Practical Issues with Decision Trees

CSE 4309 – Machine Learning

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Programming Assignment

- The next programming assignment asks you to implement decision trees, as well as a variation called “decision forests”.
- There are several concepts that you will need to implement, that we have not addressed yet.
- These concepts are discussed in these slides.

Data

- The dataset format is the same as in several previous assignments.
- For each dataset, you are given:
  - a training file, that you use to learn decision trees.
  - a test file, that you use to apply decision trees and measure their accuracy.
- All three datasets follow the same format:
  - Each line is an object.
  - Each column is an attribute, except:
  - The last column is the class label.

Data

- Values are separated by whitespace.
- The attribute values are real numbers (doubles).
  - They are integers in some datasets, just treat those as doubles.
- The class labels are integers.

Class Labels Are Not Attributes

- A classic mistake is to forget that the last column contains class labels.
- What happens if you include the last column in your attributes?

Class Labels Are Not Attributes

- A classic mistake is to forget that the last column contains class labels.
- What happens if you include the last column in your attributes?
- You get perfect classification accuracy.
- The decision tree will be using class labels to predict class labels.
  - Not very hard to do.
- So, make sure that, when you load the data, you separate the last column from the rest of the columns.

Dealing with Continuous Values

- Our previous discussion on decision trees assumed that each attribute takes a few discrete values.
- Instead, in these datasets the attributes take continuous values.
- There are several ways to discretize continuous values.
- For the assignment, we will discretize using thresholds.
  - The test that you will be choosing for each node will be specified using both an attribute and a threshold.
  - Objects whose value at that attribute is LESS THAN the threshold go to the left child.
  - Objects whose value at that attribute is GREATER THAN OR EQUAL TO the threshold go to the right child.

Dealing with Continuous Values

- For example: supposed that the test that is chosen for a node N uses attribute 5 and a threshold 30.7.
- Then:
  - Objects whose value at attribute 5 is LESS THAN 30.7 go to the left child of N.
  - Objects whose value at attribute 5 is GREATER THAN OR EQUAL TO 30.7 go to the right child.

- Please stick to these specs.
- Do not use LESS THAN OR EQUAL instead of LESS THAN.

Dealing with Continuous Values

- Using thresholds as described, what is the maximum number of children for a node?

Dealing with Continuous Values

- Using thresholds as described, what is the maximum number of children for a node?
- Two. Your decision trees will be **binary**.

Choosing a Threshold

- How can you choose a threshold?
  - What makes a threshold better than another threshold?
- Remember, once you have chosen a threshold, you get a binary version of your attribute.
  - Essentially, you get an attribute with two discrete values.
- You know all you need to know to compute the information gain of this binary attribute.
- Given an attribute A, different thresholds applied to A produce different values for information gain.
- The best threshold is which one?

Choosing a Threshold

- How can you choose a threshold?
  - What makes a threshold better than another threshold?
- Remember, once you have chosen a threshold, you get a binary version of your attribute.
  - Essentially, you get an attribute with two discrete values.
- You know all you need to know to compute the information gain of this binary attribute.
- Given an attribute A, different thresholds applied to A produce different values for information gain.
- The best threshold is which one?
  - The one leading to the highest information gain.

Searching Thresholds

- Given a node N, and given an attribute A with continuous values, you should check various thresholds, to see which one gives you the highest information gain for attribute A at node N.
- How many thresholds should you try?
- There are (again) many different approaches.
- For the assignment, you should try 50 thresholds, chosen as follows:
  - Let L be the smallest value of attribute A among the training objects at node N.
  - Let M be the largest value of attribute A among the training objects at node N.
  - Then, try thresholds: L + (M-L)/51, L + 2*(M-L)/51, …, L + 50*(M-L)/51.
  - Overall, you try all thresholds of the form L + K*(M-L)/51, for K = 1, …, 50.

Review: Decision Tree Learning

function DTL_TopLevel(*examples*) **returns** a decision tree
    attributes **=** list of all attributes in the examples
    default **=** DISTRIBUTION(*examples*)
    return DTL(*examples*, *attributes*, *default*)

- DTL_TopLevel is the top level function for decision tree learning.
- It defines two variables (*attributes* and *default*), and calls the DTL function.
  - DTL (explained in the next slides) is an auxiliary function that does the actual work.
- DTL is recursive, so the job of DTL_TopLevel is to just make the first call to DTL, with the right arguments.

Review: Decision Tree Learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree
    if *examples* is empty **then return** *default*
    else if all *examples* have the same class **then return** the class
    else
        (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
        tree = a new decision tree with root test (*best_attribute, best_threshold*)
        examples_left = {elements of *examples* with *best_attribute* < *threshold*}
        examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
        tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))
        tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))
        return *tree*

- Above you see the DTL function that we have reviewed previously, with a few modifications, to account for the assignment requirements:

Review: Decision Tree Learning

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same class then return the class
    else
        (best_attribute, best_threshold) = CHOOSE-ATTRIBUTE(examples, attributes)
        tree = a new decision tree with root test (best_attribute, best_threshold)
        examples_left = {elements of examples with best_attribute < threshold}
        examples_right = {elements of examples with best_attribute >= threshold}
        tree.left_child = DTL(examples_left, attributes, DISTRIBUTION(examples))
        tree.right_child = DTL(examples_right, attributes, DISTRIBUTION(examples))
        return tree
```

- Above you see the DTL function that we have reviewed previously, with a few modifications, to account for the assignment requirements:
  - CHOOSE-ATTRIBUTE needs to pick both an attribute and a threshold.

Review: Decision Tree Learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree
  if *examples* is empty **then return** *default*
  else if all *examples* have the same class **then return** the class
  else
    (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
    tree = a new decision tree with root test (*best_attribute, best_threshold*)
    examples_left = {elements of *examples* with *best_attribute* < *threshold*}
    examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
    tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))
    tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))
    return *tree*

- How are these DTL recursive calls different than before?

Review: Decision Tree Learning

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same class then return the class
    else
        (best_attribute, best_threshold) = CHOOSE-ATTRIBUTE(examples, attributes)
        tree = a new decision tree with root test (best_attribute, best_threshold)
        examples_left = {elements of examples with best_attribute < threshold}
        examples_right = {elements of examples with best_attribute >= threshold}
        tree.left_child = DTL(examples_left, attributes, DISTRIBUTION(examples))
        tree.right_child = DTL(examples_right, attributes, DISTRIBUTION(examples))
        return tree
```

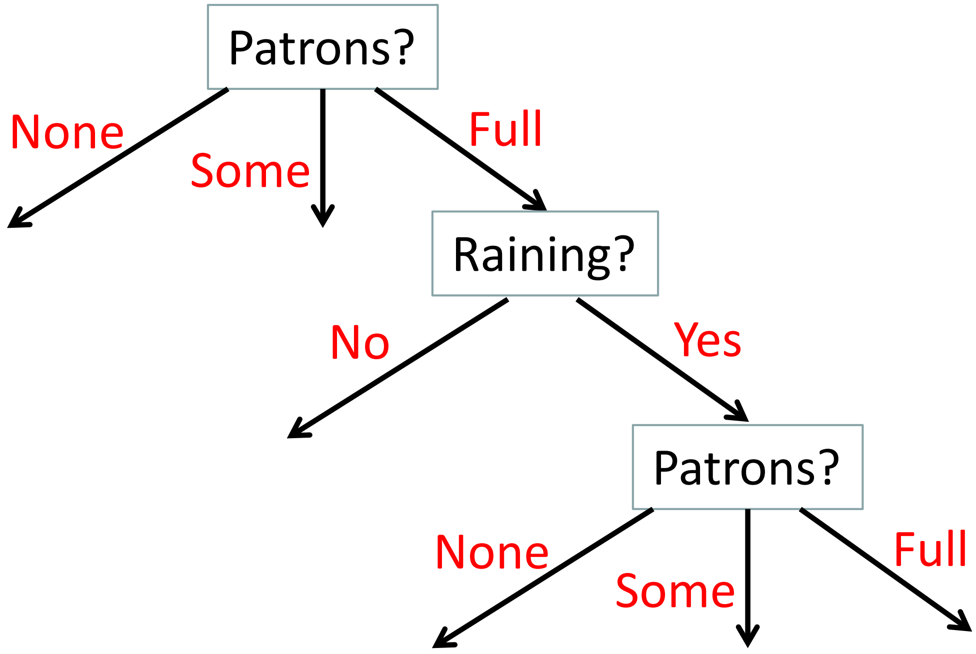
- How are these DTL recursive calls different than before?
  - Before, we were passing attributes – best_attribute.
  - Now we are passing attributes, without removing best_attribute.
  - Why?

Review: Decision Tree Learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree
    if *examples* is empty **then return** *default*
    else if all *examples* have the same class **then return** the class
    else
        (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
        tree = a new decision tree with root test (*best_attribute, best_threshold*)
        examples_left = {elements of *examples* with *best_attribute* < *threshold*}
        examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
        tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))
        tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))
        return *tree*

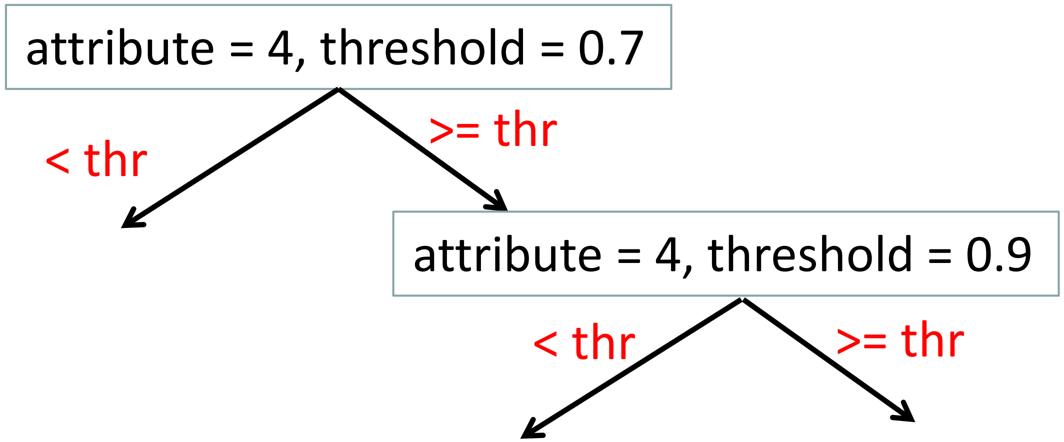
- How are these DTL recursive calls different than before?
  - Before, we were passing attributes – best_attribute.
  - Now we are passing attributes, without removing best_attribute.
  - The best attribute may still be useful later, with a different threshold.

Using an Attribute Twice in a Path



- When we were using attributes with a few discrete values, it was useless to have the same attribute appear twice in a path from the root.
  - The second time, the information gain is 0, because all training examples go to the same child.

Using an Attribute Twice in a Path



- When we use attributes with continuous values, together with a threshold, it <u>**may be useful**</u> to have the same attribute appear twice in a path from the root.
  - The second time, the information gain does not have to be 0, because we are using a different threshold.
  - The second time, all our training examples have values >= 0.7 for attribute 4.
  - Some of those values may be < 0.9, some may be >= 0.9.

Review: Decision Tree Learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree

    if *examples* is empty **then return** *default*

    else if all *examples* have the same class **then return** the class

    else

        (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)

        tree = a new decision tree with root test (*best_attribute, best_threshold*)

        examples_left = {elements of *examples* with *best_attribute* < *threshold*}

        examples_right = {elements of *examples* with *best_attribute* >= *threshold*}

        tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))

        tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))

        return *tree*

- How are these DTL recursive calls different than before?
  - There is one more difference, in addition to not removing **best_attribute** from **attributes**.

Review: Decision Tree Learning

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree
    if *examples* is empty **then return** *default*
    else if all *examples* have the same class **then return** the class
    else
        (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
        tree = a new decision tree with root test (*best_attribute, best_threshold*)
        examples_left = {elements of *examples* with *best_attribute* < *threshold*}
        examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
        tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))
        tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))
        return *tree*

- How are these DTL recursive calls different than before?
  - Instead of calling MODE(examples), we call DISTRIBUTION(examples).
  - More details on that later in these slides, when we discuss decision forests.

Search for Best Test

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same class then return the class
    else
        (best_attribute, best_threshold) = CHOOSE-ATTRIBUTE(examples, attributes)
        tree = a new decision tree with root test (best_attribute, best_threshold)
        examples_left = {elements of examples with best_attribute < threshold}
        examples_right = {elements of examples with best_attribute >= threshold}
        tree.left_child = DTL(examples_left, attributes, DISTRIBUTION(examples))
        tree.right_child = DTL(examples_right, attributes, DISTRIBUTION(examples))
        return tree
```

- In this code, where do we search for the combination of attribute and threshold that give the highest information gain?

Search for Best Test

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same class then return the class
    else
        (best_attribute, best_threshold) = CHOOSE-ATTRIBUTE(examples, attributes)
        tree = a new decision tree with root test (best_attribute, best_threshold)
        examples_left = {elements of examples with best_attribute < threshold}
        examples_right = {elements of examples with best_attribute >= threshold}
        tree.left_child = DTL(examples_left, attributes, DISTRIBUTION(examples))
        tree.right_child = DTL(examples_right, attributes, DISTRIBUTION(examples))
        return tree
```

- The search for the best combination of attribute and threshold happens in the CHOOSE-ATTRIBUTE function.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- **Note:** in the assignment, use this CHOOSE-ATTRIBUTE version when the "optimized" option is provided on the command line. More details in a bit.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- **examples** is the training data. It is a matrix, where each row is a training object, each column is an attribute, the last column contains class labels.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- To fit with this pseudocode, **attributes** can simply be an array, containing values 0, 1, …, up to the number of attributes – 1.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- The function returns the combination of attribute and threshold that produce the highest information gain.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- These variables will keep track of the attribute and threshold that have produced the highest information gain so far.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- Obviously, to find the best attribute, we must loop over all attributes.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- **attribute _values** is the array containing the values of all examples for attribute A.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- We find the minimum and maximum value of attribute A among the examples, so that we can try 50 threshold values between the min and max.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- Loop over the 50 thresholds.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- For each threshold, measure the information gain attained on these examples using that combination of attribute A and threshold.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- If we found the best combination of attribute and threshold so far, keep track of it.

CHOOSE-ATTRIBUTE, Optimized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_attribute = best_threshold = -1
    for each attribute A of attributes do
        attribute_values = SELECT-COLUMN(examples, A)
        L = min(attribute_values)
        M = max(attribute_values)
        for K = 1; K <= 50; K++
            threshold = L + K*(M-L)/51
            gain = INFORMATION-GAIN(examples, A, threshold)
            if gain > max_gain then
                max_gain = gain
                best_attribute = A
                best_threshold = threshold
    return (best_attribute, best_threshold)
```

- Return the best combination of attribute and threshold that we have found.

Using Many Different Tests

- When we have continuous-valued attributes, the number of possible tests (combinations of attribute and threshold) can be huge.
- There are also many applications where the number of attributes is itself huge (thousands, or millions).
- Can a single decision tree apply millions of tests to an object?
- In theory yes, but to learn such a tree, we would need a humongous amount of training data, more than we can handle with today's computers.

Decision Forests

- When we have too many combinations of attributes and thresholds to fit into a single tree, we can learn multiple different trees.
- Question: how do we learn multiple different trees?
  – Will our DTL algorithm work?
- No. The version we have seen is deterministic.
- Given the same training examples, it will always come up with the same tree.
  – Unless there are ties, where multiple combinations of attributes and thresholds tie for best, and we let DTL choose randomly among them.

Decision Forests

- To learn multiple different trees, we need to force the algorithm to make some random choices, so that each time it is called it produces a different tree.
- There are different approaches as to what to randomize.
- We will follow a simple approach:
  - CHOOSE-ATTRIBUTE chooses an attribute randomly.
  - For that attribute that is chosen randomly, we still need to find the best threshold.

CHOOSE-ATTRIBUTE, Randomized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_threshold = -1
    A = RANDOM-ELEMENT(attributes)
    attribute_values = SELECT-COLUMN(examples, A)
    L = min(attribute_values)
    M = max(attribute_values)
    for K = 1; K <= 50; K++
        threshold = L + K*(M-L)/51
        gain = INFORMATION-GAIN(examples, A, threshold)
        if gain > max_gain then
            max_gain = gain
            best_threshold = threshold
    return (A, best_threshold)
```

- Here is the randomized version of CHOOSE-ATTRIBUTE.
- Main modification: Now we pick a random attribute.

CHOOSE-ATTRIBUTE, Randomized

```
function CHOOSE-ATTRIBUTE(examples, attributes) returns (attribute, threshold)
    max_gain = best_threshold = -1
    A = RANDOM-ELEMENT(attributes)
    attribute_values = SELECT-COLUMN(examples, A)
    L = min(attribute_values)
    M = max(attribute_values)
    for K = 1; K <= 50; K++
        threshold = L + K*(M-L)/51
        gain = INFORMATION-GAIN(examples, A, threshold)
        if gain > max_gain then
            max_gain = gain
            best_threshold = threshold
    return (A, best_threshold)
```

- We still search to find the best threshold for that attribute, so as to maximize information gain.

Choosing CHOOSE-ATTRIBUTE Version

- So, we have now two different CHOOSE-ATTRIBUTE versions.
- Question: which one do you use in the assignment?
- Answer: both.
- The third command line argument determines which version you use.
- The third command line argument can have four possible values:
  - **optimized** - use the first CHOOSE-ATTRIBUTE version, that finds the best combination of attribute and threshold, learn a single tree.
  - **randomized** - use the second CHOOSE-ATTRIBUTE version, learn a single randomized tree.
  - **forest3 -** use the second CHOOSE-ATTRIBUTE version, learn three randomized trees.
  - **forest15 -** use the second CHOOSE-ATTRIBUTE version, learn fifteen randomized trees.

Classification with Random Forests

- When we apply multiple decision trees to the same object, obviously the trees may provide different answers.
- How can we combine those answers into a single best answer?
- Solution: the answer of each tree will be a probability distribution, assigning a probability to each class.
- To classify an object using a decision forest, consisting of multiple decision trees:
  - First, apply each tree to the object, to obtain from that tree a probability distribution.
  - Then, compute the average of those probability distributions. For each class, simply compute the average of its probabilities.
  - Finally, identify and output the class with the highest average probability.

Storing Probability Distributions

function DTL(*examples*, *attributes*, *default*) **returns** a decision tree
    if *examples* is empty **then return** *default*
    else if all *examples* have the same class **then return** the class
    else
        (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
        tree = a new decision tree with root test (*best_attribute, best_threshold*)
        examples_left = {elements of *examples* with *best_attribute* < *threshold*}
        examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
        tree.left_child = DTL(*examples_left*, *attributes*, DISTRIBUTION(*examples*))
        tree.right_child = DTL(*examples_right*, *attributes*, DISTRIBUTION(*examples*))
        return *tree*

- This is why we have replaced the MODE function with a DISTRIBUTION function.
- Suppose you have N classes, and your class labels are from 0 to N-1.
- Then, the DISTRIBUTION function simply returns an array, whose i-th position is the probability of the i-th class.

Example

- Suppose we have five classes.
- Suppose that we are at a node in the decision tree where:
  - 35 training examples are from class 0.
  - 22 training examples are from class 1.
  - 15 training examples are from class 2.
  - 37 training examples are from class 3.
  - 12 training examples are from class 4.
- What does DISTRIBUTION(*examples*) return here?

Example

- Suppose we have five classes.
- Suppose that we are at a node in the decision tree where:
  - 35 training examples are from class 0.
  - 22 training examples are from class 1.
  - 15 training examples are from class 2.
  - 37 training examples are from class 3.
  - 12 training examples are from class 4.
- What does DISTRIBUTION(*examples*) return here?
  - P(class 0) = 35 / 121 = 0.2893
  - P(class 1) = 22 / 121 = 0.1818
  - P(class 2) = 15 / 121 = 0.1240
  - P(class 3) = 37 / 121 = 0.3058
  - P(class 4) = 12 / 121 = 0.0992
  - DISTRIBUTION(*examples*) returns this array:
    [0.2893, 0.1818, 0.1240, 0.3058, 0.0992].

Classification Using a Decision Forest

- Suppose that we want to classify a test object using a decision forest of 3 trees, and there are five classes.
- The first tree outputs distribution [0.2893, 0.1818, 0.1240, 0.3058, 0.0992].
- The second tree outputs distribution [0.1289, 0.1724, 0.3579, 0.1733, 0.1675].
- The third tree outputs distribution [0.2823, 0.1098, 0.2037, 0.0680, 0.3362].
- The average distribution is: [0.2335, 0.1547, 0.2285, 0.1824, 0.2010]
- So, what is the predicted class for the test object?

Classification Using a Decision Forest

- Suppose that we want to classify a test object using a decision forest of 3 trees, and there are five classes.
- The first tree outputs distribution [0.2893, 0.1818, 0.1240, 0.3058, 0.0992].
- The second tree outputs distribution [0.1289, 0.1724, 0.3579, 0.1733, 0.1675].
- The third tree outputs distribution [0.2823, 0.1098, 0.2037, 0.0680, 0.3362].
- The average distribution is: [0.2335, 0.1547, 0.2285, 0.1824, 0.2010]
- So, what is the predicted class for the test object?
  - Class 0, since it has the highest probability among all five classes.

Ties

- Suppose that the average distribution computed from the decision forest is:
  [0.3, 0.1, 0.2, 0.3, 0.1]
- What is the predicted class?
- Class 0 and class 3 are tied with probability 0.3.
- Here, your program should pick one of these classes randomly.

Pruning

- Typically, leaf nodes that contain very few examples are not very reliable.
- The distribution of classes among those few examples may depend more on luck than on any pattern among training examples.
- One approach to handle this case is pruning.
- Pruning means that we eliminate some leaf nodes that contain few examples and are not reliable.

A Method for Pruning

function DTL(*examples*, *attributes*, *default, pruning_thr*) **returns** a decision tree
  if SIZE(*examples*) < *pruning_thr* **then return** *default*
  else if all *examples* have the same class **then return** the class
  else
    (*best_attribute, best_threshold*) = CHOOSE-ATTRIBUTE(*examples*, *attributes*)
    tree = a new decision tree with root test (*best_attribute, best_threshold*)
    examples_left = {elements of *examples* with *best_attribute* < *threshold*}
    examples_right = {elements of *examples* with *best_attribute* >= *threshold*}
    dist = DISTRIBUTION(*examples*)
    tree.left_child = DTL(*examples_left*, *attributes*, *dist, pruning_thr*)
    tree.right_child = DTL(*examples_right*, *attributes*, *dist, pruning_thr*)
    return *tree*

- Add the pruning threshold as a fourth argument to DTL.
- Instead of checking if the examples are empty, check if they contain fewer elements than the pruning threshold.

Top Level Function with Pruning

function DTL_TopLevel(*examples*, *pruning_thr*) **returns** a decision tree
    attributes **=** list of all attributes in the examples
    default **=** DISTRIBUTION(*examples*)
    return DTL(*examples*, *attributes*, *default, pruning_thr*)

- We add *pruning_thr* as an argument to the top level function.