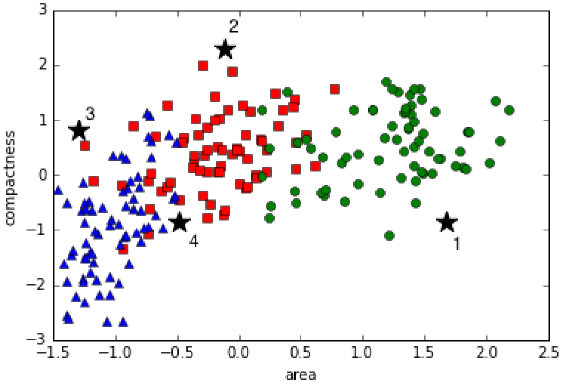
Lecture 11 – KNN and Decision Trees

ECE 597ML-697ML

Mario Parente

Nearest Neighbor Classification

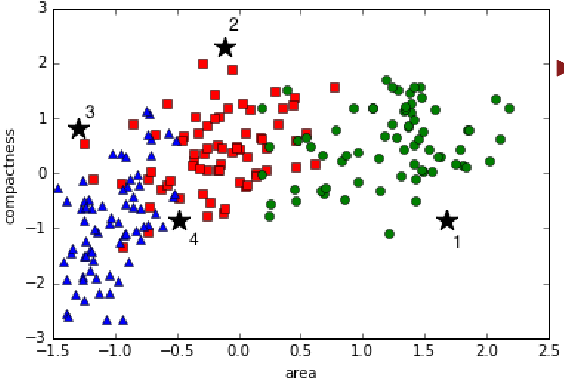
Seed classification by area and compactness



- What should we predict for unlabeled test points (stars)?

Nearest Neighbor Classification

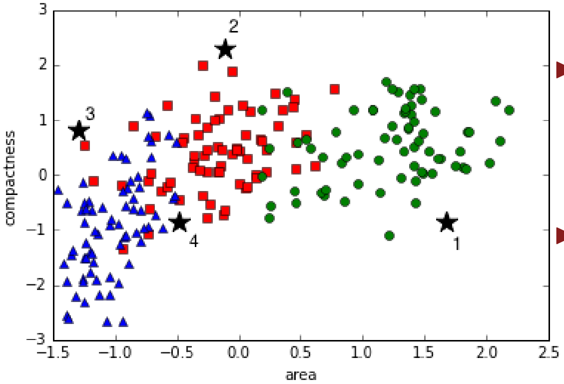
Seed classification by area and compactness



- What should we predict for unlabeled test points (stars)?
- Nearest neighbor classification: predict label of nearest training example

Nearest Neighbor Classification

Seed classification by area and compactness



- What should we predict for unlabeled test points (stars)?
- Nearest neighbor classification: predict label of nearest training example
- k-nearest neighbor: predict consensus of k nearest training examples

k-Nearest Neighbor Classification

- **Training**: store the training data (trivial!)

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}, \ldots (\mathbf{x}^{(m)}, y^{(m)})\}$$

k-Nearest Neighbor Classification

- **Training**: store the training data (trivial!)

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}, \ldots (\mathbf{x}^{(m)}, y^{(m)})\}$$

- **Prediction**: for a new instance $\mathbf{x}$, predict label that is **most frequent** among k training examples **closest** to $\mathbf{x}$

k-Nearest Neighbor Classification

- **Training**: store the training data (trivial!)

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}, \ldots (\mathbf{x}^{(m)}, y^{(m)})\}$$

- **Prediction**: for a new instance $\mathbf{x}$, predict label that is **most frequent** among k training examples **closest** to $\mathbf{x}$

- KNN can work with any distance function and any value of k. We need to choose these.

Distance and Similarity

- KNN can use any **distance function** to determine k nearest neighbors. A distance function $d(\mathbf{x}, \mathbf{x}')$ takes two data points and returns a distance. It should satisfy
  - $d(\mathbf{x}, \mathbf{x}') \geq 0$ (non-negativity)
  - $d(\mathbf{x}, \mathbf{x}') = 0$ (distance from a point to itself is zero)

Distance and Similarity

- KNN can use any **distance function** to determine k nearest neighbors. A distance function $d(\mathbf{x}, \mathbf{x}')$ takes two data points and returns a distance. It should satisfy
  - $d(\mathbf{x}, \mathbf{x}') \geq 0$ (non-negativity)
  - $d(\mathbf{x}, \mathbf{x}') = 0$ (distance from a point to itself is zero)
- Or you can use a *similarity function*
  - $s(\mathbf{x}, \mathbf{x}') \geq 0$
  - $s(\mathbf{x}, \mathbf{x}) \geq s(\mathbf{x}, \mathbf{x}')$ for all other $\mathbf{x}'$ ($\mathbf{x}$ is more similar to itself than any other point)

Euclidean Distance

- We've already seen one distance function, the **Euclidean distance**:

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|$$

  - Length of straight line between $\mathbf{x}$ and $\mathbf{x}'$

Euclidean Distance

- We've already seen one distance function, the **Euclidean distance**:

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|$$

  - Length of straight line between $\mathbf{x}$ and $\mathbf{x}'$ ($=$ vector norm of $\mathbf{x} - \mathbf{x}'$)

Minkowski Distance

- A more general class of distance functions come from **Minkowski Distance**

$$d_p(\mathbf{x}, \mathbf{x}') := \|\mathbf{x} - \mathbf{x}'\|_p$$
$$\|\mathbf{r}\|_p := \Big(\sum_{i=1}^{n} |r_i|^p\Big)^{1/p}$$

Minkowski Distance

- A more general class of distance functions come from **Minkowski Distance**

$$d_p(\mathbf{x}, \mathbf{x}') := \|\mathbf{x} - \mathbf{x}'\|_p$$
$$\|\mathbf{r}\|_p := \Big(\sum_{i=1}^{n} |r_i|^p\Big)^{1/p}$$

  - $p = 2$ is Euclidean distance (verify on own)
  - $p = 1$ is called the "Manhattan distance", intutition (2D): path "around the block"
  - $p = \infty$ is called the Chebyshev's distance $d_\infty(\mathbf{x}, \mathbf{x}') \max_i(|x_i - x_i'|)$, intutition (2D): In a warehouse, the distance between locations can be represented as Chebyshev distance if an overhead crane is used because the crane moves on both axes at the same time with the same speed.

Examples

- Jupyter Demo 1: different distance functions

KNN Implementation

- The “brute force” version of KNN is very straightforward:

KNN Implementation

- The "brute force" version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example

KNN Implementation

- The "brute force" version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example
  - Sort training examples by distance

KNN Implementation

- The "brute force" version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example
  - Sort training examples by distance
  - k-nearest neighbors $=$ first k examples in this sorted list.

KNN Implementation

- The “brute force” version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example
  - Sort training examples by distance
  - k-nearest neighbors $=$ first k examples in this sorted list.
  - Now, making the prediction is straightforward.

KNN Implementation

- The "brute force" version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example
  - Sort training examples by distance
  - k-nearest neighbors $=$ first k examples in this sorted list.
  - Now, making the prediction is straightforward.
  - **Running time**: $O(m \log m)$ for *one* prediction

KNN Implementation

- The "brute force" version of KNN is very straightforward:
  - Given test point $\mathbf{x}$, compute distances $d^{(i)} := d(\mathbf{x}, \mathbf{x}^{(i)})$ to each training example
  - Sort training examples by distance
  - k-nearest neighbors = first k examples in this sorted list.
  - Now, making the prediction is straightforward.
  - **Running time**: $O(m \log m)$ for *one* prediction

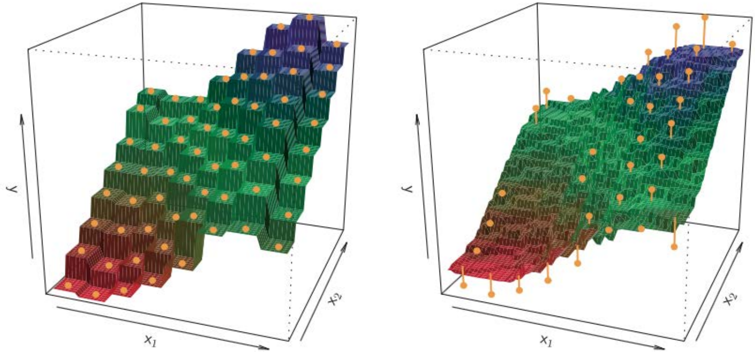
- In practice, clever data structures (e.g., KD-trees) can be constructed to find k nearest neighbors and make predictions more quickly.

KNN regression

- The KNN regression is a non-parametric regression method that simply stores the training data D and makes a prediction for each new instance $\mathbf{x}$ using an average over its set of K nearest neighbors $\mathcal{N}_K(\mathbf{x})$ computed using any distance function d as in classification.

$$f_{KNN}(\mathbf{x}) = \frac{1}{K} \sum_{i \in \mathcal{N}_K(\mathbf{x})} y^{(i)}$$

Example: 2D KNN regression (K=1, and K=9)



KNN Trade-Offs

- Strengths
  - Simple
  - Converges to the correct decision surface as data goes to infinity

KNN Trade-Offs

- Strengths
  - Simple
  - Converges to the correct decision surface as data goes to infinity
- Weaknesses
  - Lots of variability in the decision surface when amount of data is low

KNN Trade-Offs

- Strengths
  - Simple
  - Converges to the correct decision surface as data goes to infinity
- Weaknesses
  - Lots of variability in the decision surface when amount of data is low
  - Curse of dimensionality: everything is far from everything else in high dimensions (empty space phonomenon)

KNN Trade-Offs

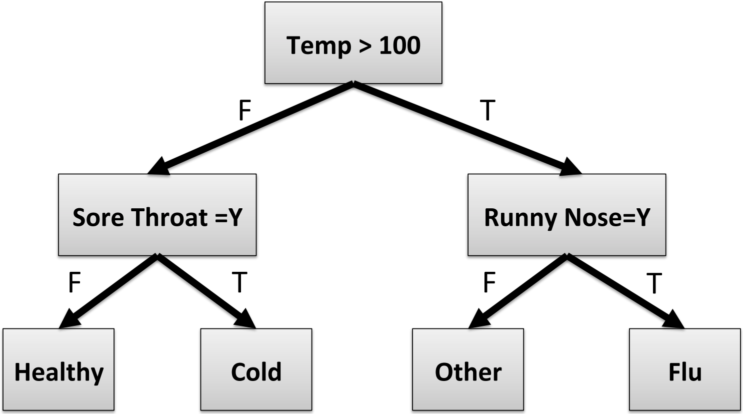
- Strengths
  - Simple
  - Converges to the correct decision surface as data goes to infinity
- Weaknesses
  - Lots of variability in the decision surface when amount of data is low
  - Curse of dimensionality: everything is far from everything else in high dimensions (empty space phonomenon)
  - Running time and memory usage: store all training data and perform neighbor search for every prediction $\rightarrow$ use a lot of memory / time

KNN Trade-Offs

- Strengths
  - Simple
  - Converges to the correct decision surface as data goes to infinity
- Weaknesses
  - Lots of variability in the decision surface when amount of data is low
  - Curse of dimensionality: everything is far from everything else in high dimensions (empty space phonomenon)
  - Running time and memory usage: store all training data and perform neighbor search for every prediction $\rightarrow$ use a lot of memory / time
- Jupyter Demo 2: KNN in action
  - Effect of k
  - KNN convergence as data goes to infinity

Decision Trees

Example: Flu decision tree



Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure
- Data instance $\mathbf{x}$ is routed from the root to leaf

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure

- Data instance $\mathbf{x}$ is routed from the root to leaf
  - Nodes = "splitting rules"

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure
- Data instance $\mathbf{x}$ is routed from the root to leaf
  - Nodes = "splitting rules"
    - Continuous variables: test if $(x_j < c)$ or $(x_j \geq c)$ (2 branches)

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure

- Data instance $\mathbf{x}$ is routed from the root to leaf

  - Nodes = "splitting rules"

    - Continuous variables: test if $(x_j < c)$ or $(x_j \geq c)$ (2 branches)
    - Discrete variables: test $(x_j = 1), (x_j = 2), \ldots$ for k possible values of x_j (k branches)

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure

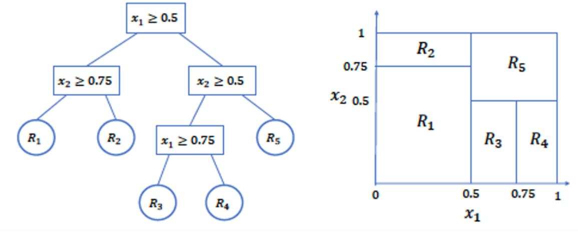
- Data instance $\mathbf{x}$ is routed from the root to leaf
  - Nodes = "splitting rules"
    - Continuous variables: test if $(x_j < c)$ or $(x_j \geq c)$ (2 branches)
    - Discrete variables: test $(x_j = 1), (x_j = 2), \ldots$ for k possible values of x_j (k branches)
  - $\mathbf{x}$ goes down branch corresponding to result of test

Decision Trees

- Classical model for making a decision or classfication using "splitting rules" organized into tree data structure

- Data instance $\mathbf{x}$ is routed from the root to leaf
  - Nodes = "splitting rules"
    - Continuous variables: test if $(x_j < c)$ or $(x_j \geq c)$ (2 branches)
    - Discrete variables: test $(x_j = 1), (x_j = 2), \ldots$ for k possible values of x_j (k branches)
  - $\mathbf{x}$ goes down branch corresponding to result of test
  - Leaf nodes are assigned labels $\rightarrow$ prediction for $\mathbf{x}$

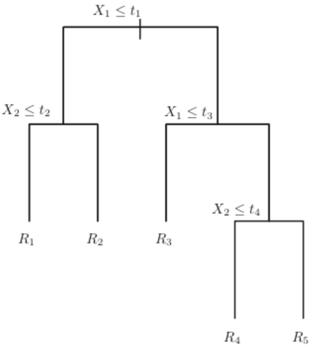
Regression Trees

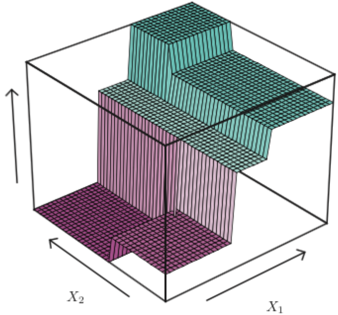


$$f(\mathbf{x}) = \sum_{m=1}^{M} c_m I(\mathbf{x} \in R_m)$$

- regression tree will divide the input data into M regions $R_1, R_2, \ldots, R_M$
- for an input data $\mathbf{x}$, it outputs c_m if $\mathbf{x} \in R_m$

Regression Trees





Regression Tree Learning

- How do we fit a regression tree to training data?

Regression Tree Learning

- How do we fit a regression tree to training data? Given $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})$, if $R_1, R_2, \ldots, R_m$ are fixed then the estimates of constants $c_1, \ldots, c_M$ are

$$\hat{c}_m = average(y^{(i)} | \mathbf{x}^{(i)} \in R_m)$$

  if MSE is cost function.

Regression Tree Learning

- How do we fit a regression tree to training data? Given $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})$, if $R_1, R_2, \ldots, R_m$ are fixed then the estimates of constants $c_1, \ldots, c_M$ are

$$\hat{c}_m = average(y^{(i)} | \mathbf{x}^{(i)} \in R_m)$$

  if MSE is cost function.

- How do we find best splits?

Regression Tree Learning

- How do we fit a regression tree to training data? Given $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})$, if $R_1, R_2, \ldots, R_m$ are fixed then the estimates of constants $c_1, \ldots, c_M$ are

$$\hat{c}_m = average(y^{(i)} | \mathbf{x}^{(i)} \in R_m)$$

  if MSE is cost function.

- How do we find best splits? Exhaustive search: NP-hard. Need greedy algorithm

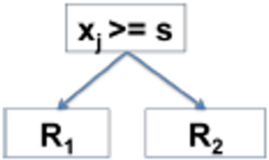
Regression Tree Learning

Idea: recursive splitting of training set

Regression Tree Learning

Idea: recursive splitting of training set

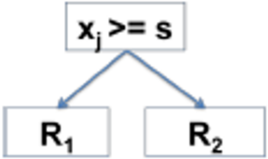
- Start with all training examples at root of tree



Regression Tree Learning

Idea: recursive splitting of training set

- Start with all training examples at root of tree



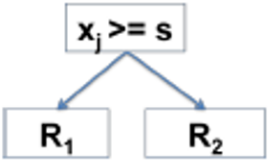
- Find "best" splitting rule at root (best pair x_j, s)

$$\min_{j,s}\left(\sum_{\mathbf{x}^{(i)}\in R_1(j,s)}(y^{(i)}-\hat{c}_1)^2+\sum_{\mathbf{x}^{(i)}\in R_2(j,s)}(y^{(i)}-\hat{c}_2)^2\right)$$

Regression Tree Learning

Idea: recursive splitting of training set

- Start with all training examples at root of tree



- Find "best" splitting rule at root (best pair x_j, s)

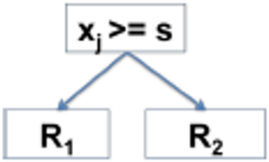
$$\min_{j,s} \left(\sum_{\mathbf{x}^{(i)} \in R_1(j,s)} (y^{(i)} - \hat{c}_1)^2 + \sum_{\mathbf{x}^{(i)} \in R_2(j,s)} (y^{(i)} - \hat{c}_2)^2 \right)$$

- Split the data based on where it falls

Regression Tree Learning

Idea: recursive splitting of training set

- Start with all training examples at root of tree



- Find "best" splitting rule at root (best pair x_j, s)

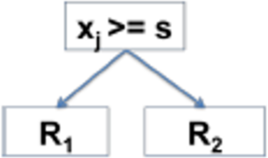
$$\min_{j,s} \left(\sum_{\mathbf{x}^{(i)} \in R_1(j,s)} (y^{(i)} - \hat{c}_1)^2 + \sum_{\mathbf{x}^{(i)} \in R_2(j,s)} (y^{(i)} - \hat{c}_2)^2 \right)$$

- Split the data based on where it falls
- Recurse on each branch

Regression Tree Learning

Idea: recursive splitting of training set

- Start with all training examples at root of tree

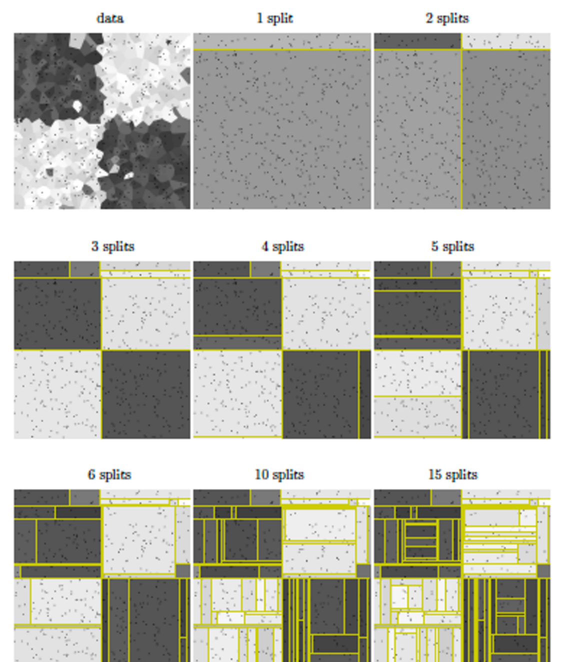


- Find "best" splitting rule at root (best pair x_j, s)

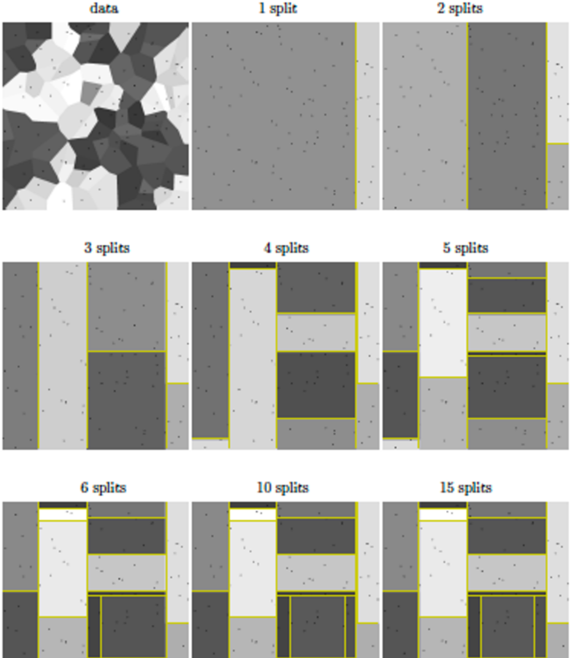
$$\min_{j,s}\left(\sum_{\mathbf{x}^{(i)}\in R_1(j,s)}(y^{(i)}-\hat{c}_1)^2+\sum_{\mathbf{x}^{(i)}\in R_2(j,s)}(y^{(i)}-\hat{c}_2)^2\right)$$

- Split the data based on where it falls
- Recurse on each branch
- Stop eventually (to control capacity of the tree) at maximum depth, at maximum number of terminal nodes, at maximum number of splits (and many others)

Regression Tree Learning: Examples



Regression Tree Learning: Examples



Complexity

The runtime of a naive implementation is pN^2 where p number of dimensions and N number of training points. We can improve this by using recursive algorithms which brings the run time to $pN \log N$.

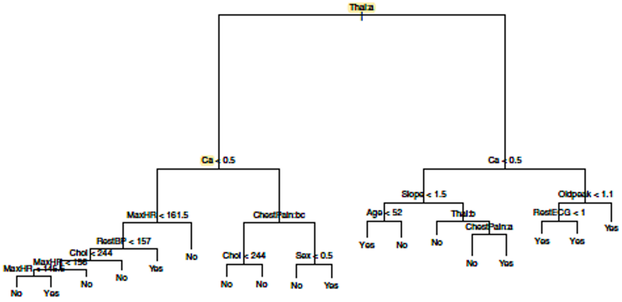
Tree Pruning

- training procedure: grow a tree up to a certain depth D. Could lead to overfitting (high complexity, low bias, high variance)
- instead of stopping early, grow large tree T_0 and prune later using complexity pruning (or weakest link pruning)

$$\sum_{m=1}^{|T|} \sum_{\mathbf{x}^{(i)} \in R_m} (y^{(i)} - \hat{y}_{R_m})^2 + \alpha |T|$$

  (first term is training error, secind penalty on number of leaves)
- cross-validate α

Classification Trees



$$f(\mathbf{x}) = \sum_{m=1}^{M} c_m I(\mathbf{x} \in R_m)$$

- A classification tree will divide the input data into M regions $R_1, R_2, \ldots, R_M$
- for an input data $\mathbf{x}$, it outputs c_m if $\mathbf{x} \in R_m$ but $c_m \in \{1, 2, \ldots, K\}$ is a class label

Classification Tree Learning

- How do we fit a regression tree to training data?

Classification Tree Learning

- How do we fit a regression tree to training data? For fixed splits, c_m is set to be the most common label in region R_m. For convenience, we will note this most common label as $k(m)$.

- How do we find best splits?

Classification Tree Learning

- How do we fit a regression tree to training data? For fixed splits, c_m is set to be the most common label in region R_m. For convenience, we will note this most common label as $k(m)$.

- How do we find best splits? We use recursive binary splitting but can't use MSE as cost.

Classification Tree Learning

- How do we fit a regression tree to training data? For fixed splits, c_m is set to be the most common label in region R_m. For convenience, we will note this most common label as $k(m)$.
- How do we find best splits? We use recursive binary splitting but can't use MSE as cost.
- First, we write the fraction of training data in region m with label k to be

$$\hat{P}_{mk} = \frac{1}{N_m} \sum_{\mathbf{x}^{(i)} \in R_m} I(y^{(i)} = k)$$

Our intuition is to let $\hat{P_{mk}}$ have low randomness.

Classification Tree Learning

We have the following choices to measure randomness.

- Misclassification Error $1 - \max_k \hat{P_{mk}}$. This is rarely used since it not sufficiently sensitive for tree growing.

- Gini Index

$$\sum_{k=1}^{K} \sum_{k \neq k'} \hat{P_{mk}} \hat{P_{mk'}} = \sum_{k=1}^{K} \hat{P_{mk}} \left(1 - \hat{P_{mk}}\right)$$

- Cross Entropy (from information theory)

$$-\sum_{k=1}^{K} \hat{P_{mk}} \log \hat{P_{mk}}$$

One can show that the entropy will take on a value near zero if the $\hat{P_{mk}}$ are all near zero or near one. In practice, Gini Index and Cross Entropy usually have similar effects.

Decision Tree Trade-Offs

- Strengths
  - **Interpretability**: the learned model is easy to understand
  - Trees can easily handle qualitative predictor
  - **Running time for predictions**: shallow trees can be extremely fast classifiers

Decision Tree Trade-Offs

- Strengths
  - **Interpretability**: the learned model is easy to understand
  - Trees can easily handle qualitative predictor
  - **Running time for predictions**: shallow trees can be extremely fast classifiers

- Weaknessees
  - **Running time for learning**: finding the optimal trees is computationally intractable (NP-complete), so we need to design greedy heuristics.
  - **Representation**: we may need very large trees to accurately model geometry of our problem with axis-aligned splits (lead to higher variance)
  - **Accuracy** By themselves, not always as performing as NN or SVM.

Decision Tree Trade-Offs

- Strengths
  - **Interpretability**: the learned model is easy to understand
  - Trees can easily handle qualitative predictor
  - **Running time for predictions**: shallow trees can be extremely fast classifiers
- Weaknessees
  - **Running time for learning**: finding the optimal trees is computationally intractable (NP-complete), so we need to design greedy heuristics.
  - **Representation**: we may need very large trees to accurately model geometry of our problem with axis-aligned splits (lead to higher variance)
  - **Accuracy** By themselves, not always as performing as NN or SVM.
- General advice: decision trees are very competitive "out-of-the-box" machine learning models for lots of problems if associated with methods such as bagging, random forests or boosting.

Bagging for Trees

- Trees have high veriance (if fit a tree to two halves of training set can get different results)
- If one can extract several training sets from population and average predictions (regression) or take majority vote (classification) over each set can lower variance
- We only have one dataset: bootstrap (select several training samples with replacement) and average predictions (regression) or take majority vote (classification). This is called bagging.
- in bagging the trees are grown deep and not pruned (low bias, high variance)
- B larger is better but higher computational complexity and diminishing returns in accuracy the larger it is.

Out-of-Bag Error Estimation

- No need for crossvalidation to estimate test error of bagged model. Each bootstrapped tree makes use of around two thirds of the observations. Can use the rest (out-of-bag or OOB) observations for testing
- single observation $\mathbf{x}$ is OOB for around $B/3$ trees (B is the $\#$ of trees created by bootstrapping). Predict output for $\mathbf{x}$ for each tree and average (regression) or majority vote (classification).

Variable Importance

- Total amount that the MSE (regression) or the Gini Index (classification) is decreased due to splits over a given predictor, averaged over all B trees. A large value indicates an important predictor.
- One can train the ensamble only using the l most important features. Slight decrease in performance but boost in computation time gain.

Random Forests

As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m = \sqrt(p)$ but can cross-validate over m.

This avoids having that all trees use particularly strong predictors in the top split, making the trees correlated.

Low value of m preferrable when we have large number of correlated predictors.