Applying the ID3 decision tree classifier to the mushrooms data set

Authors:
Yoeri van Bruchem - s1013393
Mick van Hulst - s1013954

January 2018

Abstract

This project implemented the ID3 decision tree algorithm on a mushroom dataset. Using the generated tree, predictions were made regarding mushrooms being poisonous or edible. This algorithm was implemented by Yoeri van Bruchem and Mick van Hulst. The algorithm was implemented in Python, using the course notes and description of Saed Sayad as a baseline. The tree is created by recursively using either a pre- or post-pruning approach (also called the training phase). After this phase, the test data set is classified using the generated tree. The algorithm classified the test data set perfectly using post-pruning.

1 Introduction

This report describes the algorithm that Yoeri van Bruchem and Mick van Hulst have developed during their project for the data mining course at the Radboud University. The algorithm has been developed using the programming language Python.

The algorithm that was developed is called the ID3 decision tree algorithm. The algorithm is used for generating decision trees. In the case of this project, the focus lies on handling discrete variables only, meaning that the algorithm cannot handle continuous variables.

The code for this project can be found at the following link: `https://github.com/mickvanhulst/dm_id3_dt_project`. The README file in the GitHub repo explains the steps the user has to take to use this algorithm. This report will refer to the important parts in the code using the following notation: (file name, line numbers). Where an example could be: (main.py, 100-110). For this example the report would refer to the GitHub repository mentioned above, the file main.py and the line numbers 100 to 110.

2 Problem

The type of problems the ID3 decision tree algorithm tries to solve are supervised classification problems, meaning that if one has a set of training data for which the classes are known, one can train a decision tree with that data. Then when new data is gathered, classifications can be performed by using the previously generated decision tree.

3 Data set

During this project a mushroom data set has been used for the decision tree algorithm. The mushroom data set contains information of over 8,000 species of mushrooms. Each mushroom species is classified as either poisonous or edible and has various other characteristics that can be used in determining the edibility of the mushroom [2]. The algorithm will use the characteristics to classify records of mushrooms as either poisonous (p) or edible (e).

4 Related previous work

The related previous work that has been used can be split into two categories, where the first category of sources was used as background information to fully understand the theoretical/mathematical concepts behind the algorithm. This first category of previous work consists of the following sources:

- The lecture slides of the course were used as a baseline, meaning that it was used to find the corresponding formula's and the global process of the algorithm.
- The video's of Victor Lavrenko were used to learn more about the process of training and testing the decision tree [5].
- To implement post-pruning the web page of Saed Sayad was used. This created insight in the use of error estimation [4].

The second category consists of several previously developed projects that were used as example code. These projects were carefully analyzed and used to optimize the way of writing code for the algorithm. The projects are listed below:

- Jason Brownlee implemented a decision tree algorithm in Python using the Gini index. Although the Gini index wasn't implemented, the code gave insight in how to recursively call functions within Python [1].
- Gabriele Lazaro also implemented a decision tree in Python. In contrast to Jason Brownlee, she used the Information Gain. The code gave insight in the way of calculating the Information Gain [3].

5 Algorithm

The following subsections explain the ID3 decision tree algorithm and the process of splitting the data, training the decision tree and classifying the data set. These subsections also explain the mathematical concepts behind the algorithm and the variety of scenario's that may occur when running the algorithm.

5.1 User input

The first part of the algorithm consists of processing the inputs of the user. The algorithm requires the following inputs:

- The *data set* the user wants to use. The algorithm begins by splitting this data set into three groups. A training, test and validation set (ratio: 40/30/30). The training set is used to establish the tree, the validation set is used for post-pruning and the test set is used to test if the resulting model performs well or not (where 'well' is defined as the accuracy of classifying unknown records correctly or not).
- The *features and class label* of the data set. Using these values, the algorithm knows which features it can use to build the tree and which column (class label) it must use when comparing the predicted classes to the actual classes.

Using these inputs the algorithm proceeds to the training phase where it uses the features, data set and the class label to build the tree.

5.2 Training

During the training phase, the user can set several parameters which will determine how the decision tree will handle itself. The user can choose between post- and pre-pruning and can also decide if he/she wants to apply a maximum tree depth.

The algorithm starts by recursively calling the function that is responsible for generating the tree ('decision_tree.py', line 88-130). This recursive function uses the training data, the list of features and the class label as an input. Using this data, the algorithm determines the Information Gain for each feature at a particular split. The algorithm splits using the feature that has the maximum Information Gain.

$$GAIN_{split} = Entropy(p) - (\sum_{i=1}^{k} \frac{n_i}{n} Entropy(i))$$

- $Entropy(p)$, the Entropy of the parent node.
- Parent node is split into k partitions.
- n_i is the number of records in partition i.

$$Entropy(t) = -\sum_{c} p(c|t)log(p(c|t))$$

- $p(c|t)$ is the relative frequency of class c at node t.

After choosing which feature to split on, the same function is recursively called, using the remainder of the data and features yet to be split on as its inputs.

As mentioned above, the algorithm uses several inputs regarding pre-pruning, post-pruning and maximum tree depth. These inputs have an impact on how the algorithm handles itself during the training phase. The subsections below explain the impact of these inputs.

5.2.1 Pre-pruning

Pre-pruning is used to evaluate if a split is significant or not. This significance is determined by comparing the Information Gain of the previous split to the Information Gain of the to be made split. If the new Information Gain is lower or equal to the previous Information Gain, then the corresponding part of the tree gets pruned. In such a case, the algorithm will calculate the amount of classes at that particular split and return the class that occurs the most (i.e. the dominant class). This dominant class will be the leaf of the corresponding branch.

5.2.2 Max tree depth

The max tree depth is a predefined value which can be set by the user. This parameter decides how many layers of branches the tree can consist of. When the tree hits this corresponding tree depth, the tree will be pruned. In such a case the algorithm will calculate the amount of classes at that particular split and return the class that occurs the most (i.e. the dominant class). The most dominant class will be turned into a leaf.

5.2.3 Post-pruning

Post-pruning is a method that is used to avoid overfitting in previously build trees. Overfitting occurs when the decision tree algorithm fits the training data in a way that the tree would be inaccurate while predicting the class of the untrained data. Before post-pruning, the tree is first created completely, meaning that the tree only stops looking for newer features when the amount of classes left in the corresponding branch is equal to one [4].

Post-pruning consists of several steps:

1. Using the validation set, a validation tree is created. The validation set contains the original class. Using this, the algorithm can determine for each leaf if the instance of the validation set is classified correctly. This

step will result in a validated tree with a set of values consisting of the number of correctly and incorrectly classified instances.

2. Using the previously obtained information the algorithm applies the error estimation formula to estimate the error for each node ('post_pruning.py', line: 167-182). If the total error (calculated using the ratio of each of the leaves and it's corresponding error) under a node is greater than the error of that given node, then the algorithm makes the decision to cut that part of the tree off. The node will become a leaf with the most dominant class (of it's original sub leaves).

   The error estimation formula consists of the following formula and corresponding components: $e = (f + \frac{z^2}{2N} + z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}})/(1 + \frac{z^2}{N})$

   - f is the error on the training data.
   - N is the number of instances covered by the leaf.
   - z score from the normal distribution.

3. Now that the tree is pruned, the algorithm sanitizes the tree. Sanitation consists of removing all validation information and removes this with the corresponding classes. This step is necessary so that the remainder of the algorithm can recursively classify the test data.

5.3 Classification

Classification of the test data is also performed recursively. The corresponding part of the algorithm takes one row of the test data set and then recursively goes through all values of the corresponding features until it hits a leaf ('main.py', line: 132-159). If that's the case, then a class has been found.

After classifying all the test data, the predicted classes are compared to the actual classes, resulting in the accuracy.

6 Results

Using pre-pruning, the algorithm only needed one feature to classify around 98% of the test data correctly (see Figure 1). This feature is called 'odor'. Other than that, the post-pruning algorithm didn't prune anything, meaning that the algorithm didn't find a point in the tree which could be removed so that the error estimation would improve (see Figure 2). Post-pruning classified the test-set perfectly with an accuracy of 100%.

Figure 1: Result pre-pruning with 'odor' feature

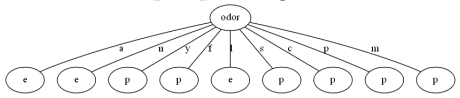
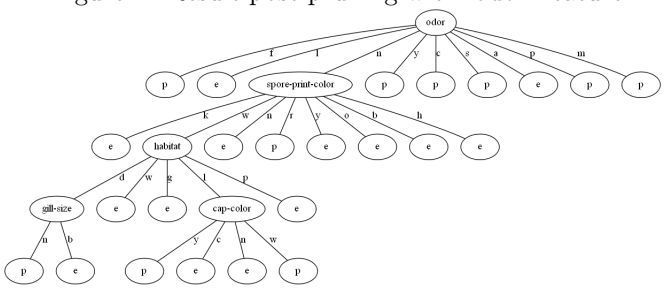


Figure 2: Result post-pruning with 'odor' feature



Another test was performed by removing the feature that seemed dominant (feature with name 'odor'). When using pre-pruning this resulted in a larger tree (see Figure 3), but in the decrease of the accuracy (around 94%). Using post-pruning it once again resulted in not removing any part of the tree (see Figure 4). Furthermore, the accuracy remained the same compared to the previous test (100%). To ensure that the post-pruning algorithm worked, the algorithm was tested with an extreme hardcoded error input. This resulted in an empty tree, proving that the algorithm does prune the tree when the error rate was significant.

Figure 3: Result pre-pruning without 'odor' feature

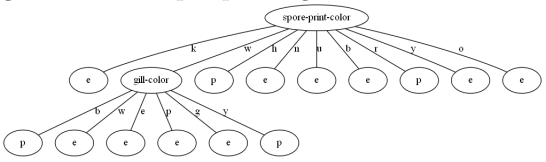
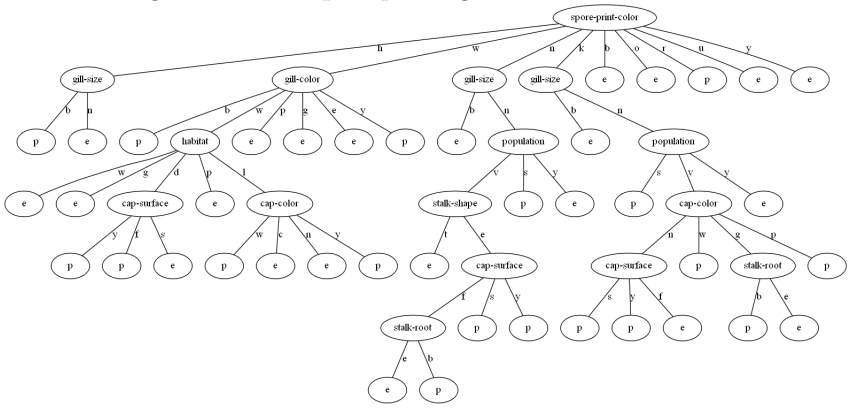


Figure 4: Result post-pruning without 'odor' feature



The results gave form to two hypotheses:

- The feature 'odor' could be a dominant feature, meaning that this single feature significantly increases the accuracy of the algorithm.
- The data has a low amount of variety. This hypothesis was formed when the tree wasn't pruned at all when applying post-pruning. Normally, not removing any part of the tree would result in an overfitted tree. In this case the test set was classified perfectly, showing that the tree wasn't being overfit.

Due to the nature of the project (i.e. implementing an algorithm), the hypotheses weren't tested. This will be further discussed in the improvements section.

7 Analysis results

This specific section will be discussed in the 'we' form as it consists of our own opinions. As a team, we expected this algorithm to be quite a challenge to implement. Looking at our code base we think we put in a lot of effort to make this project a success. The results did surprise us somewhat as we didn't think that one of the features would have such a big impact on the generated tree and on the accuracy itself (i.e. the feature with name 'odor').

Overall, we're very satisfied with the result and feel like we've learned a lot throughout this process. Implementing an algorithm from scratch instead of using a package like Sci-Kit gave us a new understanding on the several concepts that we've used. It also resulted in us respecting the developers of such packages even more, as we realized that these packages make our lives a whole lot easier.

8 Improvements

After completing the development of the algorithm, the team of two students considered the following improvements:

- The algorithm could be further tested by using a different data set. As mentioned, it seemed that there is a low amount of variety in the data set. Testing it on different data sets would give room to test whether or not removing part of the tree when post-pruning has a positive impact on the overall accuracy.
- Implementation of features like boosting which could possible increase the accuracy of the algorithm significantly.
- Extending this implementation to work with continuous variables. The data set that was used did not contain any continuous variables, so the feature wasn't implemented. For an algorithm to perform for any data set, this feature would have to be included.

References

[1] Jason Brownlee. How to implement the decision tree algorithm from scratch in python, Aug 2017.

[2] Holt. Mushroom analysis, Jun 2017.

[3] Gabriele Lanaro. Implementing decision trees in python, Mar 2016.

[4] Saed Sayad. Decision tree - classification.

[5] victorlavrenko. Id3 algorithm: how it works, Jan 2014.