Artificial Intelligence: Assignment 2

April 15, 2016

Prof. Andrea Torsello

Francesco Pelosin

Learning Problem

Definition

Given a set of observations, where each of them has a set of attributes and a correspondent labeling, a learning algorithm tries to inductively learn the underlying phenomenon which causes the classification of the observations, then it store that knowledge in a structured model. The model is then used to predict the classification of new objects, by looking to their features.

Setting

We have three basic entities in the learning process:

- *Training set :* this is the set of examples that the learning algorithm exploits to create the knowledge model, these examples are observations of the phenomenon that we want to study. Each observation is characterized by a set of attributes and a label or classification.

$$example = \underbrace{a_1, a_2, \dots, a_n}_{features}, \quad \underbrace{c_1}_{classification}$$

- *Learning algorithm :* the algorithm that creates the predictive model from the provided training set, it exploits informations and relationships by observing the examples and then it create the predictive model.
- *Test set :* this is the set of examples toward which we test the trained model. One important thing is that the examples that are in the training set, should not be used for the testing of the classifier, otherwise the output has no meaning, since examples do not represent new observations which can be useful to measure the error of the predictions.

Information Theory P.O.V. An interesting point of view comes from the Information Theory perspective, we can say that the classification process is the analogous inference process which the receiver has to perform during the channel communication between two entities, and the relative learning phase relies in the creation of the knowledge model during different communications. Figure may visualize better the problem.

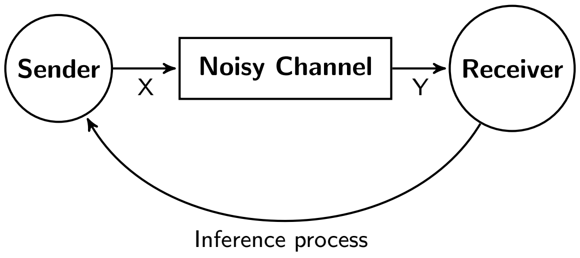

Figure 1: The learning phase is the construction of the model during different communications, and the classification is the analogous inference problem.

One of the main issues during the learning phase, is that the receiver can only see the observations after passing throughout the channel, and since each transmission of data is affected by the noise of the channel, the receiver can only see spurious data which does not represent the phenomenon in a neat way. If we do not handle this problem, the model which the algorithm creates will not have real and clean knowledge of the phenomenon, but it will simply "interpolate" the observations, this is usually referred as overfitting. So to construct a good classifier through a learning algorithm, we should take in account this fact and minimize the learning of the noise that affect data.

Overfitting

If we take the observations and create a learning algorithm that only best fits the examples, we are in fact learning also the noise present in data. The fact is that the model will badly generalize when tested to new observations due to its perfect training and this problem affects not only decision trees (topic of the assignment), but all the classification algorithms based on observations of data. Figure 2 and Figure 3 may help visualize the problem.

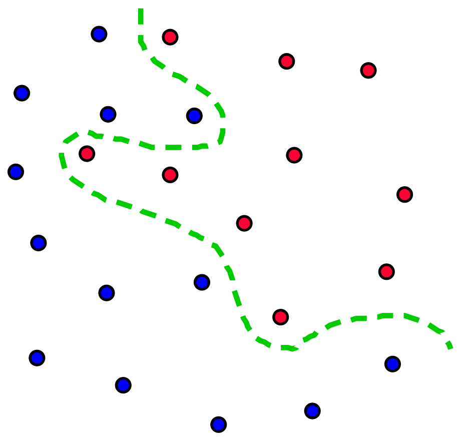

Figure 2: Overfitting, the classification doesn't generalize very well

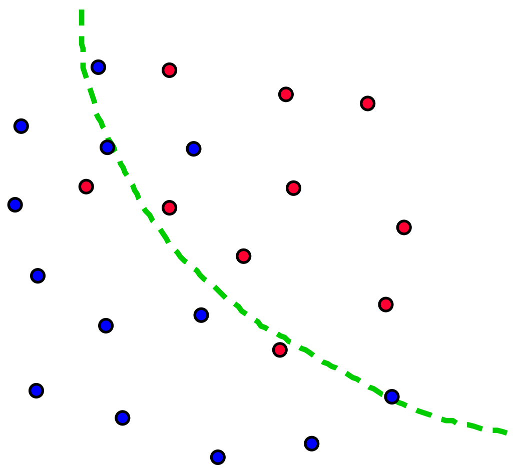

Figure 3: Classification might generalize better than the previous case

There are several ways to minimize this problem, in the case study section I will briefly cite some techniques that can be applied to decision trees in order to better interpret the phenomenon.

Underfitting

The same opposite situation may happen when we have a too small set of examples toward which we train our algorithm. In this case, we will end up with a model which has a poor knowledge of the underlying phenomenon, with poor performances when tested on new data because of its "ignorance". This can be avoided by using a reasonable amount of data to train our algorithm, but the problem now is providing a measure to "reasonable", obviously it depends from the intrinsic complexity of the problem, but now we should be able to measure the complexity of a problem, but in general: the bigger the training set, the better will be the classifier.

Learning Trees

One well known learning algorithm is the **ID3** algorithm (Quinlan '86), which relies on the decision tree structure. The ID3 takes in input a training set and ends up with a decision tree to represent the knowledge. Each decision node is a test for a particular attribute of the observation, at the bottom there are the leaf nodes that represent the predicted classification for the branch, so by starting from the root we have a set of decision nodes that conduce to a classification. It is a good algorithm for classification and, like all other learning tree algorithms, it has the nice property to be readable since the tree structure can be printed and interpreted by humans, but ID3 has some big limitations:

- It does not handle **continuous attributes** unless they can be "discretizable" in some way, maybe after a preprocessing step where we introduce some abstraction of the data. For example if we have a continuous attribute such as the percentage of the humidity in the air of a particular day - which varies from (0 to 1) - we should sub-devide it in some discrete values, maybe (low, medium, high). The loss of information during this transformation depends from the level of granularity which we use to discretize the data, obviously if the algorithm could handle continuous attributes it would gather more knowledge of the phenomenon.

- It does not handle **missing values**, so for each example, we must have every attribute specified. In real scenarios, missing values are frequent, so even if it does not provide direct information about the process, it could be useful to handle them to further classify the examples. To overcome this limitation, there could be some pre-processing step which tries to guess the correct value of the missing attribute, based on the other examples.

These two limitations are very restrictive in real problems, that's why there are several better algorithms, the evolution of ID3 is **C4.5** (Quinlan '93), which handles the two issues an can also extract some rules from the tree. Another famous algorithm is the **C.5** which in particular has better performances than C4.5 and has several improvements too.

Case study : Heart Disease

Dataset preprocessing

The heart disease dataset from the UCI repository that we used in our case study, is divided in four different files:

- `procesed.cleveland.data`
- `procesed.hungarian.data`
- `procesed.switzerland.data`
- `procesed.va.data`

In each file there are both numerical and discrete attributes, in addition there are missing values too. The classifications are 0 for a negative labeling, 1,2,3 and 4 stands for a positive labeling and respectively they quantifies the gravity of the disease. For simplicity we will consider 1 every classification which is different from 0, so there will be only positive and negative classes this is only to simplify the problem.

Normalization. Each missing value is represented as a `?` character, I decided to replace all that char occurrences with a -9 (because no other attribute has negative values), just because my implementation is in C# so I needed some uniform structure to store data.

67	0	3	130	263	0	0	97	0	1.2	2	1	7	2
28	1	2	130	132	0	2	185	0	0	?	?	?	0

...

Table 1: There are 13 attributes some discrete some continuous, the last column represent the classification, as you can see there are missing values too

After the preprocessing step, we ends up with something like that:

67	0	3	130	263	0	0	97	0	1.2	2	1	7	1
28	1	2	130	132	0	2	185	0	0	-9	-9	-9	0

...

Table 2: The previous two examples after the preprocessing step

Shuffling. The interesting thing about this dataset, is that each of them collect respectively data coming from different geographical regions. So each dataset represent the heart disease phenomenon in a particular population. In order to train the algorithm I could have merged the four files and use it as it was, but I preferred **randomly permutate** the lines, of the merged file, before use it to train the algorithm.
The motivation is that the examples from different places are not independent, because each geographical region has his own lifestyle (and maybe some examples are relatives too), so being in the same geographical region could influence the training phase.

For example if we train the algorithm in a dataset created from people of the same country with high percentage of obesity (say U.S.A.), the generated tree will be very good when asked to predict examples from the same region, but if I test the same model with a testset that comes from a different country (say Japan), performances might be not so good. So to better generalize the results and understand the phenomenon in a transversal way, I decided to randomize data.

The reason behind this is that people from the same region have more relations that constrains the phenomenon and if we remove these assumptions we are implicitly extending the model to predict "someone in the world has heart disease if" rather then "U.S.A. people has heart disease if" . This extension might require more data to train a good classifier, but I think it better capture the reason why we are studying the problem.

Data Split. After the shuffling step and the normalization, I've split the set into two main files:

- `crossvalidationset.data` - which contains the set toward which we will perform the crossvalidation and the final training step, it has 810 examples which correspond to 88% of the whole dataset.
- `testset.data` - which contains the set toward which we will perform the test step, it contains 110 examples which correspond to the remaining 12% of the whole dataset.

Classifier Implementation

Decision Tree. The tree structure is the knowledge representation model that we are going to create trough our algorithm, it has two kind of nodes:

- *Leaf node* - simply represent the predicted classification, so in our case we have `[Yes]` and `[No]` nodes which correspond to labeling 1 and 0.
- *Decision node* - tests a particular attribute with a given value and provide several branches to follow to eventually arrive to a decision node. Basically each decision node represents a "classification rule" that has been learned during the training phase.

So to test a new example, we just have to pick it and look to its attributes, follow the decision nodes, choose the right path and arrive to a leaf node.

Learning Algorithm. The implementation of the algorithm is an improved version of the ID3, what our implementation extends is that it handles continuous attributes and missing values. The basic structure is the same: it exploits the notion of *Entropy* between examples and in particular selects of the most discriminant attribute left, by calculating the *Information Gain.* The main role of the algorithm is to create the tree which will represent the inferred knowledge and specifically the parameter of the function are :

- *examples* - represents the training examples, used to build the tree. At each iteration of the algorithm, this set is restricted in order to eventually arrive to have a base case and finish the training process.
- *attributes* - represents the actives attributes i.e. not already used. At each iteration if the best attribute is discrete, it is removed, if it is numerical then we do not remove it, since it still has information to exploit.
- *default* - it's a default leaf node that is used in the base case, where there are no examples left.
- *depth* - represent the current depth of the tree. It is used to stop the algorithm during the crossvalidation phase.

```
Require: Initialization of the attributes list and a list of examples
 1: function DecisionTreeLearning(examples, attributes, default, depth)
 2:     if examples is empty then
 3:         return default
 4:     if depth = maxDepth then
 5:         return MajorityValue(examples)
 6:     if all examples has the same classification then
 7:         return a leaf node with the classification
 8:     if attributes is empty then
 9:         return MajorityValue(examples)
10:     bestAttribute = ChooseAttribute(examples, attributes)
11:     tree = decision node with attribute bestAttribute
12:     m = MajorityValue(examples)
13:     if besAttribute is discrete then
14:         for all values of bestAttribute do
15:             $examples_v$ = subset of examples which has value $v$
16:             subTree = DecisionTreeLearning($examples_v$, attributes−bestAttribute, m, depth+1)
17:             add subTree to tree in a branch with value $v$
18:     else
19:         $examples_{preSplit}$ = subset of examples which are part of the presplit
20:         $subTree_{pre}$ = DecisionTreeLearning($examples_{preSplit}$, attributes, m, depth+1)
21:         add subTree to tree in a branch with value split
22:         $examples_{postSplit}$ = subset of examples which are part of the postSplit
23:         $subTree_{post}$ = DecisionTreeLearning($examples_{postSplit}$, attributes, m, depth+1)
24:         add subTree to tree in a branch with value split
25:     return tree
```

Algorithm 1: ID3 improved

The MajorityValue function and the ChooseAttribute correspond to the classical implementations. The first one takes as input a set of examples and return a leaf node with classification equal to the majority of the labeling among the examples, the ChooseAttribute simply calculates the Information Gain for each attribute and choose the one with the highest value, in our case we extended this method to handle numerical attributes.

Handling Numerical Attributes. The new improvement from the ID3 algorithm is how we treat continuous attributes, this paragraph will explain what are the *postSplit* and *preSplit* variables cited in the algorithm. These kind of attributes do not have a small prefixed amount of feasible values, they have a great variability in the values and potentially they could be infinite. So a basic way to calculate the information gain is to :

1. Reorder the examples by their attribute value.
2. Split examples incrementally and calculate the Information Gain between the two splits.

0.1	‖	0.12	0.3	0.6	1.3	1.34	1.4	1.7	1.89	$\rightarrow InformationGain_1$
preSplit	‖	postSplit								

0.1	0.12	‖	0.3	0.6	1.3	1.34	1.4	1.7	1.89	$\rightarrow InformationGain_2$
preSplit		‖	postSplit							

$\dots$

0.1	0.12	0.3	0.6	1.3	1.34	1.4	1.7	‖	1.89	$\rightarrow InformationGain_n$
preSplit								‖	postSplit	

Table 3: Information gain in continuous attributes

3. Take the split that maximize the Information Gain and store the value to compute decision subtrees.

Although it seems correct to handle numerical attributes in this way, there is a side effect because they are always preferred to discrete attributes, numerical attributes can take a potentially very large set of values, it is simple to see that in the limit case, they can act like "identifiers" providing the maximum information. To overcome this problem, one of the solution is to substitute the Information Gain with **Information Gain Ratio** which simply reduces the bias towards numerical attributes.

Handling Missing values. In order to handle missing values I basically treated them like a separate value for the attribute. The fact that we use missing values does not automatically mean that results will be better, but if exploited in a correct way they might provide additional information. There are several ways to handle them, some very simple methods ares:

- Remove examples with missing values, this is maybe the best possible way to get a neat solution, but we are losing data which could be exploited and another problem arises when we have a very uncertain dataset. How do we handle this situation? Surely we can't remove everything.
- Assign the most common classification between the examples with missing values.
- Distribute them among all the subtrees.
- Treat them as another value, and this is what my implementation does. So in the final tree there will be some decision subtrees with missing value as roots.

Generalize the model. When we perform the training phase, we might face the problem of overfitting of data. As we discussed, overfitting decrease dramatically the predictive capacities of the model, so to better generalize, we must find a way to "extend" the model. There are different ways to achieve such goal:

- **Early-stop** - One rough method is to early stop the learning phase, maybe to some depth of the tree. This is not so effective but still limit the chances to overfit the data. In my implementation I've used this method, and the choice of the depth to which stop the algorithm is made by observing the results of the crossvalidation phase, we will cover this in the next paragraph.
- **Pre/Post-Pruning** - Pruning some branches of the tree might enhance the accuracy, there are different ways to implement this.

Crossvalidation and Results. Crossvalidation is a method to evaluate the algorithm that we have implemented, in particular I've used a 10-fold-crossvalidation on a set of 880 examples 88% of whole dataset, so at each step I pick a slice of 88 examples toward which I test the the algorithm that was trained to the 792 remaining examples, and I repeat the process 10 times with different slice of examples. The crossvalidation phase will take 10 training/test phases, after these steps I take the average of the accuracies coming from these output classifiers and this number will "evaluate" the goodness of the algorithm.
It is important to stress that if we only take in consideration the accuracy coming from one cycle *training* $\rightarrow$ *test*, we are actually measuring the performance of a single instance output classifier of the algorithm, not the algorithm per se.

As I mentioned in the previous paragraph, I exploited the crossvalidation phase to decide the maximum depth of our final tree. In particular I trained a classifier on the whole dataset to find the maximum depth that the algorithm could generate and it turns out to be 23. Since I need some generalization of the classifier to avoid overfitting, I ran 23 different depth, 10-fold-crossvalidation cycles to find the maximum average accuracy, and turns out that the maximum average accuracy is reached at depth 8, with 73.8% of accuracy as can be seen in Figure 4, and the maximum single instance accuracy that has been reached, is 83%. During the final training (performed on the 88% of the whole dataset) and test (performed on the remaining 12%) phase, the accuracy of the classifier is 74% which at the end could improve if we use the whole dataset for the a release version.

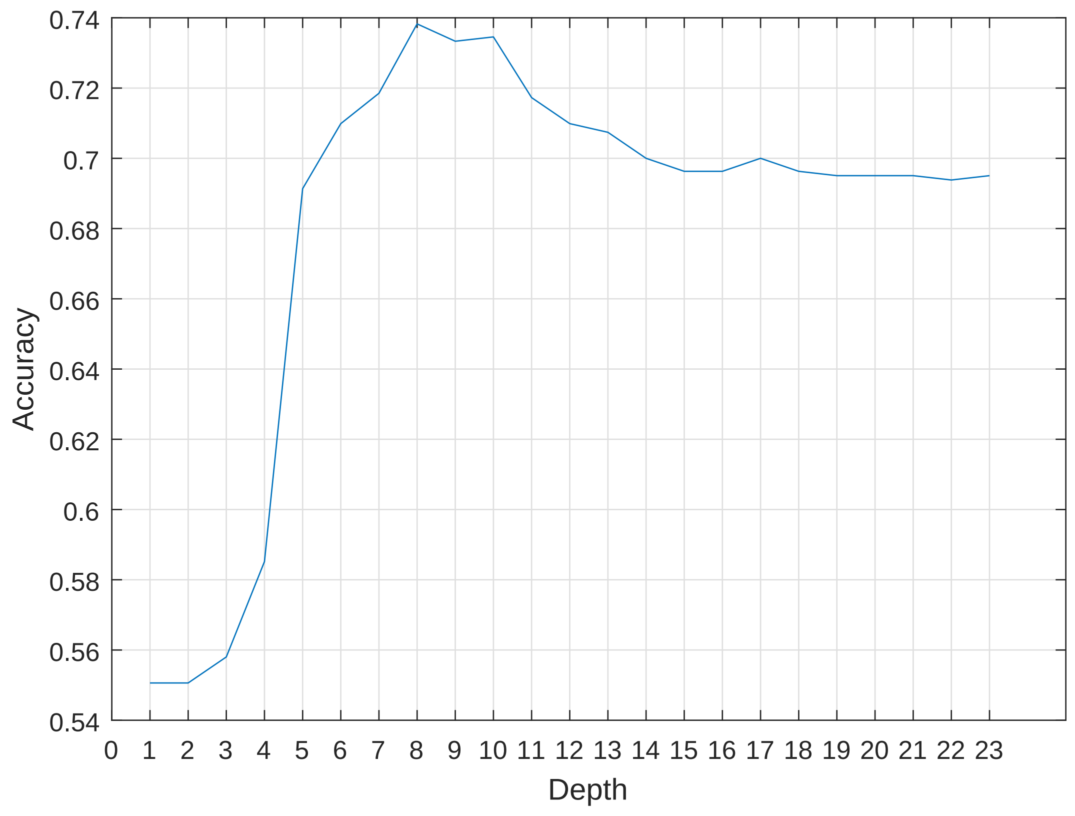

Figure 4: 10-fold-crossvalidation average accuracies for 23 different depth levels

Conclusion

Decision trees seems to me to be fairly good classifiers, although there might me some implementation inaccuracies in my work, results seems pretty good even if I think that in delicate problems, like disease classification and some other critical phenomenons, learning trees might have difficulties to very well predict facts and could not completely substitute an expert on the field. Certainly with more advanced and fine-granular techniques, there might be situation where these kind of classifiers not only can predict correct classifications, but they can visualize new relationships between some events that human may not notice. This small assignment made me realize how big and complex the topic is and provided me enthusiasm to go deeper on the field.

Student: Francesco Pelosin
Email: 839220@stud.unive.it *OR* pelosinfrancesco@live.it
Id: 839220