Post-training Data Addition to Decision Trees

Jeremy Swerdlow
Earlham College
Richmond, Indiana
jjswerd14@earlham.edu

April 4, 2018

Abstract

Decision trees are one of the most widely used machine learning techniques for classification problems. Current classifiers in real world applications create new decision trees on a regular basis, instead of modifying existing trees. This process is time consuming, adversely impacting the overall performance of the classifiers.

This study focuses on exploring different ways to minimize the time taken for the decision tree to accommodate new training sets. We propose a two-phased algorithm, which modifies an existing decision tree based on the newly available data set. In the first phase, each record from the training set is compared against the existing decision tree, in order to decide whether a new child node should be created for this node or the value of a node in the tree should be updated. In the next phase, the algorithm adjusts the weights of each node based on newly added data to represent the dataset as a whole.

By using this method instead of rebuilding the entire tree, we still gain the benefit of this new data, but reduce the time cost to adding it. While the study is not yet complete, we expect the outcome to be an increase in the accuracy of the model as compared to the original, while experiencing a decrease in the time to make use of the new data.

1 Introduction

One of the biggest phrases in the tech world right now is machine learning. Companies, individuals, and researchers are all striving to develop products and projects that use some form of machine learning. Machine Learning comes in many different forms, but generally falls into three types of problems: supervised, unsupervised, or reinforcement learning. Of these, the area of focus for this research is in supervised learning.

Supervised learning models can handle many different problems, but one of the most common is classification. When given a set of data with labels attached to it, these models can provide labels for data that do not have one already. There are many different types of model which can perform this task, such as neural networks and support vector machines. For the purposes of this research however, the focus is on the decision tree.

Decision trees are an effective and easy machine learning model. Their design is easily understood, is relatively fast to train, and is proven to be effective. However, with the rise of Big Data, the massive collections of data people wish to use for their projects, training the decision tree models has become time consuming.

In the past, decision tree algorithms have followed a system process similar to the one presented in Figure 1. It consists of several phases, which can be split into three major categories: data gathering/preparation, training, and testing. When all of these have been completed, the model is ready for production.

However, this system does not allow for any newly gathered data to be utilized. After the initial phase of gathering and preparation, the

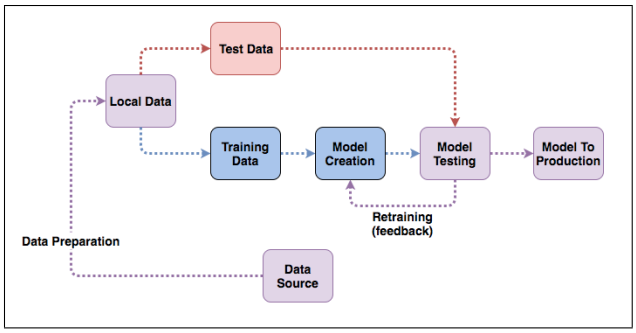

Figure 1: System diagram for the creation of supervised machine learning models

model is trained on just those records, tested, and finished. It does not implement any way to update the tree. This means the model must be retrained with the old data and the new. As this process takes a significant amount of time and resources, many choose instead to delay creating the model until

It is on this aspect of the system this research focuses. We propose a method to add data to a pre-existing tree. The expected result of this method is an increase in the model's performance on various metrics, while decreasing the time required to gain the benefits of the data. To summarize, the presented method will allow for new data to be utilized by the model, without requiring the costly process of recreating the tree to occur.

2 Background

2.1 Machine Learning

The field of machine learning is a subset of artificial intelligence (AI), which has been around almost as long as modern day computers. In 1950, Alan Turing invented the Turing Test to determine whether a computer was intelligent.[**?**] The idea a computer could learn, grow, and adapt to a given situation has existed since then, though what it means for a computer to be "intelligent" has greatly changed over time. As computers became more complex and could solve more difficult problems, the definition of a problem which would mean a computer was intelligent became more and more difficult. However, recent changes in the ways the problems were considered and in computational ability have led contributed to giant leaps forward, pushing the boundaries. In the 1990s, machine learning became a focal point of AI. This meant a shift from a focus on knowledge and rules, to a focus on learning from past examples, to learning from data. Instead of trying to learn rules and apply those, machine learning is more focused on examining copious amounts of data, and identifying patterns and rule within them.

2.2 Decision Tree

As the focus of this research revolves around the decision tree, it is important to discuss the history of the structure, as well as current versions in use. As mentioned in the introduction, decision trees are a type of supervised learning model. This requires that the data being used to train the model is labeled into the different categories which are trying to be guessed. The structure of the decision tree is based off of the branches of a tree, such that at each split, the set of data is divided by some value such to maximize the ability to classify results by the labels associated with each training sample. In one of the centerpieces to the teaching of AI, Stuart J. Russell and Peter Norvig present pseudocode for a generic algorithm to how a decision tree can be built. This algorithm, shown in Fig. 2, works recursively, splitting the data at each level before recursively attempting to split at lower levels.[**?**]

function DECISION-TREE-LEARNING($examples, attributes, parent\_examples$) **returns** a tree

  if $examples$ is empty **then return** PLURALITY-VALUE($parent\_examples$)
  else if all $examples$ have the same classification **then return** the classification
  else if $attributes$ is empty **then return** PLURALITY-VALUE($examples$)
  else
    $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE($a, examples$)
    $tree \leftarrow$ a new decision tree with root test A
    for each value v_k of A **do**
      $exs \leftarrow \{e : e \in examples$ **and** $e.A = v_k\}$
      $subtree \leftarrow$ DECISION-TREE-LEARNING($exs, attributes - A, examples$)
      add a branch to $tree$ with label $(A = v_k)$ and subtree $subtree$
    return $tree$

Figure 2: Pseudocode for the creation of a decision tree (*Artificial Intelligence*, 702).

This generic algorithm has been extended and developed further in several ways, both in terms of improvements made to the basic design, and in actually creating functional code to construct and use decision trees.

2.3 Implementations of the Algorithm

There have been many implementations of decision trees, across multiple languages. For the purposes of this research, Python will be used for its relative ease of programming, and wide variety of available packages to help support the work.

Within Python, there are some standard versions which are widely used. Of these, the most notable comes from the scikit-learn package, where the `DecisionTreeClassifier` is defined.[?] It provides a full range of methods to train, test, and predict results from a set of data.

2.4 Improvements to the Algorithm

In addition to actually creating workable version of the algorithm, many developers have sought to improve its predictive ability, or timeliness, through a variety of creative projects. One of the first and most simple improvements was that of the ID3 Algorithm. Developed first by Ross Quinlan, it has been implemented by many. As Peng et al. describe in *An Implementation of ID3 Decision Tree Learning Algorithm*, it employs "a top-down, greedy search through the given sets to test each attribute at every tree node" in an effort to minimize the depth of the tree.[?] By trying to make as small a tree as possible, it becomes more general, and faster to predict labels, due to the fewer levels of node to work through.

Another popular avenue of improvement has been towards gradient-boosted trees, ones which attempt to find the best tree, but may not always do so. The concept of boosting is covered well in Freund and Schapire's *A Short Introduction to Boosting*, which was published in 1999.[?] In Python, this method is implemented in various packages. Chen and Guestrin discuss one such package, XGBoost, in their presentational paper on it, *XGBoost: A Scalable Tree Boosting System.*[?][?]

However, these improvements do not focus on what to do with the model after it has been created. It considers each tree to be discrete, and separate from other ones that will be created or have been created.

This research attempts to break this idea, and realize that many trees are solving the same problem, using similar or overlapping sets of data. We can redirect the focus from making new trees, to updating preexisting ones.

3 Methodology

3.1 New System Flow

The first step to developing an algorithm was identifying where new data should come from, and how to bring it into the model. For the simple testing purposes, this step just involved splitting the data into three sets, instead of the standard two of training and test. We can mimic the same behaviors used for data preparation in this step, and bring data formatted in the same way to the model. This new diagram is demonstrated in Fig. 3.

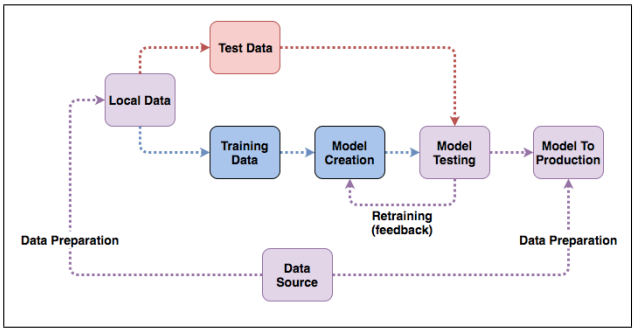

Figure 3: System diagram for the creation of decision trees with post-training data addition

By streaming the data directly from the source to the production model, we can bypass the timely cost of creating the model, yet still have a tree to use and add the data to. This way, the expensive process is only completed once, and then the new algorithm is used in place of it. With this model in mind, we can move onto the actual development of how the algorithm should work.

3.2 Algorithm

The actual algorithm works similarly to the one for creating decision trees. Its pseudocode is

found in Fig. 4. Starting from the root node, the algorithm recursively walks down the tree, checking at each node for several cases. At each node, there are two possible paths. The algorithm checks to see if one of the children matches the attribute of the data sample. If it does, it recursively calls to that node, checking if it is a leaf and updating the leaf based on the result if it is.

The second possibility is that no child node matches the attribute. In this case, a new leaf node is created, with the label of the new data point being the value of the leaf.

The process described above is repeated for each new point of data being added to the model. While this means that there is some significance to order (there are possibly more children to compare against if nodes with attributes that are not included in the original tree are added first), by the time all samples have finished being added, the tree looks the same.

By checking for leaf nodes, and then covering the possibilities of the child existing or not, all possible outcomes from comparing a data sample versus the node's splitter are accounted for.

```
function ADD-ROW-TO-DECISION-TREE(examples, tree) returns a tree

    goal = tree.goal
    dec = tree.decision
    for each value row of examples do
        add row to tree.data
        if row[dec] in tree.branches then
            if tree.branches[row[dec]] is a leaf then
                if row[goal] is equal to leaf value then return tree
                else
                    exs ⟵{e : e ∈ tree.data and e[dec] = row[dec]}
                    tree.branches[row[dec]] = DECISION-TREE-
                LEARNING(exs, tree.remaining_attributes, tree.data)
            else
                tree.branches[row[dec]] = ADD-ROW-TO-DECISION-
TREE(row, tree.branches[row[dec]])
        else
            tree.branches[row[dec]] = row[goal]
    return tree
```

Figure 4: Pseudocode for adding new data to a preexisting decision tree

3.3 Python Implementation

To be continued...