Decision trees

Data sets

Three data sets are prepared for testing the decision tree construction algorithm: lenses, car, tic. For each of these sets, a text file is available that contains subsequent feature vectors (class number in the last column) and a file with data description (attributes, classes). Attribute values and class numbers must be consecutive integers greater than zero.

Decision tree building

The **addnode** function is used for tree construction. An algorithm has been implemented in it, in which the attribute with the minimum entropy is selected in each node. The **build_lenses** script contains all commands related to data preparation, tree construction and testing.

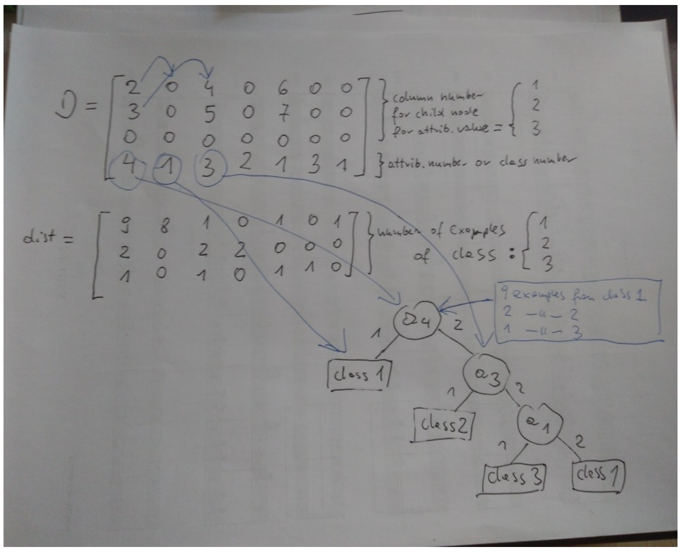
Decision tree structure

The addnode function returns a tree in the form of an array, each column of which corresponds to one node. The number of rows is one greater than the maximum number of attribute values. The last line contains the attribute number used in the node (or class number when the node is a leaf). The other lines contain column indexes corresponding to child nodes (or zeros when the node is a leaf).

For example, if a certain column has the form [2 0 4 1] ', this means that attribute 1 has been used in this node. For the first attribute value, the child node is in the 2nd column of the table, there is no branch for the second value (this situation may take place when in the training set there were no examples with this attribute value 1), for the third value of an attribute 1 the child node is in the column 4.

If the column has the form [0 0 0 3] ', this means that the node is a leaf to which class 3 has been assigned.

In addition to the described table, the constructed tree can be "drawn" in a text file using the **printnode** function.



Usable functions

1. Open the script **build_lenses.**

   In the first command, enter the path and file name of **lenses.txt**. The effect of the first fragment of the file is the division of the data set into training set p1 (odd lines) and test p2 (even lines).

   The next fragment is responsible for creating the tree (**addnode** function) and saving it in a file with the given name.

   The last instructions calculate the classification error (number of incorrectly classified / number of all vectors) for the training and test set.

2. Run the **build_lenses** script for the lenses data.

   After the tree is constructed, the following variables are available:

   D – tree in the form of an array

   TrainingVectors – learning vectors

   TrainingClasses – class numbers of learning vectors

   TestVectors – test vectors

   TestClasses – class numbers of test vectors

3. Classification. The **what_class** function allows the classification of a single feature vector

   k = what_class(D, [3 1 2 1 1])

   or the entire data set

   k = what_class(D, TestVectors)

   The function returns the class number for each example, or 0 if the example does not reach any leaf (when calculating the error, such cases are treated as examples of incorrect classification).

4. We run the **chart** script. An error graph is drawn as the tree grows. The error for the training set (red) drops to zero (during the construction of the tree, the nodes are divided until they reach only vectors of one class). The error for the test set (blue) first decreases and then begins to increase. This graph illustrates the phenomenon of overfitting a tree to learning data. You can see that it would be worth finishing the tree construction sooner or prune the complete tree.

5. When analyzing the D tree, you should consider where to prune the tree to get a smaller classification error. The distribution function may be helpful, as it returns an array containing the number of examples of individual classes reaching individual nodes. An array element with (i, j) coordinates is the number of class images in node j. Function call:
   dist = distribution(D, TrainingVectors, TrainingClasses)
   another function:

   g = depth(D)

   returns the depth at which each of the D tree nodes lies.

   We check the classification error using the function **calc_error**:
   calc_error(D, TrainingVectors, TrainingClasses) - error for training set
   calc_error(D, TestVectors, TestClasses) - error for test set
   For the D tree constructed for the lenses data, for example, you can remove the last two columns (two leaves), remembering to replace the predecessor node (number 5) with a leaf (you should assign it the label of the class most represented; in this case, assigning a class 3 label will reduce the error).

Exercise:

Fill the gaps in the script **build_xxx** with our own pruning method. After implementing our own pruning method, compare the classification error before and after pruning for the training and test sets. The reduction of average test error value should be positive. For the given data set the pruning method should be chosen so as to reduce the average classification error for the test set as much as possible (improve the generalization) regardless of the content of the training set: in the script **build_xxx.m** or **build_xxx.py** 10 experiments are used to calculate average test error with different random partitions into training and test set. Using the parameter **part_for_constr** you can additionally divide the training set into a subset for tree construction and subset for validation - for checking what generalization can look like after pruning subsequent nodes. You can also prune the tree without validation subset - by heuristic method. The use of a test set for tree construction or pruning is prohibited!