FIT2086 Lecture 9 Summary
Trees and Nearest Neighbour Methods

Dr. Daniel F. Schmidt

November 1, 2019

1 Part I: Machine Learning

Cross-Validation Revisited. In Lecture 8 we examined cross-validation (CV) as a technique for selecting which predictors should be included in a regression model. A particular strength of CV is that it is extremely general and can be applied without modification to many different problems. For this reason it is frequently used in machine learning to control the complexity of a model. Let us define γ to be some "complexity" parameter that controls how complex a model is. For linear/logistic regression models it could be the number of predictors, or the λ for ridge/lasso regression. We will also examined other "complexity" parameters in this Lecture.

Let $\hat{\mathcal{M}}(\gamma)$ be a model with complexity γ fitted to some data sample. The obvious question is: which choice of model complexity γ is appropriate for this sample? As alluded to above, A very general approach to answering this question to use cross-valididation to choose a γ. The intuition is we would like to find a complexity that leads to good prediction error (mean squared prediction error, MSPE). To estimate the MSPE for a given γ using cross validation we can use the algorithm presented in Lecture Summary 8. In practice most packages use a specialisation called K-fold cross validation. This has the following algorithm:

- Outer loop: try different model complexities γ
    1. For $i = 1$ to m
        (a) Partition data into K equal sized, disjoint subsets
        $$\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \mathbf{y}^{(3)}, \ldots, \mathbf{y}^{(K)}$$
        (b) For $k = 1$ to K
            i. Fit model $\mathcal{M}(\gamma)$ to all $\mathbf{y}^{(j)}$ except for $j = k$
            ii. Use fitted model to predict onto $\mathbf{y}^{(k)}$
            iii. Calculate and accumulate prediction errors
    2. Average all m accumulated prediction errors

Once this procedure is finished we have CV estimates of the MSPE for each model complexity γ we considered, and we choose the γ that resulted in the smallest estimated error. In general the CV errors will vary from run to run due to the random partitioning step in the K-fold algorithm, but by

taking the number of repetitions m to be larger we can obtain more stable estimates. The downside is the bigger m is, the slower the process becomes because we must fit more and more models. A choice of $K = 10$ has been empirically found to perform well in general for many problems.

Machine Learning. The models we will examine today are often considered part of the area of machine learning. This field is essentially an intersection between statistics and computer science, and is sometimes also called data mining or artificial intelligence. While clearly dilineating machine learning and statistics is difficult, one can say in general that machine learning methods are often algorithmic, usually non-linear and flexible and make relatively few assumptions about the data. Frequently it is impossible to interpret the resulting "model" as they are highly complex. Perhaps the most clear difference is that machine learning is primarily focussed on *prediction*, i.e., how well can we predict, while statistics has other aims, including learning about the nature of the data we are analysing. Some well known machine learning methods include:

- decision trees;
- random forests;
- k-nearest neighbours (kNN);
- support vector machines (SVMs);
- neural networks;
- deep neural networks (deep learning);
- clustering;
- mixture modelling;

though it is important to emphasise that virtually all of these models also appear in regular statistics literature, and indeed many were pioneered by statisticians.

2 Part II: Decision Trees

Decision Trees. Decision trees are one of the most widely used non-linear machine learning techniques. A decision tree is a specific type of supervised learning model, i.e., it takes inputs (predictors) $x_1, \ldots, x_p$ and produces a prediction

$$\hat{y}(x_1, \ldots, x_p) = f(x_1, \ldots, x_p)$$

about an individual. So in this basic sense it is similar to linear/logistic regression. The key difference is in the form of $f(\cdot)$: for a linear model, the function $f(\cdot)$ was restricted to be a linear relationship between the predictors and the target. In the case of decision trees, $f(\cdot)$ is a highly non-linear function. The strength of decision trees however is that despite being highly non-linear they still manage to retain a high degree of interpretability. That is, we can understand why the model is making the predictions it is making – and learn what the model says about the nature of the population that generated the data. Decision trees are also very general in that they can be applied equally easily to both regression and (multi-class) classification problems.

A decision tree works by taking the predictor space, that is the p-dimensional space of all the possible predictor values, and subdividing it up into L disjoint regions, say $R_1, \ldots, R_L$. By disjoint we mean that none of the regions intersect with each other, i.e., $R_i \cap R_j = \emptyset$, $i \neq j$. How does it achieve this subdivision? The way a tree does this is by sequentially asking questions of our predictors,

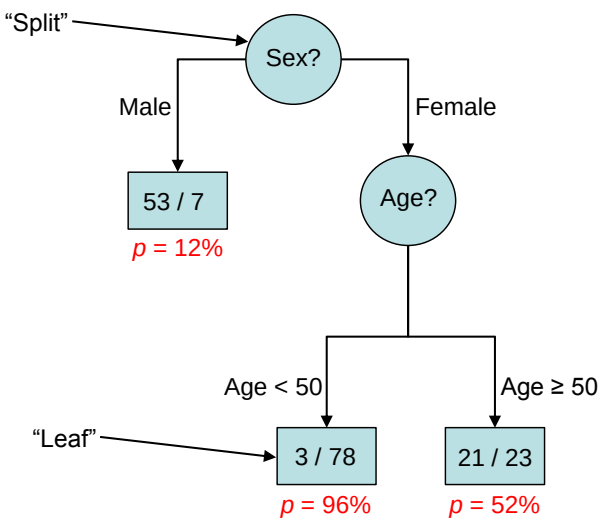


Figure 1: Example of a decision tree for predicting high blood pressure from an individual's sex and age.

and depending on the answer, moving down one path or another of a tree. In this way individuals who have the same answers to these questions are grouped together into each of the regions R_i. The decision tree then builds a seperate, simple model for each of the individuals in each of the leaves. For a regression tree, where the targets are continuous, the model would be something like a normal distribution with a different mean μ_i for each leaf R_i. For a binary classification problem, each leaf would contain a Bernoulli distribution with a different probability of success θ_i for each leaf.

Figure 2 shows an example of a decision tree for predicting high blood pressure. The rectangles/circles are the nodes of the tree. The node at the top is called the root, the circle nodes are called "splits" and the rectangular nodes are the "leaves". Each circle node asks a question about one of the predictors; depending on the answer, we either move an individual to the left or the right. The leaves show for our dataset the number of individuals that are in the leaf, and of them, the X/Y denotes that X individuals do not have high blood pressure and Y do have high blood pressure. Attached to each leaf is a probability p which is the estimated probability of having high blood pressure for individuals allocated to that leaf. So for example, we see the first question asked at the root node is what sex is the individual. If they are male, they go left; as we see that there are 53 individuals without high blood pressure in the left leaf, and 7 with high blood pressure, we can determine there were 60 males in total in this dataset. If the individual is not male, they go to the right, at which point a new question is asked regarding their age. Depending on their age, the individual is then either shuffled to the left or the right. We see the disjoint nature of the leaves – no person who answers "male" to the first question can possible end up on the right of the tree. Every individual is allocated to exactly one leaf depending on the value of their predictors and the questions being asked.

To make a prediction with a decision tree is straight forward: given the values of predictors $x_1, \ldots, x_p$ for a new individual, we simply traverse the tree, taking the appropriate path at each split until we arrive at a leaf node. When we arrive at a leaf, we use the model in that leaf to make a prediction about this individual. The number of leaves L is therefore the primary determinant of complexity of the tree. The more leaves, the more finely we divide up our data sample and the more different models we have for groups of people in our population. If we have only the root node we

reduce the tree to the basic models we have examined throughout this subject.

Learning Decision Trees: Forward Search. An obvious question is how should we learn a decision tree from our data sample $\mathbf{y}$? The problem is that given some data, if we take the number of leaves L big enough we can always fit the data *perfectly* using a decision tree. But by now this should trigger an alarm bell – if we fit perfectly, we are also very likely to be overfitting. Instead, we generally counter this problem by using a model selection approach: we can assign a score to each tree using something like an information criteria (AIC, BIC, etc.) or use cross-validation scores. We can then find the tree with the smallest score, with the aim of trading-off goodness-of-fit of the tree against the overall complexity of the tree.

A problem with learning trees in general however is that the space of possible trees is enormous if the number of different possible predictor variables p is even moderate. For this reason it is usual to use approximate search algorithms. The most basic algorithm, which is frequently the building block for more advanced methods is called the forward search with information criterion. This works in the following way:

1. Start with just one leaf node (the root node)
2. Try splitting on every predictor and compute criterion scores
3. If none of the splits improves the tree, stop
4. Otherwise, choose the split that results in tree with smallest score
5. Go back to Step 2

The algorithm works by iteratively growing the tree forwards by trying to find the best variable to split on, splitting on that variable and then repeating the search. The algorithm is greedy and is not guaranteed to find the optimal solution, but it is reasonably fast and often gives adequate fits. The key question that remains is: how to determine if one split is better than another?

Splitting. At every step of our forward search there are usually many predictors we could use to split our data. We try splitting on all of them, and end up choosing one as the "best split". But how do we say that one split is better than another? One way is to measure the goodness of a split through the negative log-likelihood. For simplicity of exposition, let us just consider a tree with binary targets, i.e., the target data is binary. Let $\mathbf{y} = (y_1, \ldots, y_n)$ be the data in some leaf, and let

$$n_1 = \sum_{i=1}^{n} y_i$$

be the total number of "1"s in assigned to that leaf, with $n_0 = n - n_1$ be the total number of "0"s. For binary data, we use the Bernoulli distribution to model the individuals who have been allocated into a leaf. From Lecture 2 we now that the likelihood for a Bernoulli distribution is

$$p(\mathbf{y} \,|\, \theta) = \theta^{n_1}(1-\theta)^{n_0} \tag{1}$$

and that the maximum likelihood estimator of the success probability θ is $\hat{\theta} = n_1/n$, i.e., the observed proportion of successes in the leaf. The maximised likelihood is then found by plugging $\hat{\theta}$ into (1), which yields:

$$p(\mathbf{y} \,|\, \hat{\theta}) = \left(\frac{n_1}{n}\right)^{n_1} \left(\frac{n_0}{n}\right)^{n_0}$$

y	1	1	0	1	0	0	0	1	1	0
x_1	0	0	0	0	0	1	1	1	1	1
x_2	1	1	0	1	1	0	0	0	1	0

Table 1: Example data on $n = 10$ individuals. There are two predictors we could split on: x_1 and x_2.

The minimised negative log-likelihood of the data is then found by taking negative logarithms of this which yields

$$-\log p(\mathbf{y} \mid \hat{\theta}) = -n_1 \log\left(\frac{n_1}{n}\right) - n_0 \log\left(\frac{n_0}{n}\right)$$

This quantity tells us how well our Bernoulli model with success parameter $\hat{\theta}$ fits the data in the leaf. This quantity is: (i) largest when $n_1/n = 1/2$ (least "pure"), and (ii) smallest when $n_1 = 0$ or $n_1 = n$ (most "pure"). By "pure" we mean how imbalanced the distribution of 0s and 1s is in the leaf. The "purer" the data, i.e., the more imbalanced, the smaller the negative-log likelihood is because the model fits the data better. If we had a leaf with only 1s and no zeros, then the negative log-likelihood is just zero, because the model explains the data perfectly. On the flipside, if a leaf contains a 50-50 mix of 1s and 0s then we can't say much about whether an individual is likely to be a 1 or a 0. There is a lot of uncertainty. The aim when growing a tree is to try and make splits that result in purer leaves, because if a variable partitions our individuals neatly into two groups with one containing mostly 1s and one containing mostly 0s, then we have obviously learned something about the individuals that lets us predict their target well.

Example: Splitting. We demonstrate the use of negative log-likelihood to characterise a split on a simple toy example. Consider the data shown in Table 1. We have $n = 10$ individuals in this leaf, with two binary predictors x_1 and x_2. We could choose to split this leaf into two new leaves using either x_1 or x_2; but which variable would lead to be a better split? As a reference we first compute the negative log-likelihood of the data without splitting. In this case we have $n_1 = 5$ (i.e., five of the $n = 10$ individuals in the leaf have $y = 1$ and five have $y = 0$), so that $\hat{\theta} = 5/10$ and the minimised negative log-likelihood is

$$-\log p(\mathbf{y} \mid \hat{\theta}) = -5\log(5/10) - 5\log(5/10) \approx 6.9315.$$

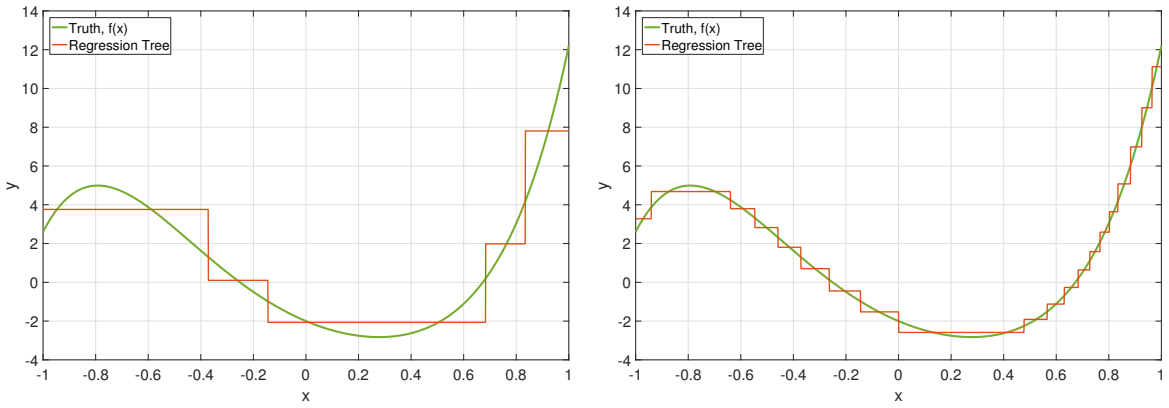
So without splitting the minimised negative log-likelihood is 6.9315 units. Now let us first try splitting on predictor x_1. In this case, we subdivide $\mathbf{y}$ into two groups, one for each new leaf, depending on the associated value of x_1. We have:

- Take y when $x_1 = 0 \Rightarrow (1, 1, 0, 1, 0)$, $\hat{\theta}_{x_1=0} = 3/5$
- Take y when $x_1 = 1 \Rightarrow (0, 0, 1, 1, 0)$, $\hat{\theta}_{x_1=1} = 2/5$

The minimised negative-log-likelihood of the two leaves is then:

$$-\log p(\mathbf{y} \mid \hat{\theta}, x_1 = 0) = -3\log(3/5) - 2\log(2/5) \approx 3.3651$$
$$-\log p(\mathbf{y} \mid \hat{\theta}, x_1 = 1) = -2\log(2/5) - 3\log(3/5) \approx 3.3651$$

and the total negative-log likelihood found by adding together the minimised negative log-likelihoods for each leaf is approximately 6.7302. This is a bit better than "no split" neg-log-like of 6.9315, but not greatly so. The reason is that splitting on x_1 leads to leaves that are not much purer than the original, unsplit data. In original leaf there was an estimated probability of 0.5 of being a 1; in the



(a) $L = 5$ leaves (b) $L = 20$ leaves

Figure 2: Two different decision trees approximating the non-linear relationship (shown in green) between a continus predictor x and a continuous target y, each with a different number of leaves (complexity).

two new leaves the estimated probabilites are 0.6 in the leaf when $x_1 = 0$ and 0.4 in the leaf when $x_1 = 1$, which are not much different from 0.5. The predictor x_1 does not do a particular good job of discriminating $y = 0$ individuals from $y = 1$ individuals. Let us now consider splitting on x_2:

- Take y when $x_2 = 0 \Rightarrow (0, 0, 0, 1, 0)$, $\hat{\theta}_{x_2=0} = 1/5$
- Take y when $x_2 = 1 \Rightarrow (1, 1, 1, 0, 1)$, $\hat{\theta}_{x_2=1} = 4/5$

The minimised negative-log-likelihood of the two leaves is then:

$$\begin{aligned} -\log p(\mathbf{y} \,|\, \hat{\theta}, x_2 = 0) &= -1\log(1/5) - 4\log(4/5) \approx 2.5020 \\ -\log p(\mathbf{y} \,|\, \hat{\theta}, x_2 = 1) &= -4\log(4/5) - 1\log(1/5) \approx 2.5020 \end{aligned}$$

so that the total negative-log likelihood is approximately 5.04. This a lot better than "no split" negative log-likelihood of 6.9315 because the allocation of individuals to leaves based on their x_2 results in a much more imbalanced distribution of 0s and 1s in each leaf. So, we can see that splitting on x_2 leads to purer leaf nodes than splitting on leaf 1, and therefore we would prefer to split on x_2.

Splitting on Numeric Variables. In general some of our predictors will be numeric. In this case most programs create a binary split by choosing a value c and then subdividing individuals based on whether the variable $x_j \leq c$ or $x_j > c$. The value of c that is chosen is usually the one that minimises the negative log-likelihood we obtain when we partition the individuals between the two leaves. To understand how this splitting approach gives decision trees the ability to model arbitrary non-linear relationships, consider the problem of modelling some numerical target y by a numerical predictor x.

Figure 2 shows the resulting tree formed with two different number of leaves; Figure 2(a) shows the tree approximation formed using $L = 5$ leaves, and Figure 2(b) shows the tree approximation formed using $L = 20$ leaves. It is clear that the tree divides the x predictor variable into a set of regions, and assigns an "average" value to each region as its prediction of y. When $L = 5$ we see the tree captures the gross, overall behaviour of the nonlinear population curve (plotted in green), but is quite coarse. When $L = 20$ the approximation is substantially better, though of course, it still exhibits the

"stepped" appearance because it assigns a different average value to each of the regions. In this way trees can approximate nonlinear relationships arbitrarily well for sufficient numbers of leaves, though the approximation will always be discontinous in nature due to the way trees divide up the predictor space into disjoint regions.

Learning Decision Trees with Cross Validation. How do we use cross-validation to learn a decision tree? To answer this question, we first introduce a variant of the forward algorithm for growing trees based on tree "pruning". In this approach, we first grow a large, overfitted tree with more leaves than would be optimal. To do this we use a forward search and split on predictors that decreases negative log-likelihood by the most at each step. Then, given this big overfitted tree, we "prune" some of it back until we get a good number of leaves. How much we prune is frequently determined by cross-validation. This is only an approximate search method, but it often performs better than just using a forward search.

The full algorithm for learning a tree with CV is:

- Input: $L_{\max}$, maximum number of leaves to consider
  1. For $i = 1$ to m
     (a) Partition data into K equal sized, disjoint subsets
     $$\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \mathbf{y}^{(3)}, \ldots, \mathbf{y}^{(K)}$$
     (b) For $k = 1$ to K
        i. Grow a tree with $\gg L_{\max}$ leaves using all $\mathbf{y}^{(j)}$ except for $j = k$
        ii. Prune this tree back to $L = 1, \ldots, L_{\max}$ leaves
        iii. For each of the $L_{\max}$ fitted models, predict onto the testing fold $\mathbf{y}^{(k)}$
        iv. Calculate and accumulate prediction errors
  2. Average all m accumulated prediction errors
- Choose the size L that had the smallest accumulated CV error
- Grow a tree with L leaves on all data as final model

The basic idea is to grow a big, complex tree for each CV iteration, and then see how much improvement in prediction error we get as we prune the tree back. This gives us an idea of the best size of tree for our data, and we can then use that many leaves to learn our final tree from the entire data. So the first part of the algorithm figures out a value for L, and the final step grows our final tree on all the data using the "best" value of L.

Summary. Decision trees are popular because the basic approach has many strengths:

- They are highly interpretable; it is easy to explain why the decision tree makes the predictions it does, even to a non-expert
- They handle non-linear relationships
- They naturally handle interactions between variables; for example, if we split on one variable, then another, these two variables are naturally interacting in our model
- They seamlessly handle continuous, categorical and count predictor variables
- They easily generalise to multiclass (categorical) target variables

- The problem of variable selection is naturally incorporated into the learning of the model; if we do not use a variable at split in the tree, it has automatically been excluded from the model

Despite these many advantages, the decision tree approach also suffers from a number of weaknesses:

- It is sometimes difficult to find a good tree due to the computationally formidable problem of searching for good trees
- They are very sensitive to small perturbations in the data. A slight change can result in a drastically different tree (they are statistically unstable)
- They can be potentially inefficient models for explaining simple relationships. For example, if the true model is well explained by a linear model, then tree needs many leaves, and many parameters to approximate the linear structure well

Researchers have attacked these weaknesses over the years, and the first two problems are to a large degree dealt with by moving beyond a single tree to many (a "forest") of trees.

3 Part III: Random Forests

Ensembles of Trees. A key problem with trying to learn a single decision tree is that the tree we end up with is very sensitive to the algorithm we use to grow the tree, and also very sensitive to small changes in data. We will find that many trees will fit the data almost as well as each other – that is many trees will have information scores that are very close. Furthermore, the greedy forward growing approach is very statistically unstable. If we change the first variable that we choose to split on when growing our tree, then tree we end up with will be very different.

Growing Random Forests. To overcome this problem the random forest idea is to build a collection (a "forest") of many trees, rather than try to find a single best tree. This is a specific example of a more general approach called ensemble learning, in which we have a collection of models rather than a single model. The obvious question is: how do we build the forest? There are a number of ways of approaching this problem, but the random forest packages use the following strategy. They loop q times, and each time they grow a new tree in a controlled but random forward selection approach. When performing the forward growing algorithm, the set of possible variables that can be split on is randomly selected from the full set of all variables, and the best split (based on likelihood) is made from within this smaller subset. This helps overcome the "greedy" nature of forward selection by forcing the algorithm to sometimes split on variables that would never be considered if we were considering all variables as candidates.

Making Predictions. Once we have grown a forest of q trees, we need to somehow use them to make a single prediction for a new individual. The usual approach is to go through each of the q trees in the forest and make a prediction for each of these trees. We then combine these together predictions together and use the average prediction from the forest. For regression trees this will just be the average value predicted by all the trees, and for classification problems it will be the average probability of success.

How does this help? The key idea is that each individual tree in the forest has low bias, because a tree can learn any non-linear pattern given a sufficient number of leaves. The problem is that growing the trees is very statistically unstable, as discussed above, and so the variance of any one tree is high. By averaging the trees together we retain the low bias but the aggregation helps to reduce the variance

through averaging.

Strengths and Weaknesses. The biggest drawback with random forests is that they lack the interpretability of an individual tree. While we can look at and understand why a single tree is making the predictions it is making, it is very difficult to do so with a forest of $q = 10$ trees, say, and impossible if $q = 100$ or more, which is standard. In general the only real interpretible measure we get from a random forest is a basic measure of which variables are important: most packages return a measure of how much the predictive performance of the forest would be expected to degrade if each variable was omitted, and we can use this to rank how important a variable is.

In summary, random forests have a number of strengths:

- They are very stable, in the sense that they are resistant to perturbations in data (i.e., small changes to data result in only small changes to the predictions of the forest), and they are resistant to "local minima" in search space
- In general they yield improved predictive accuracy over any single tree
- Easily handle multiclass or count targets
- Flexible and handle non-linearities just like regular trees, as they are simply a collection of individual trees
- Inherently incorporate variable/predictor selection as per regular decision trees

However, random forests also suffer from several weaknesses:

- Poor interpretability – it is essentially impossible to understand why a random forest has produced a certain prediction. We can obtain an idea of how important a variable is, but that is about it.
- They are complex to learn and there are a lot of variations and algorithm parameters to tweak. Furthermore, there is not much theory available to guide us in setting these parameters due to the very algorithmic nature of random forests.

4 Part IV: Nearest Neighbour Methods

Motivation. The final machine learning method we will examine is the k-nearest neighbours methods, sometimes also called "nearest neighbours smoothing", or just k-NNs. The interesting difference between k-NN approaches and everything else that we have learned so far is that they do not explicitly produce a statistical model of our population. Instead, the k-NN setup is as follows: we have a training set of n example predictor/target pairs, with the predictor values $x_{i,1}, \ldots, x_{i,p}$ paired with each target y_i, and we want to predict target value for *new individual* with predictor values $x'_1, \ldots, x'_p$.

Rather than approach the problem in the usual way by first building a model for the population from our training data, and then using that model to make a prediction about the new individual, the k-NN approach works by first finding the k individuals in our training data that are "most similar" to the new individual in terms of their predictor values, and then using the target values of these k individuals to make a prediction about the target value for our new individual.

The strength of this approach is that we only make very weak assumptions. The primary assumption we make is that individuals that are similar to each in other in terms of predictor values will also be similar in terms of targets. This seems reasonable and is in a way an assumption we make for every method we have seen so far, including trees, which for example, group people together into leaves based on whether their predictors satisfy the same set of conditions. The difference with a k-NN

is that is the only assumption we make – there is no explicit model or distribution we choose to try and capture the underlying relationships.

Similarity Measures. While there is no explicit model, we do have to choose how we measure the "similarity" between individuals. This is a crucial ingredient in k-NN methods and to a large extent determines the performance of the algorithm. The most common, and almost default method, is the Euclidean distance between the two in predictor space:

$$d(\mathbf{x}, \mathbf{x}') = \left[\sum_{j=1}^{p} (x_j - x'_j)^2 \right]^{\frac{1}{2}}$$

This treats the two individuals as p-dimensional points in space and calculates how far apart they are. While this is simple to compute, and in some sense seems reasonable, it's appropriateness really does depend on the particular problem. For example, if some of the predictors are categorical then Euclidean distance is meaningless and we need to use an alternative, and more appropriate, measure of distance. However, different distance measures produce different k-nearest neighbour predictions, so in theory this is a delicate problem that needs to be carefully addressed. In practice however, most packages (and users of those packages) simply resort to Euclidean distance for the sake of convenience.

The k-Nearest Neighbours Algorithm. Imagine we have a set of training data consisting of n individuals with targets y_i and predictors $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,p})$. We are asked to predict the target value for a new individual with predictor values $\mathbf{x}' = (x'_1, \ldots, x'_p)$. The k-NN algorithm works as follows:

1. Calculate the distance from each individual in the training data to our new individual:
$$d_i = d(\mathbf{x}_i, \mathbf{x}'), \quad i = 1, \ldots, n$$
2. Sort the individuals from smallest d_i to largest (i.e., nearest to furthest)
3. Denote the targets in the sorted list by $y^{(1)}, \ldots, y^{(n)}$
   - $y^{(1)}$ is the observed target for nearest individual
   - $y^{(n)}$ is the observed target for furthest individual
4. Use the k individuals with smallest d_i to predict y'
$$\hat{y}' = f(y^{(1)}, \ldots, y^{(k)})$$

The function $f(\cdot)$ is an aggregator function; it takes the target values of the k nearest individuals and combines them to make the prediction for y'. The question is how do we choose the prediction function $f(\cdot)$? For classification problems, we can use voting; that is, we can output the class that is most frequent amongst all the k nearest neighbours. So if $k = 10$, and seven of our nearest neighbours were class 1, we would use $\hat{y}' = 1$. For regression problems, we can use averaging, i.e.,

$$\hat{y}' = \frac{1}{k} \sum_{i=1}^{k} y^{(i)}$$

Such a simple average function treats all of k nearest individuals as equally important. But maybe individuals who are closer should be treated as more relevant than those who are far away? This is frequently done in k-NN packages by using a weighted average

$$\hat{y}' = \frac{1}{k} \sum_{i=1}^{k} g(y^{(i)})$$

where $g(d)$ is a weighting function that gets smaller as d gets larger, i.e., the further away the individual is from the point we want to predict, the less weight their target gets in the average. The functions $g(\cdot)$ are often called "kernel functions" in the various k-NN packages, and there exist a large number of options we could use.

Using k-NN Methods in Practice. While k-NN methods are conceptually simple, they do have a large number of "tunable" parameters/options that a user has to set to use them, and the particular choice can have a substantial impact on the performance of the resulting algorithm. For example, a basic k-NN package will require the following tuning parameters to be set:

- Neighbourhood size k, i.e., how many individuals to use to make a prediction?
- Distance functions, i.e., how to measure similarity between individuals?
- Kernel weighting functions, i.e., how much weighting should be put on closer individuals?

It is natural to ask: how can we choose these in an objective fashion? The standard approach is to appeal back to cross-validation. We can use cross-validation to try and estimate the predictive performance of the k-NN algorithm with different choices for the above tunable parameters, and choose those values that lead to the best (estimated) prediction accuracy. For k-NN methods the leave-one-out cross validation approach is probably the most natural and straightforward to implement. Essentially, for each combination of parameter choices (neighbourhood size k, distance function, weighting function) we do the following:

1. Predict each example y_i using k-NN (with y_i removed from the training set)
2. Calculate and accumulate the error in predicting y_i

Once this is done we choose the combination of tuning parameters that lead to the smallest accumulated cross-validation error. This is the standard approach used in most packages.

Summary. Nearest neighbour methods are popular because they are conceptually simple, straight forward to implement in software and make relatively few assumptions. They have a number of strengths:

- They make quite weak assumptions about data (individuals similar in predictors are likely to be similar in targets);
- They are efficient at learning in high dimensions because they do not need to learn explicit model parameters
- They easily handle continuous and categorical variables with appropriate choices of distance measures

However, as ever, they also have a number of weaknesses:

- Lots of parameters to configure
  - How many neighbours (k) to use?
  - How to measure "nearest"?
  - How to average or weight the neighbours
  - Should all predictors be treated the same?

- Predictor selection is not straightforward; most k-NN packages do not have easy, automatic options for doing variable selection and will always include all predictors in the distance measure. If many predictors are not associated these can lead to poor predictions.
- No interpretability; the drawback of having no model to learn is that you end up with no description of the population. There is no way of interpreting k-NN because there is nothing to interpret!