2nd hand in – trees

Emmely Lundberg cph-el69

A decision tree is used for classification problems.

A decision tree consists of internal nodes (nodes with subtrees) and leaves (terminal nodes, nodes without subtrees). The links from the root node to a leaf is called a branch.

You have a collection of information when you use the tree. Each internal node contains a question, and the supplied information will determine which link to follow from the node. In the general case there can be several answers and different questions in the different branches and branches of different length. The leaves will contain the final classification.

In the simple case we will use here with a binary tree there will be two answers (yes/no – 1/0) and the questions will be the same in each branch.

Make a decision tree for warning students.

A student will come with 4 items of information : "Read textbook", "Hand ins made in time", "Attend lectures", "Make exercises". E.g. for a nice student (yes, yes, yes, yes).

The classification should be "You could easily fail the exam" in the following cases : "Hand ins made in time" = no "Attend lectures" and "Read textbook" = no and no "Attend lectures" and "Make exercises" = no and no The classification should be "You should be able to pass the exam" in all other cases.

The algorithm or method I use to get to the solution

1. First I ask the tree for the root node. And ask it to print the question that is stored in the node data.
2. Then I use the following method that takes the answer "yes" or "no" and return the child (left or right node) that matches the answer.

```
// Special function that depending on the answer retrun the node to the left or to the right.
func NavigateNode(n *Node, b string) *Node {
  if (b == "yes") {
     return n.Left
  } else {
     return n.Right
  }
}
```

3. Then I iterate over this method until I reached the last question where the node contains the answer about the student.

The implementation in summary

For the code for the Decision Tree (DecisionTree.go) implementation I have taken inspiration from this article about Binary Search Tree in go (https://appliedgo.net/bintree/).

The implementation of this solutions consist of a balanced Binary Search Tree. The nodes contain data about the question/answer to the student. The nodes are inserted nodes so the binary search tree would be shaped as the Decision Tree in the task.

Go language has been used as the programming language.

The program consists of two files **DecisionTree.go** and **Main.go**.

DecisionTree.go

- Decision tree contains all the logic regarding the decision tree.
- Implementation of the Binary Tree

Main.go

- The Main.go is used to run the program and make the tests.
- Initialize Decision Tree with key and data.
- Runs the tests on the tree and sets up test data.

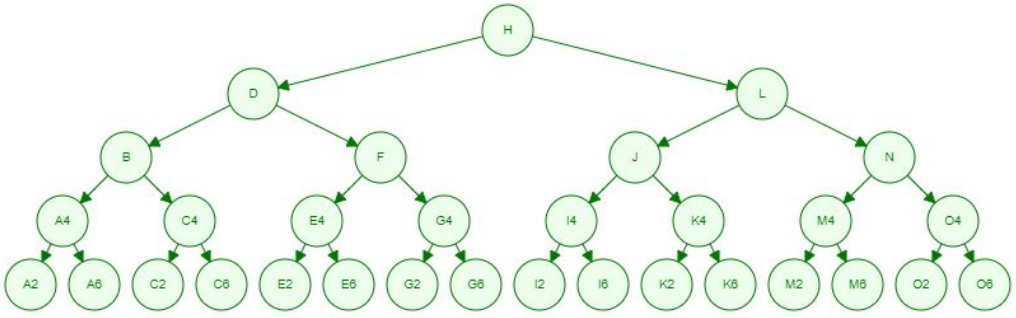
This is what the Node implementation looks like found in DecisionTree.go. (question acts as the answer for the final nodes (leaves))

```
type Data struct {
  id       string
  answer   string
  question string
}
type Node struct {
  Value string
  Data  Data
  Left  *Node
  Right *Node
}
```

I used this site (https://www.cs.usfca.edu/~galles/visualization/BST.html) to verify that the

tree were build up as I wanted. The graphical representation can be seen in the picture below.

This is what the tree looks like. The tree has been inserted with the keys shown in the picture. Positive answers "yes/true" - goes to the left. Negative answers "no/false" goes to the right.



This code returns the left/right node depending on answer:

```
// Special function that depending on the answer retrun the node to the left or to
the right.
func NavigateNode(n *Node, b string) *Node {
  if (b == "yes") {
     return n.Left
  } else {
     return n.Right
  }
}
```

Code for inserting the keys (Main.go). Initializing the Decision tree:

```
// Data for tree
data := []Data{
  Data{"H", "true", "Read textbook?"},

  Data{"D", "true - ", "Hand ins made in time?"},
  Data{"L", "false - ", "Hand ins made in time?"},

  Data{"B", "true", "Attend lectures?"},
  Data{"F", "false", "Attend lectures?"},
  Data{"J", "true", "Attend lectures?"},
  Data{"N", "false", "Attend lectures?"},

  Data{"A4", "true", "Make exercises?"},
  Data{"C4", "false", "Make exercises?"},
  Data{"E4", "true", "Make exercises?"},
  Data{"G4", "false", "Make exercises?"},
  Data{"I4", "true", "Make exercises?"},
```

```
  Data{"K4", "false", "Make exercises?"},
  Data{"M4", "true", "Make exercises?"},
  Data{"O4", "false", "Make exercises?"},


  Data{"A2", "true", "You should be able to pass the exam"},
  Data{"A6", "false", "You should be able to pass the exam"},
  Data{"C2", "true", "You should be able to pass the exam"},
  Data{"C6", "false", "You could easily fail the exam"},
  Data{"E2", "true", "You could easily fail the exam"},
  Data{"E6", "false", "You could easily fail the exam"},
  Data{"G2", "true", "You could easily fail the exam"},
  Data{"G6", "false", "You could easily fail the exam"},
  Data{"I2", "true", "You should be able to pass the exam"},
  Data{"I6", "false", "You should be able to pass the exam"},
  Data{"K2", "true", "You could easily fail the exam"},
  Data{"K6", "false", "You could easily fail the exam"},
  Data{"M2", "true", "You could easily fail the exam"},
  Data{"M6", "false", "You could easily fail the exam"},
  Data{"O2", "true", "You could easily fail the exam"},
  Data{"O6", "false", "You could easily fail the exam"},
}

//Create a tree and fill it from the data values.
tree := &Tree{}
for i := 0; i < len(data); i++ {
  err := tree.Insert(data[i].id, data[i])
  if err != nil {
     log.Fatal("Error inserting value '", data[i].id, "': ", err)
  }
}
```

Getting the results

Code

```
func makeTest(answers [] string, tree *Tree, verbose bool) {
  fmt.Printf("%v", answers)
  fmt.Println("")
  myRoot := GetRoot(tree)
  if (verbose) {
     fmt.Println("Q: " + myRoot.Data.question + " - Node Id " + myRoot.Value)
     fmt.Println("User input: " + answers[0])
  }
  level2 := NavigateNode(myRoot, answers[0])
  if (verbose) {
     fmt.Println("Q: " + level2.Data.question + " Your previous answer was : " +
level2.Data.answer + " - Node Id " + level2.Value)
     fmt.Println("User input: " + answers[1])
  }
```

```
  level3 := NavigateNode(level2, answers[1])
  if (verbose) {
     fmt.Println("Q: " + level3.Data.question + " Your previous answer was : " +
level3.Data.answer + " - Node Id " + level3.Value)
     fmt.Println("User input: " + answers[2])
  }
  level4 := NavigateNode(level3, answers[2])
  //print(level4)
  if (verbose) {
     fmt.Println("Q: " + level4.Data.question + " Your previous answer was : " +
level4.Data.answer + " - Node Id " + level4.Value)
     fmt.Println("User input: " + answers[3])
  }
  level5 := NavigateNode(level4, answers[3])
  //print(level4)
  fmt.Println("Final answer: " + level5.Data.question + " Your previous answer was
: " + level4.Data.answer + " - Node Id " + level4.Value)
}
```

Running the tests:

```
// Student tests
verbose := true;
fmt.Println("\nGood student")
values := []string{"yes", "yes", "yes", "yes"}
makeTest(values, tree, verbose)

fmt.Println("\nHand ins made in time = no")
values = []string{"yes", "no", "yes", "yes"}
makeTest(values, tree, verbose)

fmt.Println("\nAttend lectures" and "Read textbook" = no and no")
values = []string{"no", "yes", "no", "yes"}
makeTest(values, tree, verbose)

fmt.Println("\nAttend lectures" and "Make exercises" = no and no")
values = []string{"yes", "yes", "no", "no"}
makeTest(values, tree, verbose)
```

Results

```
Good student
[yes yes yes yes]
Final answer: You should be able to pass the exam Your previous answer was : true
- Node Id A4
```

```
Hand ins made in time = no
[yes no yes yes]
Final answer: You could easily fail the exam Your previous answer was : true -
Node Id E4

Attend lectures” and “Read textbook” = no and no
[no yes no yes]
Final answer: You could easily fail the exam Your previous answer was : false -
Node Id K4

Attend lectures” and “Make exercises” = no and no
[yes yes no no]
Final answer: You could easily fail the exam Your previous answer was : false -
Node Id C4
```

Decision Tree Implementation Code

```go
package main
/** This code is copies to a large extent from Binary Search Tree implementation
https://appliedgo.net/bintree/ */

import (
  "errors"
)

type Data struct {
  id       string
  answer   string
  question string
}
type Node struct {
  Value string
  Data  Data
  Left  *Node
  Right *Node
}
// Special function that depending on the answer retrun the node to the left or to
the right.
func NavigateNode(n *Node, b string) *Node {
  if (b == "yes") {
     return n.Left
  } else {
     return n.Right
  }
}
```

```go
func (n *Node) Insert(value string, data Data) error {

  if n == nil {
     return errors.New("Cannot insert a value into a nil tree")
  }

  switch {
  //If the data is already in the tree, return.
  case value == n.Value:
     return nil
     //If the data value is less than the current node's value, and if the left
child node is nil, insert a new left child node. Else call Insert on the left
subtree.
  case value < n.Value:
     if n.Left == nil {
        n.Left = &Node{Value: value, Data: data}
        return nil
     }
     return n.Left.Insert(value, data)
     //If the data value is greater than the current node's value, do the same but
for the right subtree.
  case value > n.Value:
     if n.Right == nil {
        n.Right = &Node{Value: value, Data: data}
        return nil
     }
     return n.Right.Insert(value, data)
  }
  return nil
}

// todo change to get
func (n *Node) Find(s string) (Data, bool) {

  if n == nil {
     d1 := Data{"", "", ""}
     return d1, false
  }

  switch {
  //If the current node contains the value, return the node.
  case s == n.Value:
     return n.Data, true
     //If the data value is less than the current node's value, call Find for the
left child node,
  case s < n.Value:
     return n.Left.Find(s)
     //else call Find for the right child node.
  default:
     return n.Right.Find(s)
  }
}
```

```go
//TREE STRUCT
func GetRoot(t *Tree) *Node {
  return t.Root
}

type Tree struct {
  Root *Node
}

//Insert calls Node.Insert unless the root node is nil
func (t *Tree) Insert(value string, data Data) error {
  //If the tree is empty, create a new node,…
  if t.Root == nil {
     t.Root = &Node{Value: value, Data: data}
     return nil
  }
  //…else call Node.Insert.
  return t.Root.Insert(value, data)
}

//Find calls Node.Find unless the root node is nil
func (t *Tree) Find(s string) (Data, bool) {
  if t.Root == nil {
     //d := Data("","")
     d1 := Data{"", "", ""}
     return d1, false
  }
  return t.Root.Find(s)
}
```