Machine Learning:

Decision Tree Algorithm

Will Townsend

Advanced Data Structures and Algorithm Analysis

December 11, 2020

Dr. E.A. Lu

Salisbury University

Abstract

The idea of decision trees are not only useful to business when charting out decision, but programmers use them for many applications including machine learning. The idea can be applied to whether a person is likely to get a loan to a very simple AI algorithm to make decisions based on the environment and/or input criteria (the algorithm utilizes supervised learning not reinforcement learning like other gaming AI). The algorithm uses a tree graph structure, much like binary trees in various other system processes with one big difference. This means it uses much of the same terminology such as root nodes, interior nodes, and leaf nodes and the traversal of the tree is very similar. The applications looked at include the loan applicant example and a simple RPG video game AI which utilize a decision tree toward a machine learning goal. The decision tree algorithm offers a simple and effective tool for machine learning through applications like simple artificial intelligence and some data mining techniques.

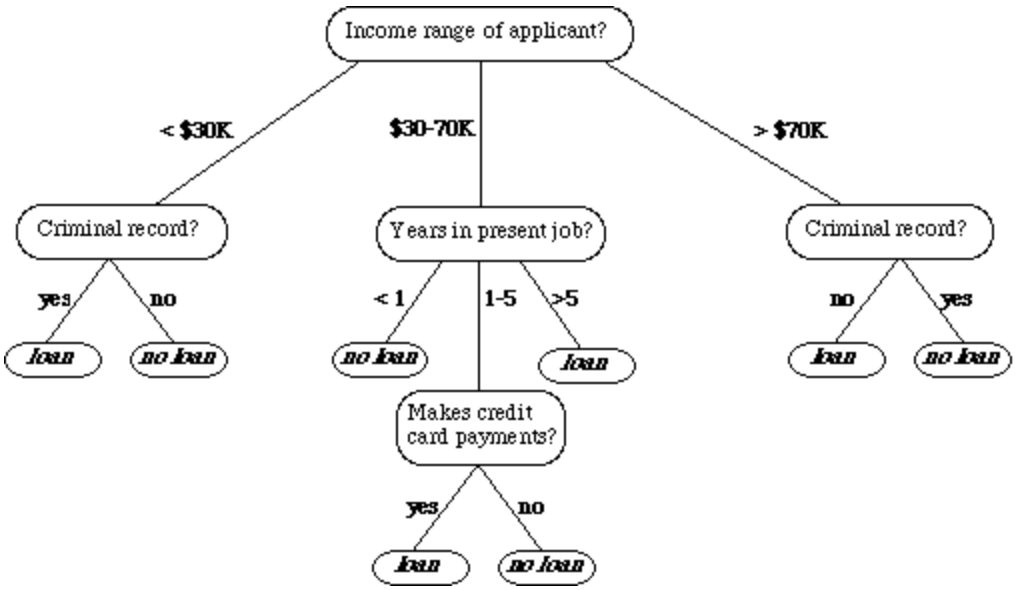
1) Introduction

Machine learning is the study of automatic improvement in a system and has a plethora of applications via different algorithms. There are three different types of machine learning types: supervised, unsupervised, and reinforcement. The simplest of these algorithms the Decision Tree Algorithm. A decision tree is a tool that uses a tree-like model decision that branch to possible outcomes. There are many uses for decision trees including regression and classification type problems: classification referring to discrete outcomes and regression referring to real or continuous. The most general decision trees are known as C.A.R.T or classification and regression trees. These trees are one of the simplest tools for classification and prediction. There are many uses for this simple algorithm/structure including but not limited to very simple artificial intelligence, diagnostics, image classification, and everyday business use. This algorithm is also the basis for many other important machine learning algorithms including but not limited to, random forest and boosted decision trees.

2) General Implementation

The structure to use for decision trees is a tree graph with n children per interior node where n is a constant integer above one. Decision trees' interior nodes and root node can have up to infinitely many children and can vary in the sub trees. In other words, a tree's root node can contain 4 different children while one of those children can only have 2 children. Decision trees must have at least two children. However, when the number of nodes or "decisions" become too large the tree can become muddled. The root node is the initial branch of decisions and can branch into n sub-trees. The algorithm always hits the root node. The interior nodes are the "sub-decisions" and are not always hit when parsing through the tree. The leaf nodes represent the output and final "prediction" of the decision tree.

#1 Application for a Loan



(Cited https://towardsdatascience.com/what-is-a-decision-tree-22975f00f3e1)

Take the tree labeled "Application for a Loan" for example. The tree presented predicts the likelihood of a person gaining a loan based on up to four criteria or inputs: yearly income of the applicant, if applicant has a criminal record, years in current job, and whether the applicant has made their credit card payments. Note that the tree does not consider criminal record for middle income and other factors like home ownership or renting a home etc. So, this tree is not 100% accurate, but it demonstrates the idea of a decision tree using a simplified representation of a loan decision. The root node is represented with the question of yearly income of the applicant. The root node splits into three subtrees based on the income. These interior nodes then take the other criteria into consideration based on the last decision (i.e. criminal record or length in current job). All these trees eventually make their way to the leaf nodes and output the machine's "decision" or prediction based on the search through the tree. The parsing through the tree is the exact same as a binary tree so the time complexity of the tree graph would be the same: O(Depth). This brings up the question of how to actually implement this into a program.

3) Implementing the Loan Decision Tree (C++)

The implementation of a tree graph structure is very similar to that of a binary tree, but each node, again, can have up to n children where n is any integer greater than one. In the picture labeled DTnode.h, notice the one key difference from the implementation of a binary tree: node<T>* center. Again, theoretically the implementation can have up to n number of nodes, but for the purposes of the two examples found in this report three is enough. When the node only requires k number of children instead of n (maximum allowed), the creation of the node looks like: node<T>* node1= new node (T, node2, NULL, node3, node4, NULL…nodeN). When a node does not need to be used, use the keyword NULL to nullify the path.

#2 DTnode.h

```
#ifndef DTNODE
#define DTNODE

#include <cstddef>

template <typename T>
class node
{
    public:
        T nodeValue;
        node<T> *left, *center, *right;
        node(){}
        node (const T& item, node<T> *ptr01=NULL, node<T> *ptr02=NULL, node<T>
        *ptr03=NULL):
            nodeValue(item), left(ptr01), center(ptr02), right(ptr03)
        {}
};
#endif
```

(My Personal Code)

3.1) Constructing the Decision Tree

In order to implement the loan application tree, the tree needs to be constructed. The similarity between the binary tree and decision tree cannot be overstated enough. Just like the binary tree in order to properly construct the tree it must be built from level n to the root based on the depth. In the code labeled createDT(), the number of nodes in the tree is 13 all declared as integer node pointers.

Each node represents the different decisions and outputs of the tree. The outputs are distinguished by their lack of children implemented when created (i.e., nodes n4, n5, n6, n8, n9, n10, n11, n12). The rest are interior nodes with the exception of the node labeled *root*. The numbers attribute to the comparison value. 70 refers to the income, 5 refers to the years in the applicant's job, 0 (with nodes attached) refer to criminal record, and -1 refers to the credit card payments. In a more complex situation, a way to implement the tree could be using structures or objects to demonstrate different variables and the specifics of the node, but for the sake of visualization and learning the integers are the best variable to use.

#3 createDT()

```
node<int>* createDT(){
    node<int>* n12=new node<int>(0);//n7 (False)
    node<int>* n11=new node<int>(1);//n7 (True)

    node<int>* n10=new node<int>(0);//n3 (False)
    node<int>* n9=new node<int>(1);//n3 (True)

    node<int>* n8=new node<int>(1);//n2 years>5 (Loan)
    node<int>* n7=new node<int>(-1,n11,NULL,n12);//n2 1<years<5 (Check CC Payments)
    node<int>* n6=new node<int>(0);//n2 years<1 (No Loan)

    node<int>* n5=new node<int>(0);//n1 False
    node<int>* n4=new node<int>(1);//n1 True

    node<int>* n3=new node<int>(0,n9,NULL,n10);//Criminal Record

    node<int>* n2=new node<int>(5,n6,n7,n8);//Years in Current Job

    node<int>* n1=new node<int>(0,n4,NULL,n5);//Criminal Record?

    node<int>* root=new node<int>(70,n1,n2,n3);//Yearly Income
    return root;
}
```

(My Personal Code)

3.2) Traversing the Decision Tree

Again, the actual traversing through the tree is very similar to a binary tree. Looking at the photo labeled decide() function. The main difference between binary tree traversal and a decision tree traversal is the basis on the specific parameters the decision tree requires. Binary trees typically search through the entire tree, but much like a binary search tree, decision trees only have to parse through one path. So, all the if-else and nested if-else statements determine where on the decision tree the search function currently is to make said "decision" just like a binary search tree uses the current node's value to determine which path to take.

#4 decide() function

```
template <typename T>
bool decide(node<T>* root,int income,int jobLength,bool cr, bool credit){
    if(root->left&&root->right){
        if(root->center&&root->nodeValue==70){
            if(income>root->nodeValue)
                return decide(root->right,income,jobLength,cr,credit);
            else if(income>30)
                return decide(root->center,income,jobLength,cr,credit);
            else
                return decide(root->left,income,jobLength,cr,credit);
        }
        else if(root->center){
           if(jobLength>root->nodeValue)
                return decide(root->right,income,jobLength,cr,credit);
            else if(jobLength>1)
                return decide(root->center,income,jobLength,cr,credit);
            else
                return decide(root->left,income,jobLength,cr,credit);
        }
        else if(root->nodeValue==0){
            if(cr)
                return decide(root->right,income,jobLength,cr,credit);
            else
                return decide(root->left,income,jobLength,cr,credit);
        }
        else{
            if(credit)
                return decide(root->left,income,jobLength,cr,credit);
            else
                return decide(root->right,income,jobLength,cr,credit);
        }
    }
    else{
        if(root->nodeValue==0)
            return false;
        else
            return true;
    }
}
```

(My Personal Code)

3.3) Main and Output

The main of the program is simple in its output. It uses the decide() function in an if statement to output the correct outcome of the decision tree. The user is prompted to enter all the information for the machine to determine whether the applicant is most likely to receive or be denied a loan. A sample output would be something like these below. Keep in mind that the source tree for the code does not take into account criminal record for incomes between 30-70 (inclusive) and the tree is only supposed to display a general example of using decision trees.

Enter Yearly Income (Enter 1=1000k): 63
Enter Years at Current Job: 3
Criminal Record (1=Yes 0=No): 0
Credit Card Payments (1=Yes 0=No): 0
You are most likely not eligible for a loan…

OR

Enter Yearly Income (Enter 1=1000k): 70
Enter Years at Current Job: 5
Criminal Record (1=Yes 0=No): 1
Credit Card Payments (1=Yes 0=No): 1
You are most likely eligible for a loan!

#5 Main

```
int main(){
    bool crim,credit;
    int c,income,years;
    printf("Enter Yearly Income (Enter 1=1000k): ");
    std::cin>>income;
    printf("Enter Years at Current Job: ");
    std::cin>>years;
    printf("Criminal Record (1=Yes 0=No): ");
    std::cin>>c;
    if(c==1)
        crim=true;
    else
        crim=false;
    printf("Credit Card Payments (1=Yes 0=No): ");
    std::cin>>c;
    if(c==1)
        credit=true;
    else
        credit=false;

    node<int>* root=createDT();
    if(decide(root,income,years,crim,credit))
        puts("You are most likely eligible for a loan!");
    else
        puts("You are most likely not eligible for a loan...");
    return 0;
}
```

(My Personal Code)

Overall, this implementation is a very general introduction to the idea of a decision tree that displays the idea of supervised machine learning. The next example will dive into a clearer machine learning aspect of the algorithm by illustrating an enemy in an RPG game (i.e., an enemy AI).

4) Simple A.I. via an RPG Enemy

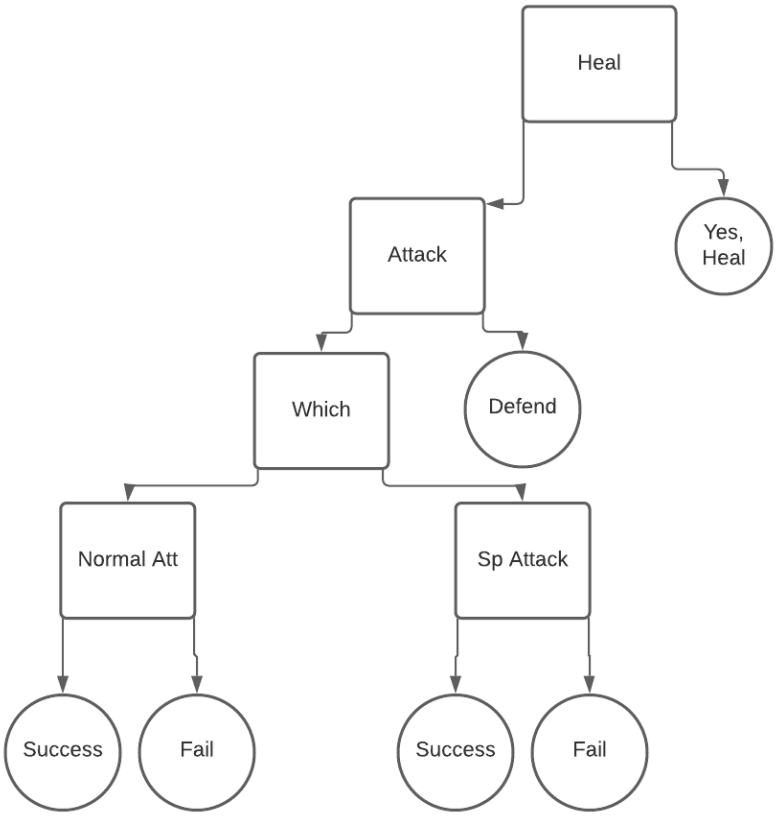
The implementation of this program involves an extraneous header file called player.h that will not be shown in its entirety but will be briefly discussed in relation to the main and enemy.h which is where the decision tree is located. This decision tree in this program is created the minute the enemy object is created in the main (Note: the root is declared in the private members of the class). The creation of the decision tree looks much smaller and simpler (the diagram labeled AI Tree displays the tree visually), but the place where it is more complex is the way the algorithm is designed to traverse through the tree using percentages and changing those percentages based on the status of the user and the enemy itself. Just like the loan applicant tree, the distinguishing factors are the lack of children added on the leaf nodes versus present children on the root and interior nodes. Keep in mind that this is an application for supervised machine learning, so this AI and the decision tree does not adhere to the reward system in reinforcement learning, hence it is a *simple* turn-based RPG AI.

#6 AI Decision Tree

```
//The constructor which builds the Decision Tree
//when the enemy object is created
enemy(){
    health=100;
    node<std::string>* n6=new node<std::string>("Sp Attack");
    node<std::string>* n5=new node<std::string>("Normal Att");
    node<std::string>* n4=new node<std::string>("Defend");
    node<std::string>* n3=new node<std::string>("Which",n5,NULL,n6);
    node<std::string>* n2=new node<std::string>("Attack",n3,NULL,n4);
    node<std::string>* n1=new node<std::string>("Yes Heal");
    root=new node<std::string>("Heal",n1,NULL,n2);
}
```

(My Personal Code)

#7 AI Tree



Circles=leaf nodes, Rectangles=decision nodes (Made using Lucidchart)

4.2) Traversing the AI Decision Tree

The action() function is the traversing function for the enemy decision tree. In the code labeled action(), the number one important feature is the attVal variable which changes the chance that the enemy will attack the user. It changes based on player health (userHealth), its own health (health), and whether the player is defending or not (user). The chart indicates the chance of enemy attack if all variables are true in the row. Also, because the program has nested if-else statements, the chart must be looked at from top to bottom, and once the program reaches a true statement, it does not check the other statements and change the attVal variable. This traversal indirectly utilizes the input values rather than directly like the loan decision tree.

User Health	CPU Health	User Defend?		Attack %	Defend %
>50	<50	N/A	then	20%	80%
N/A	<50	True	then	40%	60%
>50	>50	N/A	then	70%	30%
<50	N/A	False	then	85%	15%
else	else	else	then	50%	50%

The inputs change the chance for a certain path the AI “decides” to take based on those three criteria. This is a blatant example of supervised machine learning. In a more complex scenario, the programmer could easily compile more parameters to look at multiple previous moves made by the player to help the enemy come to a conclusion, perhaps even through another simpler decision tree. The bounds to make this more complex are considerably endless, but this is a simple way to look at supervised machine learning and decision trees.

#8 action() function

```
//This is the bulk of where the Decision Tree works
//it requires the state of the user blocking for
//the attack() and spAttack() functions and the
//int value of the damage done by the user
int action(bool user,int userHealth){
    int random;
    int attVal=50;
    srand(time(0));
    def=false;

    //This is the part of the Algorithm that
    //makes this machine learning taking the user
    //input and modifying the percentage of attack
    //very simply based on user and cpu health and
    //the user's previous move (0 Defended or Healed !0 attacked)
    //in a more complicated setting the computer
    //could be programmed to use the last multiple
    //inputs to come out with an output or plan of attack
    if(health<50&&userHealth>50)
        attVal=20;
    else if(user&&health<50)
        attVal=40;
    else if(userHealth>50&&health>50)
        attVal=70;
    else if(userHealth<50&&!user)
        attVal=85;
```

(My Personal Code)

4.3) Rest of the action() Function

All of the actual traversing down the tree is fairly simple just like before. The only two differences are that this travels through the tree iteratively and using std::string type instead of int type. The randomly generated number from 1 to 100 is the computer's "thinking" of what to do and does this all the way down until it comes to a decision on whether to attack, special attack, defend, or heal. The attVal is used once the computer reaches the node labeled "Attack" where it decides whether to attack or defend based on the random number drawn within that if statement. Once again using a simple parameter instead of a struct or object helps tremendously for the sake of visualization. There is also two static percentages with the healing-10/90 when the health is above 50 and 40/60 when below 50-and missing attacks-always 5/95.

#9 Traversing action()

```
//This part traverses through the Decision Tree iteratively
//and uses the percentages to come to an output/decision
//One could code a tree with objects to better and more concisely
//use the percentages but my implementation does it outside
//to make it more visual
for(node<std::string>* curr=root;root->left&&root->right;curr){
    if(curr->nodeValue=="Heal"&&health>50){
        random=(rand()%100)+1;
        if(random>10)
            curr=curr->right;
        else
            curr=curr->left;
    }
    else if(curr->nodeValue=="Heal"){
        random=(rand()%100)+1;
        if(random>40)
            curr=curr->right;
        else
            curr=curr->left;
    }
    else if(curr->nodeValue=="Yes Heal"){
        heal();
        std::cout<<"Enemy gains 25 health\n";
        return 0;
    }
    else if(curr->nodeValue=="Attack"){
        random=(rand()%100)+1;
        if(random<attVal)
            curr=curr->left;
        else
            curr=curr->right;
    }
    else if(curr->nodeValue=="Defend"){
        std::cout<<"The enemy guards.\n";
        defend();
        return 0;
    }
    else if(curr->nodeValue=="Which"){
        random=(rand()%100)+1;
        if(random>30)
            curr=curr->left;
        else
            curr=curr->right;
    }
    else if(curr->nodeValue=="Normal Att"){
        random=(rand()%100)+1;
        if(random>5)
            return attack(user);
        else{
            std::cout<<"The enemy swung at you but missed!"<<std::endl;
            return 0;
        }
    }
    else if(curr->nodeValue=="Sp Attack"){
        random=(rand()%100)+1;
        if(random>5)
            return spAttack(user);
        else{
            std::cout<<"The enemy went for a special and missed!"<<std::endl;
            return 0;
        }
    }
```

(My Personal Code)

4.4) The Battle

#10 The Battle (My Personal Code)

```
int main()
{
    srand(time(0));
    player user;
    enemy cpu;
    int dam;
    int move=1;

    while(user.getHealth()>0&&cpu.getHealth()>0)
    {
        if(user.getHealth()>0&&cpu.getHealth()>0)
        {
            std::cout<<"============================\n";
            std::cout<<"User Health: "<<user.getHealth()<<" HP\n";
            std::cout<<"CPU Health: "<<cpu.getHealth()<<" HP\n";
            std::cout<<"============================\n";
            printf("User Move %d:|\n",move);
            puts("=============");
            dam=user.actionMenu(cpu.defState());
            cpu.setHealth(cpu.getHealth()-dam);
        }
        puts("");
        if(user.getHealth()>0&&cpu.getHealth()>0)
        {
            std::cout<<"============================\n";
            std::cout<<"User Health: "<<user.getHealth()<<" HP\n";
            std::cout<<"CPU Health: "<<cpu.getHealth()<<" HP\n";
            std::cout<<"============================\n";
            printf("CPU Move %d:|\n",move++);
            puts("=============");
            dam=cpu.action(user.defState(),user.getHealth());
            user.setHealth(user.getHealth()-dam);
        }
    }
    std::cout<<"============================\n";
    std::cout<<"User Health: "<<user.getHealth()<<" HP\n";
    std::cout<<"CPU Health: "<<cpu.getHealth()<<" HP\n";
    std::cout<<"============================\n";
    if(cpu.getHealth()<=0)
        std::cout<<"You Win!\n";
    if(user.getHealth()<=0)
        std::cout<<"You Lose!\n";
    return 0;
}
```

The main is a very simple game loop with a repeated output of health and an option menu for the player under a function, actionMenu(), made in the player.h file. The main utilizes the usual get and set functions in order to find the health and set it since the functions return the amount of damage delt by the user or the enemy. The output looks like the text below. This will keep going until someone's health completely depletes and the program declares a winner. The simple algorithm is able to create a simple RPG enemy and a simulated battle.

```
============================
User Health: 100 HP
CPU Health: 100 HP
============================
User Move 1:|
=============
1.Attack
2.Sp Attack
3.Defend
4.Heal
```

```
What will you do: 2
You use a special attack.
They took 15 damage.
============================
User Health: 100 HP
CPU Health: 85 HP
============================
CPU Move 1:|
============
The enemy guards.
============================
User Health: 100 HP
CPU Health: 85 HP
============================
```

5) Advantages and Disadvantages

Any algorithm and data structure are going to have their advantages and disadvantages and the decision tree is no exception. These factors can come with the speed and efficiency of said algorithms. According to various sources and with my own analysis there are three main advantages and disadvantages all having to do with the relative simplicity of the algorithm and the computative risks and qualities.

5.1) Advantages

First, the simplicity of the algorithm helps to make the rules and parameters understandable and easy to compute. This also make the algorithm easy to modify for various uses and purposes (not just machine learning). Secondly, the idea of the decision tree and tree graphs in general makes it fairly easy to visualize potentially conceptually confusing problems without typing a single line of code into the system. Lastly, this machine learning algorithm, unlike some others can handle both classification and regression problems. Another tiny advantage of the decision tree algorithm is, as said earlier, the time complexity of O(Depth). The time complexity is rather fast for what it can accomplish on a grand scheme.

5.2) Disadvantages

The first disadvantage is that although the idea is simple, if the tree becomes too big, it comes with a computational cost. In other words, the construction of a decision tree can be computationally “expensive” which, on a grand scale, can slow down the programs if the tree is too big. The second is the fact that even though decision trees can solve regressive problems with continuous outcomes, there are better and more efficient algorithms to solve regression type problems. The trees are also more prone to errors when dealing with regressive type problems, and when dealing with precise numbers, this can land a person in trouble.

6) Conclusion

A decision tree is an efficient, simple machine learning algorithm that can lead to more complex algorithms and have many applications for both regression and classification type problems. Much like the binary tree, the tree graph used to construct a decision tree is easily visualized and implemented into a program despite a few drawbacks. This simple but powerful tool is used for so many applications and just shows how simple these cool and useful algorithms can be.

References

https://towardsdatascience.com/what-is-a-decision-tree-22975f00f3e1

https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

https://www.cs.cmu.edu/~bhiksha/courses/10-601/decisiontrees/

https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html

https://www.geeksforgeeks.org/decision-tree/?ref=lbp

https://www.berkeleybridge.com/decision-trees/?gclid=Cj0KCQiA48j9BRC-ARIsAMQu3WRw0VUWh5NZrrCdUvLcIaJHgvU8kUE_RcZv7ORWaAIxSVTIbnXGOiMaAosMEALw_wcB

https://www.wordstream.com/blog/ws/2017/07/28/machine-learning-applications

Appendix

#1: https://towardsdatascience.com/what-is-a-decision-tree-22975f00f3e1

#2: DTnode.h

#3, #4, & #5: generalEx.cpp

#6 & #8: enemy.h

#7: Made with Lucidcart (https://www.lucidchart.com/)

#9 & #10: worldEx.cpp