Decision Trees

Reading

- Chapters 8.1, 8.2
  - James, Gareth, et al. An introduction to statistical learning. Vol. 112. New York: springer, 2013.
  - Available online: https://www.statlearning.com/

Overview

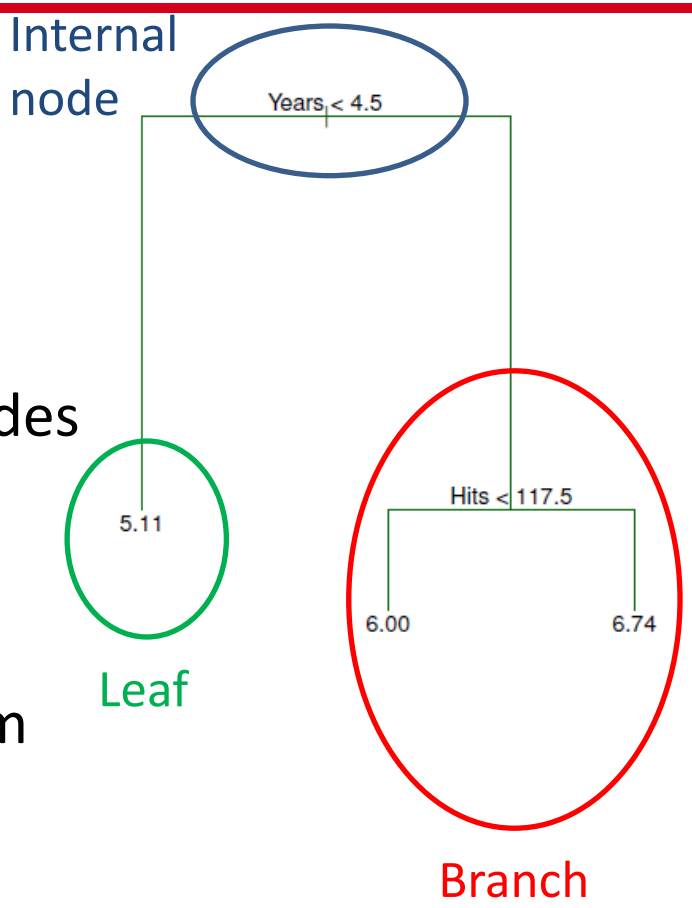
- Decision trees are a popular classification/regression model
- They are often preferred because they are intuitive and easy to interpret
  - Similar to a standard computer program
- Vanilla decision tree performance is often inferior to other methods
- Many improvements have been proposed such as random forests and so on
  - Random forests are on par with some of the best methods in classification, at a cost in interpretability

High-level description

- A decision tree is a predictive model based on if-cases
- Predict baseball players' salary (in log-scale) based on years and number of hits last year
- Very easy to interpret the tree's prediction
  - E.g., if a player has at least 4.5 years of experience and made less than 117.5 hits the previous year, their predicted salary is $10^6 = \$1M$
- Can split each branch arbitrarily for finer precision predictions
- This is a regression tree since it predicts continuous values
  - However, it can only output finitely many values, so the distinction with classification is blurry

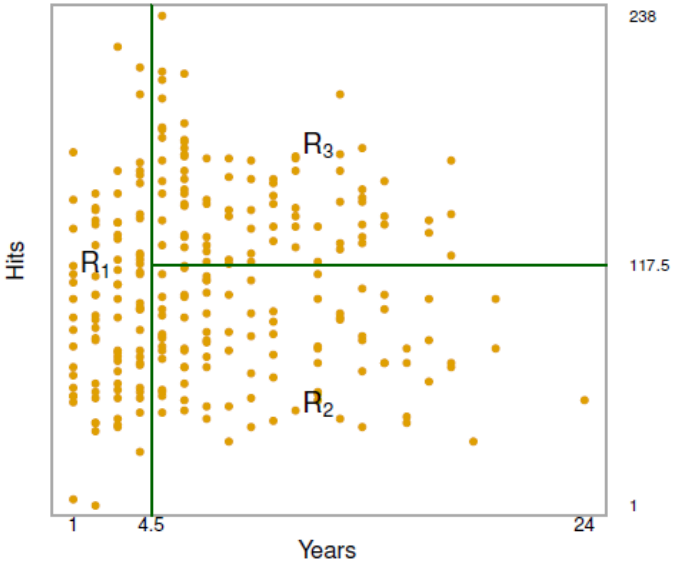
Elements of a decision tree

- Internal nodes
  - Where the predictor space is split
- Branches
  - Subsets of the tree that connect nodes
- Terminal nodes or leaves
  - Where outputs are produced
- Decisions are made from top to bottom by convention



Decision Tree Intuition

- The decision tree works by producing linear cuts in the feature space
  - For each region R_j, the prediction is the average over all points in R_j
- Can achieve arbitrary precision given enough cuts
  - A bit rudimentary for a small number of cuts



- Its main advantage is its interpretability and graph structure
  - Decision trees received increased attention with the recent push for interpretable AI

Training the Decision Tree

- Decision tree training is more an art than a science
  - This is true for many ML techniques in general
- Users need to make several decisions before even starting
  - How many splits to include?
  - Are the splits axis-aligned or arbitrary lines?
  - Which variable to split on first?
  - Some or all of these can be chosen algorithmically also

Training the Decision Tree, cont'd

- Suppose we want to have J regions: $R_1, \ldots, R_J$
  - How do we find the best regions?
    - Least squares!

$$\min_{R_1,\ldots,R_J} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}} \left(y_i - f(\boldsymbol{x}_i)\right)^2 =$$

$$\min_{R_1,\ldots,R_J} \sum_{j=1}^{J} \sum_{i:\boldsymbol{x}_i \in R_j} \left(y_i - f_j(\boldsymbol{x}_i)\right)^2$$

  - i.e., find regions to minimize the sum of squared errors
  - where $f_j(\boldsymbol{x})$ is the mean of all $y_i \in R_j$, call it $\hat{y}_{R_j}$
- What is the challenge with this approach?
  - There are exponentially many (in J and p) tree shapes
    - Unclear which tree shapes lead to better performance

Least Squares for Decision Trees

- Suppose first $J = 2$
- Need to pick a threshold t_d along some dimension d
  - Let x^d denote dimension d of input $\boldsymbol{x}$
  - Left branch is taken if $x^d < t_d$
  - Need to go through all dimensions and pick the best one
  - So far so good (linear in the number of dimensions)
- What if $J = 3$?
  - Need to pick two thresholds
    - But which one goes first?
    - Also, how do we arrange the tree – longer left or right branch?
    - Hard to say which shape will generalize better

Least Squares for Decision Trees, cont'd

- If we can't try all tree shapes, how do we grow the tree?
  - A greedy approach!
  - It's a standard approximation technique for combinatorial problems
    - Sometimes produces quite good (or even optimal) solutions
- Greedy means that we only choose the best next split without considering how it might affect future splits

Greedy Least Squares

- For 1st split, need to pick a threshold t_d along dimension d
  - That would create potential split regions

$$R_1(d, t_d) = \left\{\boldsymbol{x} \in \mathbb{R}^p \middle| x^d < t_d\right\} \text{ and } R_2(d, t_d) = \left\{\boldsymbol{x} \in \mathbb{R}^p \middle| x^d \geq t_d\right\}$$

  - Dataset is now split according to examples in R_1 and R_2

$$\mathcal{D}_1 = \{(\boldsymbol{x}_i, y_i) \in \mathcal{D} | \boldsymbol{x}_i \in R_1\}$$
$$\mathcal{D}_2 = \{(\boldsymbol{x}_i, y_i) \in \mathcal{D} | \boldsymbol{x}_i \in R_2\}$$

  - where $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$
  - What is the prediction in each region?

$$\hat{y}_{R_1} = \frac{1}{|\mathcal{D}_1|} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_1} y_i$$

$$\hat{y}_{R_2} = \frac{1}{|\mathcal{D}_2|} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_2} y_i$$

Greedy Least Squares, cont'd

- For 1st split, need to pick a threshold t_d along dimension d
  - That would create potential split regions

    $R_1(d, t_d) = \{\boldsymbol{x} \in \mathbb{R}^p | x^d < t_d\}$ and $R_2(d, t_d) = \{\boldsymbol{x} \in \mathbb{R}^p | x^d \geq t_d\}$
  - What is the prediction in each region?

$$\hat{y}_{R_1} = \frac{1}{|\mathcal{D}_1|} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_1} y_i$$

$$\hat{y}_{R_2} = \frac{1}{|\mathcal{D}_2|} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_2} y_i$$

  - What is the total squared error in each region?

$$e_{R_1} = \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_1} \left(y_i - \hat{y}_{R_1}\right)^2$$

$$e_{R_2} = \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_2} \left(y_i - \hat{y}_{R_2}\right)^2$$

Greedy Least Squares, cont'd

- For 1st split, need to pick a threshold t_d along dimension d
  - That would create potential split regions

$$R_1(d, t_d) = \left\{ \boldsymbol{x} \in \mathbb{R}^p \middle| x^d < t_d \right\} \text{ and } R_2(d, t_d) = \left\{ \boldsymbol{x} \in \mathbb{R}^p \middle| x^d \geq t_d \right\}$$

- Need to pick d and t_d to minimize mean squared error:

$$MSE(d, t_d, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \left(\sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d < t_d}} \left(y_i - \hat{y}_{R_1} \right)^2 + \sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d \geq t_d}} \left(y_i - \hat{y}_{R_2} \right)^2 \right)$$

  - As usual, we'll drop the $\frac{1}{|\mathcal{D}|}$ factor since it doesn't affect minimum (but will keep abbreviation MSE for consistency)

$$MSE(d, t_d, \mathcal{D}) = \sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d < t_d}} \left(y_i - \hat{y}_{R_1} \right)^2 + \sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d \geq t_d}} \left(y_i - \hat{y}_{R_2} \right)^2$$

Greedy Least Squares, cont'd

- For 1st split, need to pick a threshold t_d along dimension d
  - That would create potential split regions

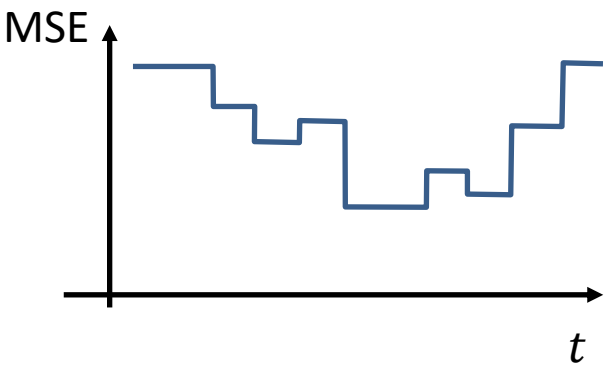
$$R_1(d, t_d) = \left\{ \boldsymbol{x} \in \mathbb{R}^p \middle| x^d < t_d \right\} \text{ and } R_2(d, t_d) = \left\{ \boldsymbol{x} \in \mathbb{R}^p \middle| x^d \geq t_d \right\}$$

- Need to pick d and t_d to minimize mean squared error:

$$MSE(d, t_d, \mathcal{D}) = \sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d < t_d}} \left(y_i - \hat{y}_{R_1} \right)^2 + \sum_{\substack{(\boldsymbol{x}_i, y_i) \in \mathcal{D} \\ x_i^d \geq t_d}} \left(y_i - \hat{y}_{R_2} \right)^2$$
$$= e_{R_1} + e_{R_2}$$

- Iterate through all p dimensions (recall $\boldsymbol{x}_i \in \mathbb{R}^p$)
  - For each dimension d, find threshold t_d that minimizes $MSE(d, t_d, \mathcal{D})$ on the training data (how?)

Greedy Least Squares, cont'd



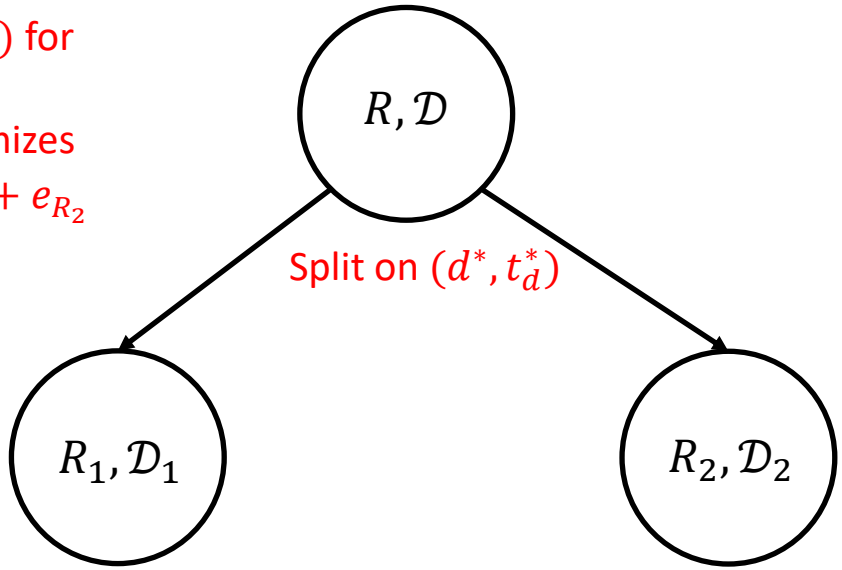
- MSE may not be convex in t, so we can't just set the derivative to 0
- But MSE is piecewise-constant on the training set
  - Why?
    - Because the prediction per region is only changed if an example is added or removed
    - Small threshold changes may not change this
- One can do an exhaustive search over the range of t
  - Set a small enough step size and step through the range of t
  - Pick the t^* that results on lowest MSE
- Alternatively, can sort all examples along dimension d
  - Increment threshold to include, e.g., 5%, 10%,… of data

Greedy Least Squares, cont'd

- Iterate through all p dimensions
  - For each dimension d, find threshold t_d that minimizes $MSE(d, t_d, \mathcal{D})$ on the training data
  - Finally, pick the combination (d, t_d) that minimizes $MSE(d, t_d, \mathcal{D})$
  - We have now created regions R_1 and R_2
- To create future regions, we split R_1 or R_2 in the same way
  - Terminate when we have J regions (or too few data points per region)

Algorithm Illustration

1. Compute $MSE(d, t_d, \mathcal{D})$ for each (d, t_d) pair
2. Find (d^*, t_d^*) that minimizes $MSE(d^*, t_d^*, \mathcal{D}) = e_{R_1} + e_{R_2}$



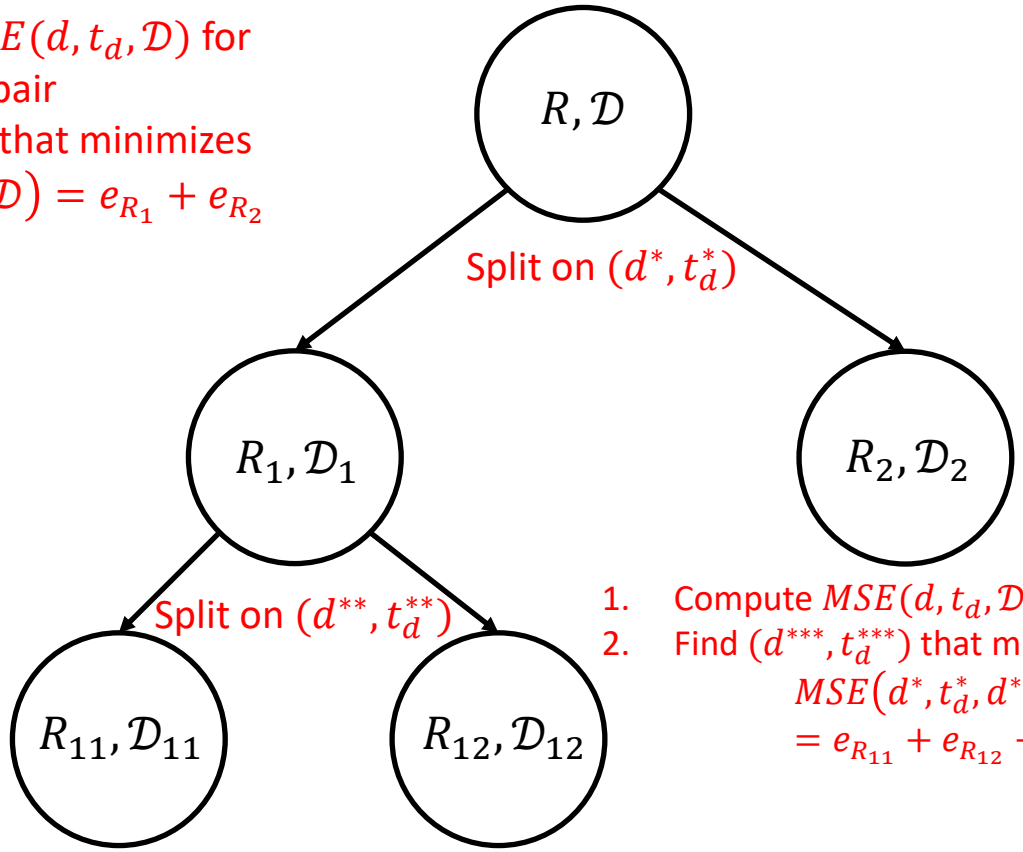
1. Compute $MSE(d, t_d, \mathcal{D}_1)$ for each (d, t_d) pair
2. Find (d^{**}, t_d^{**}) that minimizes $MSE(d^*, t_d^*, d^{**}, t_d^{**}, \mathcal{D}) = e_{R_2} + e_{R_{11}} + e_{R_{12}}$

1. Compute $MSE(d, t_d, \mathcal{D}_2)$ for each (d, t_d) pair
2. Find (d^{**}, t_d^{**}) that minimizes $MSE(d^*, t_d, d^{**}, t_d^{**}, \mathcal{D}) = e_{R_1} + e_{R_{21}} + e_{R_{22}}$

Suppose splitting R_1 results in lower loss

Algorithm Illustration

1. Compute $MSE(d, t_d, \mathcal{D})$ for each (d, t_d) pair
2. Find (d^*, t_d^*) that minimizes $MSE(d^*, t_d^*, \mathcal{D}) = e_{R_1} + e_{R_2}$



1. Compute $MSE(d, t_d, \mathcal{D}_2)$ for each (d, t_d) pair
2. Find (d^{***}, t_d^{***}) that minimizes
$MSE(d^*, t_d^*, d^{**}, t_d^{**}, d^{***}, t_d^{***}, \mathcal{D})$
$= e_{R_{11}} + e_{R_{12}} + e_{R_{21}} + e_{R_{22}}$

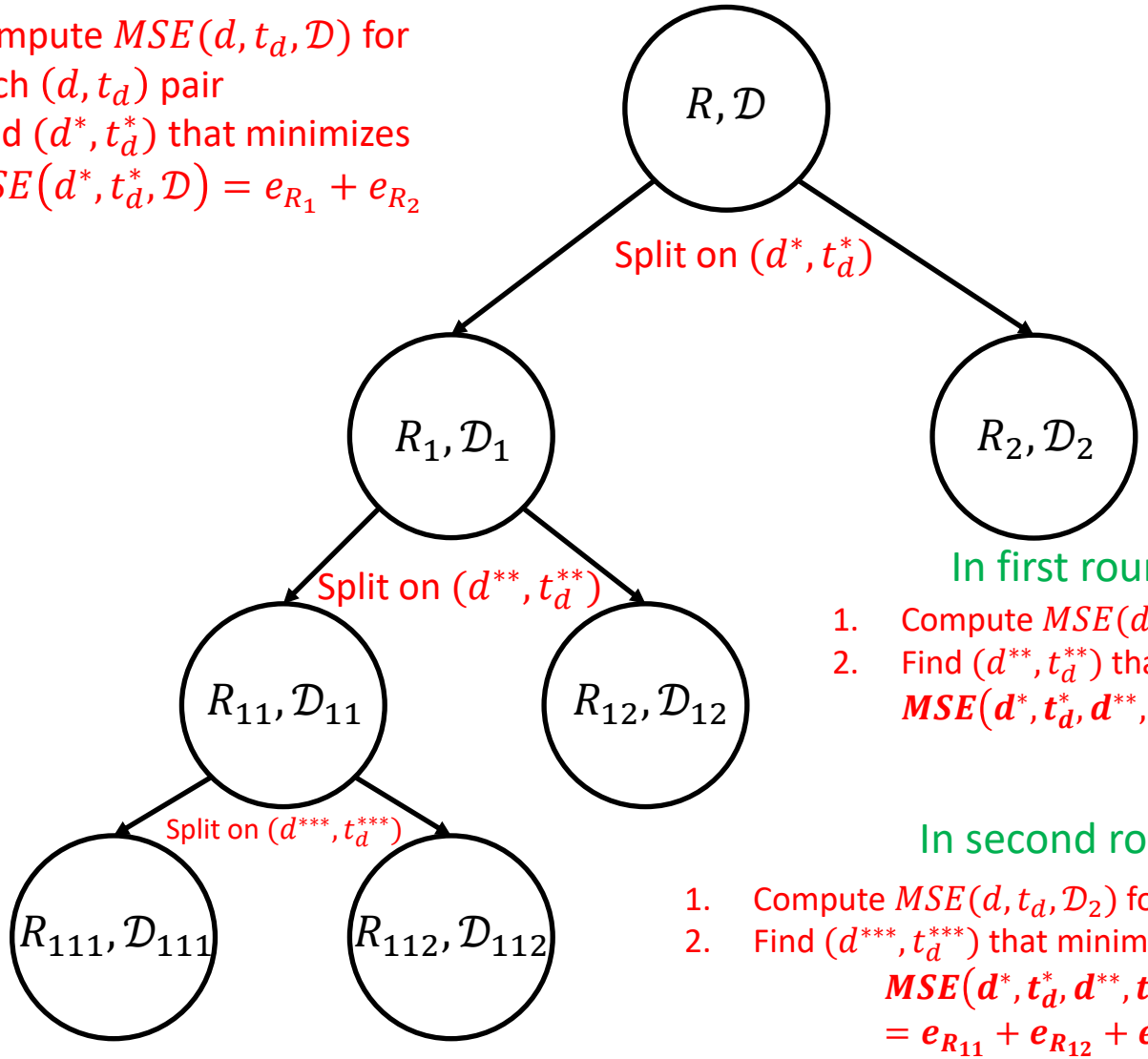
1. Compute $MSE(d, t_d, \mathcal{D}_{11})$ for each (d, t_d) pair
2. Find (d^{***}, t_d^{***}) that minimizes
$MSE(d^*, t_d^*, d^{**}, t_d^{**}, d^{***}, t_d^{***}, \mathcal{D})$
$= e_{R_2} + e_{R_{12}} + e_{R_{111}} + e_{R_{112}}$

1. Compute $MSE(d, t_d, \mathcal{D}_{12})$ for each (d, t_d) pair
2. Find (d^{***}, t_d^{***}) that minimizes
$MSE(d^*, t_d^*, d^{**}, t_d^{**}, d^{***}, t_d^{***}, \mathcal{D})$
$= e_{R_2} + e_{R_{11}} + e_{R_{121}} + e_{R_{122}}$

Suppose splitting R_{11} results in lowest loss

Algorithm Illustration

1. Compute $MSE(d, t_d, \mathcal{D})$ for each (d, t_d) pair
2. Find (d^*, t_d^*) that minimizes $MSE(d^*, t_d^*, \mathcal{D}) = e_{R_1} + e_{R_2}$



In first round loss was

1. Compute $MSE(d, t_d, \mathcal{D}_2)$ for each (d, t_d) pair
2. Find (d^{**}, t_d^{**}) that minimizes $\boldsymbol{MSE(d^*, t_d^*, d^{**}, t_d^{**}, \mathcal{D}) = e_{R_1} + e_{R_{21}} + e_{R_{22}}}$

In second round loss was

1. Compute $MSE(d, t_d, \mathcal{D}_2)$ for each (d, t_d) pair
2. Find (d^{***}, t_d^{***}) that minimizes

$$\boldsymbol{MSE(d^*, t_d^*, d^{**}, t_d^{**}, d^{***}, t_d^{***}, \mathcal{D})}$$
$$\boldsymbol{= e_{R_{11}} + e_{R_{12}} + e_{R_{21}} + e_{R_{22}}}$$

Do we need to recalculate each time?

Loss Improvement

- Loss calculation for R_2 was first

$$e_{R_1} + e_{R_{21}} + e_{R_{22}}$$

- and then

$$e_{R_{11}} + e_{R_{12}} + e_{R_{21}} + e_{R_{22}}$$

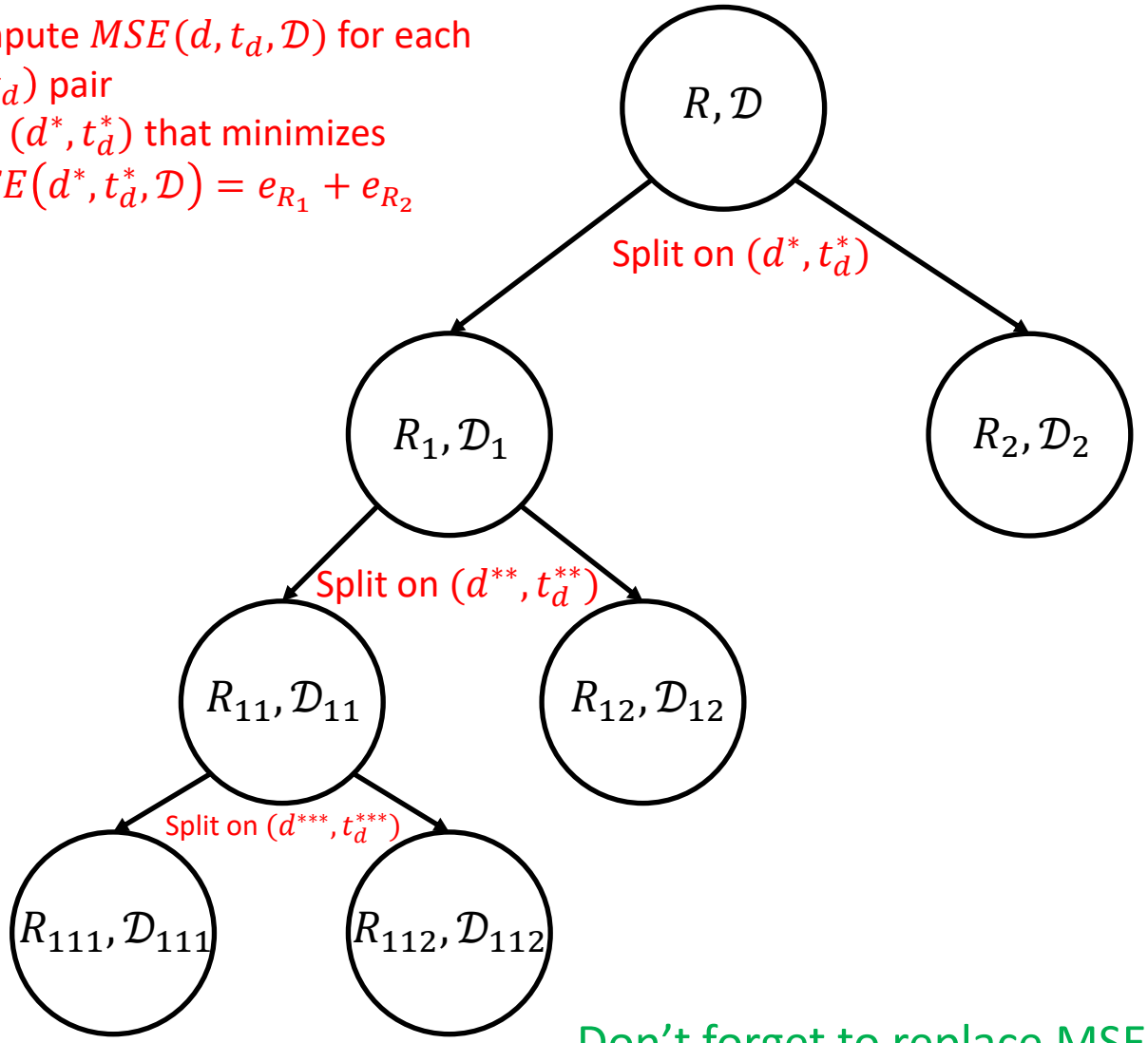
- Notice that splitting on R_2 does not affect the rest of the loss
  - After a split, the node's contribution to the total loss changes from e_{R_2} to $e_{R_{21}} + e_{R_{22}}$
- The loss improvement associated with R_2 is then

$$e_{R_2} - (e_{R_{21}} + e_{R_{22}})$$

- Needs to be calculated once (when node is created)
- Then always split on node with highest loss improvement

Algorithm Illustration

1. Compute $MSE(d, t_d, \mathcal{D})$ for each (d, t_d) pair
2. Find (d^*, t_d^*) that minimizes $MSE(d^*, t_d^*, \mathcal{D}) = e_{R_1} + e_{R_2}$



Don't forget to replace MSE with 0-1 loss in the case of classification

Classification Trees

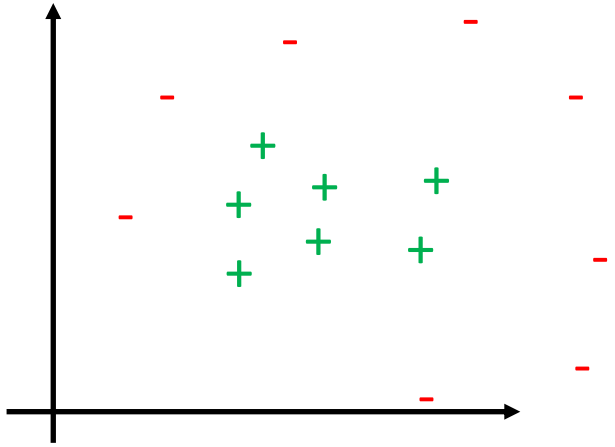
- Very similar to regression trees
- Instead of outputting the average label per region, they output the majority class
- You can use standard classification losses
  - E.g., 0-1 loss (0/1 for correct/wrong prediction, respectively)
  - Other losses are possible as well

Toy Training Example

- We have two classes and the training data is

$((2,2),+),((2,2.5),+),((2.2,2.8),+),((2.5,2.2),+),((2.52,2.53),+),$
$((3,2,2.1),+),((3.1,2.6),+)$

$((1,2.4),-),((1.5,3.5),-),((2.15,3.8),-),((3,0.1),-),((3.3,4),-),$
$((3.8,3.49),-),((3.8,0.5),-),((3.9,2.05),-)$



Toy Training Example, root

- We have two classes and the training data is

$((2,2),+),((2,2.5),+),((2.2,2.8),+),((2.5,2.2),+),((2.52,2.53),+),$
$((3,2,2.1),+),((3.1,2.6),+)$

$((1,2.4),-),((1.5,3.5),-),((2.15,3.8),-),((3,0.1),-),((3.3,4),-),$
$((3.8,3.49),-),((3.8,0.5),-),((3.9,2.05),-)$

Positive examples: 7

$((2,2),+),$
$((2,2.5),+),((2.2,2.8),+),$
$((2.5,2.2),+),((2.52,2.53),+),$
$((3,2,2.1),+),((3.1,2.6),+)$

Negative examples: 8

$((1,2.4),-),((1.5,3.5),-),$
$((2.15,3.8),-),((3,0.1),-),$
$((3.3,4),-),((3.8,3.49),-),$
$((3.8,0.5),-),((3.9,2.05),-)$

Loss: 7 (all positive examples are classified incorrectly)

Toy Training Example, split along x axis

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

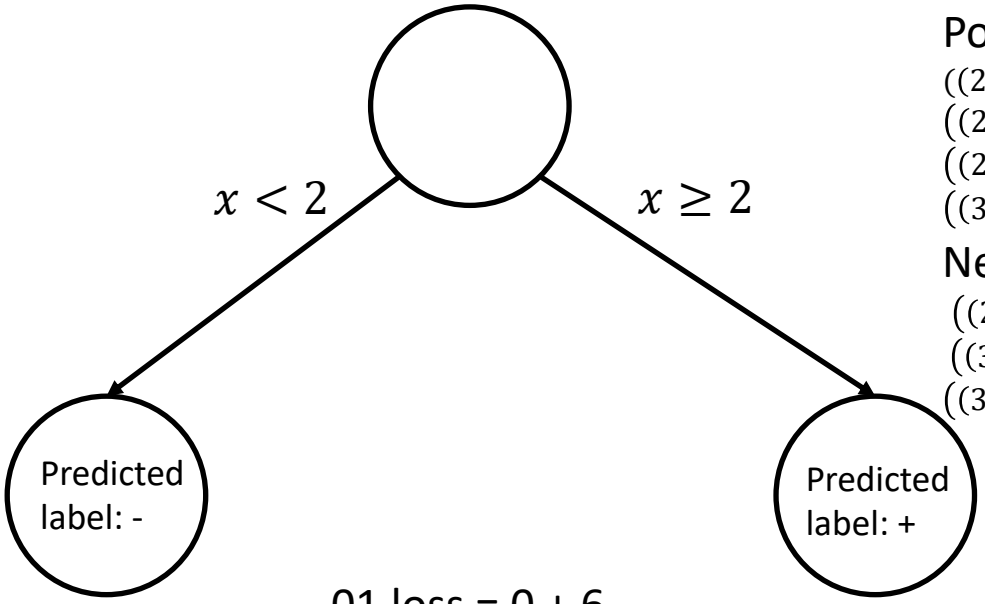
Predicted label: -

Negative examples: 8

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

X axis ranges from 1 to 3.9. With a step size of 0.1, you will have 29 thresholds to try.

Let's try $t = 2$



Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 6

$((2.15,3.8),-)$, $((3,0.1),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

Negative examples: 2

$((1,2.4),-)$, $((1.5,3.5),-)$

01 loss = 0 + 6

Toy Training Example, split along x axis

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

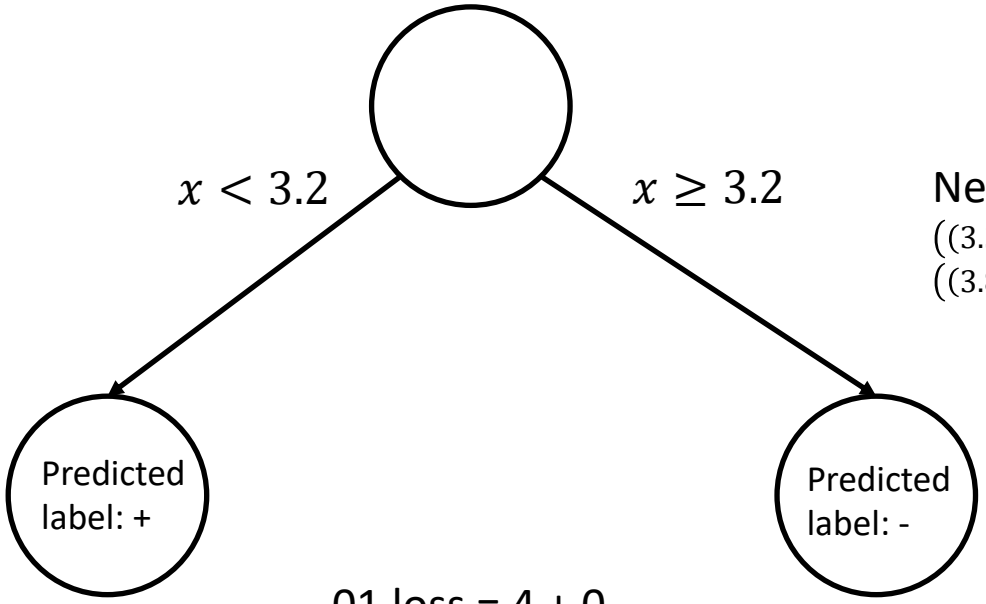
Predicted label: -

Negative examples: 8

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

X axis ranges from 1 to 3.9. With a step size of 0.1, you will have 29 thresholds to try.

Let's try $t = 3.2$



Negative examples: 4

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 4

$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

01 loss = 4 + 0

- Best threshold along x axis is 3.2, with a loss of 4!

Toy Training Example, split along y axis

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

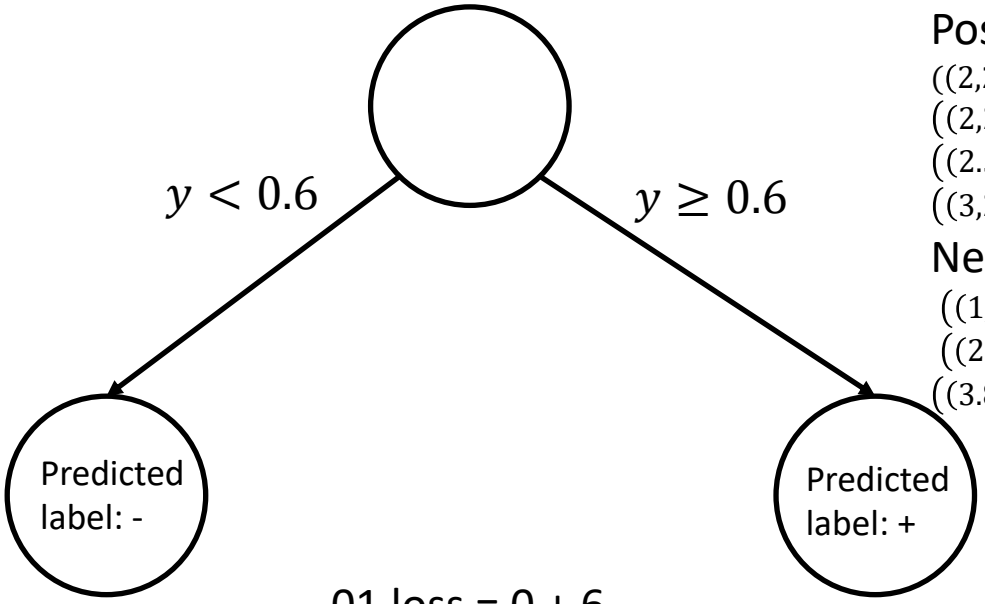
Predicted label: -

Negative examples: 8

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

Y axis ranges from 0.1 to 3.8. With a step size of 0.1, you will have 37 thresholds to try.

Let's try $t = 0.6$



Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 6

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3.3,4),-)$,
$((3.8,3.49),-)$, $((3.9,2.05),-)$

Negative examples: 2

$((3,0.1),-)$, $((3.8,0.5),-)$

01 loss = 0 + 6

Toy Training Example, split along y axis

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

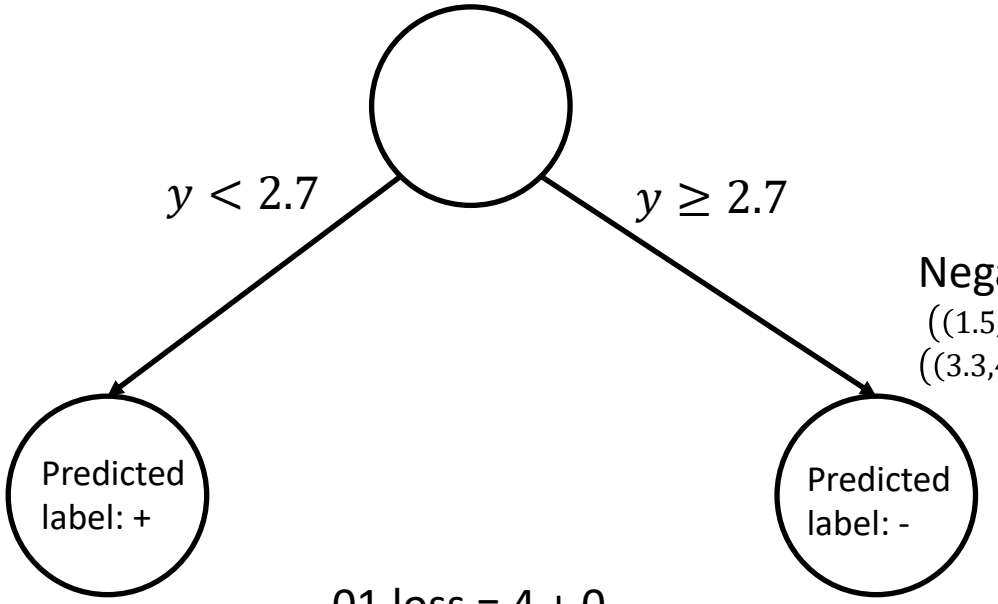
Predicted label: -

Negative examples: 8

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

Y axis ranges from 0.1 to 3.8. With a step size of 0.1, you will have 37 thresholds to try.

Let's try $t = 2.7$



Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 4

$((1,2.4),-)$, $((3,0.1),-)$,
$((3.8,0.5),-)$, $((3.9,2.05),-)$

Negative examples: 4

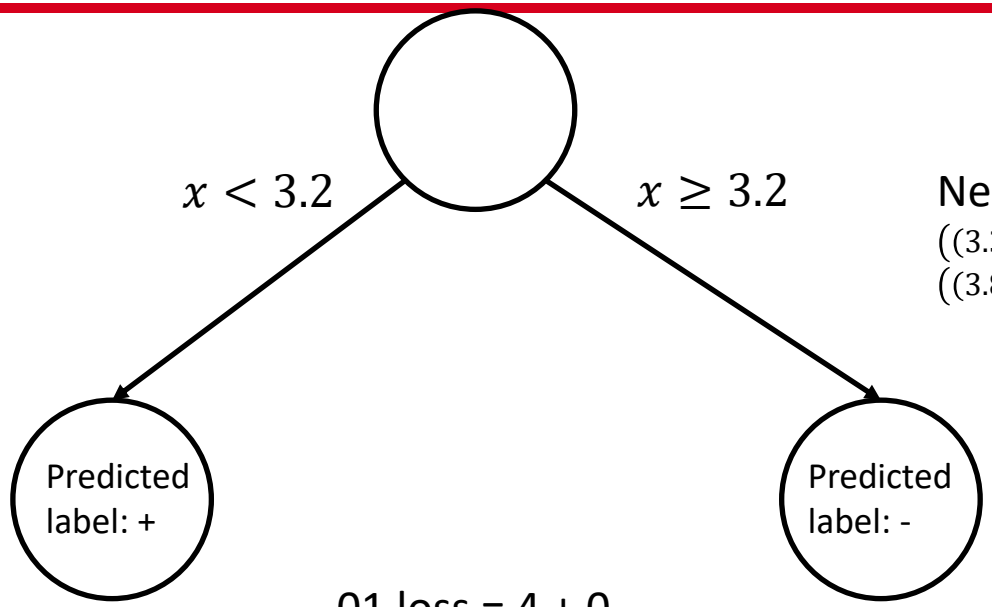
$((1.5,3.5),-)$, $((2.15,3.8),-)$,
$((3.3,4),-)$, $((3.8,3.49),-)$,

01 loss = 4 + 0

- Best threshold along y axis is 2.7, with a loss of 4!

- Split along x axis

Toy Training Example, new tree



Negative examples: 4
$((1,2.4),-),((1.5,3.5),-),$
$((2.15,3.8),-),((3,0.1),-)$

Positive examples: 7
$((2,2),+),$
$((2,2.5),+),((2.2,2.8),+),$
$((2.5,2.2),+),((2.52,2.53),+),$
$((3,2,2.1),+),((3.1,2.6),+)$

Negative examples: 4
$((3.3,4),-),((3.8,3.49),-),$
$((3.8,0.5),-),((3.9,2.05),-)$

01 loss = 4 + 0

Next split

- Right leaf is already pure, so nothing to improve
- Consider left leaf only

Toy Training Example, split left leaf along x axis

Positive examples: 7

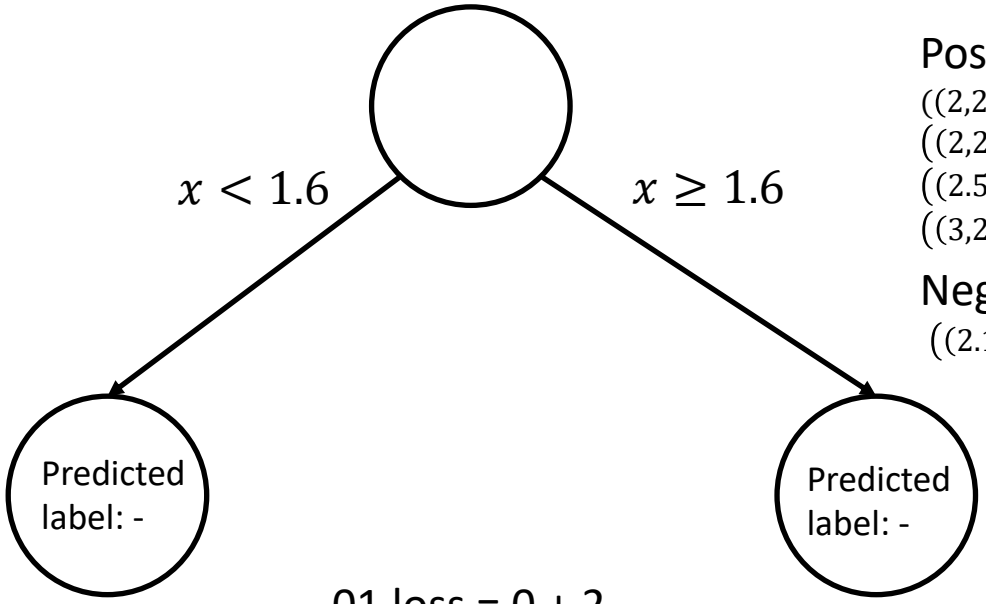
$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 4

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$

X axis ranges from 1 to 3.1. With a step size of 0.1, you will have 21 thresholds to try.

Let's try $t = 1.6$



Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

Negative examples: 2

$((2.15,3.8),-)$, $((3,0.1),-)$

Negative examples: 2

$((1,2.4),-)$, $((1.5,3.5),-)$

01 loss = 0 + 2

Toy Training Example, split left leaf along y axis

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

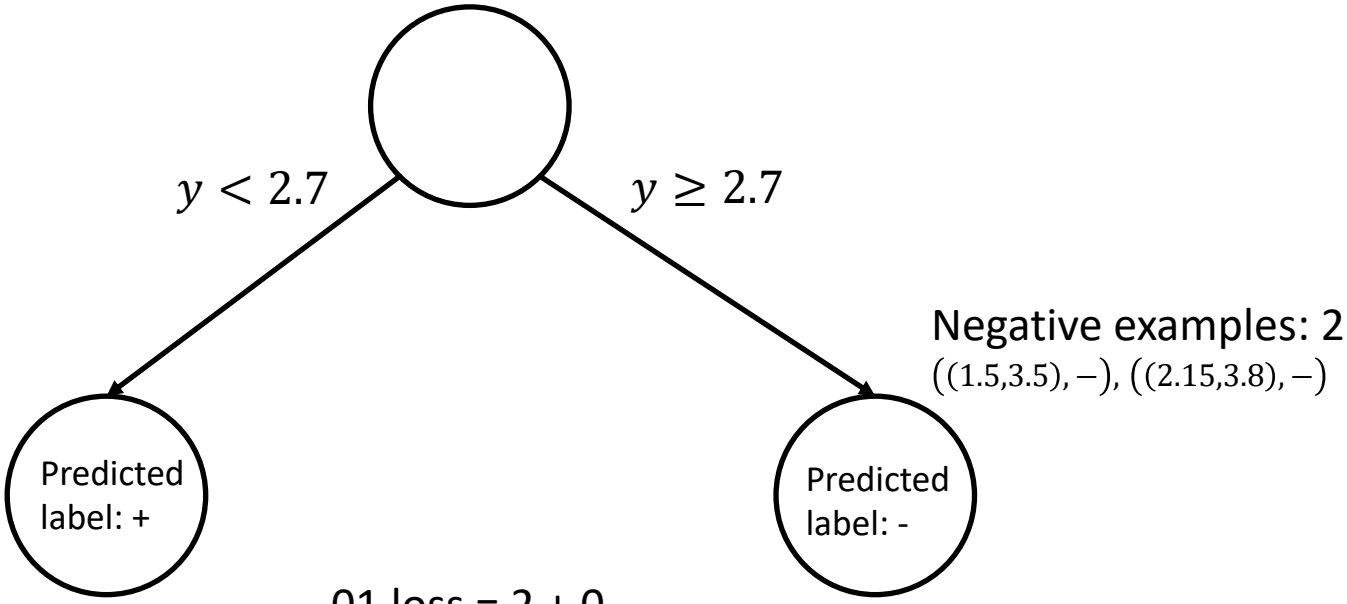
Predicted label: -

Negative examples: 4

$((1,2.4),-)$, $((1.5,3.5),-)$,
$((2.15,3.8),-)$, $((3,0.1),-)$

Y axis ranges from 0.1 to 3.8. With a step size of 0.1, you will have 38 thresholds to try.

Let's try $t = 2.7$



Negative examples: 2

$((1,2.4),-)$, $((3,0.1),-)$

Positive examples: 7

$((2,2),+)$,
$((2,2.5),+)$, $((2.2,2.8),+)$,
$((2.5,2.2),+)$, $((2.52,2.53),+)$,
$((3,2,2.1),+)$, $((3.1,2.6),+)$

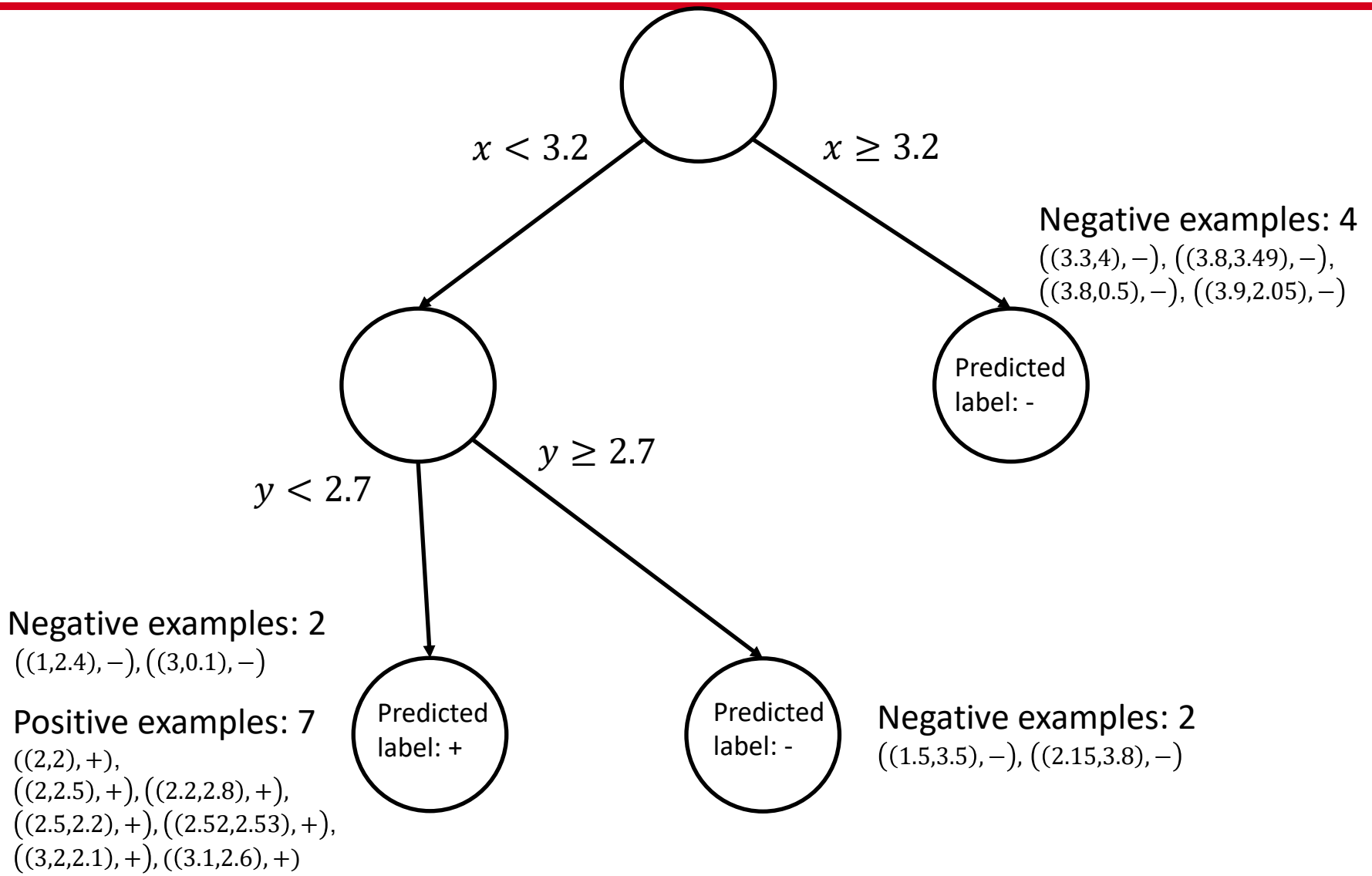
Negative examples: 2

$((1.5,3.5),-)$, $((2.15,3.8),-)$

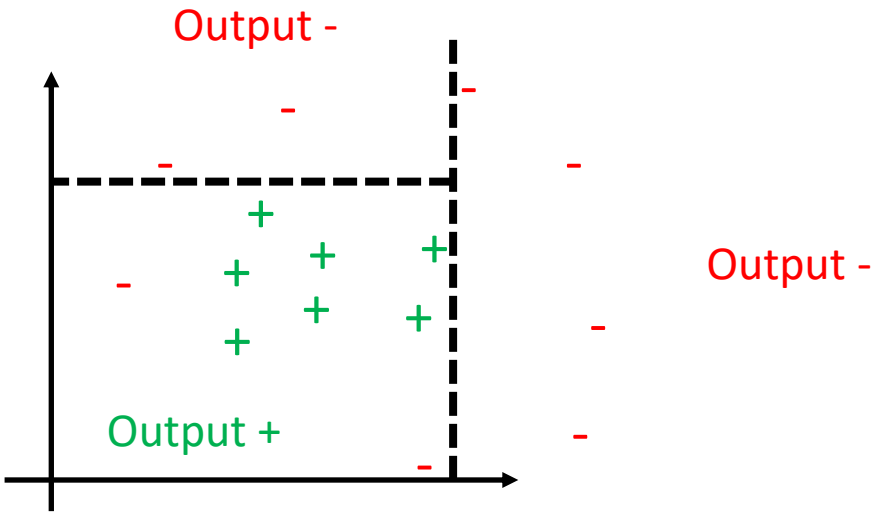
01 loss = 2 + 0

- Split along y axis

Current Tree



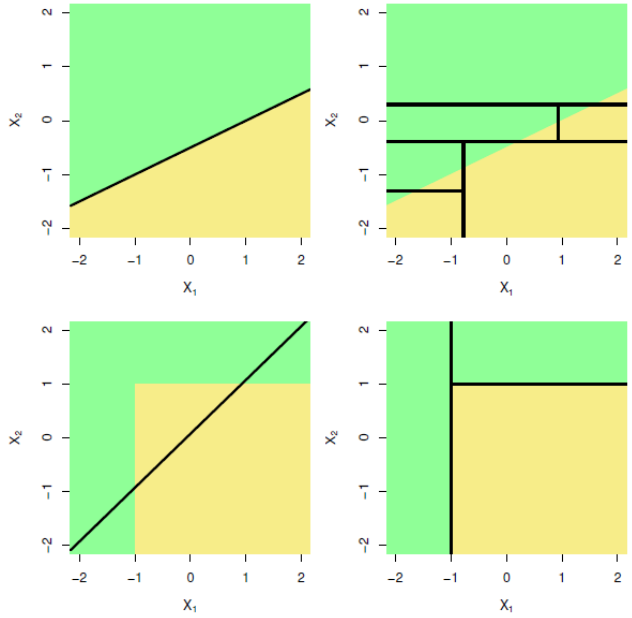
Current Splits



- Can continue building the tree for perfect training accuracy

Trees vs. Linear Regression

- Linear regression is a well understood and robust algorithm
  - However, does not work very well when data is highly nonlinear
- Trees can capture all sorts of non-linearities
  - But very susceptible to overfitting



Tree Pruning

- If we pick J to be too large, the decision tree might become very complex
  - In the extreme case of $J = N$, the tree becomes a table that just remembers all training data
  - What is the issue with that?
  - Overfitting!
- We want the tree to capture patterns in the data without being too sensitive to noise in the training data
  - We will talk about overfitting in more detail later
- How do we prune?
  - Regularization!

Cost Complexity Pruning using Regularization

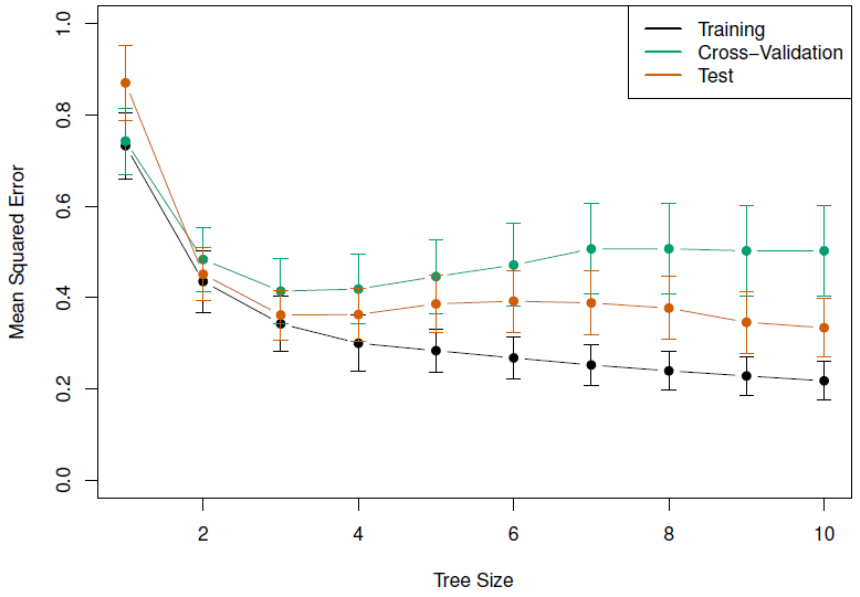
- Suppose we want a subtree with low MSE and few leaves
- A principled way to do that is to add a term to the loss function
- Suppose the original tree is T_0
- Let α be a small positive number
- Then the new loss is

$$\sum_{j=1}^{|T|} \sum_{i:x_i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2 + \alpha |T|$$

  - i.e., find a $T \subset T_0$ that minimizes the above loss
  - where $|T|$ denotes the number of leaves in T
- This technique is called regularization (more later)
- If interested, see slides at the back of deck for more detail

Effect of Regularization

- Notice that training error keeps decreasing for larger trees
  - We can bring it down to 0 with a very large tree
- However, test error starts increasing after some point
  - Overfitting!
  - A very common phenomenon (more later)
- Cross validation produces a better estimate of test error
  - Will discuss more later



Bagging and Random Forests

- Decision trees are nice and intuitive but they produce worse predictions than other methods in general
- Many improvements have been proposed over the years
- Bagging: train multiple trees by creating multiple datasets using sampling with replacement
  - Trees might be correlated
- Random forests: decrease correlation by only training on a subset of features per split
  - Forces trees to have different structure

Summary

- Decision trees are a nice graphical and easy-to-interpret model
  - Unfortunately, they are inferior to other classical methods
    - Splits are too simplistic, focusing on one feature at a time
    - Training on high-dimensional data is very slow
- Can be used with high-level features of the data
  - E.g., brightness, symmetry, etc.
- Bagging and random forests provide a significant boost in performance
- Random forests became quite popular recently with the latest push for interpretability

Cost Complexity Pruning

- Suppose we want to pick a subtree with the property that it has low MSE and few leaves
- A principled way to do that is to add a term to the loss function
- Suppose the original tree is T_0
- Let α be a small positive number
- Then the new loss is

$$\sum_{j=1}^{|T|} \sum_{i:x_i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2 + \alpha |T|$$

  - i.e., find a $T \subset T_0$ that minimizes the above loss
  - where $|T|$ denotes the number of leaves in T

Cost Complexity Pruning, cont'd

- Then the new loss is

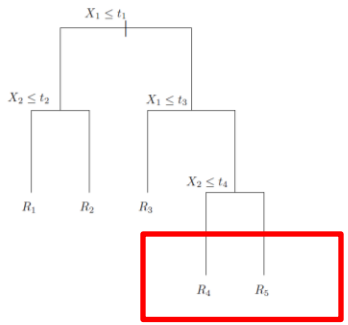
$$\sum_{j=1}^{|T|} \sum_{i:x_i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2 + \alpha|T|$$

- Note that when $\alpha = 0$, the above loss becomes MSE
- As we increase α we penalize larger trees
  - As $\alpha \to \infty$, the optimal tree converges to a one-leaf tree
  - For intermediate α, the loss balances between trees with low MSE and few leaves
- This technique is called regularization
  - Will talk more about regularization later
- If interested, see slides at the back of deck for more detail

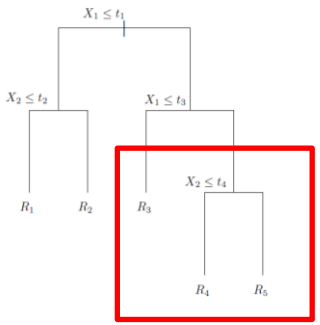
Cost Complexity Pruning, cont'd

- How do we pick the optimal T for a given α?
- Keep in mind $MSE(T) > MSE(T_0)$ for any $T \subset T_0$
  - Why?
  - By construction, when we refine a tree, we reduce the MSE
- We can recursively construct the optimal T from T_0
  - Start from the bottom of each branch
  - Compute the current loss vs. the loss if leaves are merged
  - If merging reduces loss, then merge; otherwise, move up

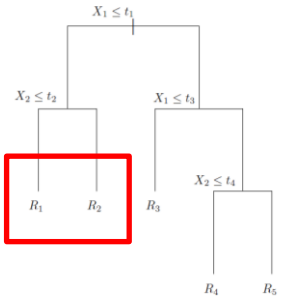
1



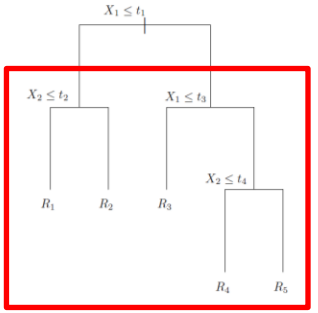
2



3



4



Cross Validation

- How do we pick α?
  - α is called a hyper-parameter: a parameter we pick at design-time that is not optimized during training proper
- Cross validation!
- Classic cross validation is used to estimate a model's test error
  - Split the data randomly into 90% training and 10% testing
  - Train on the training data and record the test accuracy
  - Repeat multiple (e.g., 10) times
  - Take the average test error over all runs
  - A better estimate of generalization error than a single split
- Try different values for α and pick the one that results in lowest cross-validation error

Cross Validation, cont'd

- Cross validation is especially useful for small datasets when it is hard to get a good test error estimate
- Not widely used today since datasets are quite large
  - Performing well on modern test sets is usually a good sign
  - Re-splitting the data and retraining can be quite costly
- Cross validation is an important tool when it comes to generalization
  - We'll talk more about generalization next