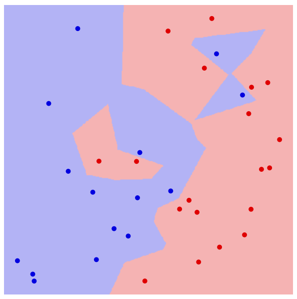
It is easy to see that the resulting function $f(x\,;\,D)$ is piecewise constant, with abrupt (discontinuous) changes in its predictions only when the identity of the nearest data point changes. Consider the classification example in Figure 2.1, with two features x_1, x_2, and the target value y indicated by color. We can think of each training example, pictured in Figure 2.1(a), as "claiming" a region of the feature space that is closer to that example than any other. For Euclidean distance, the set of points that are equidistant between $x^{(i)}$ and $x^{(j)}$ is simply the line (or in higher dimensions, plane or hyperplane) that bisects the line segment connecting $x^{(i)}$ and $x^{(j)}$. Thus, the data points partition the feature space into a set of polygons, whose faces are located midway between two neighboring data; this partitioning is called the "Voronoi regions" of the data points (see Figure 2.1(b)). Within each polygon, our prediction is the class of its associated data point. Recall that the decision boundary is the set of points at which our classifier (or decision function) f changes value; here, it is easy to see that the decision boundary is piecewise linear, made up of the segments of the partitioning whose data have different values (Figure 2.1(c)).

The same reasoning applies to regression in more than one dimension; each data point $x^{(i)}$ "claims" a region of feature space around it that is closer to $x^{(i)}$ than any other point $x^{(j)}$, $j \neq i$, in the data set, and our prediction within this region is the constant value, $y^{(i)}$.

One point about nearest neighbor methods, and exemplar-based models more generally, is that their potential to represent complex functions increases with the number of exemplars they store. For example, a nearest neighbor model with only four data points must correspond to a relatively simple decision function: the function can only partition the feature space into four parts. But, the more data are stored by the model, the more complex the predictor f may be.

In general, the more data we are able to use, the better our model will fit. In fact, a well-known result assures us that, given sufficiently many data, a nearest-neighbor classifier will have a misclassification rate that is at most twice the Bayes optimal error rate. Intuitively, at any point x, the Bayes optimal predictor predicts the most likely class, $\hat{y} = \arg\max_y p(Y = y|X = x)$, and has only one source of error: that the test data point happens to be drawn from some less likely class, which happens with probability $\rho = 1 - p(Y = \hat{y}|x)$. Now, consider a nearest neighbor classifier at x. Given enough data ($m \to \infty$), there will be a training point extremely close to x, so that this point's target value has distribution $p(Y|X = x)$. Thus, with probability $p(Y = \hat{y}|X = x)$ the training point comes from class $\hat{y}$, and the classifier is (at x) the same as the Bayes optimal predictor. Like the test point, the chance that the training point comes from some less probable class is $\rho = 1 - p(Y = \hat{y}|x)$, and thus the probability of an error cannot be greater than 2ρ.

So, a sufficient number of data guarantees our error rate will be within a small factor of the optimal performance. However, increasing the number of data will also dramatically increase the complexity of the predictor; we discuss this in more depth in Sections 2.2 and 2.4.



This also illustrates the point that, given enough data, a significant source of error for nearest neighbor comes from observing a training data point of a less-probable class for that location. Clearly this can happen whenever $p(Y|X = x)$ is nonzero for more than one value of Y (or equivalently, whenever two class-conditional distributions $p(X|Y = y)$ overlap at feature value x). We see this effect in the figure at right, where two red (class 1) data point can be found in a region that is, visually, mostly blue data (class 0), and vice versa. Looking more holistically, it seems probable that new data in the resulting small red or blue regions given by the nearest neighbor rule are more likely to be of the other class. This motivates the idea of looking more broadly than just the nearest point when making our prediction, for example looking at several nearby data points to "smooth out" such noise.

2.2 K-Nearest neighbor methods

K-nearest neighbor predictors comprise a simple generalization of nearest neighbor, in which we identify the k nearest data points and combine them to make our prediction.[1] Typically, for regression problems, we predict the average of the target values of the k nearest points, while for classification we predict the class by a majority vote of the points (breaking ties in any pre-defined way).

Conceptually, the idea is that rather than selecting only one training example as an analogue for x, we instead select k examples, treating them as equally valid stand-ins for the test point. As we saw in Chapter 1, the squared error loss is minimized by predicting the average, or expected value of Y, while the 0/1 classification loss is minimized by choosing the most probable class. Hence for these losses we use the average or majority vote of the k points. For other loss measures, we might substitute a different function of the k points meant to minimize the desired loss.

Increasing k makes the decision function more smooth, in the sense that it depends on data from a larger region of the feature space. This often leads to simpler- and smoother-looking decision boundaries; see Figure 2.2. This statement is imprecise, of course – the actual decision boundary remains piecewise linear, since it corresponds to locations at which the k nearest points change identity (which, by the

[1]When $k = 1$, this reduces to the standard nearest neighbor method.

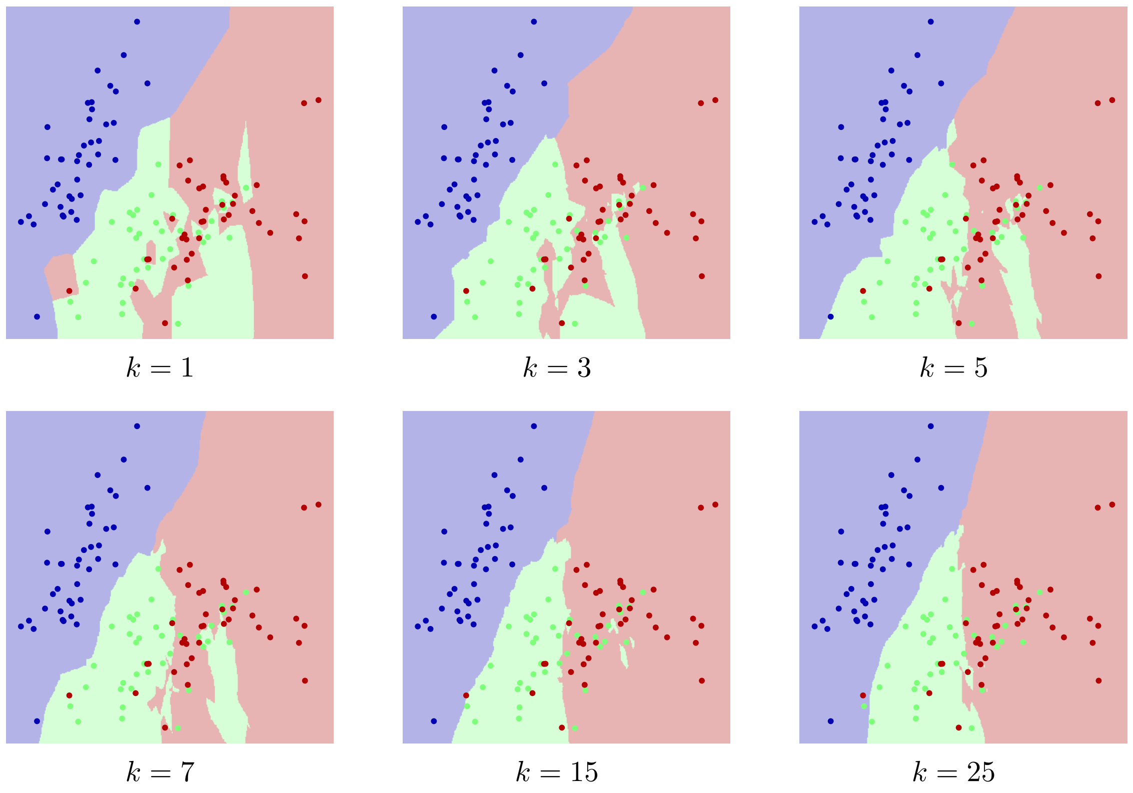


Figure 2.2: A k-nearest neighbor classifier on the Iris data, for varying settings of k. At $k = 1$, each training point claims a region and predicts its target value, making many small "islands" of different predictions. As k increases, each location looks at a larger area to decide its prediction, so that a single red data point in a region dominated by blue, for example, will start to predict blue instead. By $k = 25$, our predictions are holistically simpler and more regional, predicting blue in the upper left, green at the bottom, and red to the right. The resulting classifier looks "simpler"; we can compare the decision boundary, for example, to the linear boundaries formed by the equal-covariance Gaussian Bayes classifier in Example 1-6.

same argument as nearest neighbor, must correspond to locations where the kth nearest point becomes equidistant to the $k + 1$st). Since this involves more potential subsets of points, the decision boundary may actually have more segments. But, speaking imprecisely, since at each point x the function depends on the average of several data points, there is less emphasis on individual points (which may be noisy, e.g., have lower-probability class values Y), and we can see that the function looks less complex and "noisy" as k increases.

We can see basically the same effect in k-nearest neighbor regression, illustrated in Figure 2.3. Here, we have a single feature X, and the target value Y of each data point is shown on the vertical axis. Our prediction is the average of the k nearest points. When $k = 1$, the function changes values rapidly to follow noise in the training data, giving it a too-jagged, noisy look. At $k = 5$, we are averaging over a few data points, and our function looks smoother and more visually appealing. By $k = 25$, our average is starting to include data that are far enough away that we begin to lose the apparent local features of the function (the dip around $X = 0.6$, for example).

More generally, we can see the over- and under-fitting effects as a function of k. When $k = 1$, the training error will be zero, since the nearest training data point to $x^{(i)}$ is always itself, which correctly

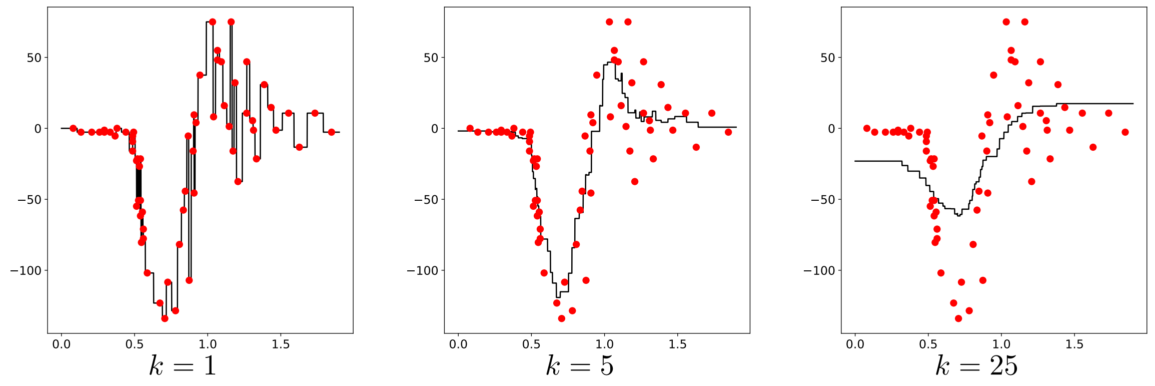


Figure 2.3: A k-nearest neighbor regressor. Again, at $k = 1$, we predict the value of the nearest data point; noise in the target Y makes this function look jagged. As k increases and we average over more nearby points, the predicted relationship begins to look "smoother" (though still technically piecewise constant!). If we increase k even more, we begin to average over large regions and may start to miss details in the relationship between X and Y.

outputs $y^{(i)}$. As k increases, we begin to get some of these data wrong (since, for example, their class does not match that of the other data in their neighborhood), and our training error rate increases. At the other extreme, when $k = m$, the number of data, our prediction function is simply a constant – we output the majority class value in our training dataset. Our training error is then quite large – in a balanced, binary classifier it will be nearly 50% – but it is likely to be quite similar to our test error rate, assuming we have enough training data to accurately estimate the relative fraction of each class.

This matches the basic pattern of overfitting – when k is large, we have high training and test error; as k decreases, our model fits better; and finally, as k becomes too small, we continue to fit the training data better but our test performance degrades. Selecting k is thus a model complexity selection problem, which means it cannot be done using training data alone (since $k = 1$ always gives the best training performance); we must use a validation set or a cross-validation procedure.

2.3 Weighted neighborhood methods

Another approach related to k-nearest neighbors is to again use several nearby data points in the prediction, but not treat these data points equally – instead, we pay more attention to data that are close, compared to data further away. Suppose we are predicting at a point x. We associate a weight $w^{(i)}(x)$ with each data point i, which measures how similar $x^{(i)}$ is to the test point x; this is usually expressed using a *kernel function* $K(x, x')$ which measures similarity, and then normalizing the values to sum to one:

$$w^{(i)}(x) \propto K(x, x^{(i)}) \quad \text{where} \quad \sum_i w^{(i)}(x) = 1.$$

Then, we use a weighted average (for regression) or weighted vote (for classification) to determine our predictor.

For example, the Gaussian or *radial basis function* (RBF) kernel is specified as,

$$K_{\text{rbf}}(x, x') = \exp\left(-\frac{1}{2C}\|x - x'\|^2\right) \propto \mathcal{N}(x\,;\,x', C)$$

A kernel typically augments some distance $d(x, x')$, for example the Euclidean length $\|x - x'\|$, with a functional expression of how relevant data observed at a given distance should be to our prediction. For

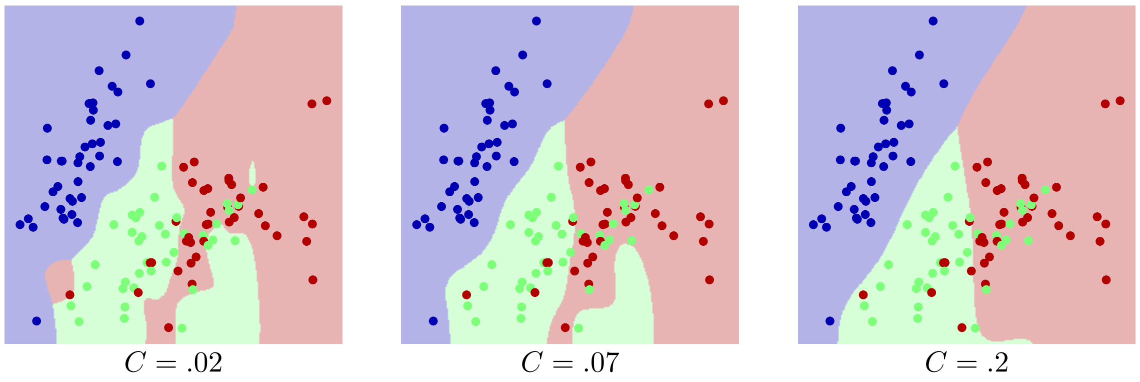


Figure 2.4: Weighted neighborhood classifier. At each test point x, we weight the data and use a weighted vote to determine our prediction. The continuous weight function makes the decision boundary more smooth, and similarly to k, the width parameter C can be used to control how many data have significant weights, and thus the degree of overfitting.

example, in the RBF kernel, the scale (or *width*) parameter C determines how far data can be before their weight becomes close to zero and are effectively ignored (compared to nearby data). If C is sufficiently large, all the data points will have approximately equal weight, while if C is very small, only the nearest point will have high weight. Since the weight typically decreases to zero as $\|x - x'\|$ becomes large, this also means that the value of k (number of neighbors) becomes less critical, and can be viewed as purely a computational convenience – even if $k = m$ (all data are included), the kernel width C will still determine which data are near enough to participate in the prediction, and thus control under- and over-fitting in a manner similar to k.

Because the weights vary smoothly with x, the resulting regression function will now also change smoothly, rather than being locally constant with abrupt changes. Similarly, when weights are included for classification we typically obtain smooth (rather than piecewise linear) decision boundaries. The decision boundaries for the Iris data set for various widths C is shown in Figure 2.4.

Weighted neighbor classifiers and kernel density estimation. In classification, weighted neighborhood predictors can also be interpreted as a Bayes classifier using a particular model, called a kernel density estimator (KDE), for the class distributions. A kernel density estimate approximates the probability density function of a given data set using a "smoothed" version of the empirical distribution,

$$\hat{p}_{\text{kde}}(X) = \frac{1}{m} \sum_i K(X, x^{(i)}),$$

where K is a kernel function, such as the Gaussian kernel, $K(X, x) = \mathcal{N}(X\,;\, x, \sigma^2)$. Intuitively, the KDE places probability mass $\frac{1}{m}$ around each observed data point $x^{(i)}$, and then smooths or spreads that mass to nearby, unobserved values according to the function K.

It is easy to see that the weighted neighborhood rule is equivalent to a Bayes classifier decision for a particular choice of estimated probability distributions. Recall that for a Bayes classifier, we estimate the class probability $p(Y)$ and the distribution of features for each class, $p(X|Y)$. Suppose that for $p(Y = y)$ we use the empirical frequency of class y in the data set, i.e., $\hat{p}(Y = y) = \frac{m_y}{m}$, and use a kernel density estimate on the data with $y^{(i)} = y$ for $\hat{p}(X|Y = y)$. Then the Bayes classifier is,

$$\begin{aligned}\hat{y}(x) &= \arg\max_y \; \hat{p}(Y = y) \cdot \hat{p}(X = x|Y = y) \\ &= \arg\max_y \; \frac{m_y}{m} \cdot \frac{1}{m_y}\Big[\sum_{i:y^{(i)}=y} K(x, x^{(i)})\Big],\end{aligned}$$

which simply selects the class y whose sum of weights $K(x, x^{(i)})$ is largest.

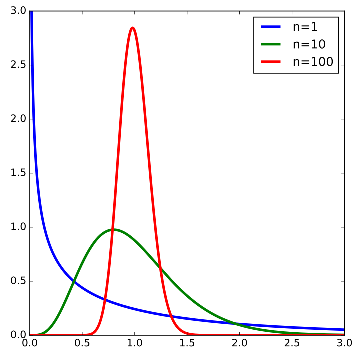
Weighted neighbor regression. The weighted average predictor is equivalent to a local regression model called the Nadaraya-Watson kernel method [Nadaraya, 1964]. **expand? Postpone until after linear regression? (Local regressors, etc.)**

2.4 The "Curse of Dimensionality"

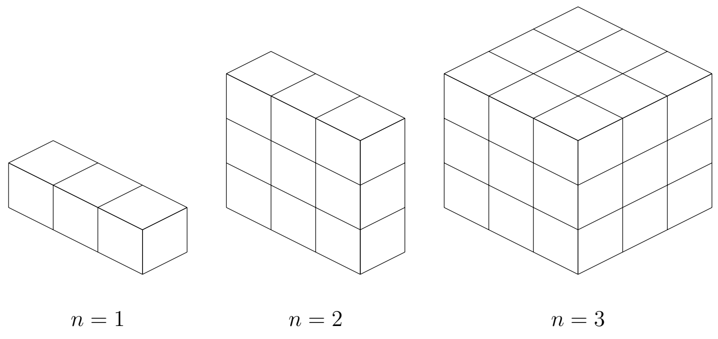
Unfortunately, nearest-neighbor based methods tend to perform poorly in high dimension. One reason for this is termed the *curse of dimensionality*: that as the dimension n increases, a sample of fixed size m tend to become equidistant from one another.

To see why this is the case using a concrete example, consider data from a fixed distribution $P(X) = \mathcal{N}(X\,;\,0, 0.5I)$, i.e., each feature has an independent Gaussian distribution with zero mean and variance $\frac{1}{2}$. Now, let us evaluate the squared distance between two points x and x' drawn from $P(X)$ in n dimensions. It is straightforward to see that this squared distance is a random variable with a chi-squared distribution, $\chi^2(n)$. For $n = 1$ feature, there is a significant probability that the squared distance is near zero, meaning that x and x' are close to each other. However, as n increases, by the central limit theorem, the scaled squared distance, $\frac{1}{n}\sum_j (x_j - x'_j)^2$, becomes asymptotically Gaussian with mean one and variance $\frac{2}{n}$. The distribution of the scaled distance for several n (dimensions $1, 10, 100$) is shown in Fig. 2.5(a). We can see that, as n increases, the variance becomes small and almost every pair x, x' will have (scaled) distance close to one. This shows the limiation of relying on Euclidean distance to indicate which data are similar in very high dimension.

A related perspective is to imagine discretizing the features into bins of some fixed size ϵ, representing how close two data points would need to be to ensure that they are sufficiently "similar" for prediction. Then, the number of such bins grows exponentially with the dimension n (see Fig. 2.5(b)). This suggests that for accurate prediction we will likely need exponentially many training data to ensure that we have a nearby training example.



(a)



(b)

Figure 2.5: Curse of dimensionality. (a) The distribution of scaled distances, $P(\|x - x'\|^2/n)$, in n dimensions. As n increases, nearly all pairs of points x, x' are approximately equidistant. (b) As dimension increases, the number of fixed-size bins needed to partition the space increases exponentially in n, so that the number of data we need to ensure that one is "close" also increases exponentially in n.

2.5 Computational considerations

Nearest neighbor methods also tend to have a high computational cost. For the naïve algorithm, which simply runs through all the training points to find the closest, the cost of each prediction is $O(m \cdot n)$, linear in number of data and number of features. So, to predict at m' test points would cost $O(m \cdot m' \cdot n)$; if $m' \approx m$ this is quadratic in the number of data.

If m is large, this can be overly costly. However, it is possible to use spatial data structures to speed up the performance of nearest neighbor methods. Intuitively, consider how we might speed up a nearest neighbor method for data with only one feature ($n = 1$). Rather than the naïve algorithm, we could preprocess the data by sorting it by its feature value, for cost $O(m \log m)$; then, finding the nearest training example is equivalent to finding the insertion points of the test data, which can be done in time $O(\log m)$ for each test point.

One multi-dimensional generalization of this idea is to use a k-d tree (short for k-dimensional tree), which organizes the data to try to ensure that nearby data (in the feature space) are grouped and stored together. Each node of the k-d tree stores statistics of the data grouped under that node, such as a bounding box that encloses all the points in that set; then, as we search for the nearest data examples to a given test point, we can test to see if any points within the box could be close enough to be relevant. If no point in the bounding box could be close enough to be a neighbor, all the data under that node can be pruned from the search.

There are also many techniques that provide a speed-up by approximating the process of finding the nearest point (i.e., points that are nearby but perhaps not the nearest), such as locality sensitive hashing [Andoni and Indyk, 2008] or other random projection methods [Dasgupta and Sinha, 2015]. Other methods, such as coreset selection methods, work by discarding some of the data, reducing m while identifying which data should be kept in order to provide a good approximation of the predictor.

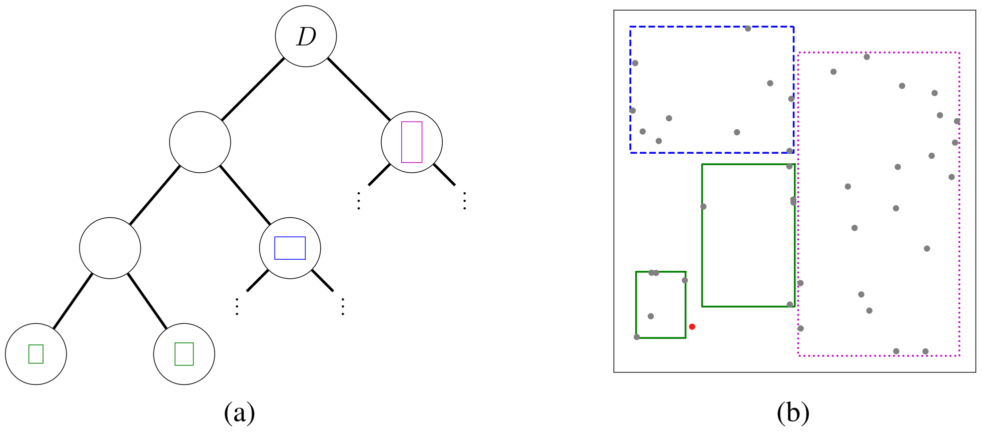


Figure 2.6: A k-d tree organizes the data into a hierarchy and saves useful statistics of the data stored below each node of the tree. Here we show the nodes and bounding boxes relevant for finding the nearest neighbor to a particular target (red point). The tree allows us to quickly find a small number of promising candidate points (e.g., those in the green boxes), and then to rule out data that are provably more distant (e.g., those in the blue and magenta boxes).