Classification: Kernels and Decision Trees

Siva Balakrishnan
Data Mining: 36-462/36-662

February 14th, 2018

Chapter 8.1 - 8.2 of ISL (Decision Trees)

Recap: Soft-Margin SVM

- To overcome problems with the Hard-Margin SVM (can be unstable, data may not be linearly separable) we introduced *slack variables* to allow points to “violate the margin”.

ϵ_i - slack vars.

$$\text{Maximize}_{M,\beta,\beta_0,\epsilon}\ \boxed{M}$$

$$\text{subject to}\ \sum_{j=1}^{p}\beta_j^2 = 1,\ \varepsilon_i \geq 0,\ \underbrace{\sum_{i=1}^{n}\varepsilon_i \leq C}_{}$$

$$y_i(\beta_0 + x_i^T\beta) \geq M(1 - \underline{\underline{\varepsilon_i}})$$

- When we solve this program the value of the slack variables tells us where the point is: $\epsilon_i = 0$, x_i is correctly classified at least M away from boundary.

$\underline{\epsilon_i > 1}$, x_i is misclassified.

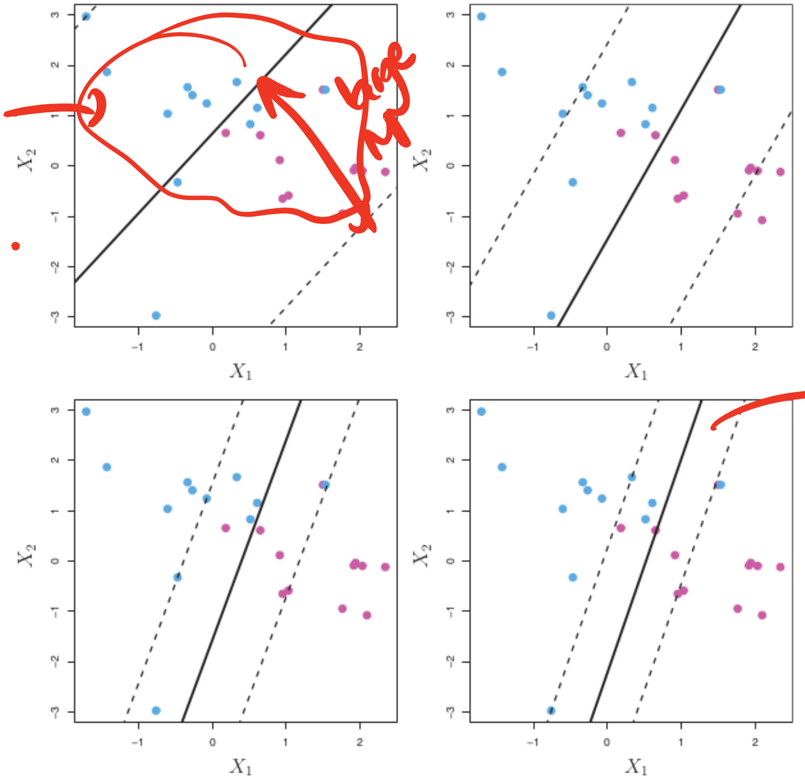
$0 < \epsilon_i < 1$, x_i is correctly classified but not outside margin.

- The tuning parameter C is critical. Increasing C, usually increases bias and decreases variance. We typically choose it by cross-validation.

Recap: The parameter C

- One way to think about C is to note that it is an upper bound on the number of training errors.
- If we want to understand its effect on bias and variance we should recall:

C is large
many SVs
large margin.



C is small.

Recap: Kernels

- SVMs give us a way to obtain a linear classifier with a large margin. Suppose we want a non-linear classifier.
- The usual answer is to use feature expansions, i.e. we take our features and concatenate new features which are combinations of existing features.

$$\Phi((\underline{\text{balance}}, \underline{\text{income}})) = (\boxed{\text{b}}, \boxed{\text{i}}, \underline{\text{b} \times \text{i}}, \underline{\text{b}}^2, \underline{\text{i}}^2).$$

- A linear classifier in the expanded feature space is a non-linear classifier in the original space.
- Can be computationally very annoying – we have to create, store and manipulate these much (much) larger feature vectors.

Recap: Kernels

- If the optimal hyper-plane was a linear combination of our data-points (it always is):

n^2 inner products.

$$\widehat{\beta} = \underline{\sum_{i=1}^{n} \alpha_i x_i},$$

then SVMs could be written only in terms of inner products $x_i^T x_j$ for the training data (and of course the labels).

- To obtain the SVM classifier (after feature expansion) we do not need to store the big feature vectors, we just need to be able to compute their inner products quickly, i.e. we need some way of computing $\Phi(x)^T \Phi(x')$ for pairs of training examples.

Recap: Kernels

- For many interesting, non-linear kernels, we can compute $\Phi(x)^T\Phi(x')$ very easily using a kernel function:

$$\boxed{K(x, x')} = \underline{\Phi(x)^T\Phi(x')}.$$

- For example, suppose our original data is 2 dimensional, if we choose a quadratic feature map (so we can learn quadratic decision boundaries):

$$\Phi(x) = (\underline{1}, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \underline{\sqrt{2}x_1x_2}).$$

→ $\Phi(x)^T\Phi(x')$.

Instead of computing $\Phi(x)^T\Phi(x')$ by this feature expansion we can see that this just corresponds to:

$$\underline{K(x, x') = (1 + x^Tx')^2}.$$

→ don't need to do feature exp.

- For higher-order polynomials we use the polynomial kernel:

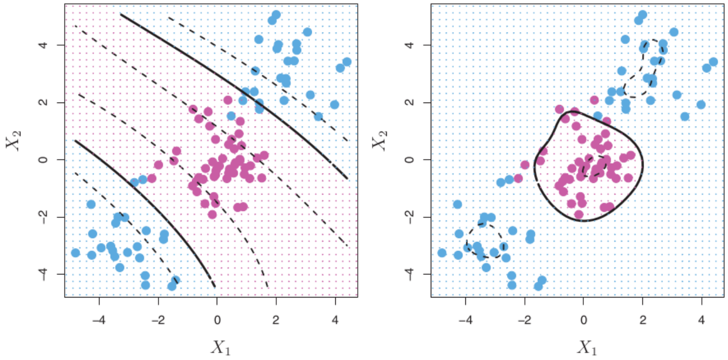
$$K(x, x') = (1 + x^Tx')^p.$$

Another popular kernel is the Radial Basis Function kernel:

$$K(x, x') = \exp(-\gamma\|x - x'\|_2^2).$$

Recap: Kernels Main Points

- We can make linear classifiers non-linear by feature expansion.
- Many classifiers only need inner products between the training examples.
- We can often compute inner-products between the feature expanded training examples *directly* using kernels.
- This gives us a way to quickly "non-linearize" (kernelize) classifiers without having to carefully craft feature expansions.



Recap: How do we use kernel SVMs?

- When we run a kernel SVM package (e1071) the coefficients it returns to us are now β_0 and α_i, and our classification function takes the form:

$\beta_0 + \beta^T x.$

$$\hat{f}(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x_i, x).$$

- To classify a new point: x

Compute $\hat{f}(x)$

Class +1, if $\hat{f}(x) > 0$

−1 if $\hat{f}(x) \leq 0.$

non-linear classifier.

SVM: a different perspective

There is another important way of thinking about the linear SVM. It turns out that we can re-write the optimization problem (its a bit of work) as solving:

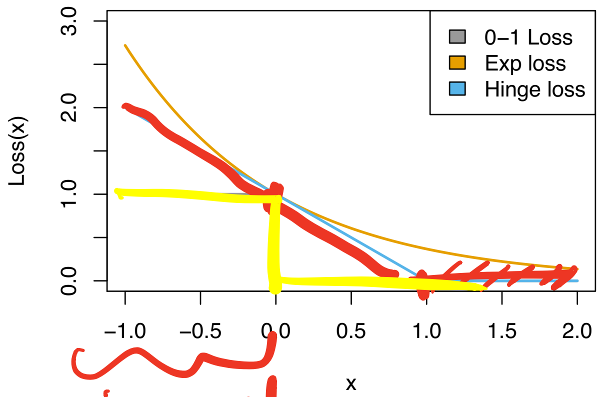
Soft margin

min loss + ridge penalty

$\max\{1-y_i f(x_i), 0\}$

$$\min_{\beta}\sum_{i=1}^{n}\underbrace{(1-y_i f(x_i))_+}_{\text{hinge loss}}+\frac{\lambda}{2}\|\beta\|_2^2$$

This is like replacing 0-1 loss with a hinge function ~~and adding~~ a ridge penalty to keep things regularized!



incorrect

If $y\beta^T x > 1$. then no loss.

If $y\beta^T x < 0$. loss

$1 + |y\beta^T x| > 1$.

Empirical risk minimization

This general pattern:

1. We want to minimize:

$$\boxed{\mathbb{E}\mathbf{1}\{Y \neq f(X)\}}$$

2. And so we actually try to minimize its empirical version:

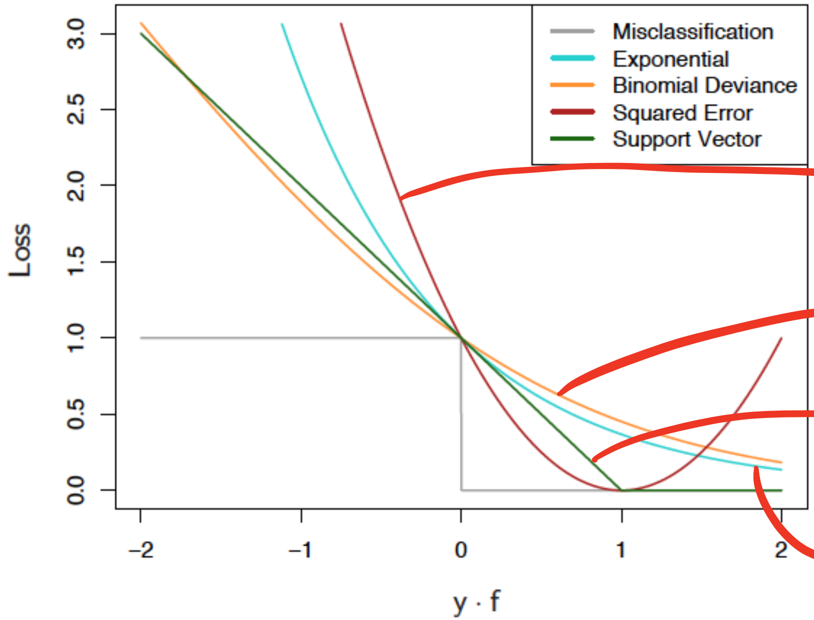
$$\boxed{\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{y_i \neq f(x_i)\}} \rightarrow \text{min. training 0/1 loss.}$$

3. But we can't even do that for classification. So we introduce a nicer loss L and minimize

$$\underline{\frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i))} + \text{regularizer}$$

The first two steps are known as *empirical risk minimization*. The last step almost always follows for classification.

Empirical risk minimization



→ loss of least squares.

→ logistic regression loss

→ hinge loss

→ $\exp\{-y \cdot \beta^T x\}$.

"bigger than 0/1 loss"

- What do you observe about all these losses?
- Why is this a nice thing? → classifier with small hingeloss than must have 0/1 loss

Multiclass SVM

- Unlike logistic regression and LDA there is no particularly natural way to take the (binary) SVM and use it in multi-class settings.
- Two popular methods are:

class 1 vs everything else

class K vs " " .

1. One-versus-all classification: Here we fit K different SVMs $\{(\widehat{\beta}_{01}, \widehat{\beta}_1), \ldots, (\widehat{\beta}_{0K}, \widehat{\beta}_K)\}$ by comparing each class to all the other classes.
   To finally classify a point we use:
   $$\widehat{f}(x) = \arg\max_k \widehat{\beta}_{0k} + \widehat{\beta}_k^T x.$$
2. One-versus-one classification: Here we fit $\binom{K}{2}$ different SVMs, by comparing each class with every other class.
   To classify a new test example: we classify it according to each of the classifers and pick the class that is chosen most often.

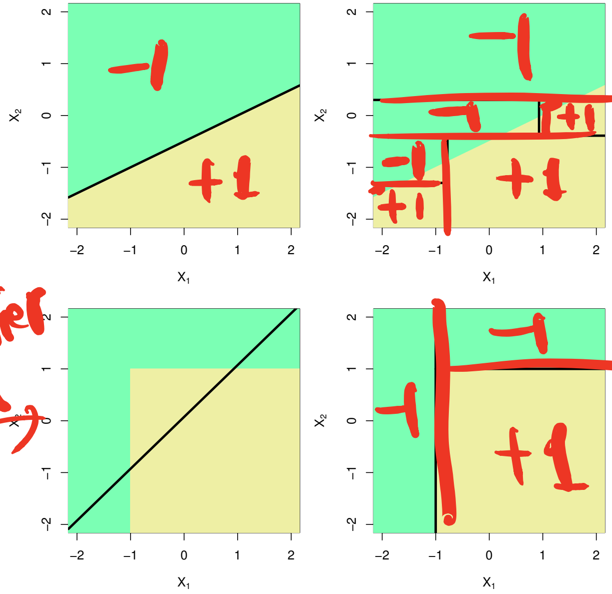
Overview: Tree-based methods (Disc. models)

Tree-based based methods for predicting y from a feature vector $x \in \mathbb{R}^p$ divide up the feature space into rectangles, and then fit a very simple model in each rectangle. This works both when y is discrete and continuous, i.e., both for classification and regression

Class → $\mathbb{P}(y=k|X=x)$

Reg → $\mathbb{E}[y|X=x]$.

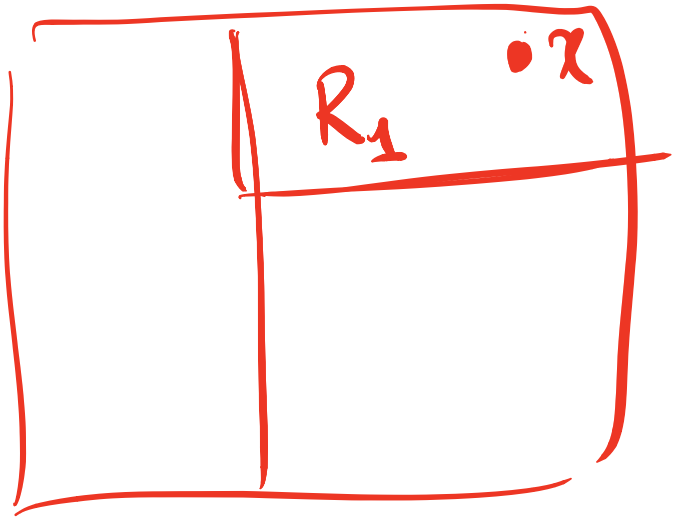
linear classifier has too much bias.



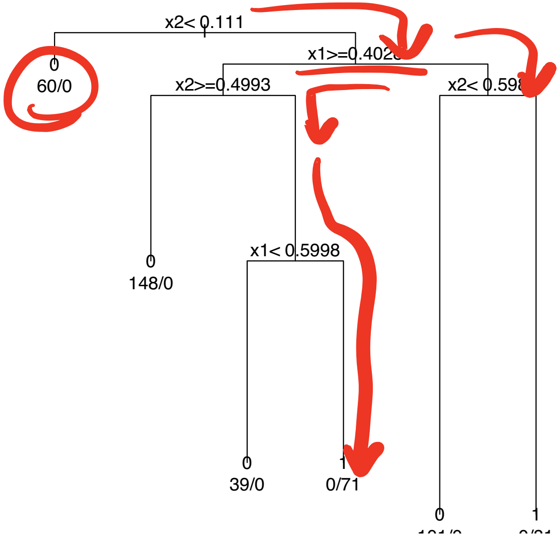
→ trees are great.

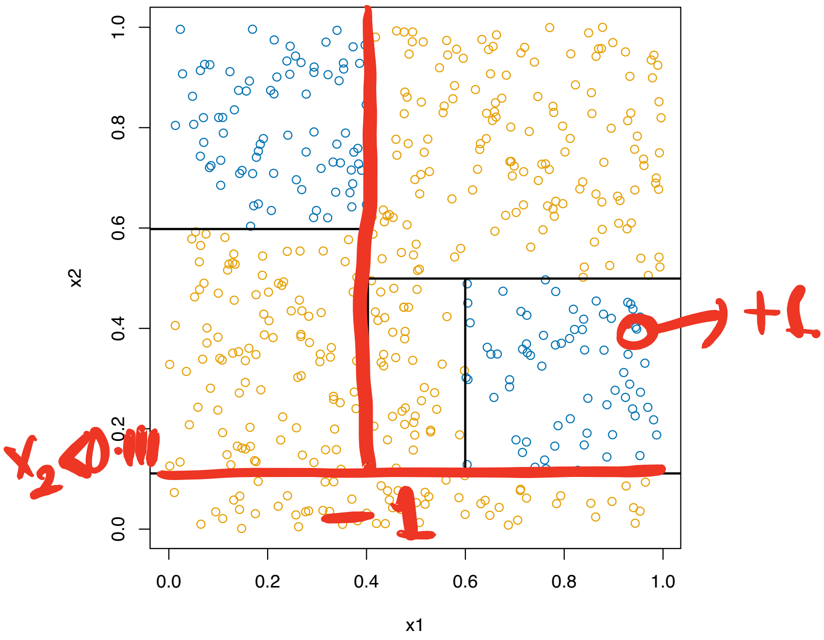
(ISL Figure 8.7)

This is a big shift from thinking about linear-style models!



$$\mathbb{P}(y=k \mid X=x) \approx \mathbb{P}(y=k \mid X \in R_1).$$

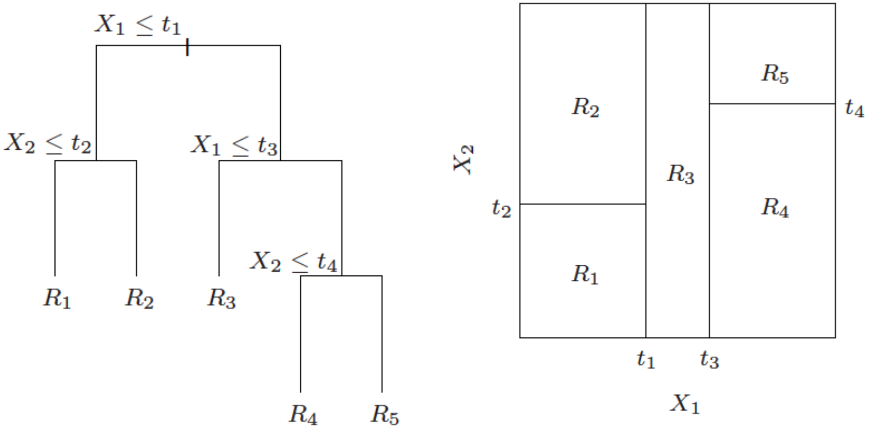




This gives a rule that is easy to understand, easy to explain, and easy to implement!

No more coefficients to interpret!

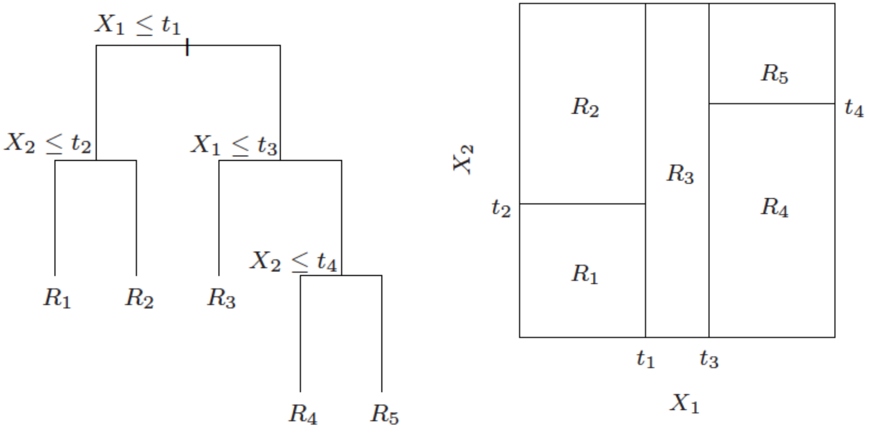
Classification trees



The classification tree can be thought of as defining m regions (rectangles) $R_1, \ldots R_m$, each corresponding to a leaf of the tree

We assign each R_j a class label $c_j \in \{1, \ldots K\}$. We then classify a new point $x \in \mathbb{R}^p$ by

$$\hat{f}^{\text{tree}}\underline{\underline{(x)}} = \sum_{j=1}^{m} c_j \cdot 1\{x \in R_j\} = c_j \text{ if } x \in R_j$$



$$\hat{f}^{\text{tree}}(x) \; = \; \sum_{j=1}^{m} c_j \cdot 1\{x \in R_j\}$$

Finding out which region a given point x belongs to is easy since the regions R_j are defined by a tree—we just scan down the tree. Otherwise, it would be a lot harder (need to look at each region)

Estimated class probabilities

Note that each region R_j contains some subset of the training data (x_i, y_i), $i = 1, \dots n$, say, n_j points.

We have been predicting class $\underline{c_j}$ using the most common class among points in R_j.

For each class k, we can also estimate the probability that a point has that class, given that it falls in R_j, $\underline{\mathrm{P}(C = k | X \in R_j)}$, by

$$\underline{\widehat{p}_k(R_j)} = \frac{1}{n_j} \sum_{x_i \in R_j} \underline{1\{y_i = k\}} \longrightarrow \text{just a multinomial.}$$

the proportion of points in the region that are of class k.

We can even think of our predicted class as

$$c_j = \underline{\underset{k=1,\dots K}{\operatorname{argmax}} \ \widehat{p}_k(R_j)}$$

How to build trees?

There are two main issues to consider in building a tree:

1. How to choose the splits?
2. How big to grow the tree? → how to not overfit

→ Class & Reg. trees.

The CART Algorithm:

1. Choose splits greedily for best improvement at each step, starting from the root.
2. Grow the tree very deep to avoid getting stuck locally
3. Prune the tree back to a reasonable size to reduce variance.

Recall that in a region R_m, the proportion of points in class k is

$$\widehat{p}_k(R_m) = \frac{1}{n_m} \sum_{x_i \in R_m} 1\{y_i = k\}.$$

The CART algorithm begins by considering splitting on variable j and split point s, and defines the regions

$$\underline{R_1 = \{X \in \mathbb{R}^p : X_j \le s\}}, \quad \underline{R_2 = \{X \in \mathbb{R}^p : X_j > s\}}$$

We then greedily chooses j, s by minimizing the misclassification error

$$\operatorname*{argmin}_{j,s} \Big(n_{R_1}\big[1 - \underline{\widehat{p}_{c_1}(R_1)}\big] + n_{R_2}\big[1 - \widehat{p}_{c_2}(R_2)\big] \Big)$$

mis-class of this split.

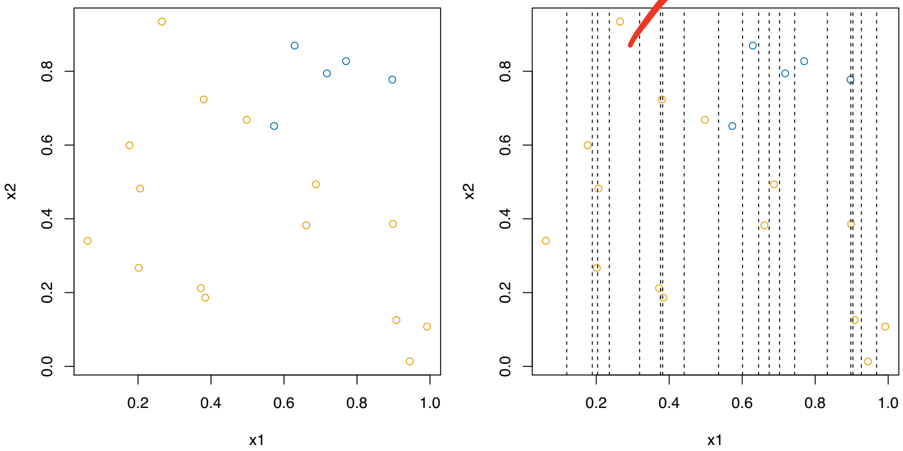
Here $c_1 = \underline{\operatorname{argmax}_{k=1,\dots K} \widehat{p}_k(R_1)}$ is the most common class in R_1, and $c_2 = \operatorname{argmax}_{k=1,\dots K} \widehat{p}_k(R_2)$ is the most common class in R_2

We now repeat this within each of the newly defined regions R_1, R_2. Again consider all variables and split points for each of R_1, R_2, greedily choosing the biggest improvement in misclassification error.

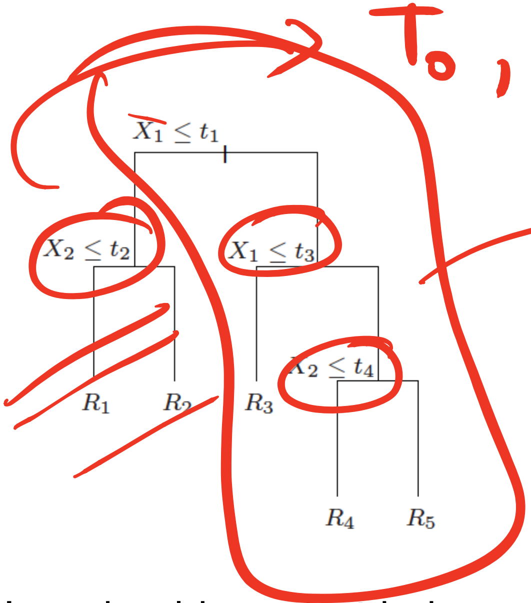
How do we find the best split s?

Aren't there infinitely many to consider?

n_j points, => only consider ≈ n_j splits



T_0 , $\{T_0^1, T_0^2, T_0^3\}$. $\{$ $\}$



Continuing on in this manner, we will get a big tree T_0. Its leaves define regions $R_1, \ldots R_m$. We then prune this tree, meaning that we collapse some of its leaves into the parent nodes

How should we decide how much to prune a tree?

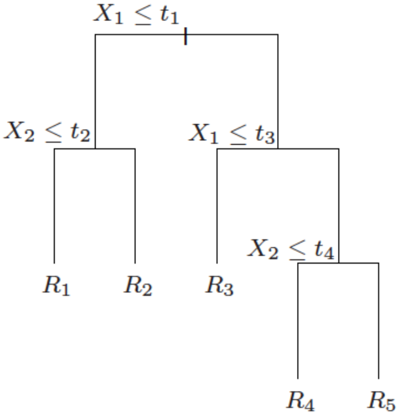
Like to use Cross-validation.

What is weird about applying this here?

No obvious parameter.

Order trees by eliminating "leaves", pick best one on held out set.

Details are not very important but basic idea is



For any tree T, let $|T|$ denote its number of leaves. We define

$$C_\alpha(T) = \sum_{j=1}^{|T|} n_{R_j}\left[1-\widehat{p}_{c_j}(R_j)\right]+\alpha|T|$$

try to reduce variance by eliminating small leaves

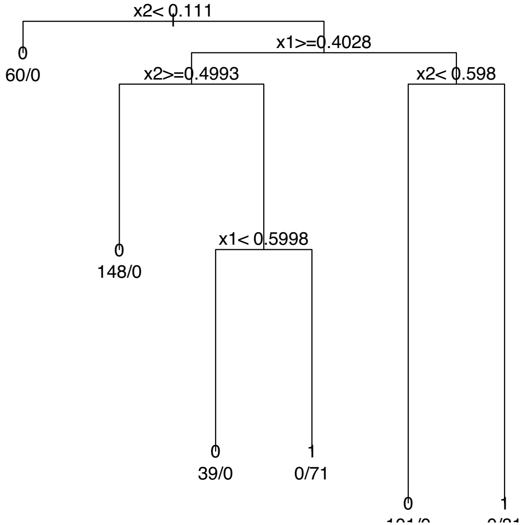
We seek the tree $T \subseteq T_0$ that minimizes $C_\alpha(T)$. It turns out that this can be done by pruning the weakest leaf one at a time.

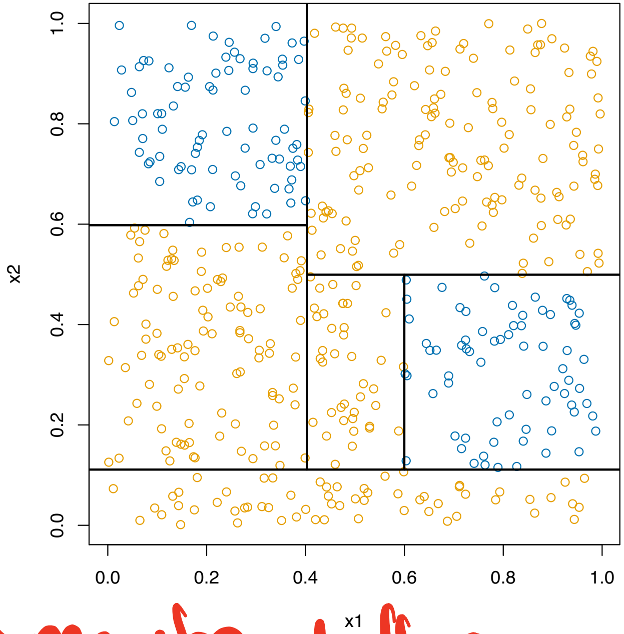
Note that α is a tuning parameter, and a larger α yields a smaller tree. CART picks α by 5- or 10-fold cross-validation

use CV to determine "how much to prune"

Example: simple classification tree

Example: $n = 500$, $p = 2$, and $K = 2$. We ran CART:





maybe better

In R, can use `rpart` or `tree`.

Example: spam data

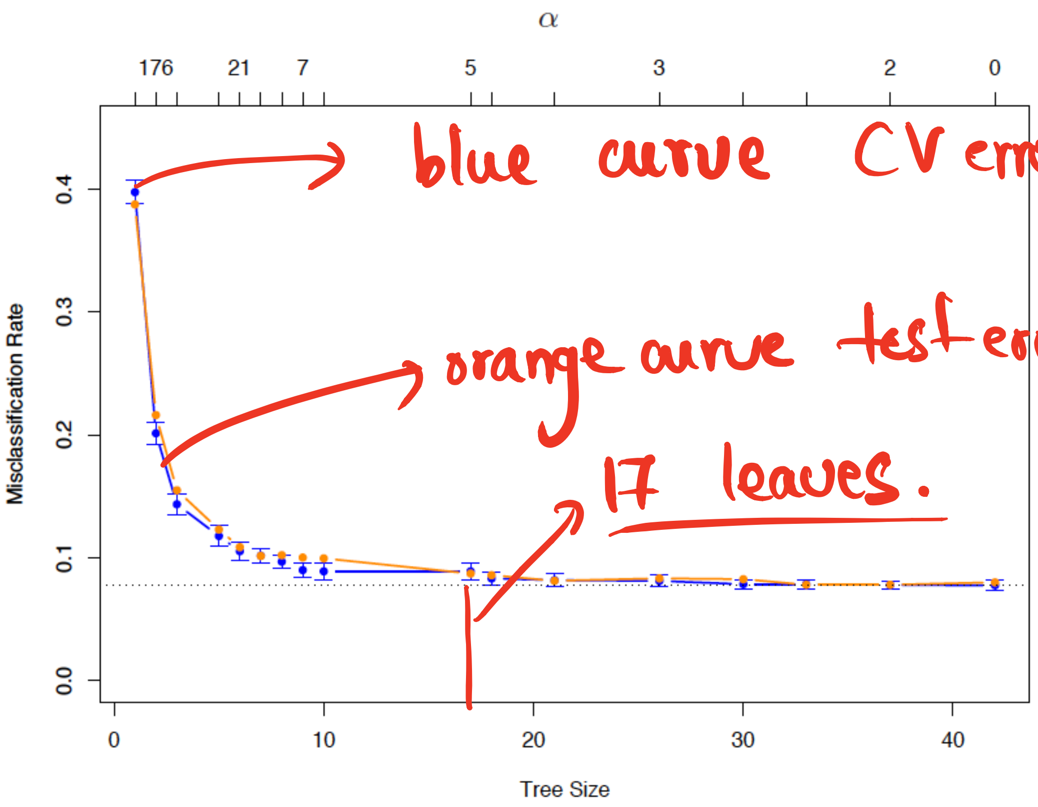
Example: $n = 4601$ emails, of which 1813 are considered spam. For each email we have $p = 58$ attributes.

The first 54 features measure the frequencies of 54 key words or characters (e.g., “free”, “need”, “\$”). The last 3 measure

- the average length of uninterrupted sequences of capitals;
- the length of the longest uninterrupted sequence of capitals;
- the sum of lengths of uninterrupted sequences of capitals

(Data from ESL section 9.2.5)

Cross-validation error curve for the spam data (from ESL page 314):

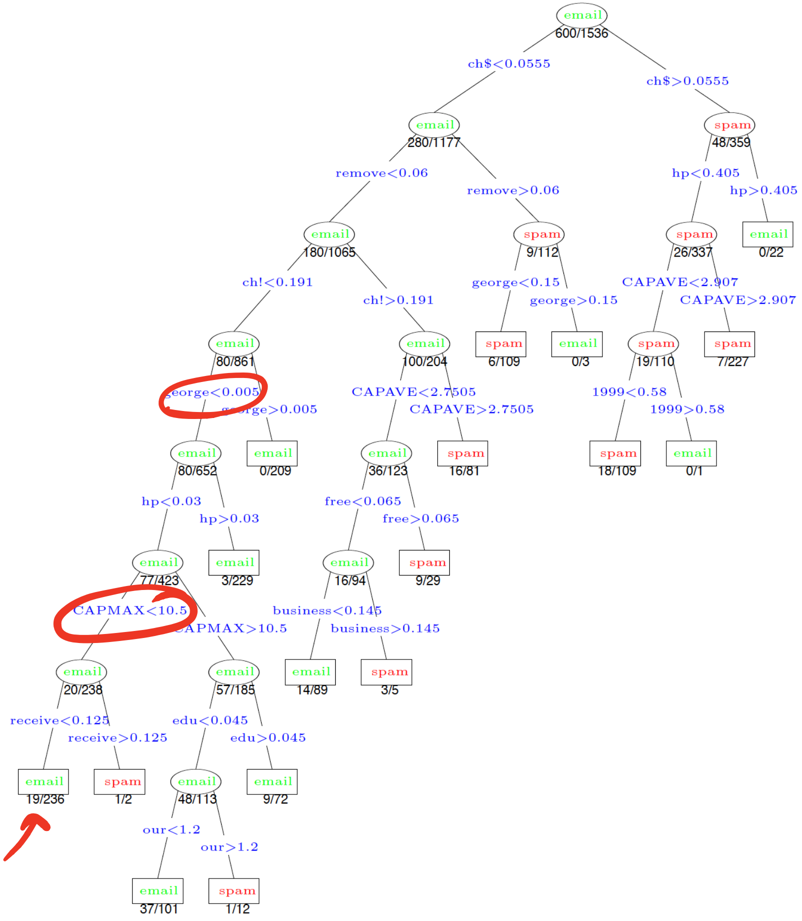


blue curve CV error

orange curve test error

17 leaves.

Tree of size 17, chosen by cross-validation (from ESL page 315):



Note: The leaf annotations are a bit different here.

Other impurity measures

We used misclassification error as a measure of the impurity of region R_j,

$$\underline{1 - \widehat{p}_{c_j}(R_j)}$$

if mis-class err is 0 then two pure nodes.

But there are other useful measures too: the Gini index:

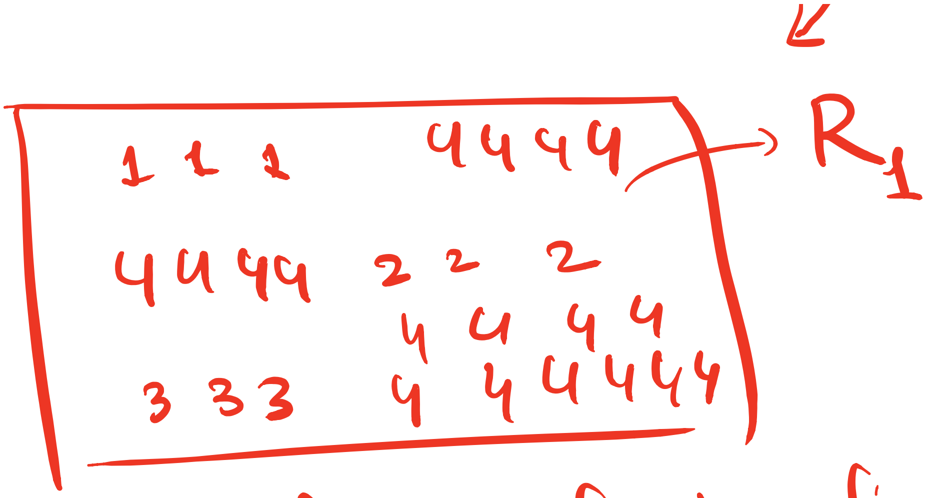
$$\sum_{k=1}^{K} \widehat{p}_k(R_j)\left[1 - \widehat{p}_k(R_j)\right],$$

multinomial

and the cross-entropy or deviance:

$$-\sum_{k=1}^{K} \widehat{p}_k(R_j) \log\left\{\widehat{p}_k(R_j)\right\}.$$

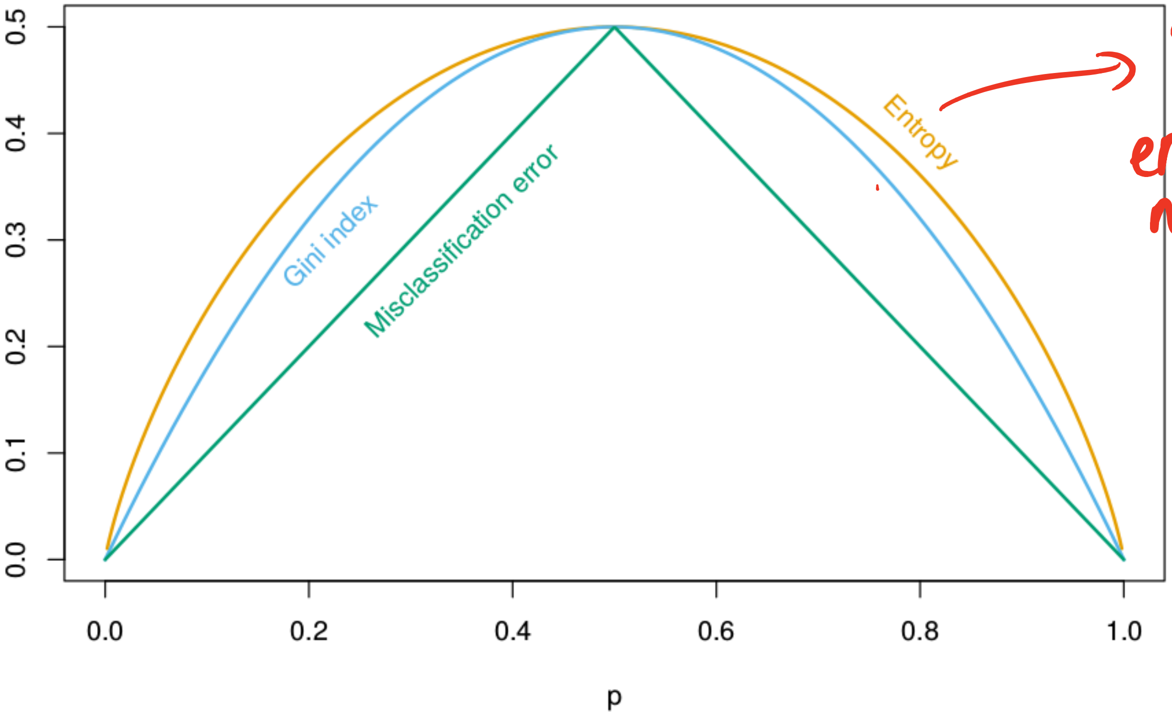
Using these measures instead of misclassification error is sometimes preferable because they are more sensitive to changes in class probabilities.



$$\left(\text{fraction in } \underline{1},\ \text{fr in } 2,\ \text{fr in } 3,\ \text{fr in } 4\right)$$

$$1 - \max_{k} p(\text{class } k).$$

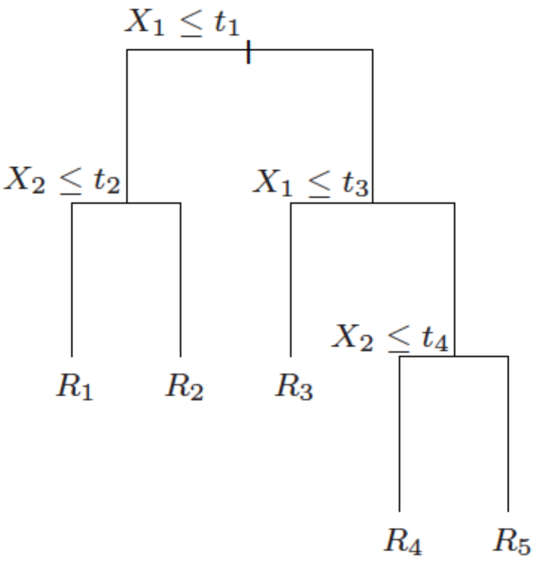
Other impurity measures

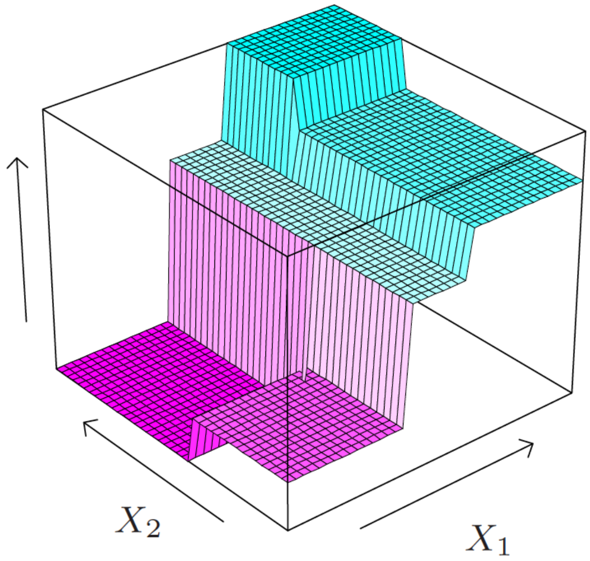


"stronger encouragement for purity".

Regression trees

Suppose that now we want to predict a continuous outcome instead of a class label. Essentially, everything follows as before, but now we just fit a constant inside each rectangle





The estimated regression function has the form

$$\hat{f}^{\text{tree}}(x) \;=\; \sum_{j=1}^{m} c_j \cdot 1\{x \in R_j\} \;=\; c_j \;\text{ such that }\; x \in R_j$$

just as it did with classification. The quantities c_j are no longer predicted classes, but instead they are real numbers: the average response within each region:

$$c_j = \frac{1}{n_j} \sum_{x_i \in R_j} y_i$$

The main difference in building the tree is that we use squared error loss instead of misclassification error (or Gini index or deviance) to decide which region to split.

Categorical predictors

- If a categorical predictor takes on q different values then how many splits do we have to consider?

$\{1, 2, 3, 4, 5\}$.

$\{1, 2, 3\}$ $\quad$ $\{4, 5\}$

$\{1, 4, 5\}$ $\quad$ $\{2, 3\}$

$\vdots$

$2^{q-1} - 1$.

Trees provide a good balance

	Model assumptions?	Estimated probabilities?	Interpretable?	Flexible?
LDA	Yes	Yes	Yes	No
LR	Yes	Yes	Yes	No
k-NN	No	A bit	No	Yes
Trees	No	Yes	Yes	Somewhat

maybe?

How well do trees predict?

Trees seem to have a lot of things going in the favor. So how is their predictive ability?

Unfortunately, the answer is not great.

Trees tend to suffer from high variance because they are quite unstable: a small change in the observed data can lead to a dramatically different sequence of splits, and hence a different prediction.

This instability comes from their greedy nature; once a split is made, it is permanent and can never be "unmade" further down in the tree

However, we will see that trees form the building blocks for some very powerful predictive methods.