Assignment 2: Decision Trees Algorithm

H. Blum, D. Cavezza, A. Paudice and M. Rohbeck
Machine Learning CO395
Imperial College London

November 10, 2015

1 Training

Decision trees are a conceptually simple, yet powerful model for solving classification problems. The ID3 algorithm is one of the simplest training algorithms for decision trees: it builds the tree by choosing at each step the attribute that maximizes the *Information Gain*, a measure that represents the discriminatory power of an attribute over the class values.

The problem we are given consists in classifying facial expressions into 6 emotion classes via decision trees; each facial expression is represented as a feature vector of 45 *Action Units* (AU), where each feature is a binary value that indicates the presence or absence of each AU in the example. We decompose the problem in 6 binary decision problems according to the *1-vs-all model*, with each problem corresponding to the recognition of one specific emotion. Therefore, we use ID3 to train 6 different trees, each one trained to recognize one specific emotion.

Our implementation of the ID3 algorithm is provided in the files `train.m`, `DTTrain.m` and `choose_best_attr.m`, containing the functions with the same name.

The function `train` is the entry point for the training algorithm. It takes as inputs the training examples, the array of attribute ids (in our case, an array from 1 to 45), and an array of labels of the same length as the number of examples; such labels are numbers from 1 to 6, each one indicating one of the emotions in the original dataset. It returns the 6 desired trees.

Each tree is trained on a transformed version of the input data: at the i-th iteration of the for loop, the labels are made binary; the vector `binary_targets` contains a 1 in the positions corresponding to the i-th emotion, a 0 in all the other positions. The function `DTTrain` is called to train each tree on such transformed data.

```matlab
function [ T ] = train( examples, attributes, labels )
%Train 6 binary trees on the training set (examples, labels) and return
%them into the tree array T.

[m, n] = size(examples);
binary_targets = zeros(m, 1);

for i = 1:6

    %Switch to binary labels
    index = find(labels == i);
    binary_targets(index) = 1;

    %Train a tree to learn an emotion
    T(i) = DTTrain(examples, attributes, binary_targets);

%    DrawDecisionTree(T(i));

    binary_targets = zeros(m, 1);

end

end
```

The function `DTTrain` is a recursive function implementing the ID3 algorithm. Every time a recursion occurs, a check is performed on the entropy of the current node: if it is not 0, the attribute that maximizes the Information Gain is chosen and associated to the current node, then the recursion occurs.
In the first call it is passed the whole training sample, the array 1:45 of attribute ids, and the binary labels.

```matlab
function [ tree ] = DTTrain( examples, attributes, binary_targets )
%Train a decision tree on the dataset (examples, binary_target)
%according to the ID3 algorithm. The examples have the attributes listed in
```

```
 4 %attributes
```

First, an empty tree is initialized.

```
 6 %Initialize an empty tree
 7 tree.op = [];
 8
 9 tree.kids = [];
10 tree.class = [];
```

Then a check on the input's entropy is performed. If the sample entropy in the labels is 0, the algorithm is in a pure node, that is all the current examples belong to the same class: in such case, the algorithm must stop, and the value returned by the current node is the class value held by the majority of the examples. Likewise, the algorithm must stop if there are no more attributes to decide about. In these cases, `tree.op` and `tree.kids` are not set, so as to mark the node as a leaf.

```
12 if (sample_entropy(binary_targets) == 0 || isempty(attributes))
13     % either pure targets or no more attributes to base decision on
14
15     tree.class = maj_value(binary_targets);
```

Else, the attribute that yields the highest Information Gain is chosen

```
17 else
18
19     tree.op = choose_best_attr(examples, attributes, binary_targets);
```

and the examples are split by the value they contain in the chosen attribute.

```
 1     tree.kids = cell(1, 2);
 2
 3     % raise up the kids
 4
 5     for j = 1:2
 6
 7         %fprintf('#Examples: %d \n', length(examples));
 8         child_index = find(examples(:, tree.op) == (j - 1));
 9         child_examples = examples(child_index, :);
10         child_binary_targets = binary_targets(child_index);
```

At this point, the set of examples with a specific value of the chosen attribute may be empty. In that case, the corresponding kid node is made leaf and the returned class value is the value held by the majority of the nodes in the current node.

```
36         if (isempty(child_examples))
37             % we can't train this child, make a leaf with the majority
38             % value of all training data coming to the parent
39
40             tree.kids{j}.op = [];
41             tree.kids{j}.kids = [];
42             tree.kids{j}.class = maj_value(binary_targets);
```

Otherwise, the corresponding subtree is created. Note that in the recursive calls, the chosen attribute is deleted from the list of candidate attributes, since its value is fixed for the samples in each of the kids (0 for the left kid and 1 for the right).

```
44         else
45
46             % remove the used attribute from the list
47             index = find(attributes == tree.op);
48             attributes(index) = [];
49
50             % recursive training with this child
51             tree.kids{j} = DTTrain(child_examples, attributes, ...
52                 child_binary_targets);
53
54         end
```

At line 19, the above functions calls `choose_best_attr`. This function computes the Information Gain of every available attribute and returns the id of the attribute that yields the highest Information Gain.

```
function [ best_attr ] = choose_best_attr( examples, attributes, binary_targets )
%chooseBestAtt computes and return in bestAtt the attribute in attributes
%which determine the maximum information gain for the set
%(examples, binary_targets). Return -1 if the set is pure because no split
%is required

%Compute the sample entropy
n_ex = length(binary_targets);          %Dataset size
E = sample_entropy(binary_targets); %Sample entropy

% initial values
best_gain = 0;
best_attr = attributes(1); % for now this is as good as anything

for i = 1:length(attributes)

    % examples where attribute is 1
    one_attr = binary_targets(examples(:, attributes(i)) == 1);
    % examples where attribute is 0
    zero_attr = binary_targets(examples(:, attributes(i)) == 0);

    E_partition = length(one_attr)/n_ex * sample_entropy(one_attr) ...
        + length(zero_attr)/n_ex * sample_entropy(zero_attr);

    gain = E - E_partition;

    if (gain > best_gain)

       best_gain = gain;
       best_attr = attributes(i);

    end

end

end
```

Finally, the files `sample_entropy.m` and `maj_value.m` simply contain the computation of the entropy and the most frequent label value in a sample respectively.
The learned trees are shown in Figure 1.

2 Decision making strategies

Since we have different trees, and each of these trees may commit mistakes in its classification, their outputs may be contradictory. Specifically, more than one tree may return 1 in response to the same example, or all trees may return 0. Therefore, we have devised different decision policies to resolve conflicts.

2.1 Random choice

One possible strategy is choosing randomly: if more than a tree returns 1, the label of the example is chosen randomly among the classes corresponding to such trees; if all the trees return 0, the label is picked randomly among all the classes. The implementation of this strategy is in the file `testTreesRandomChoice.m`.

```
function [ predictions ] = testTreesRandomChoice(T,x2)
% TestTreesRandomChoice performs a random selection among the emotions of
% all the trees that return a prediction of 1, or among all the emotions if
% all the trees return 0.

[m,n] = size(x2);
binary_predictions = zeros(m,6);
predictions = zeros(m,1);

for i = 1:m
    for j = 1:6
        binary_predictions(i,j) = predictionBinaryTree(T(j),x2(i,:));
    end
    % If all the predictions are 0, pick an emotion randomly
    if(max(binary_predictions(i,:))==0)
        predictions(i) = randi(6);
    % Else pick an emotion randomly among the 1s
```

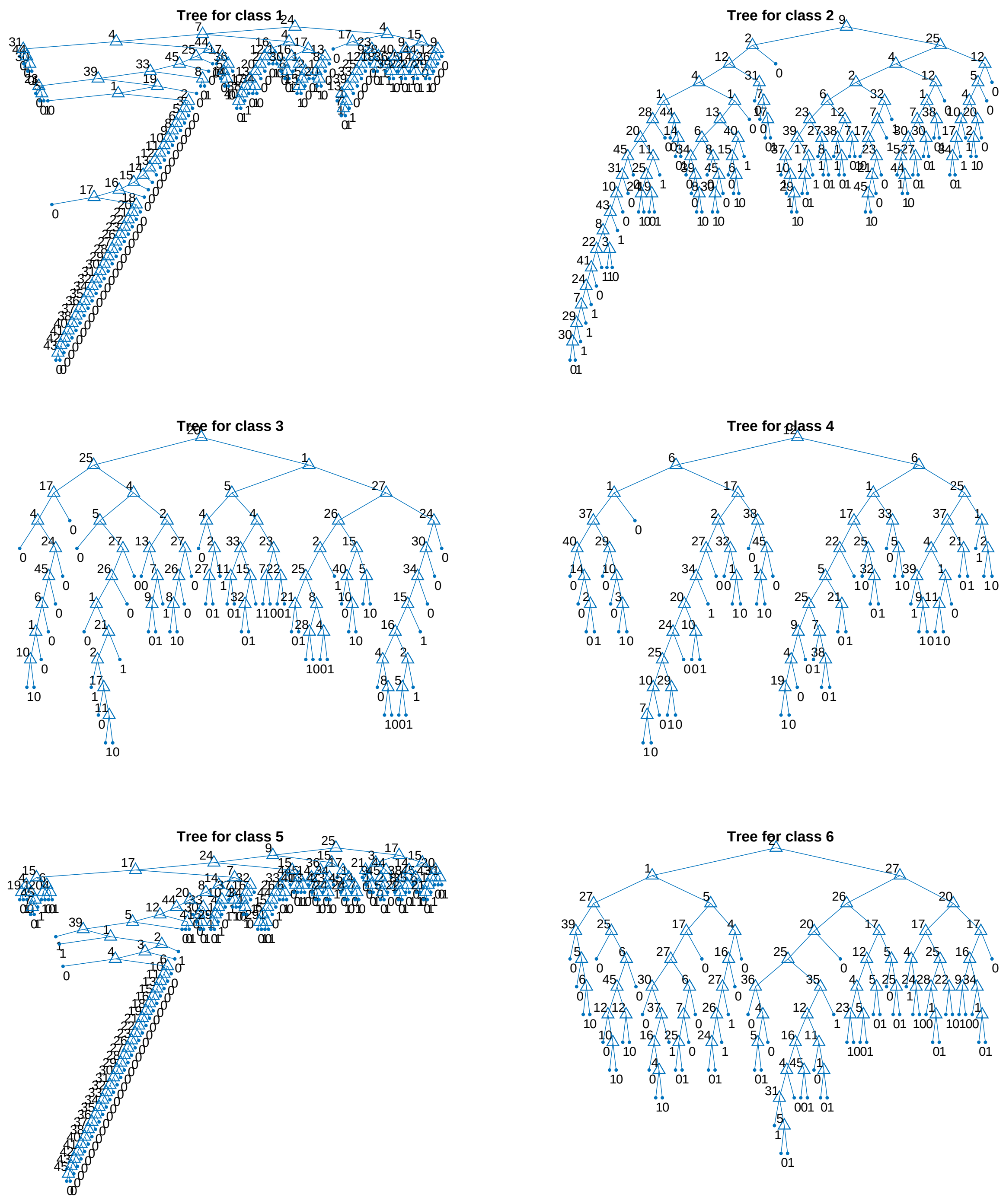


Figure 1: Trees learnt on clean data

```
18      else
19          candidateEmotions = find(binary_predictions(i,:));
20          predictions(i) = candidateEmotions(randi(length(candidateEmotions)));
21      end
22
23  end
24
25
26  end
```

The function `predictionBinaryTree` called in line 12 is defined in the same file: it passes through the branches of the tree corresponding to the attribute values of the input example and returns the binary label corresponding to the reached leaf.

```
28  function [ class ] = predictionBinaryTree( tree, x )
29  %predictionBinaryTree walks along the tree to find the classification of
30  %the istance x
31
32      if (isempty(tree.op))   %Check if it is a leaf
33
34          class = tree.class;
35
36      else                    %Test the attribute
37
38          x_to_test = x(tree.op);
39
40          if (x_to_test == 0) %Follow the left branch
41
42             class = predictionBinaryTree(tree.kids{1}, x);
43
44          else %Follow the right branch
45
46             class = predictionBinaryTree(tree.kids{2}, x);
47
48          end
49
50      end
51
52  end
```

This strategy is one of the simplest to consider in resolving conflicts. It is chosen as our baseline strategy.

2.2 Score-based choice

2.3 Tree confidence choice

With this decision strategy, in case of a tie, we choose the tree which achieved the smallest classification error on the training set. The key assumption here is that a tree with a small classification error represents a confident model, while a tree with a high classification error is less confident. This assumption is justified observing that a tree with small confidence is likely to be a poor model of the true function because of underfitting.

In particular, this function first computes the classification error achieved by every tree on the training set (lines 39-52). Then it computes the predictions made by the trees for each point in the test set (lines 57-65). Finally, it checks whether there is a tie between two or more trees and makes the final prediction picking the tree with the smallest training classification error (lines 68-84). The implementation of this strategy is in the file `testTreesTreeConfidence.m`.

```
28  function [ predictions ] = testTreesTreeConfidence( T, x, y, test_data )
29
30  [m, n] = size(test_data);
31  binary_predictions = zeros(m, 6);
32  binary_errors = zeros(1, 6);
33  confidence = zeros(1, 6);
34  predictions = zeros(m, 1);
35
36  [m_train, n_train] = size(x);
37
38  %Evaluate the error in the training set
39  for i = 1:6
40
41      for j = 1:m_train
42
43          binary_predictions(j, i) = predictionBinaryTree(T(i), x(j, :));
```

```
        binary_errors(i) = binary_errors(i) + (not(binary_predictions(j, i)==1 && y(j)==i)...
            && not(binary_predictions(j, i)==0 && y(j)~=i));

    end

    binary_errors(i) = 1/m_train * binary_errors(i);
    confidence(i) = 1 - binary_errors(i);

end

binary_predictions = zeros(m, 6);

%Compute the predictions of each tree
for i = 1:6

    for j = 1:m

        binary_predictions(j, i) = predictionBinaryTree(T(i), test_data(j, :));

    end

end

%Compute the prediction
for i = 1:m
    candidateClasses = find(binary_predictions(i,:)==1);
    if(isempty(candidateClasses))
        % If all the trees return 0, pick the prediction with minimum
        % confidence. This prediction, indeed, is the most error-prone and
        % most likely its correct value should be 1 (only according to our
        % confidence heuristic, there is no mathematical justification
        % here)
        [~, predictions(i)] = min(confidence);
    else
        % Take the prediction of the tree with most confidence among the
        % ones returning 1
        [~, imaxConfidence] = max(confidence(candidateClasses));
        predictions(i) = candidateClasses(imaxConfidence);
    end

end

end

function [ class ] = predictionBinaryTree( tree, x )
%predictionBinaryTree walks along the tree to find the classification of
%the istance x

    if (isempty(tree.op))   %Check if is a leaf

        class = tree.class;

    else                    %Test the attribute

        x_to_test = x(tree.op);

        if (x_to_test == 0) %Follow the left branch

           class = predictionBinaryTree(tree.kids{1}, x);

        else %Follow the right branch

           class = predictionBinaryTree(tree.kids{2}, x);

        end

    end

end
```

This function makes use of `predictionBinaryTree`, defined above.

3 Cross-validation

Cross-validation is generally used to obtain an appropriate estimate of the performance of a model on unseen data, or to tune the parameters of a model appropriately (i.e. to perform model selection), if the size of the data set at one's disposal is low.
It consists in splitting the data set into k partitions, called *folds*, and testing the training algorithm iteratively on these folds. In each iteration, k-1 folds are used to train the model, while the remaining one is used for testing or validating the trained model. After k iterations, the performances computed in each folds are averaged.
Such partitioning and iterations are necessary for two reasons:

- the performance statistics computed on the same data used for training are not reliable estimates of the generalization power of the learnt model; thus, it is necessary to leave some data outside the training set for testing purposes;
- the performance statistics are more reliable as the test set size increases; iterating over all the folds is a device to increase the number of examples on which the learnt model is tested.

We provide an implementation of *stratified* cross-validation: in this version, the proportion of class values inside each fold reproduces approximately the proportion of class values inside the entire dataset. Our implementation is provided in the five m-files `crossValidate.m`, `stratifySampleIndexed.m`, `getFoldIndexed.m`, `getTrainingSetIndexed.m`, and `getTestSetIndexed.m`.
The function `crossValidate` requires as inputs the attributes and label values of every example, and the number of desired folds; `stratified` is a Boolean parameter specifying whether or not the cross-validation should be stratified.

```
1  function [confusionMatrix, accuracy, precision, recall, fmeasure] = crossValidate(xvalues, labels
       , k, stratified)
```

If `stratified` is `false`, the function just performs a split of the data set into k folds (lines 27-31). Otherwise, the split is performed according to the stratified cross-validation (lines 15-25). First, the dataset is split by class value (line 16); then, each fold is constructed by taking a subfold from the examples of each class, and concatenating such subfolds (lines 18-25).

```
14  if(stratified)
15      % Obtain the example indices divided by class label
16      indicesPerClass = stratifySampleIndexed(labels);
17  
18      for i=1:k
19          % Perform the division in k folds in every class. Merge the
20          % subfolds from all the classes to obtain a single fold
21          for j=1:numClasses
22              foldsIndices{i} = [foldsIndices{i}; getFoldIndexed(indicesPerClass{j},k,i)];
23          end
24  
25      end
26  
27  else
28      % Non-stratified cross-validation
29      for i=1:k
30          foldsIndices{i} = getFoldIndexed(1:length(labels),k,i);
31      end
32  end
```

Notice that there is no randomness in the split: if the input is the same, the folds will not change over different calls.
After the split, the training and test sets are constructed for each iteration, the performance indicators are computed for each fold and the averages are returned. For further details, refer to the indicated files and their comments.

4 Performance

We report the performance of a 10-fold stratified cross-validation executed on each of the decision strategies presented in Section 2. Figures 2-4 show the total confusion matrix (that is the sum of confusion matrices over all the folds) and the average performance metrics over the 10 folds.

The metrics reveal that every approach classifies better clean data than noisy ones in terms of accuracy and per-class precision and recall. Classes 1 and 5 consistently show the worst performances; class 1, in particular, has precision and recall around 25% on noisy data. Class 3 is not recognized well on noisy data in terms of recall.
For details on the possible explanations of such results, see Section 5.1.

$$\begin{bmatrix} 87 & 15 & 6 & 4 & 17 & 3 \\ 16 & 141 & 8 & 10 & 13 & 10 \\ 6 & 7 & 78 & 6 & 11 & 11 \\ 5 & 12 & 4 & 174 & 14 & 7 \\ 14 & 14 & 10 & 11 & 73 & 10 \\ 4 & 8 & 16 & 8 & 8 & 163 \end{bmatrix}$$

(a) Confusion matrix for clean data

Class	Precision	Recall	F_1	Accuracy
1	67.0%	65.9%	65.7%	
2	73.0%	71.1%	71.3%	
3	64.5%	65.6%	64.4%	71.3%
4	82.6%	80.5%	81.2%	
5	54.5%	55.4%	54.4%	
6	80.4%	78.8%	79.3%	

(b) Performance metrics for clean data

$$\begin{bmatrix} 21 & 11 & 18 & 11 & 17 & 10 \\ 14 & 123 & 12 & 16 & 10 & 12 \\ 20 & 14 & 98 & 18 & 12 & 25 \\ 7 & 12 & 20 & 145 & 9 & 16 \\ 13 & 15 & 8 & 5 & 57 & 12 \\ 9 & 11 & 19 & 11 & 14 & 156 \end{bmatrix}$$

(c) Confusion matrix for noisy data

Class	Precision	Recall	F_1	Accuracy
1	26.6%	23.8%	24.2%	
2	67.6%	65.9%	65.9%	
3	56.0%	52.4%	53.9%	59.9%
4	70.6%	69.4%	69.6%	
5	47.4%	51.8%	49.3%	
6	67.9%	70.9%	69.1%	

(d) Performance metrics for noisy data

Figure 2: Performance of Random Choice

$$\begin{bmatrix} 96 & 10 & 7 & 3 & 13 & 3 \\ 15 & 147 & 4 & 8 & 10 & 14 \\ 8 & 3 & 82 & 7 & 7 & 12 \\ 5 & 10 & 3 & 180 & 10 & 8 \\ 14 & 19 & 5 & 7 & 77 & 10 \\ 3 & 6 & 16 & 8 & 6 & 168 \end{bmatrix}$$

(a) Confusion matrix for clean data

Class	Precision	Recall	F_1	Accuracy
1	68.8%	72.7%	70.2%	
2	76.2%	74.2%	74.8%	
3	70.7%	69.0%	69.6%	74.7%
4	85.0%	83.3%	83.9%	
5	63.6%	58.4%	60.3%	
6	78.5%	81.2%	79.6%	

(b) Performance metrics for clean data

$$\begin{bmatrix} 21 & 13 & 15 & 13 & 19 & 7 \\ 15 & 135 & 12 & 14 & 2 & 9 \\ 18 & 19 & 97 & 17 & 18 & 18 \\ 9 & 14 & 10 & 152 & 11 & 13 \\ 12 & 12 & 8 & 10 & 57 & 11 \\ 9 & 5 & 15 & 12 & 16 & 163 \end{bmatrix}$$

(c) Confusion matrix for noisy data

Class	Precision	Recall	F_1	Accuracy
1	24.7%	23.6%	23.8%	
2	68.9%	72.2%	70.0%	
3	62.5%	51.8%	56.3%	62.4%
4	69.7%	72.7%	71.0%	
5	47.2%	51.8%	49.2%	
6	74.6%	74.1%	74.0%	

(d) Performance metrics for noisy data

Figure 3: Performance of Score-based Choice

Among the conflict-resolving approaches, we choose the **Score-based Choice**, which has a better accuracy on both clean and noisy data.
NOTE: Since we have run cross-validation on different approaches and chosen the best performing one, we used cross-validation for *validating* the model: the performance estimates are then biased and cannot be considered as performance estimates of the chosen model on unseen data. To have such estimate, we need to hold out a separate test set and compute the performance of the chosen model on this set.
NOTE: Since the Random Choice approach has randomness in it, it may produce slightly different confusion matrices on different cross-validations, even with the same folds.

5 Questions

5.1 Noisy-Clean Datasets

All the alternative approaches perform worse on noisy data than on clean ones. This is because ID3 is particularly prone to overfit, since decision trees are a very expressive model. Overfitting a corrupted training set leads to a model which captures a wrong correlation between features and labels. Thus, the learnt model is not able to predict the target function outside the training set.
More specifically, on both clean and noisy data we observe the worst performance in classes 1 and 5. Class 3, instead, exhibits bad performance on noisy data. This is related to the particular shape of the corresponding trees: they use many

$$\begin{bmatrix} 101 & 13 & 6 & 3 & 8 & 1 \\ 34 & 152 & 2 & 4 & 3 & 3 \\ 19 & 5 & 80 & 2 & 4 & 9 \\ 20 & 9 & 2 & 175 & 6 & 4 \\ 43 & 18 & 2 & 6 & 59 & 4 \\ 27 & 6 & 16 & 3 & 2 & 153 \end{bmatrix}$$

(a) Confusion matrix for clean data

Class	Precision	Recall	F_1	Accuracy
1	41.9%	76.5%	54.0%	
2	75.4%	76.7%	75.8%	
3	74.2%	67.3%	70.1%	71.7%
4	90.8%	81.0%	85.4%	
5	73.9%	44.6%	54.5%	
6	88.2%	74.0%	80.3%	

(b) Performance metrics for clean data

$$\begin{bmatrix} 46 & 10 & 12 & 6 & 12 & 2 \\ 34 & 134 & 6 & 8 & 1 & 4 \\ 47 & 16 & 83 & 18 & 7 & 16 \\ 28 & 18 & 8 & 145 & 4 & 6 \\ 26 & 13 & 4 & 8 & 50 & 9 \\ 32 & 5 & 11 & 9 & 7 & 156 \end{bmatrix}$$

(c) Confusion matrix for noisy data

Class	Precision	Recall	F_1	Accuracy
1	22.4%	52.1%	31.0%	
2	68.2%	71.8%	69.7%	
3	66.9%	44.3%	52.7%	61.3%
4	75.3%	69.4%	71.9%	
5	64.4%	45.5%	52.4%	
6	81.2%	70.9%	75.3%	

(d) Performance metrics for noisy data

Figure 4: Performance of Tree confidence Choice

attributes to make their decisions, and thereby are the most prone to overfitting. The trees for class 1 and 5 exhibit this shape in both the clean and noisy trainings, while class 3 does just in the noisy one.

5.2 Ambiguity

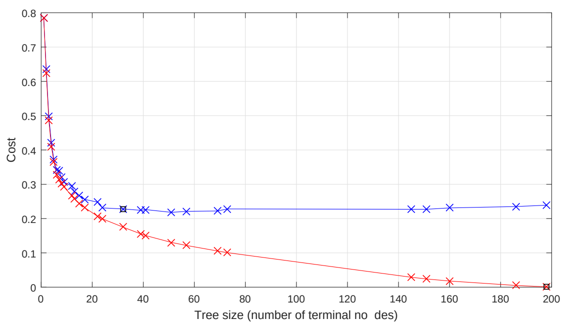
- To be sure that only one prediction is returned, we provided the decision making solutions described in Section 2.
- The required descriptions are in Section 2 as well.
- Section 4 contains the commented description of our approaches' performance. The observations are consistent with our previous answer.

5.3 Pruning

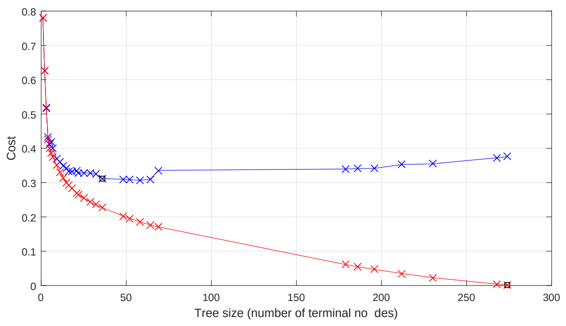
The *pruning_example* function plots the training and test error of a tree learned on the data as a function of the tree size (number of leaves).
The tree is learned by the MATLAB function `classregtree`, which also computes the optimal pruning path. The returned tree object is the basis for the subsequent testing, run via the MATLAB function `test`: this function returns the classification errors obtained by pruning the starting tree at different sizes; the tree size is expressed as the number of leaves. Besides, the function returns the tree size for which the minimum error has been recorded.
Two tests are performed, one on the whole training set and the other using cross-validation. The plot represents the classification error recorded in the two tests as a function of the number of leaves: the red line represents the error on the training set, while the blue line is the average error recorded on unseen data. The figures also report the optimal values on both the curves. The plots for clean and noisy data are shown in Figure 5
The curves have this shape because, as long as the tree size increases, the tree fits the training data better, but it progressively loses generalization power. Therefore, while the red curve has its optimum for 200 leaf nodes, the blue one has its optimum at 32 leaves.
==

6 Implementations

From the set of 6 Decision Trees, each tree will return a binary classification for a given data sample. Therefore, we need an Algorithm that decides on Basis of the output of the 6 trees, which class the given sample falls into. In fact, this algorithm does not even have to depend on the output of the classification trees. For example, one could think of an algorithm that simply outputs a randomly choosen classification and will have an accuracy of 17% for a balanced test set. However, we searched for algorithms with a better performance.



(a) Pruning Example with clean data



(b) Pruning Example with noisy data

Figure 5: Plots produced by *pruning_example*

6.1 Stacking

As we already implemented the basis function to train a decision tree, we tried to approximate the decision function (the function which outputs a class for a given example) by another tree which is trained on basis of the output of all 6 binary classification trees for the training set. Therefore, given a sample, the decision algorithm works as follows:

1. give the sample to the 6 binary classification trees
2. give the output of all 6 trees to the decision function tree and output it's output

6.1.1 Statistics

The stacked tree method achieves a general accuracy of 72.4% on clean data and 58.8% in noisy data.

6.2 Probabilistic Trees

In this implementation, we assign a score to each leaf of every tree between 0 and 1. The score is correlated to the number of training examples that can be classified by the given leaf with respect to the total number of training examples in every class.

$$\text{score} = \frac{\text{\# correctly classified training data by this leaf}}{\text{\# training data in this tree}}$$

Of all binary classification trees returning 1, the decision algorithm will pick the class corresponding to the tree with the highest decision score. If no tree returns 1, it will pick at random.

```
function [ predictions ] = decide_by_score(trees, testset)
% decide_by_score performs a classification based on the fraction of
% correctly classified training examples given by the tree leafs

[m,n] = size(testset);
predictions = zeros(m,1);

for i = 1:m
    % test in all trees, find the one with the best score
    best_score = 0;
    predictions(i) = NaN;
```

$$\begin{bmatrix} 105 & 9 & 2 & 1 & 9 & 6 \\ 37 & 145 & 1 & 5 & 3 & 7 \\ 21 & 4 & 79 & 1 & 4 & 10 \\ 13 & 10 & 4 & 177 & 5 & 7 \\ 39 & 13 & 3 & 6 & 64 & 7 \\ 18 & 5 & 13 & 6 & 8 & 157 \end{bmatrix}$$

(a) confusion matrix for clean data

Class	Precision	Recall	F_1
1	48.3%	79.6%	57.6%
2	78.5%	73.7%	75.5%
3	76.5%	64.9%	69.8%
4	90.4%	82.5%	86.0%
5	66.3%	47.6%	54.7%
6	85.8%	76.4%	79.7%

(b) performance for the different classes with clean data

$$\begin{bmatrix} 19 & 7 & 16 & 4 & 9 & 33 \\ 13 & 130 & 7 & 8 & 2 & 27 \\ 13 & 16 & 98 & 11 & 4 & 45 \\ 10 & 14 & 13 & 141 & 1 & 30 \\ 21 & 8 & 4 & 6 & 43 & 28 \\ 13 & 8 & 21 & 9 & 11 & 158 \end{bmatrix}$$

(c) confusion matrix for noisy data

Class	Precision	Recall	F_1
1	23.5%	21.6%	NaN%
2	70.2%	68.1%	68.8%
3	60.8%	51.1%	54.8%
4	78.1%	67.6%	72.0%
5	59.9%	38.1%	45.4%
6	49.1%	71.8%	57.7%

(d) performance for the different classes with noisy data

Figure 6: statistics of the stacked tree

```
    for t = 1:6
        [pred, score] = prediction_with_score(trees(t),testset(i,:));
        if pred == 1
            % the tree recognises this item as his class
            if score > best_score
                predictions(i) = t;
                best_score = score;
            end
        end
    end
    % If all the predictions are 0, pick a random class
    if(isnan(predictions(i)))
        predictions(i) = randi(6);
    end
end

end
```

6.2.1 Statistics

The described method achieves a general accuracy of 73.2% on clean data and 64.4% in noisy data.

$$\begin{bmatrix} 88 & 15 & 7 & 4 & 11 & 7 \\ 15 & 145 & 5 & 7 & 14 & 12 \\ 7 & 5 & 83 & 2 & 5 & 17 \\ 4 & 8 & 6 & 188 & 5 & 5 \\ 15 & 19 & 2 & 10 & 73 & 13 \\ 3 & 9 & 14 & 11 & 4 & 166 \end{bmatrix}$$

(a) confusion matrix for clean data

Class	Precision	Recall	F_1
1	66.3%	63.1%	63.4%
2	72.4%	75.1%	73.3%
3	67.1%	65.9%	66.2%
4	79.1%	84.0%	81.0%
5	60.5%	55.1%	57.5%
6	83.3%	81.1%	82.0%

(b) performance for the different classes with clean data

$$\begin{bmatrix} 21 & 11 & 20 & 8 & 19 & 9 \\ 16 & 132 & 13 & 15 & 6 & 5 \\ 14 & 13 & 108 & 22 & 11 & 19 \\ 8 & 13 & 10 & 161 & 10 & 7 \\ 15 & 10 & 8 & 7 & 59 & 11 \\ 13 & 9 & 14 & 10 & 10 & 164 \end{bmatrix}$$

(c) confusion matrix for noisy data

Class	Precision	Recall	F_1
1	25.5%	23.3%	NaN
2	71.4%	69.7%	69.9%
3	64.4%	56.7%	59.2%
4	71.6%	77.6%	74.0%
5	51.0%	52.9%	50.7%
6	76.2%	74.1%	74.6%

(d) performance for the different classes with noisy data

Figure 7: statistics on the probabilistic tree