CS-330 Project 2020 - Cardiovascular diseases diagnosis

Christophe MARCIOT, Titouan RENARD

2020-05-25

1 Introduction

What is intended in this project is to create a tool capable of predicting whether a patient suffers from a cardiovascular disease given a set of information. The information given for each patient are :

- A variable called *target*, that takes value 0 or 1 indicating if the patient does actually suffer from a cardiovascular disease or not (0 means healthy, 1 means sick). This is the variable whose behavior we try to predict.
- 13 other variables called respectively *sex*, *cp*, *trestbps*, *chol*, *fbs*, *restecg*, *tahlach*, *exang*, *oldpeak*, *slope*, *ca* and *thal* that can take up to 4 different values.

From now on, we will refer to those 14 variables as *attributes* and the numbers taken by those attributes as *values* of the given attribute for the patient. We are provided with a dataset of 143 points for training and 80 points for testing.

2 Discussion of the results

2.1 Task 1

The goal here is to generate a decision tree able to predict the value of *target* for a given patient using a first set of data - given in the file *train_bin.csv* - this is achieved using the *ID3* algorithm as implemented in the exercises. We get the following information about the generated tree :

- It's composed of 94 nodes 70 of which are leaves,
- The size of the tree is 8, which means that the tree will make at most 8 decisions before coming to a conclusion,
- The average number of children for a node is 3.88 - without taking leaves into account,
- The average length of a branch is 3.83. This means that, on average, the tree will make 4 decisions before giving a guess of the value of *target*.

We first notice that the *age* attribute is the most prevalent attribute present in the nodes. This seems actually relevant as one would expect age to be an attribute that minimizes the entropy as older people tend to be more suffering from cardiovascular disease. Also we note that the *fbs* and *oldpeak* attributes are never used in the tree. This allows to say that, in this particular context, these two variables are useless to the construction of a tree using our *ID3* algorithm.

2.2 Task 2

The goal was to evaluate the accuracy of the tree obtained in Task 1 on a second set of data - given in the file *test_public_bin.csv*. Our tree has an accuracy of 56.25% on the testing data. This is obviously a disappointing result, we partly attribute the low accuracy of ID3 on disparities between training and testing data. Indeed, a quick look at the input datasets shows us that the average value for the *target* attribute in the *train_bin.csv* file is 0.2307 whereas it is 0.8250 for the *test_public_bin.csv* dataset. This unbalance in data selection is likely what throws off the ID3 algorithm. This can be shown by shuffling the data of *train_bin.csv* and *test_public_bin.csv* together and then separating them in new training and testing datasets (of the same sizes as the ones provided). Doing this gives on average an accuracy of 69.43%, this shows the way the data is split is "especially bad". We have written an small example program to show this, it is included in the files we are submitting as *shuffled_data.py*.

On the training data, the accuracy goes as high as 100%. Knowing this allows us to use the PAC method of statical analysis to give a lower bound on the number of examples needed to obtain a good tree. As seen in course we have: $N \geq log(\delta/|H|)/log(1-\epsilon)$ where N is the number of examples, ϵ is the error rate we admit for our tree, δ is the probability of having a tree with error rate $\geq \epsilon$ and H is the set of all possible trees. Here, as we operate in a medical context, we would like to set ϵ to be low. Let $\delta = \epsilon = 0.01$. Now we have to estimate $|H|$. Computing an exact value of $|H|$ is a hard task because some attributes in this problem can take up to 4 values, but for some, such as *sex* and *fbs*, it can take less (only 2 for the mentioned attributes). Here we will assume that all attributes can take up to 4 different values for the upper bound and 2 for the lower one. We get: $8.28 * 10^5 \leq \frac{log(0.01/|H|)}{log(0.99)} \leq 1.82 * 10^9$.

This allows us to say that with $N \geq 1.82 * 10^9$ we know that we have a probability of 0.01 that the algorithm gives us a tree with an error rate lower than 0.01, and that N must at least greater than $8.28 * 10^5$ for it to be the case.

2.3 Task 3

We want to predict the value of *target* using only the aforementioned rules and to use them to give a justification of the prediction. We use a *DFS* algorithm on the tree generated in Task 1 and generate rules based on the nodes covered in its path. Doing this provides one rule per leaf, that is one rule per path. The rules generated are of the form $[(\text{`attribute}_1\text{`}, \text{`?}x\text{`}, \text{`value}_1\text{`}), \ldots, (\text{`attribute}_k\text{`}, \text{`?}x\text{`}, \text{`value}_k\text{`})] \Rightarrow (\text{`}target\text{`}, \text{`?}x\text{`}, \text{`guess`})$ where '?x' denotes a variable.
To make a deduction from those facts, we use forward chaining while giving to the engine initial facts of the form $[(\text{`}age\text{`}, \text{`patient no } k\text{`}, \text{`}age(\text{patient no } k)\text{`}), \ldots, (\text{`}thal\text{`}, \text{`patient no } k\text{`}, \text{`}thal(\text{patient no } k)\text{`})]$. We format the deductions of the inference engine to display the diagnosis in a more readable way as can be found in the file *Task3_ data.txt*. Those deductions are made from the test data set.
We notice that one of the guesses cannot be done by the inference engine (patient no. 61). This is due to the fact that there are no rules that can explain this case. The *classifie* method of the *NoeudDeDecision* class classifies it by simply affecting it to the predominant class. This did not seem to us to be a useful justification, therefore we chose to produce no deductions using the inference engine.

2.4 Task 4

In this part, we implement a tool capable of prescribing a treatment when necessary. We assume that a treatment is necessary for a given patient if the diagnosis proposed in Task 3 concludes that $target = 1$ *or* if the procedure in Task 3 could not draw any conclusion.
We produce a treatment by modeling this as a problem of diagnosis by induction. Here, we assume that each value to each attribute can be considered as defective. For a given patient, we use a *BFS* algorithm on the following tree:

- To each node is associated a *change*. A change is a list of attributes and values such that the value associated to the given attribute is different from the value associated to the same attribute and the patient.
- The initial node is associated to the empty change.
- The children of a given node are the changes that are just the change of the current node to which we add an attribute that is not already present in it with a value that is different to the value associated to the given attribute and the patient.
- When reaching a node, we diagnose the patient whose information are updated with the values present in the change. If the diagnosis returns $target = 0$, we then consider that the change is a treatment. Also we use the rule used from diagnosis to justify the treatment. If the diagnosis returns $target = 1$ we then queue up the children of the current node in the queue - as it is usually done in *BFS*.

Restricting us to changes of size smaller or equal to 2, we see that we can actually treat all patients that were diagnosed to be ill. In fact we notice that there are usually more than 400 changes available of size smaller or equal to 2 and thus it is not really surprising to see a treatment to be possible. Also we have designed a function that is able to represent the diagnosis and the treatment, as well as the justifications for both of them. A list of all diagnosis and treatments cases be found in the file *Task4_ data.txt*.

2.5 Task 5

In this part of the project, we modify the ID3 algorithm so that it is able to deal we continuous instead of binned values data attributes. Unlike the previous implementation, the new algorithm produces binary trees. At each node, patients are separated in two subsets according to a threshold value on the attribute that maximizes : $[H(C) - H(C|A)]/values(A)$. This threshold value is selected to minimize the entropy of the target value after partitioning.
Because of the small size of the dataset, we did not implement a root-finding algorithm, we simply try every value in between two existing values in the dataset for the attribute we are evaluating. That being said, such an algorithm would definitely be worth implementing if we were to consider significantly larger datasets.
The performance of the continuous diagnosis algorithm is slightly but not significantly better than discrete one. We get an accuracy of 58.75% on the testing dataset. As for task 2 we think that this is partly due to the way the data-points are separated between the *test_ public_ continuous.csv* and *train_ continuous.csv*.

3 Supplementary results

3.1 Task 2 bis

In this supplementary section, we tried to implement the *random forests* algorithm. We generate trees with randomly chosen subparts of the original training data set and then consider only the ones that can treat the whole original training data set. While testing the trees, we also compute their accuracy on the original data set.Then we can use 3 different classifiers. Here we implemented the *BESTTREE*, the *MAJORITYVOTE* and the *ADABOOST* algorithms. We describe them the following way :

- *BESTTREE* algorithm : Choose the tree with the greater accuracy on the training data set and use it as a classifier,
- *MAJORITYVOTE* algorithm : Consider the guesses of all the trees generated and then gives a guess corresponding to the majority of the guesses,
- *ADABOOST* algorithm : Consider only the trees with a probability of error smaller than 50% - as weak methods - and construct a classifier using the *ADABOOST* algorithm presented in class.

We tested the accuracy of these algorithms on the test data set by generating 1000 trees from a randomly chosen subsets of the training data set. The size of the subset is determined by the value of *subsampling*. For a value n of subsampling, we get that the subsets are of size $\lfloor 143/n \rfloor$. As the randomness plays a big role in the computation, we computed the accuracy 100 times and considered the mean of these accuracies. The means of the computed accuracies are given in the following table.

subsampling/algorithm	*BESTTREE*	*MAJORITYVOTE*	*ADABOOST*
2	≈ 0.53	≈ 0.50	≈ 0.53
3	≈ 0.51	≈ 0.47	≈ 0.53
4	≈ 0.48	≈ 0.42	≈ 0.55

We clearly see that the *BESTTREE* and *MAJORITYVOTE* algorithms perform as good as or worse than the expected accuracy of guessing randomly, which allows us to conclude that they are practically useless when the subsampling is greater than 2. We also notice that while both *BESTTREE* and *MAJORITYVOTE* perfom worse when the subsampling gets greater, the performance of *ADABOOST* does not change. As the trees are generated from smaller training sets every row, it seems that on average, those trees get less effective in their guesses. The impact is visible on *BESTTREE* and *MAJORITYVOTE*. On *ADABOOST* we see that despite the augmentation of the subsampling value, the accuracy stays the same. We conclude that the *ADABOOST* algorithm is in fact the better one. This said we also have to observe that generating a tree with *ID3* is still the better solution.