CHAPTER 13

Non-parametric methods

1 Intro

We will continue to broaden the class of models that we can fit to our data. 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.

We now 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. Some non-parametric models, such as decision trees, which we might call *semi-parametric methods*, 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 semi-parametric methods tend to have the form of a composition of simple models. We'll look at:

- *Tree models*: partition the input space and use different simple predictions on different regions of the space; this increases the hypothesis space.
- *Additive models*: train several different classifiers on the whole space and average the answers; this decreases the estimation error.

Boosting is a way to construct an additive model that decreases both estimation and structural error, but we won't address it 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 hyper-parameters to tune.
- They often work as well or better than more complicated methods.
- Both 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.

2 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," which recursively splits the space.

These methods differ by:

- The class of possible ways to split the space at each node; these are generally linear splits, either aligned with the axes of the space, or sometimes 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 for each individual training example.
- The algorithm for making the partitions and fitting the models.

The primary 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.

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. They 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.

We'll concentrate on the CART/ID3 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.

2.1 Regression

The predictor is made up of

- a partition function, π, mapping elements of the input space into exactly one of M regions, $R_1, \dots, R_M$, 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; that is:

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

Define the error in a region as

$$E_m = \sum_{\{i \mid x^{(i)} \in R_m\}} (y^{(i)} - O_m)^2 \quad .$$

Ideally, we would select the partition to minimize

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

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.

2.1.1 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. We will select the partition of the data that *minimizes the sum of the mean squared errors of each partition.*

Given a data set D, let

- $R^{+}_{j,s}(D) = \{x \in D \mid x_j \geqslant s\}$ be the set of examples in data set D whose feature value in dimension j is greater than or equal to split point s;
- $R^{-}_{j,s}(D) = \{x \in D \mid x_j < s\}$ be the set of examples in D whose feature value in dimension j is less than s;
- $\hat{y}^{+}_{j,s} = \text{average}_{\{i \mid x^{(i)} \in R^{+}_{j,s}(D)\}} y^{(i)}$ be the average y value of the data points in set $R^{+}_{j,s}(D)$; and
- $\hat{y}^{-}_{j,s} = \text{average}_{\{i \mid x^{(i)} \in R^{-}_{j,s}(D)\}} y^{(i)}$ be the average y value of the data points in set $R^{-}_{j,s}(D)$.

Now, here is the pseudocode.

```
BuildTree(D, k):
  • If |D| ⩽ k: return Leaf(D)
  • Find the dimension j and split point s that minimizes: $E_{R^{-}_{j,s}(D)} + E_{R^{+}_{j,s}(D)}$ .
  • Return Node(j, s, BuildTree($R^{-}_{j,s}$(D, k)), BuildTree($R^{+}_{j,s}$(D, k)))
```

Each call to **BuildTree** considers $O(dn)$ splits (for d dimensions, since we only need to split between each data point in each dimension); each requires $O(n)$ work where n is the number of data points considered ($n = |D|$ if all data points in D are used).

Study Question: Concretely, what would be a good set of split-points to consider for dimension j of a dataset D?

2.1.2 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.

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

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m(T) + \alpha|T| \ .$$

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 criterion.

We can choose an appropriate α using cross validation.

2.2 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:

$$O_m = \text{majority}_{\{i|x^{(i)}\in R_m\}} y^{(i)} \ .$$

Define the error in a region as the number of data points that do not have the value O_m:

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

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

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

where N_m is the number of training points in region m. We'll define the empirical probabilities of split values, as well, for later use.

$$\hat{P}_{mjv} = \hat{P}(R_{mj})(v) = \frac{\left|\{i \mid x^{(i)} \in R_m \text{ and } x_j^{(i)} \geqslant v\}\right|}{N_m}$$

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*:
$$Q_m(T) = \frac{E_m}{N_m} = 1 - \hat{P}_{mO_m}$$
- *Gini index*:
$$Q_m(T) = \sum_k \hat{P}_{mk}(1 - \hat{P}_{mk})$$
- *Entropy*:
$$Q_m(T) = H(R_m) = -\sum_k \hat{P}_{mk} \log_2 \hat{P}_{mk}$$

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

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

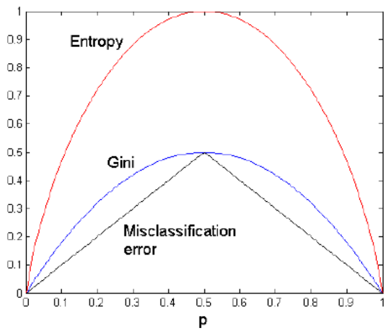
Choosing the split that minimizes the entropy of the children is equivalent to maximizing the *information gain* of the test $X_j = v$, defined by

$$\text{infoGain}(X_j = v, R_m) \quad = \quad H(R_m) - \left(\hat{P}_{mjv} H(R^{+}_{j,v}) + (1 - \hat{P}_{mjv}) H(R^{-}_{j,v})\right)$$

In the two-class case, all the criteria have the values

$$\begin{cases} 0.0 & \text{when } \hat{P}_{m0} = 0.0 \\ 0.0 & \text{when } \hat{P}_{m0} = 1.0 \end{cases}$$

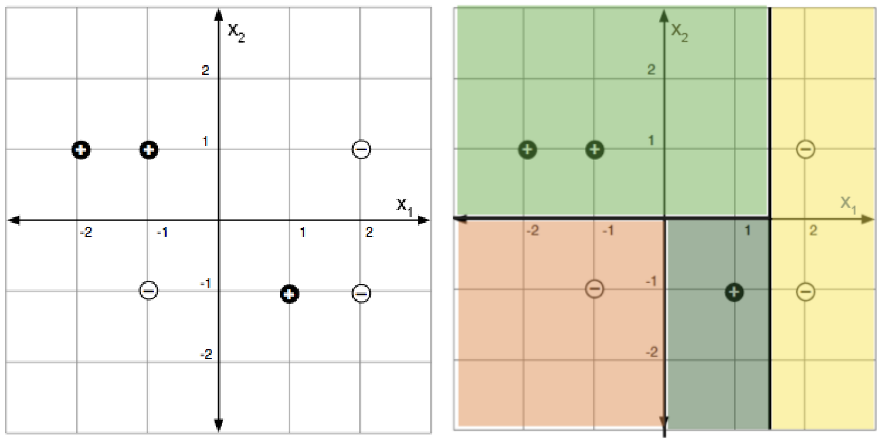
The respective impurity curves are shown below, where $p = \hat{p}_{m0}$:



There used to be endless haggling about which one to use. It seems to be traditional to use:

- Entropy to select which node to split while growing the tree
- Misclassification error in the pruning criterion

As a concrete example, consider the following images:



The left image depicts a set of labeled data points and the right shows a partition into regions by a decision tree.

Points about trees There are many variations on this theme:

- Linear regression or other regression or classification method in each leaf

- Non-axis-parallel splits: e.g., run a perceptron for a while to get a split.

What's good about trees:

- Easily interpretable
- Fast to train!
- Easy to handle multi-class classification
- Easy to handle different loss functions (just change predictor in the leaves)

What's bad about trees:

- High estimation error: small changes in the data can result in very big changes in the hypothesis.
- Often not the best predictions

Hierarchical mixture of experts Make a "soft" version of trees, in which the splits are probabilistic (so every point has some degree of membership in every leaf). Can be trained with a form of gradient descent.

3 Bagging

Bootstrap aggregation is a technique for reducing the estimation error of a non-linear predictor, or one that is adaptive to the data.

- Construct B new data sets of size n by sampling them with replacement from $\mathcal{D}$
- Train a predictor on each one: $\hat{f}^b$
- *Regression case*: bagged predictor is

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

- *Classification case*: majority bagged predictor: let $\hat{f}^b(x)$ be a "one-hot" vector with a single 1 and $K-1$ zeros, so that $\hat{y}^b(x) = \arg\max_k \hat{f}^b(x)_k$. Then

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

  which is a vector containing the proportion of classifiers that predicted each class k for input x; and the predicted output is

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

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

3.1 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 the best classification performance among large collections of much fancier methods.

For $b = 1..B$

- Draw a bootstrap sample $\mathcal{D}_b$ of size n from $\mathcal{D}$
- Grow a tree on data $\mathcal{D}_b$ by recursively repeating these steps:
  - Select m variables at random from the d variables
  - Pick the best variable and split point among them
  - Split the node
- Return tree T_b

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

4 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

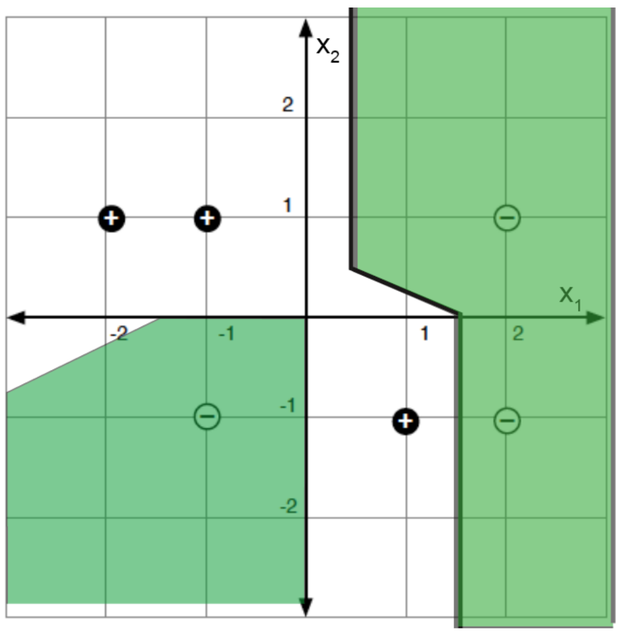
$$h(x) = y^{(i)} \quad \text{where} \quad i = \arg\min_i d(x, x^{(i)}) \ ,$$

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 6 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).