IMPLEMENTING C4.5 DECISION TREE ALGORITHM FOR MEDICAL DATA MINING

by

Keelin Sekerka-Bajbus

B00739421

CSCI 4144 Data Mining and Warehousing

Dalhousie University

Halifax, Nova Scotia

April 10, 2022

1 Introduction

The magnitude of medical information records has dramatically increased in recent years, simultaneously allowing computing to play a more prominent role in clinical settings. Medical researchers and healthcare professionals increasingly use information technologies to collect large quantities of information, waiting for further analysis and knowledge discovery [1]. Within the framework of Knowledge Discovery in Databases (KDD), data mining serves as a valuable process to systematically identify patterns in many real-world applications like medicine [1]. Specifically, data mining clinical data through the analysis of medical records and research data has the potential to aid researchers in identifying disease development and symptom patterns to predict health condition diagnoses accurately and efficiently [2]. Such pattern identification is particularly helpful to researchers in clinical medicine and pharmaceuticals [2].

Within the problem of clinical data mining, mining for pattern discovery in the scope of disease and health conditions as a classification problem is a core issue tackled by the field. In particular, we consider the work of Jacob and Ramani [2] in using data mining algorithms to identify classification patterns in clinical medicine to define the scope of this project. In their research work, they applied 20 classification algorithms to the University of California Irvine (UCI) Machine Learning Repository [3] clinical medicine datasets to develop an efficient and accurate framework for illness classification [2]. The Mammography, SPECT-Heart disease, Thyroid disease, Orthapaedic abnormality, and Dermatology infection datasets were employed in developing this framework [2,3]. Among the 20 algorithms employed, Jacob and Ramani [2] explored the performance of Quinlan's [4] C4.5 algorithm, a decision tree classifier with proven success in clinical classification problems [2].

In this project, we implement Quinlan's [4] C4.5 decision tree algorithm and follow Jacob and Ramani's [2] suit in applying this classifier to the UCI medical datasets [3]. In particular, we will use the UCI Thyroid dataset, the largest of these datasets with 2800 training records [3], for the core data mining experiments in our project**.** Specifically, the *allbp* dataset is employed, which contains 2800 training instances and 972 test instances [3]. The algorithm is implemented using the Python programming language, and its performance upon experimentation with the UCI datasets is compared to that of Jacob and Ramani's [2] results. We also look to analyze the algorithm's performance by taking a closer look at the generated decision trees for medical data mining.

2 Algorithm

The C4.5 algorithm is a greedy, divide-and-conquer decision tree algorithm created by Ross Quinlan as an extension to the preceding ID3 decision tree algorithm [5]. The algorithm is known for its robust performance accuracy and speed in machine learning and data mining applications for classification tasks [6]. Decision tree algorithms like C4.5 use training tuples with a specified class label to recursively generate a decision tree; a data structure made up of a series of nodes and leaves [5,6].

In constructing the decision tree, the algorithm is initialized with a set of training data tuples, each of which is made up of a collection of discrete or continuous values specified as attributes in addition to the class label [6]. At each node in the decision tree, a statistical test is conducted on the attributes to choose the 'best' attribute to proceed with to grow branches from the node, breaking the data into further partitions until a class label can be effectively assigned at a leaf [2, 5, 6]. In the case of the C4.5 algorithm, the attribute selection criterion at each node uses the *information gain ratio* to select the highest scoring attribute upon statistically testing each attribute at a given node [2]. We note that the preceding ID3 algorithm uses only *information gain* for the attribute selection criterion, a measure that is known to be biased in multi-output

test cases [5]. With this, C4.5 is designed to overcome the observed bias through normalization by using a *split information* value in computing the gain ratio [5].

The derivation of the attribute selection criterion comes from information theory, looking to minimize the amount of information needed for tuple classification and to reduce partition impurity in building a decision tree [5]. The *information* of an attribute *A* for a set of tuples *D*, partitioned into *v* subsets, is given by

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$ [5, eq. 8.2]

where

$$Info(D) = -\sum_{i=1}^{m} p_i \cdot \log_2(p_i)$$ [5, eq. 8.1]

is the entropy and p_i is the probability that a tuple in D will belong to a class C_i given by

$$p_i = \frac{|C_{i,D}|}{|D|}$$ [5].

Thus, the *information gain* reflects the change in information required for tuple classification once the set is partitioned by *A*, given by

$$Gain(A) = Info(D) - Info_A(D)$$ [5, eq. 8.3].

In the case of C4.5, the *information gain ratio* is given by

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$ [5, eq. 8.6]

where

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2(\frac{|D_j|}{|D|})$$ [5, eq. 8.5]

is the split information value that provides a normalization for the ratio [5].

Looking now to the decision tree construction algorithm as shown in Figure 2.1 below, we see that the general pseudocode follows a recursive, divide-and-conquer strategy with three termination conditions. The algorithm follows a greedy strategy, meaning it will look to make the best local decision without backtracking [5,6]. First, the algorithm will create a new node, then it will check for the early termination cases. It checks the class frequency of the data partition *D*, and returns the node *N* if all tuples are from the same class *C*. If the set of candidate attributes is empty, *N* will be returned as a classification leaf by the algorithm with a class label assigned by computing the highest class frequency in *D* [5]. Next, the algorithm proceeds with the attribute selection testing by computing *the information gain ratio* as necessary in the case of C4.5, iterating through each attribute and then labelling the node with the top scoring attribute outcome. Once the node is labelled, the attribute is removed from the attribute list. Then, based on the outcomes of the splitting criterion (if multi-way splits are permitted), partitions are generated to grow their respective subtrees through recursion. If the partitions are empty, no recursion will occur and a classification leaf will be attached to the node before the method returns *N* [5, 6].

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition, *D*.

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C, then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let Dj be the set of data tuples in D satisfying outcome j; // a partition
(12)     if Dj is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(Dj, attribute_list) to node N;
     endfor
(15) return N;
```

Figure 2.1. Decision Tree Generation Algorithm Pseudocode. Adapted from [5, Fig.8.3]

3 Data Preparation and Algorithm Implementation

3.1 C4.5 Algorithm Implementation

For this project, our implementation uses the Python programming language. It employs Python's *collections* and *math* modules [7] and the *NumPy* [8], *pandas* [9] and *sklearn* [10] libraries for various data processing and computation tasks. Two classes were defined to build out the decision tree codebase: *Node* and *C45Tree*. The *Node* class implements a simple data structure to store information about the decision tree at a given point, storing details such as the tree depth, current data partition, attribute list, the split criterion, whether the node is a classification leaf, and a list of children. Additionally, it implements simple methods to predict the majority class given the node's data partition using the Python *collections* [7] module and a method to print the node's information. For the *C45Tree* class, the class stores the variables required to construct the decision tree, including a list of tree nodes, the root node, and the full training dataset. The class also implements the core methods required to grow the decision tree and the information gain ratio for the splitting criterion selection process, as shown in the pseudocode in Figure 2.1 above. Additionally, methods for predicting the classification of samples by parsing the decision tree, data partitioning employing the *pandas* [9] library, and a simple method to print the decision tree's nodes are implemented in this class.

3.2 Dataset and Preparation

For our experiments, we applied the C4.5 algorithm to the UCI Thyroid dataset, specifically the *allpb* dataset, consisting of health records for diagnoses related to thyroid hormone health markers [3]. This dataset comprises of 2800 training instances and 972 testing instances with 3 class labels: 'negative', 'increased binding protein', and 'decreased binding protein'. The dataset has a significant class imbalance, with most instances in the training and test sets falling under the 'negative' label. Tables 3.1 and 3.2 below display the full class distribution for the training and test datasets.

The dataset was cleaned using a short Python script to convert the data files to comma-separated value files, as the original files were formatted for parsing by Quinlan's original C4.5 system implementation [3]. This process included removing an unknown sequence of numbers found at the end of each line following the class label in the data and test files. To pre-process the data for our experiments, the train and test data files were parsed using the *pandas* library [9] to convert the data to a data frame to streamline the pre-processing and ensure easy manipulation by the classifier. The index columns of the dataset were removed, and missing values (denoted by '?') were replaced by the most common values in the respective column. We note that there were too many missing values throughout the dataset to consider dropping incomplete instances. Class labels were cleaned by removing extra punctuation from data instances such that the class labels were 'negative', 'increased binding protein' and 'decreased binding protein'. Finally, the training and test data were split into data and label sets for use by the classifier.

Table 3.1. UCI Thyroid Allbp training dataset class distribution [3]

Class Label	Count
Negative	2667
Increased Binding Protein	124
Decreased Binding Protein	9

Table 3.2. UCI Thyroid Allbp testing dataset class distribution [3]

Class Label	Count
Negative	942
Increased Binding Protein	25
Decreased Binding Protein	5

4 Experimental Results

To evaluate the performance of our C4.5 classifier implementation, we conducted several experiments where we considered the training accuracy, the testing accuracy, and the number of nodes and leaves generated. We employed a global seed to ensure experimental results were reproducible. Additionally, the training and testing datasets were randomly shuffled between experiments. We performed three initial experiments, generating three decision trees using a 100-sample subset of the training data, a 500-sample subset of the training data, and the complete 2800 instance training set. The 100-sample tree was tested using 24 unseen samples, the 500-sample tree with 124 samples, and the fully trained tree used the full 972 instance test dataset for validation. Table 4.1 below summarizes the performance of each trained tree. All three trees achieved robust training and test accuracy, all above 95%. We note that the 500-sample tree achieves the strongest performance of the three. The fully trained tree was the only tree to improve its performance accuracy upon testing in these experiments. Table 4.2 below describes the generated trees' structure in terms of the number of nodes and classification leaves. We note that the number of nodes and

leaves in the 500-sample are marginally fewer than that of the 2800-sample tree, which is indicative that the algorithm can efficiently build the fully trained tree despite having significantly larger amount of training data.

Table 4.1. Experimental Results of the C4.5 Implementation

No. of Training Samples	No. of Testing Samples	Train Accuracy (%)	Test Accuracy (%)
100	24	96.00	95.83
500	124	99.00	97.58
2800	972	95.32	96.91

Table 4.2. Experimental Decision Tree Structures

No. of Training Samples	No. of Nodes	No. of Leaves
100	45	21
500	61	29
2800	65	31

Table 4.3. Full C4.5 Tree Trials with *allbp* Data

Trial	Train Accuracy (%)	Test Accuracy (%)
1	95.32	96.91
2	95.29	96.91
3	95.36	96.91
Avg.	*95.32*	*96.91*

Further experiments were conducted using C4.5 trees generated using the full training dataset to validate the preliminary results and ensure that the classifier's performance was reproducible. We generated three C4.5 trees using the full dataset using a separate global seed for each set of experiments. The trees achieved an average training accuracy of 95.32% and testing accuracy of 96.91% over the three trials. These results are very robust and indicate that the classifier can capture the data's characteristics well. The full results from these trials are displayed in Table 4.3 above. From the C4.5 algorithm, the decision rules at a node are generated by the attribute with the highest information gain ratio. A sample rule generated by our C4.5 implementation is given in Figure 4.1 below.

```
T4U <= 0.595 THEN = negative
        T4U measured > 0.595
```

Figure 4.1. Sample Decision Rule Generated by C4.5 Implementation

Looking to Jacob and Ramani's [2] work, where they employed the *discordant* Thyroid dataset from the UCI Thyroid data [3], they achieved 100% accuracy using Quinlan's C4.5 classifier implementation. This dataset comprises 2800 training instances, similar to the *allbp* dataset, with 28 Boolean and continuous attributes [3]. However, this dataset generates a binary classification problem with two target classes in contrast to the multi-class problem posed by the *allbp* dataset. We also note that the C4.5 classifier applied to the *discordant* data generated a tree with 98 nodes, and 50 leaves, much larger than the trees generated by our implementation for a similar dataset. Our experimental results yield

a comparable performance level to the traditional C4.5 implementation within similar medical data mining applications.

5 Conclusion

In this project, we have implemented Quinlan's [4] C4.5 decision tree algorithm from scratch using the Python programming language. We have applied the classifier to the UCI Thyroid *allbp* dataset to explore using this algorithm in medical data mining applications as proposed by Jacob and Ramani in their research [2]. Our experiments with our implementation of the C4.5 algorithm yielded robust results, with the fully trained decision trees producing training accuracies of 95.32% and testing accuracies of 96.91% on average. The fully trained tree produced 65 nodes with 31 classification leaves, which is comparable to the structure of the 500-sample decision tree generated in the initial experiments. This is indicative of how efficiently Quinlan's [4] C4.5 algorithm can build a decision tree that effectively captures the complexity and characteristics of a dataset. When compared to similar medical data mining experiments conducted by Jacob and Ramani [2], we observe that our implementation achieves comparable, auspicious results. Based on the experiments conducted in this project, it is evident that C4.5 is a very efficient and effective algorithm for data mining, especially in clinical medicine applications. Applying Quinlan's [4] C4.5 algorithm to medical data mining classification tasks, at minimum, to benchmark problem complexity, would be incredibly beneficial in discovering interesting and useful patterns that have the potential to ameliorate healthcare and medical research.

References

[1] S. Aljawarneh, A. Anguera, J. W. Atwood, J. A. Lara, and D. Lizcano, “Particularities of data mining in medicine: lessons learned from Patient Medical Time Series Data Analysis,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, Nov. 2019.

[2] Shomona Gracia Jacob and R. Geetha Ramani. 2012. Mining of classification patterns in clinical data through data mining algorithms. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI '12)*. Association for Computing Machinery, New York, NY, USA, 997–1003. DOI:https://doi.org/10.1145/2345396.2345557

[3] D. Dua and C. Graff, “UCI Machine Learning Repository”. University of California, Irvine, School of Information and Computer Sciences, 2017.

[4] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, California: Morgan Kaufmann, 1993.

[5] J. Han, M. Kamber and J. Pei, “8 – Classification: Basic Concepts” in Data Mining: Concepts and Techniques, 3rd ed. Morgan Kauffman, 2012, ch.8, pp.327-391. DOI: https://doi.org/10.1016/B978-0- 12-381479-1.00008-3

[6] S. Ruggieri, "Efficient C4.5 [classification algorithm]," in IEEE Transactions on Knowledge and Data Engineering, vol. 14, no. 2, pp. 438-444, March-April 2002, doi: 10.1109/69.991727.

[7] G. Van Rossum, *The Python Library Reference, release 3.10.4*. Python Software Foundation, 2022.

[8] C. R. Harris *et al.*, “Array programming with NumPy”, *Nature*, vol 585, no 7825, bll 357–362, Sep 2020.

[9] J. Reback et al., *pandas-dev/pandas: Pandas*. Zenodo, 2020.

[10] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python”, *Journal of Machine Learning Research*, vol 12, bll 2825–2830, 2011.