Hochschule Karlsruhe – Technik und Wirtschaft

Decision Tree Learning

Game AI Coursework

Fabian Manz (52596)

supervised by
Dr.Patrick Glauner

September 25, 2019

1 Introduction

Decision Trees are tree graphs which are used to make and visualize decisions with machine learning through test examples. Fields of application are for example to visualize the behaviour of an artificial intelligence or to perform any risk analysis.

The present report describes the theory and implementation of decision tree learning. A Python program was implemented based on the algorithm discussed in class.

The next chapter deals with the idea and mathematical principles of decision tree learning (section 2). On this basis, the results of an implementation in python will be discussed (section 3). The last chapter is about a summary and possible further improvements (section 4).

2 Methodology

This chapter describes the methodology of decision tree learning. The goal is to learn decision processes from an initial state which are leading to explicit results. If you have a decision tree and you want to reach a specific result, you could look up the decisions you have to choose, to reach this result.

To learn a decision tree, there are specific information needed, called attributes and examples. *Attributes* are (in this report binary) conditions, which will represent states in the decision tree. An *example* is a result through different combinations of attribute values. In Table 1 is an example set of data which could be input data for a decision tree. For example the attribute combination *rainy & warm & high & strong & warm & change* (example 3) leads to the result + (true).

	sky	air	humid	wind	water	forecast	attack
1	sunny	warm	normal	strong	warm	same	+
2	sunny	warm	high	strong	warm	same	+
3	rainy	warm	high	strong	warm	change	-
4	sunny	warm	high	strong	cool	change	+
5	sunny	warm	normal	weak	warm	same	-
6	rainy	warm	high	strong	warm	change	-

Table 1: example set of Attributes and results from last exam

Following the learning process will be explained by a specific algorithm which searchs in each state for the most important attribute by using the entropy theory. This algorithm does not focus on learning the smallest possibile decision tree.

The learning process is recursevly and is called at the beginning and everytime the tree splits. First of all the most important attribute must be found. If the learning process just started, the most important attribute will turn into the tree root, otherwise into the root of a new branch. The most important attribute is defined as the attribute with the biggest *Information Gane*.

The entropy function (in the information theory) calculates the value of which amount of randomness an event has. A high value of entropy means a high chance of uncertainty. The following function calculates the entropy **H** of an binary event that will come true with the propability **q**.

$$H(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$$

The Information Gane **IG** of an attribute **A** is the entropy of the probability that the result (**Goal**) will get positive in general, minus the *Remainder* of the attribute.

$$IG(Goal, A) = H(\tfrac{p_{Goal}}{all_{Goal}}) - Remainder(A)$$

The Remainder of an attribute is the summary of all entropies from each value the attribute contains, from **k** (index which represents the current value) to **d** (count of all values in the attribute). Each specific entropy has to be multiplied by the propability that **k** appears in the set of all values **E** of the attribute, which is the cardinality of the value **k** in **E** divided by the cardinality of **E**.

$$Remainder(A) = \sum_{k=1}^{d} \tfrac{E_k}{E} H(\tfrac{p_k}{all_k})$$

As mentioned before, after the calculation of the information gane of each attribute, the attribute with the highest information gane will become the root or sub tree root of the tree. The tree will now split. Each specific value of the attribute is an *action* which will recursively start the algorithm again. For the recursive call, all examples that not contain the action value will be deleted from the list of examples. For example if a tree learning from Table 1 has the root *sky* and a branch with the action *sunny*, then the algorithm would start without the rows *3* and *6* which are belonging to the action *rainy*. The action *rainy* will be executed on a diferent branch in the same *level* as *sunny*. The level represents the depth of the tree. The

root Attribute will be deleted of the set of Attributes for all recursive calls descending by this root.

A branch will end splitting if all remaining values are the same (e.g. all values are 1 or 0), this value will be the last output of the branch. If the list of examples is empty, the most common value of the level before will end the a branch.

3 Results and Discussion

For this coursework the algorithm discussed in section 2 was implemented in Python. The program can create a decision tree from a data set with examples and binary attributes. The data structure for the tree is a Python *dictionary*. The keys in the dictionary are the levels of the tree. The values are all tree entries in the specific levels. There are two types of tree entries, dictionaries which are holding attributes and results (1 or 0). A result means, that the tree branch is finished. The attribute name is a key which returns the actions of the attribute. Which action the program executes first is randomly chosen by Python which stores *sets* unordered. However the tree dictionary itself is ordered and can be read level by level from left to right. For testing the algorithm, the data of Table 1 was used as input data. The unformatted output of the decision tree learning function would look as following:

```
{{0: {'sky': ['sunny', 'rainy']},
1: [{'wind': ['weak', 'strong']}, 0],
2: [0, 1]}}
```

Because the attribute *sky* has listed the action *sunny* as index *0*, the branch with action *sunny* will be stored at index *0* in *level 1*. The action *rainy* will be placed in the second index. The output can be formatted like this:

```
root:                              <sky>
actions:                   [sunny]     [rainy]
level 1:                   <wind>      0
actions:           [weak]      [strong]
level 2:           0           1
```

For the implementation there were two packages imported. The *math* package to calculate with logarithm and the *copy* package. At some point it was necessary to do a deepcopy of a list which isn't possible with something like "list_temp = list", because if entries of *list_temp* will be removed, it will also affect the entries of *list*. The problem could be solved by importing the *copy* package and explicity use a deepcopy function.

4 Conclusion and outreach

The algorithm itself was relatively short and easy to implement after the fundamental functionalites were clear. The most difficult part was to come up with a data structure to store the tree and expand it through recursion. Python was useful for implementing the algorithm because it allows dynamic typing. So the attribute values aren't limited to be *strings*, it is possible to use other types like *integers*. This can be accomplished because the algorithm only compares the attribute values which can be done with any type.

The algorithm could be extended to work with examples that can result in more than two states. Furthermore the output could be improved by using a library for visualizing graphs.