Classification and Regression Trees

Code

Ruoqing Zhu

Last Updated: September 24, 2023

Classification and Regression Trees

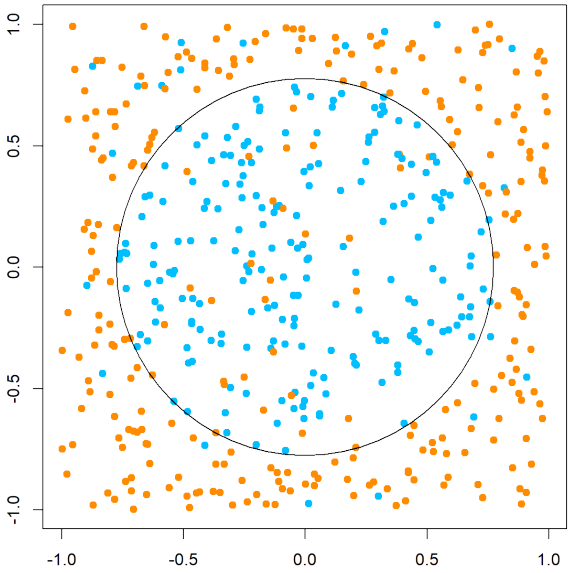
A tree model is very simple to fit and enjoys interpretability. It is also the core component of random forest and boosting. Both trees and random forests can be used for classification and regression problems, although trees are not ideal for regressions problems due to its large bias. There are two main stream of tree models, Classification and Regression Trees (CART, Breiman et al., 1984) and C4.5 [Quinlan, 1994], which is an improvement of the ID3 (Iterative Dichotomiser 3) algorithm. The main difference is to use binary or multiple splits and the criteria of the splitting rule. In fact the splitting rule criteria is probably the most essential part of a tree.

Example: Classification Tree

Let's generate a model with nonlinear classification rule.

Hide

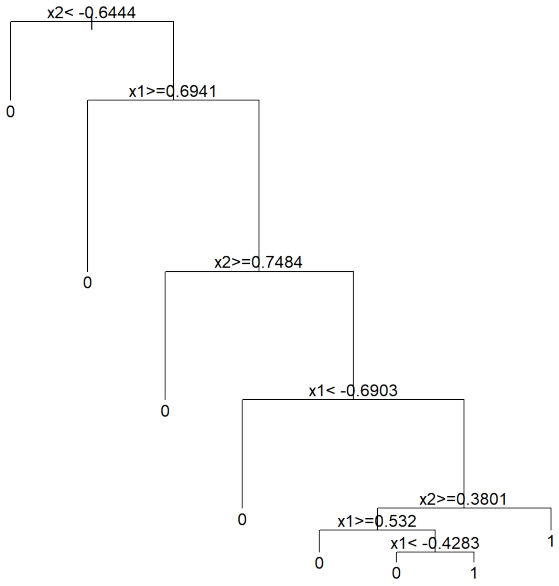
```
set.seed(1)
n = 500
x1 = runif(n, -1, 1)
x2 = runif(n, -1, 1)
y = rbinom(n, size = 1, prob = ifelse(x1^2 + x2^2 < 0.6, 0.9, 0.1))

par(mar=rep(2,4))
plot(x1, x2, col = ifelse(y == 1, "deepskyblue", "darkorange"), pch = 19)
symbols(0, 0, circles = sqrt(0.6), add = TRUE, inches = FALSE, cex = 2)
```



A classification tree model is recursively splitting the feature space such that eventually each region is dominated by one class. We will use `rpart` as an example to fit trees, which stands for recursively partitioning.

Hide

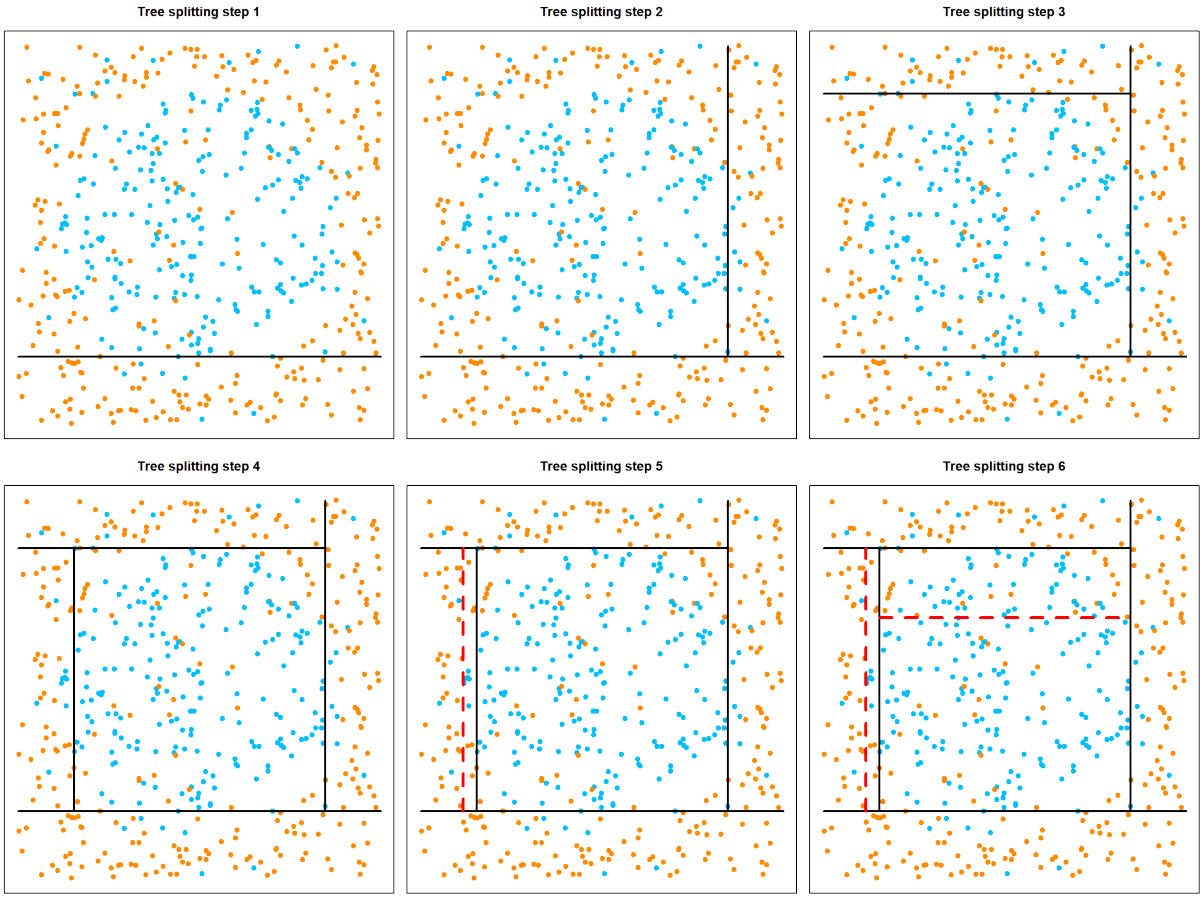
```
set.seed(1)
library(rpart)
rpart.fit = rpart(as.factor(y)~x1+x2, data = data.frame(x1, x2, y))

# the tree structure
par(mar=rep(0.5, 4))
plot(rpart.fit)
text(rpart.fit)
```



Hide

```
    # if you want to peek into the tree
    # note that we set cp = 0.012, which is a tuning parameter
    # we will discuss this later
    rpart.fit$cptable
##           CP nsplit rel error    xerror       xstd
## 1 0.17040359      0 1.0000000 1.0000000 0.04984280
## 2 0.14798206      3 0.4843049 0.7264574 0.04692735
## 3 0.01121076      4 0.3363229 0.4484305 0.04010884
## 4 0.01000000      7 0.3004484 0.4035874 0.03852329
    prune(rpart.fit, cp = 0.012)
## n= 500
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 500 223 0 (0.55400000 0.44600000)
##    2) x2< -0.6444322 90   6 0 (0.93333333 0.06666667) *
##    3) x2>=-0.6444322 410 193 1 (0.47073171 0.52926829)
##      6) x1>=0.6941279 68   8 0 (0.88235294 0.11764706) *
##      7) x1< 0.6941279 342 133 1 (0.38888889 0.61111111)
##       14) x2>=0.7484327 53   7 0 (0.86792453 0.13207547) *
##       15) x2< 0.7484327 289  87 1 (0.30103806 0.69896194)
##         30) x1< -0.6903174 51   9 0 (0.82352941 0.17647059) *
##         31) x1>=-0.6903174 238  45 1 (0.18907563 0.81092437) *
```

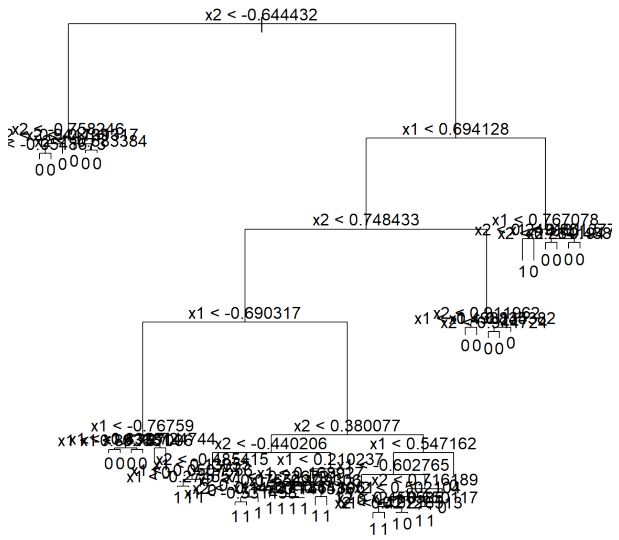
The model proceed with the following steps. Note that steps 5 and 6 may not be really beneficial (consider that we know the true model).



Alternatively, there are many other packages that can perform the same analysis. For example, the `tree` package. However, be careful that this package uses a different splitting rule by default If you want to match the result, use `split = "gini"` . Note that this plot is very crowded because it will split until pretty much only one class in each terminal node. Hence, you can imaging that there will be a tuning parameter issue. We will discuss this later.

Hide

```
    library(tree)
    tree.fit = tree(as.factor(y)~x1+x2, data = data.frame(x1, x2, y), split = "gin
i")
    plot(tree.fit)
    text(tree.fit)
```



Splitting a Node

In a tree model, the splitting mechanism performs in the following way, which is just comparing all possible splits on all variables. For simplicity, we will assume that a binary splitting rule is used, i.e., we split the current node into to two child nodes, and apply the procedure recursively.

- At the current node, go through each variable to find the best cut-off point that splits the node.
- Compare all the best cut-off points across all variable and choose the best one to split the current node and then iterate.

So, what error criterion should we use to compare different cut-off points? There are three of them at least:

- Gini impurity (CART)
- Shannon entropy (C4.5)
- Mis-classification error

Gini impurity is used in CART, while ID3/C4.5 uses the Shannon entropy. These criteria have different effects than the mis-classifications error. They usually prefer more “pure” nodes, meaning that it is more likely to single out a set of pure class terminal node if we use Gini impurity and Shannon entropy. This is because their measures are nonlinear.

Suppose that we have a population (or a set of observations) with p_k proportion of class k, for $k = 1, \dots, K$. Then, the Gini impurity is given by

$$\text{Gini} = \sum_{k=1}^{K} p_k(1 - p_k) = 1 - \sum_{k=1}^{K} p_k^2.$$

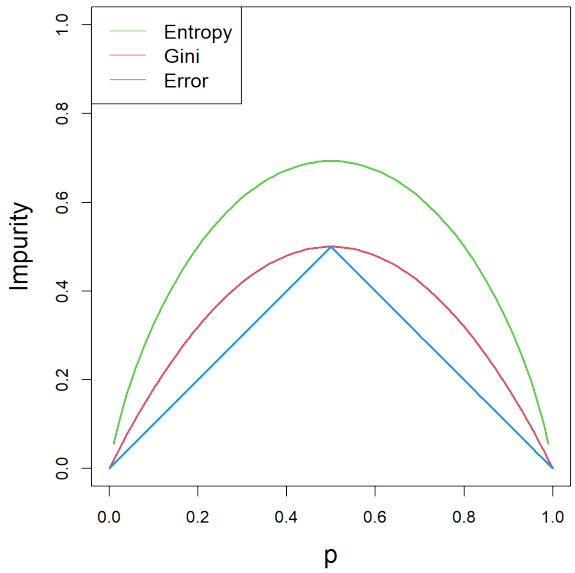
The Shannon theory is defined as

$$-\sum_{k=1}^{K} p_k \log(p_k).$$

And the classification error simply adds up all mis-classified portions if we predict the population into the most prevalent one:

$$1 - \max_{k=1,\ldots,K} p_k$$

The following plot shows all three quantities as a function of p, when there are only two classes, i.e., $K = 2$.



For each quantity, smaller value means that the node is more "pure", hence, there is a higher certainty when we predict a new value. The idea of splitting a node is that, we want the two resulting child node to contain less variation. In other words, we want each child node to be as "pure" as possible. Hence, the idea is to calculate this error criterion both before and after the split and see what cut-off point gives us the best reduction of error. Of course, all of these quantities will be calculated based on the sample version, instead of the truth. For example, if we use the Gini impurity to compare different splits, we use the following quantity for an **internal node** $\square$:

$$\text{score}(j, c) = \text{Gini}(\square) - \left(\frac{N_{\square_L}}{N_{\square}} \text{Gini}(\square_L) + \frac{N_{\square_R}}{N_{\square}} \text{Gini}(\square_R) \right).$$

Here, $\square_L$ (left child node) and $\square_R$ (right child node) denote the two child nodes resulted from a potential split on the jth variable at a cut-off point c, such that

$$\square_L = \{\mathbf{x} : \mathbf{x} \in \square, x_j \leq c\}$$

and

$$\square_R = \{\mathbf{x} : \mathbf{x} \in \square, x_j > c\}.$$

Then $N_{\square}$, $N_{\square_L}$, $N_{\square_R}$ are the number of observations in these nodes, respectively. The implication of this is quite intuitive: $\text{Gini}(\square)$ calculates the uncertainty of the entire node $\square$, while the second quantity is a summary of the uncertainty of the two potential child nodes. Hence a larger score indicates a better split, and we may choose the best index j and cut-off point c to proceed,

$$\arg\max_{j,c} \text{score}(j, c)$$

and then work on each child node separately using the same procedure.

Regression Trees

The basic procedure for a regression tree is pretty much the same as a classification tree, except that we will use a different way to evaluate how good a potential split is. Note that the variance is a simple quantity to describe the noise within a node, we can use

$$\text{score}(j, c) = \text{Var}(\square) - \left(\frac{N_{\square_L}}{N_\square} \text{Var}(\square_L) + \frac{N_{\square_R}}{N_\square} \text{Var}(\square_R) \right).$$

Tuning a Tree Model

Tree tuning is essentially about when to stop splitting. Or we could look at this reversely by first fitting a very large tree, then see if we could remove some branches of a tree to make it simpler without sacrificing much accuracy. One approach is called the **cost-complexity pruning**. This is another penalized framework that we use the accuracy as the loss function, and use the tree-size as the penalty part for complexity. Formally, if we have any tree model $\square$, consider this can be written as

$$\begin{aligned} C_\alpha(\square) &= \sum_{\text{all terminal nodes } t \text{ in } \square} N_t \cdot \text{Impurity}(t) + \alpha|\square| \\ &= C(\square) + \alpha|\square| \end{aligned}$$

Now, we can start with a very large tree, say, fitted until all pure terminal nodes. Call this tree as $\square_{\max}$. We can then exhaust all its sub-trees by pruning any branches, and calculate this $C(\cdot)$ function of the sub-tree. Then the tree that gives the smallest value will be our best tree.

But this can be computationally too expensive. Hence, one compromise, instead of trying all possible sub-trees, is to use the **weakest-link cutting**. This means that, we cut the branch (essentially a certain split) that displays the weakest banefit towards the $C(\cdot)$ function. The procedure is the following:

- Look at an internal node t of $\square_{\max}$, and denote the entire branch starting from t as $\square_t$
- Compare: remove the entire branch (collapse $\square_t$ into a single terminal node) vs. keep T_t. To do this, calculate

  $$\alpha \leq \frac{C(t) - C(\square_t)}{|T_t| - 1}$$

  Note that $|\square_t| - 1$ is the size difference between the two trees.
- Try all internal nodes t, and cut the branch t that has the smallest value on the right hand side. This gives the smallest α value to remove some branches. Then iterate the procedure based on this reduced tree.

Note that the α values will get larger as we move more branches. Hence this produces a solution path. Now this is very similar to the Lasso solution path idea, and we could use cross-validation to select the best tuning. By default, the `rpart` function uses a 10-fold cross-validation. This can be controlled using the `rpart.control()` function and specify the `xval` argument. For details, please see the documentation (https://cran.r-project.org/web/packages/rpart/rpart.pdf). The following plot using `plotcp()` in the `rpart` package gives a visualization of the relative cross-validation error. It also produces a horizontal line (the dotted line). The smallest tree under this line should be used. To show some details, `$cptable` shows the cross-validation results

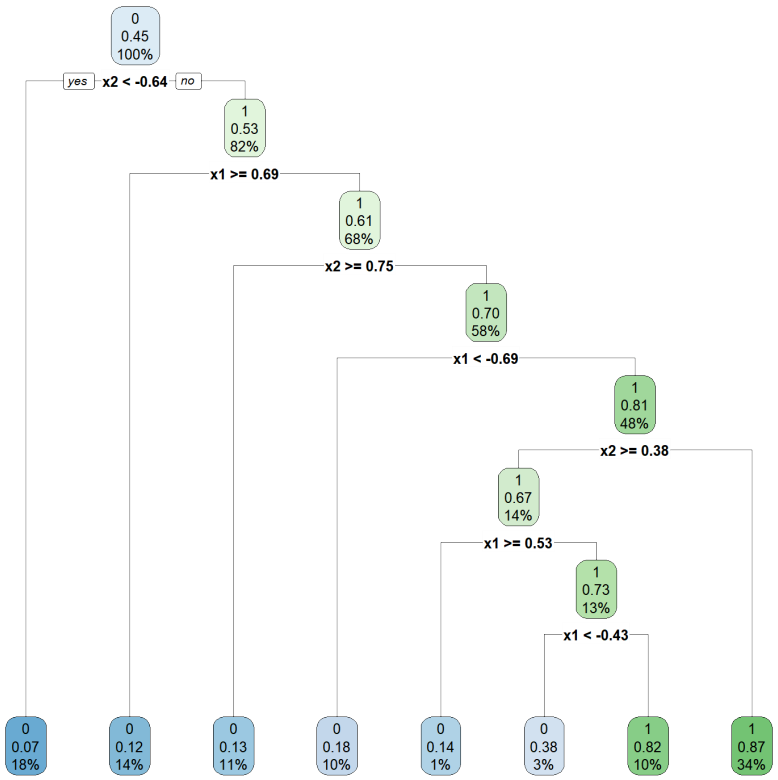
Hide

```
  rpart.fit$cptable
##           CP nsplit rel error    xerror       xstd
## 1 0.17040359      0 1.0000000 1.0000000 0.04984280
## 2 0.14798206      3 0.4843049 0.7264574 0.04692735
## 3 0.01121076      4 0.3363229 0.4484305 0.04010884
## 4 0.01000000      7 0.3004484 0.4035874 0.03852329
```

`xerror` shows the (relative) cross-validation error, and `xstd` shows the corresponding standard deviation across all folds. We then add 1-SD to the smallest cross-validation error, i.e., 0.4035874 + 0.03852329 = 0.4421107. The way that this is constructed is similar to the `lambda.1se` choice in `glmnet`. Then we find the CP value (α) for the tree with the largest cross-validation error less than 0.4421107, and use the `prune()` function to get the optimal tree for us. This turns out to select a model with 8 terminal nodes.

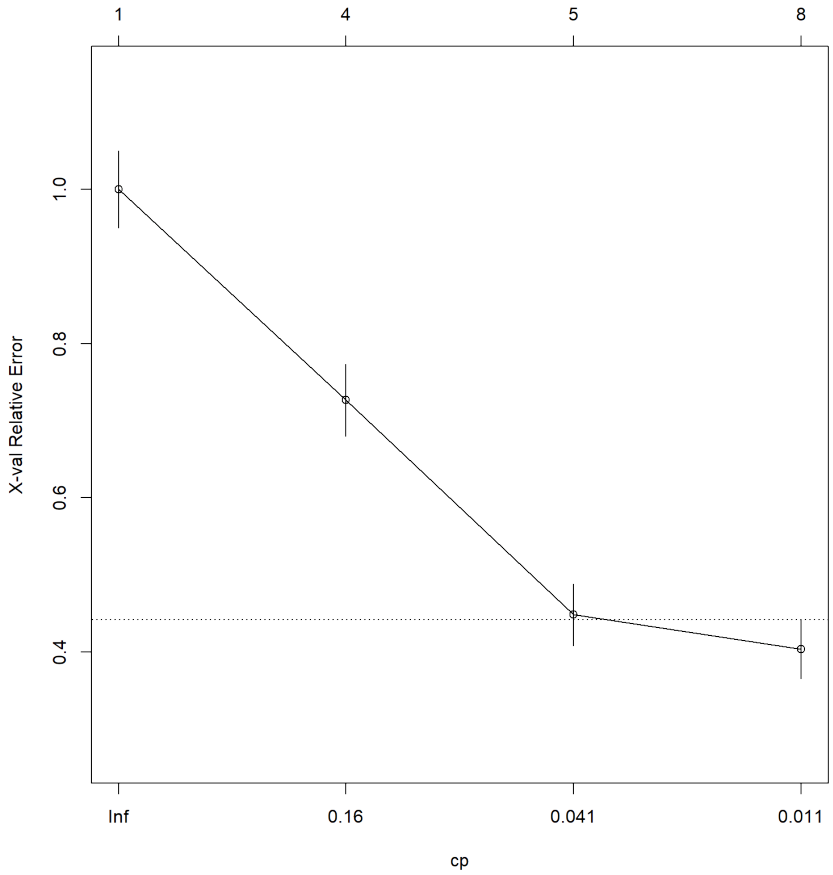
Hide

```
  # Get the middle value between the one below and above 0.4421107
  cptarg = sqrt(rpart.fit$cptable[4,1]*rpart.fit$cptable[3,1])
  cptarg
## [1] 0.01058809

  # prune the tree
  prunedtree = prune(rpart.fit,cp=cptarg)

  # we use a different function to display the result
  library(rpart.plot)
  rpart.plot(prunedtree)
```



Hide

```
  # and the tuning parameter
  plotcp(rpart.fit)
```



Hide

```
  printcp(rpart.fit)
##
## Classification tree:
## rpart(formula = as.factor(y) ~ x1 + x2, data = data.frame(x1,
##     x2, y))
##
## Variables actually used in tree construction:
## [1] x1 x2
##
## Root node error: 223/500 = 0.446
##
## n= 500
##
##         CP nsplit rel error  xerror     xstd
## 1 0.170404      0   1.00000 1.00000 0.049843
## 2 0.147982      3   0.48430 0.72646 0.046927
## 3 0.011211      4   0.33632 0.44843 0.040109
## 4 0.010000      7   0.30045 0.40359 0.038523
```

Prediction and Adaptive Kernel View

When we have a new target point $\mathbf{x}_0$ to predict, the basic strategy is to "drop it down the tree". This is simply starting from the root node and following the splitting rule to see which terminal node it ends up with. Note that a fitted tree will have a collection of terminal nodes, say, $\{\square_1, \square_2, \dots, \square_L\}$, where L is the number of terminal nodes (leaves). These terminal nodes are typically hyper-cubes defined by the tree construction process. Suppose $\mathbf{x}_0$ falls into terminal node $\square_l$, we can then calculate the average of training data outcomes (or probability) $\bar{y}_{\square_l}$ within this terminal node. Hence, the terminal node prediction is essentially a local averaging, defined by a zero-one kernel function in the form of

$$\bar{y}_{\square_l} \frac{\sum_{i=1}^{n} y_i \mathbf{1}\{\mathbf{x}_i \in \square_l\}}{\sum_{i=1}^{n} \mathbf{1}\{\mathbf{x}_i \in \square_l\}}$$

which is similar to the Nadaraya-Watson kernel regression form. However, the difference her is that the NW kernel is pre-fixed and do not depend on the observed data. Overall, a tree prediction at any target point can be viewed as checking which temrinal node $\mathbf{x}_0$ falls into, and then calculate the local average using the tree kernel.

$$\begin{aligned}\widehat{f}(\mathbf{x}_0) &= \sum_{l=1}^{L} \bar{y}_{\square_l} \mathbf{1}\{\mathbf{x}_0 \in \square_l\} \\ &= \sum_{l=1}^{L} \frac{\sum_{i=1}^{n} y_i \mathbf{1}\{\mathbf{x}_i \in \square_l\}}{\sum_{i=1}^{n} \mathbf{1}\{\mathbf{x}_i \in \square_l\}} \mathbf{1}\{\mathbf{x}_0 \in \square_l\}.\end{aligned}$$

However, tree kernels are able to adapt to the strcuture of the true model. To see this, let's perform a simulation using the `RLT` package. You need to install the package from GitHub, following this guild (https://teazrq.github.io/random-forests-tutorial/rlab/basics/packages.html). The current CRAN version is not able to complete this task.

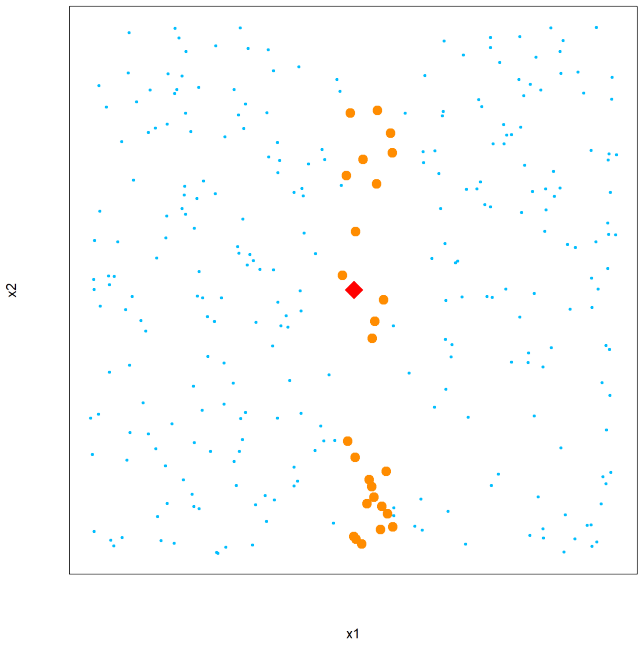
For a regular kernel estimator, the neighberhood points are defined using the distance to the target point. However, since the underlying true model only depends on the first variable, the tree model was able to adaptively use points that are close in the first dimension, but ignores the distance of the second dimension. This adaptiveness is able to combat with some dimensional issues kernel estimators have.

Hide

```
# you may need to install the devtools package first
library(devtools)

# install the RLT package from GitHub
# Rtools is required in windows
install_github("teazrq/RLT")
```

Hide

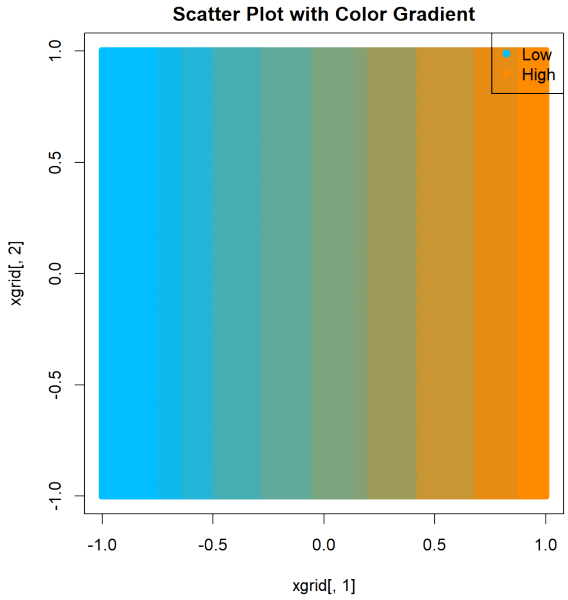
```
  library(RLT)
## RLT and Random Forests v4.2.5
## pre-release at github.com/teazrq/RLT

  # generate some data
  set.seed(2)
  n = 300
  x1 = runif(n, -1, 1)
  x2 = runif(n, -1, 1)
  y = 2*x1 + 0.2*rnorm(n)

  # fit one tree
  rf.fit = RLT(x = data.frame(x1, x2), y = y, model = "regression",
               ntrees = 1, resample.replace = FALSE, resample.prob = 1,
               mtry = 2, nmin = 40,
               param.control = list(resample.track = TRUE))

  # calculate kernel
  rf.kernel = forest.kernel(rf.fit, X1 = data.frame("x1" = 0, "x2" = 0),
                            X2 = data.frame(x1, x2), vs.train = TRUE)$Kernel

  # kernel weights
  plot(x1, x2, pch = 19, yaxt="n", xaxt = "n",
       cex = rf.kernel + 0.5,
       col = ifelse(rf.kernel != 0, "darkorange", "deepskyblue"))
  points(0, 0, col = "red", pch = 18, cex = 3)
```



However, the prediction of a tree model is non-smooth. The following plot shows different levels of predictions in this two dimensional domain.

Hide

```
  xgrid = expand.grid(x1 = seq(-1, 1, 0.01), x2 = seq(-1, 1, 0.01))

  grid.pred = predict(rf.fit, testx = xgrid)$Prediction

  # Create a color palette from blue to red
  palette <- colorRampPalette(c("deepskyblue", "darkorange"))(100)

# Create a mapping from z values to colors
  col_map <- palette[cut(grid.pred, breaks = 100)]

  # Create scatter plot
  plot(xgrid[, 1], xgrid[, 2], col = col_map, pch = 19, main = "Scatter Plot with C
olor Gradient")
   legend("topright", legend = c("Low", "High"), col = c("deepskyblue", "darkorang
e"), pch = 19)
```



However, we can also see that the model is not smooth. We will later learn the random forest model that try to address this issue.

Hide

```
  # generate some data
  set.seed(2)
  n = 300
  x1 = runif(n, -1, 1)
  x2 = runif(n, -1, 1)
  y = 2*x1 + x2 + 0.2*rnorm(n)
  xgrid = expand.grid(x1 = seq(-1, 1, 0.01), x2 = seq(-1, 1, 0.01))

  # fit one tree
  rf.fit = RLT(x = data.frame(x1, x2), y = y, model = "regression",
               ntrees = 1, resample.replace = FALSE, resample.prob = 1,
               mtry = 2, nmin = 30,
               param.control = list(resample.track = TRUE))
  # prediction
  grid.pred = predict(rf.fit, testx = xgrid)$Prediction

  # Create a color palette from blue to red
  palette <- colorRampPalette(c("deepskyblue", "darkorange"))(100)

# Create a mapping from z values to colors
  col_map <- palette[cut(grid.pred, breaks = 100)]

  # Create scatter plot
  plot(xgrid[, 1], xgrid[, 2], col = col_map, pch = 19, main = "Scatter Plot with C
olor Gradient")
   legend("topright", legend = c("Low", "High"), col = c("deepskyblue", "darkorang
e"), pch = 19)
```