CHAPTER 12

Non-parametric methods

Neural networks have adaptable complexity, in the sense that we can try different structural models and use cross validation to find one that works well on our data. Beyond neural networks, we may further broaden the class of models that we can fit to our data, for example as illustrated by the techniques introduced in this chapter.

Here, we turn to models that automatically adapt their complexity to the training data. The name *non-parametric methods* is misleading: it is really a class of methods that does not have a fixed parameterization in advance. Rather, the complexity of the parameterization can grow as we acquire more data.

Some non-parametric models, such as decision trees, can be seen as dynamically constructing something that ends up looking like a more traditional parametric model, but where the actual training data affects exactly what the form of the model will be. Other non-parametric methods, such as nearest-neighbor, rely directly on the data to make predictions and do not compute a model that summarizes the data.

The non-parametric methods we consider here tend to have the form of a composition of simple models:

- *Tree models*: (Section 12.1) where we partition the input space and use different simple predictions on different regions of the space; the hypothesis space can become arbitrarily large allowing finer and finer partitions of the input space.
- *Ensemble models*: (Section 12.1.3) in which we train several different classifiers on the whole space and average the answers; this decreases the estimation error. In particular, we will look at bootstrap aggregation, or *bagging* of trees.
- *Nearest neighbor models*: (Section 12.2) where we don't process data at training time, but do all the work when making predictions, by looking for the closest training example(s) to a given new data point.
- *Boosting* is a way to construct a model composed of a sequence of component models (e.g., a model consisting of a sequence of decision trees, each subsequent tree seeking to correct errors in the previous trees) that decreases both estimation and structural error. We won't consider this in detail in this class.

Why are we studying these methods, in the heyday of neural networks?

- They are fast to implement and have few or no hyperparameters to tune.
- They often work as well or better than more complicated methods.
- Predictions from some of these models can be easier to explain to a human user: decision-trees are fairly directly human-interpretable, and nearest neighbor methods can justify their decision to some extent by showing a few training examples that the prediction was based on.

12.1 Decision Trees

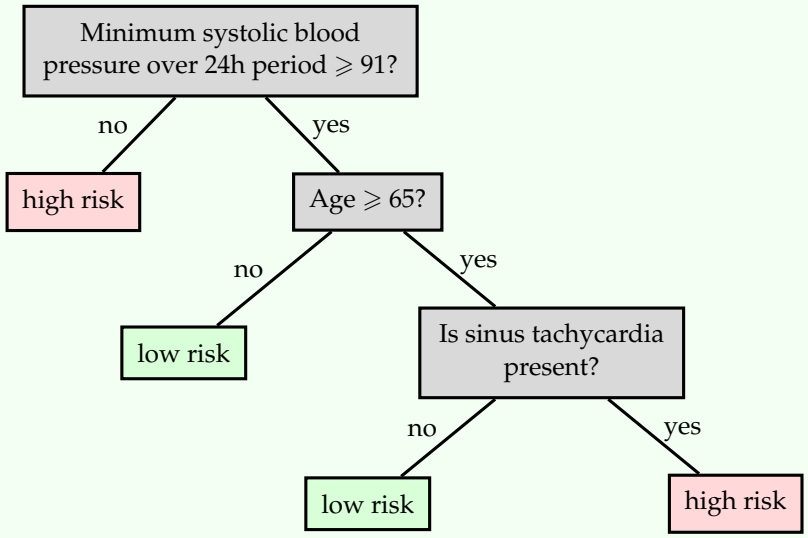
The idea here is that we would like to find a partition of the input space and then fit very simple models to predict the output in each piece. The partition is described using a (typically binary) "decision tree" that recursively splits the space.

Decision tree methods differ by:

- The class of possible ways to split the space at each node; these are typically linear splits, either aligned with the axes of the space, or sometimes using more general classifiers.
- The class of predictors within the partitions; these are often simply constants, but may be more general classification or regression models.
- The way in which we control the complexity of the hypothesis: it would be within the capacity of these methods to have a separate partition element for each individual training example.
- The algorithm for making the partitions and fitting the models.

One advantage of tree models is that they are easily interpretable by humans. This is important in application domains, such as medicine, where there are human experts who often ultimately make critical decisions and who need to feel confident in their understanding of recommendations made by an algorithm. Below is an example decision tree, illustrating how one might be able to understand the decisions made by the tree.

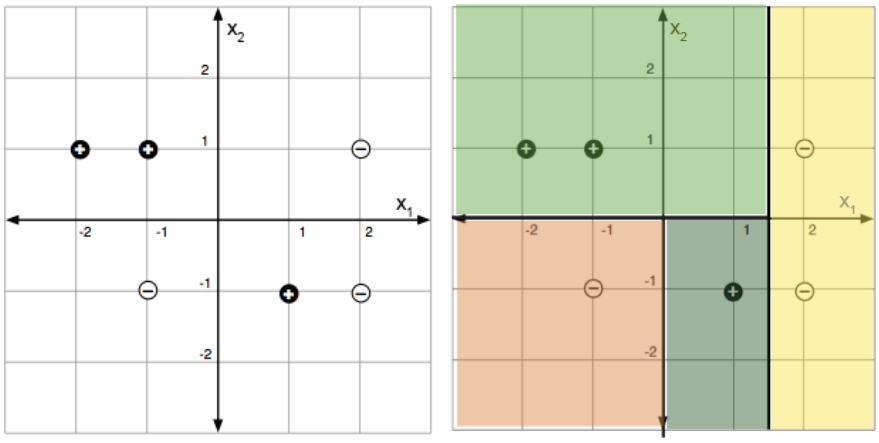
Example: Here is a sample decision tree (reproduced from Breiman, Friedman, Olshen, Stone (1984)):



These methods are most appropriate for domains where the input space is not very high-dimensional and where the individual input features have some substantially useful information individually or in small groups. Decision trees would not be good for image input, but might be good in cases with, for example, a set of meaningful measurements of the condition of a patient in the hospital, as in the example above.

We'll concentrate on the CART/ID3 ("classification and regression trees" and "iterative dichotomizer 3", respectively) family of algorithms, which were invented independently in the statistics and the artificial intelligence communities. They work by greedily constructing a partition, where the splits are *axis aligned* and by fitting a *constant* model in the leaves. The interesting questions are how to select the splits and how to control complexity. The regression and classification versions are very similar.

As a concrete example, consider the following images:



The left image depicts a set of labeled data points in a two-dimensional feature space. The right shows a partition into regions by a decision tree, in this case having no classification errors in the final partitions.

12.1.1 Regression

The predictor is made up of

- a partition function, π, mapping elements of the input space into exactly one of M regions, $R_1, \ldots, R_M$, and
- a collection of M output values, O_m, one for each region.

If we already knew a division of the space into regions, we would set O_m, the constant output for region R_m, to be the average of the training output values in that region. For a training data set $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}, i = 1, \ldots n$, we let I be an indicator set of all of the elements within $\mathcal{D}$, so that $I = \{1, \ldots, n\}$ for our whole data set. We can define I_m as the subset of data set samples that are in region R_m, so that $I_m = \{i \mid x^{(i)} \in R_m\}$. Then

$$O_m = \text{average}_{i \in I_m}\ y^{(i)} \quad .$$

We can define the error in a region as E_m. For example, E_m as the sum of squared error would be expressed as

$$E_m = \sum_{i \in I_m} (y^{(i)} - O_m)^2 \quad . \tag{12.1}$$

Ideally, we would select the partition to minimize

$$\lambda M + \sum_{m=1}^{M} E_m \quad , \tag{12.2}$$

for some regularization constant λ. It is enough to search over all partitions of the training data (not all partitions of the input space!) to optimize this, but the problem is NP-complete.

Study Question: Be sure you understand why it's enough to consider all partitions of the training data, if this is your objective.

12.1.1.1 Building a tree

So, we'll be greedy. We establish a criterion, given a set of data, for finding the best single split of that data, and then apply it recursively to partition the space. For the discussion below, we will select the partition of the data that *minimizes the sum of the sum of squared errors of each partition element.* Then later, we will consider other splitting criteria.

Given a data set $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}, i = 1, \ldots n$, we now consider I to be an indicator of the subset of elements within $\mathcal{D}$ that we wish to build a tree (or subtree) for. That is, I may already indicate a subset of data set $\mathcal{D}$, based on prior splits in constructing our overall tree. We define terms as follows:

- $I^+_{j,s}$ indicates the set of examples (subset of I) whose feature value in dimension j is greater than or equal to split point s;
- $I^-_{j,s}$ indicates the set of examples (subset of I) whose feature value in dimension j is less than s;
- $\hat{y}^+_{j,s}$ is the average y value of the data points indicated by set $I^+_{j,s}$; and
- $\hat{y}^-_{j,s}$ is the average y value of the data points indicated by set $I^-_{j,s}$.

Here is the pseudocode. In what follows, k is the largest leaf size that we will allow in the tree, and is a hyperparameter of the algorithm.

BUILDTREE(I, k)

if $|I| \leqslant k$
  Set $\hat{y} = \text{average}_{i \in I}\, y^{(i)}$
  return LEAF(value $= \hat{y}$)
else
  for each split dimension j and split value s
    Set $I^+_{j,s} = \{i \in I \mid x^{(i)}_j \geqslant s\}$
    Set $I^-_{j,s} = \{i \in I \mid x^{(i)}_j < s\}$
    Set $\hat{y}^+_{j,s} = \text{average}_{i \in I^+_{j,s}}\, y^{(i)}$
    Set $\hat{y}^-_{j,s} = \text{average}_{i \in I^-_{j,s}}\, y^{(i)}$
    Set $E_{j,s} = \sum_{i \in I^+_{j,s}} (y^{(i)} - \hat{y}^+_{j,s})^2 + \sum_{i \in I^-_{j,s}} (y^{(i)} - \hat{y}^-_{j,s})^2$
  Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
return NODE$(j^*, s^*,$ BUILDTREE$(I^-_{j^*,s^*}, k),$ BUILDTREE$(I^+_{j^*,s^*}, k))$

In practice, we typically start by calling BUILDTREE with the first input equal to our whole data set (that is, with $I = \{1, \ldots, n\}$). But then that call of BUILDTREE can recursively lead to many other calls of BUILDTREE.

Let's think about how long each call of BUILDTREE takes to run. We have to consider all possible splits. So we consider a split in each of the d dimensions. In each dimension, we only need to consider splits betwee two data points (any other split will give the same error on the training data). So, in total, we consider $O(dn)$ splits in each call to BUILDTREE.

Study Question: Concretely, what would be a good set of split-points to consider for dimension j of a data set indicated by I?

12.1.1.2 Pruning

It might be tempting to regularize by using a somewhat large value of k, or by stopping when splitting a node does not significantly decrease the error. One problem with short-sighted stopping criteria is that they might not see the value of a split that will require one more split before it seems useful.

Study Question: Apply the decision-tree algorithm to the XOR problem in two dimensions. What is the training-set error of all possible hypotheses based on a single split?

So, we will tend to build a tree that is too large, and then prune it back.

We define *cost complexity* of a tree T, where m ranges over its leaves, as

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m(T) + \alpha|T| \quad , \tag{12.3}$$

and $|T|$ is the number of leaves. For a fixed α, we can find a T that (approximately) minimizes $C_\alpha(T)$ by "weakest-link" pruning:

- Create a sequence of trees by successively removing the bottom-level split that minimizes the increase in overall error, until the root is reached.
- Return the T in the sequence that minimizes the cost complexity.

We can choose an appropriate α using cross validation.

12.1.2 Classification

The strategy for building and pruning classification trees is very similar to the strategy for regression trees.

Given a region R_m corresponding to a leaf of the tree, we would pick the output class y to be the value that exists most frequently (the *majority value*) in the data points whose x values are in that region, i.e., data points indicated by I_m:

$$O_m = \text{majority}_{i \in I_m} \, y^{(i)} \ .$$

Let's now define the error in a region as the number of data points that do not have the value O_m:

$$E_m = \left| \{ i \mid i \in I_m \text{ and } y^{(i)} \neq O_m \} \right| \ .$$

We define the *empirical probability* of an item from class k occurring in region m as:

$$\hat{P}_{m,k} = \hat{P}(I_m, k) = \frac{\left| \{ i \mid i \in I_m \text{ and } y^{(i)} = k \} \right|}{N_m} \ ,$$

where N_m is the number of training points in region m; that is, $N_m = |I_m|$. For later use, we'll also define the empirical probabilities of split values, $\hat{P}_{m,j,s}$, as the fraction of points with dimension j in split s occurring in region m (one branch of the tree), and $1 - \hat{P}_{m,j,s}$ as the complement (the fraction of points in the other branch).

Splitting criteria In our greedy algorithm, we need a way to decide which split to make next. There are many criteria that express some measure of the "impurity" in child nodes. Some measures include:

- *Misclassification error*:
$$Q_m(T) = \frac{E_m}{N_m} = 1 - \hat{P}_{m,O_m} \tag{12.4}$$
- *Gini index*:
$$Q_m(T) = \sum_k \hat{P}_{m,k}(1 - \hat{P}_{m,k}) \tag{12.5}$$
- *Entropy*:
$$Q_m(T) = H(I_m) = -\sum_k \hat{P}_{m,k} \log_2 \hat{P}_{m,k} \tag{12.6}$$

  So that the entropy H is well-defined when $\hat{P} = 0$, we will stipulate that $0 \log_2 0 = 0$.

These splitting criteria are very similar, and it's not entirely obvious which one is better. We will focus on entropy, just to be concrete.

Analogous to how for regression we choose the dimension j and split s that minimizes the sum of squared error $E_{j,s}$, for classification, we choose the dimension j and split s that minimizes the weighted average entropy over the "child" data points in each of the two corresponding splits, $I^+_{j,s}$ and $I^-_{j,s}$. We calculate the entropy in each split based on the empirical probabilities of class memberships in the split, and then calculate the weighted average entropy $\hat{H}$ as

$$\begin{aligned}
\hat{H} &= (\text{fraction of points in left data set}) \cdot H(I^-_{j,s}) \\
&\quad + (\text{fraction of points in right data set}) \cdot H(I^+_{j,s}) \\
&= (1 - \hat{P}_{m,j,s}) H(I^-_{j,s}) + \hat{P}_{m,j,s} H(I^+_{j,s}) \\
&= \frac{|I^-_{j,s}|}{N_m} \cdot H(I^-_{j,s}) + \frac{|I^+_{j,s}|}{N_m} \cdot H(I^+_{j,s}) \ .
\end{aligned} \tag{12.7}$$

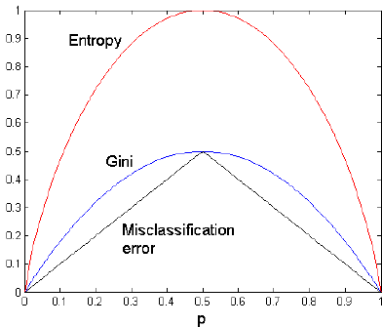
Choosing the split that minimizes the entropy of the children is equivalent to maximizing the *information gain* of the test $x_j = s$, defined by

$$\text{INFOGAIN}(x_j = s, I_m) \quad = \quad H(I_m) - \left(\frac{|I_{j,s}^-|}{N_m} \cdot H(I_{j,s}^-) + \frac{|I_{j,s}^+|}{N_m} \cdot H(I_{j,s}^+) \right) \tag{12.8}$$

In the two-class case (with labels 0 and 1), all of the splitting criteria mentioned above have the values

$$\begin{cases} 0.0 & \text{when } \hat{P}_{m,0} = 0.0 \\ 0.0 & \text{when } \hat{P}_{m,0} = 1.0 \end{cases}.$$

The respective impurity curves are shown below, where $p = \hat{P}_{m,0}$; the vertical axis plots $Q_m(T)$ for each of the three criteria.



There used to be endless haggling about which impurity function one should use. It seems to be traditional to use *entropy* to select which node to split while growing the tree, and *misclassification error* in the pruning criterion.

12.1.3 Bagging

One important limitation or drawback in conventional decision trees is that they can have high estimation error: small changes in the data can result in very big changes in the resulting tree.

Bootstrap aggregation is a technique for reducing the estimation error of a non-linear predictor, or one that is adaptive to the data. The key idea applied to decision trees, is to build multiple trees with different subsets of the data, and then create an ensemble model that combines the results from multiple trees to make a prediction.

- Construct B new data sets of size n. Each data set is constructed by sampling n data points with replacement from $\mathcal{D}$. A single data set is called *bootstrap sample* of $\mathcal{D}$.
- Train a predictor $\hat{f}^b(x)$ on each bootstrap sample.
- *Regression case*: bagged predictor is

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x) \ . \tag{12.9}$$

- *Classification case*: Let K be the number of classes. We find a majority bagged predictor as follows. We let $\hat{f}^b(x)$ be a "one-hot" vector with a single 1 and $K-1$ zeros, and define the predicted output $\hat{y}$ for predictor f^b as $\hat{y}^b(x) = \arg\max_k \hat{f}^b(x)_k$. Then

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x), \tag{12.10}$$

which is a vector containing the proportion of classifiers that predicted each class k for input x. Then the overall predicted output is

$$\hat{y}_{\text{bag}}(x) = \arg\max_k \hat{f}_{\text{bag}}(x)_k \quad . \tag{12.11}$$

There are theoretical arguments showing that bagging does, in fact, reduce estimation error. However, when we bag a model, any simple intrepetability is lost.

12.1.4 Random Forests

Random forests are collections of trees that are constructed to be de-correlated, so that using them to vote gives maximal advantage. In competitions, they often have excellend classification performance among large collections of much fancier methods.

In what follows, B, m, and n are hyperparameters of the algorithm.

RANDOMFOREST($\mathcal{D}$; B, m, n)

for $b = 1, \ldots, B$
    Draw a bootstrap sample $\mathcal{D}_b$ of size n from $\mathcal{D}$
    Grow a tree T_b on data $\mathcal{D}_b$ by recursively:
        Select m variables at random from the d variables
        Pick the best variable and split point among the m variables
        Split the node
return tree T_b

Given the ensemble of trees, vote to make a prediction on a new x.

12.1.5 Decision tree variants and tradeoffs

There are many variations on the decision tree theme. One is to employ different regression or classification methods in each leaf. For example, a linear regression might be used to model the examples in each leaf, rather than using a constant value.

In the relatively simple decision trees that we've considered, splits have been based on only a single feature at a time, and with the resulting splits being axis-parallel. Other methods for splitting are possible, including consideration of multiple features and linear classifiers based on those, potentially resulting in non-axis-parallel splits. Complexity is a concern in such cases, as many possible combinations of features may need to be considered, to select the best variable combination (rather than a single split variable).

Another generalization is a *hierarchical mixture of experts*, where we make a "soft" version of trees, in which the splits are probabilistic (so every point has some degree of membership in every leaf). Such trees can be trained using a form of gradient descent. Combinations of bagging, boosting, and mixture tree approaches (e.g., *gradient boosted decision trees*) and implementations are readily available (e.g., XGBoost).

Decision trees have a number of strengths, and remain a valuable tool in the machine learning toolkit. Some benefits include being relatively easy to interpret, fast to train, and ability to handle multi-class classification in a natural way. Decision trees can easily handle different loss functions; one just needs to change the predictor and loss being applied in the leaves. Methods also exist to identify which features are particularly important or influential in forming the tree, which can aid in human understanding of the data set. Finally, in many situations, decision trees perform surprisingly well, often comparable to more complicated regression or classification models. Indeed, in some settings it is considered good practice to start with decision trees (especially random forest or boosted decision trees) as a "baseline" machine learning model, against which one can evaluate performance of more sophisticated models.

12.2 Nearest Neighbor

In nearest-neighbor models, we don't do any processing of the data at training time – we just remember it! All the work is done at prediction time.

Input values x can be from any domain $\mathcal{X}$ ($\mathbb{R}^d$, documents, tree-structured objects, etc.). We just need a distance metric, $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$, which satisfies the following, for all $x, x', x'' \in \mathcal{X}$:

$$\begin{aligned} d(x, x) &= 0 \\ d(x, x') &= d(x', x) \\ d(x, x'') &\leqslant d(x, x') + d(x', x'') \end{aligned}$$

Given a data-set $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, our predictor for a new $x \in \mathcal{X}$ is

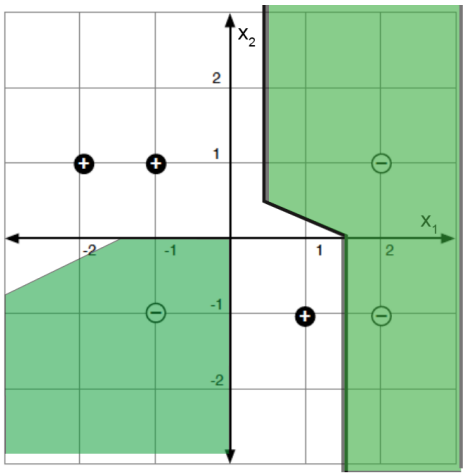
$$h(x) = y^{(i)} \quad \text{where} \quad i = \arg\min_i d(x, x^{(i)}) \ , \tag{12.12}$$

that is, the predicted output associated with the training point that is closest to the query point x.

This same algorithm works for regression *and* classification!

It's a floor wax *and* a dessert topping!

The nearest neighbor prediction function can be described by a Voronoi partition (dividing the space up into regions whose closest point is each individual training point) as shown below:



In each region, we predict the associated y value.

Study Question: Convince yourself that these boundaries do represent the nearest-neighbor classifier derived from these six data points.

There are several useful variations on this method. In k-*nearest-neighbors*, we find the k training points nearest to the query point x and output the majority y value for classification or the average for regression. We can also do *locally weighted regression* in which we fit locally linear regression models to the k nearest points, possibly giving less weight to those that are farther away. In large data-sets, it is important to use good data structures (e.g., ball trees) to perform the nearest-neighbor look-ups efficiently (without looking at all the data points each time).