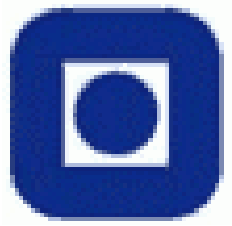


Department of Computer and Information Science

Norwegian University of Science and Technology

Induction of decision trees from partially classified data using belief functions

Marte Skarstein Bjanger

February 28, 2000

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR FYSIKK, INFORMATIKK OG MATEMATIKK

HOVEDOPPGAVE

Kandidatens navn: Marte Skarstein Bjanger

Fag: Datateknikk, Kunnskapssystemer

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Induction of decision trees from partially classified data using belief functions

Oppgavens tekst:

Most of the work on pattern classification and supervised learning has focused on the induction of decision rules from learning examples consisting of objects with known classification. In certain real-world problems, however, such ”perfect” observations are not always available. Instead, we may have an ”uncertain” training set of objects with partially known classification. For instance, an expert or a group of expert may have expressed (possibly conflicting) opinions regarding the class membership of objects contained in the data base. It may then be advantageous to take into account the resulting uncertainty in the design of a classification rule.

For that purpose, an approach based on tree-structured classifiers and the Dempster-Shafer theory of belief functions will be investigated. Using previous work on parametric inference for the Bernouilli distribution using an evidential approach, the entropy measure classically used to assess the ”purity” of nodes in decision trees will be replaced by evidence-theoretic uncertain measures allowing to take into account not only the class proportions, but also the number of objects in each node. This approach will be extended to training data whose class membership is only partially specified in the form of a belief function. The method will be evaluated on electroencephalogram data from a sleep staging experiment.

Oppgaven gitt:	1. september 1999
Besvarelsen leveres innen:	1. mars 2000
Besvarelsen levert:	28. februar 2000
Utført ved:	Université de Téchnlogie de Compiégne, Genie Informatique
Veileder:	Thierry Denoeux og Jan Komorowski

Trondheim, 28. februar 2000

Jan Komorowski
Faglærer

Abstract

The work described in this report concerns the problem of reasoning with uncertainty, in particular the problem of building classifiers based on uncertain data. The kind of uncertainty we have been concerned with has been uncertain classification, i.e., classification of data for which the classification labels are not crisp.

We have proposed a method to handle this kind of uncertainty that is sometimes present in the training objects used to build classifiers. Our method proposes to introduce the concept of belief functions, as defined in the Dempster-Shafer theory of evidence, in the well known decision tree learning method. Belief functions is a means of stating the kind of uncertainty we are interested in and give valuable information as output from the classifier. This makes the classifier able to give a more differentiated result. Also, the classifier will be able to use the information given in uncertain labeled training objects in a profitable way.

We have implemented our method in MATLAB, to test it on a real world classification problem. The results obtained from these experiments show that our method performs at least as well as the ordinary decision tree learning method. In addition they show that our method offers a way of handling problems for which the classification is not entirely known for the training objects. This means that the method will be able to handle classification problems which for instance the ordinary decision tree learning method is not able to handle.

In order to be able to obtain substantial conclusions about the method, further work will have to be done to test the method more extensively and to improve it. However, our results are promising and encourage further work with this method.

Preface

This report presents the results of work constituting a master thesis at the Department of Computer and Information Science (IDI), Faculty of Physics, Informatics and Mathematics (FIM) at the Norwegian University of Science and Technology (NTNU).

The work has been done at the Faculty of Informatics (Génie Informatique, GI) at the Technical University of Compiègne (Université de Technologie de Compiègne, UTC) in France. My teaching supervisor at UTC has been Professor *Thierry Denoeux*. My teaching supervisor at NTNU during my period of work has been Professor *Jan Komorowski*.

The assignment was given by Professor Thierry Denoeux and concerns a part of his current research on reasoning with uncertainty.

Acknowledgements

I would like to thank Professor Thierry Denoeux for introducing me to an interesting area of research and for sharing his ideas on the subject with me. He has enthusiastically provided the foundation for this work. I am grateful for all the valuable discussions we have had during the period of work.

I would also like to thank UTC for making it possible for me to come to Compiègne to do my work and for making my stay a very enjoyable one.

Thank you as well to Professor Jan Komorowski for encouraging me to go to Compiègne and carry out this work, and for giving me useful comments during my period of work.

Finally, I would like to thank my husband, Bjørn, for being interested and supportive and for his continually encouraging smile.

Compiègne, February 24, 2000

Marte Skarstein Bjanger

Contents

Chapter 1

Introduction

1.1 The purpose of our work

The work we present in this report adresses the problem of reasoning with uncertainty, in particular the problem of building classifiers based on uncertain data. The kind of uncertainty we were concerned with is uncertain classification, i.e., training a classifier on data for which the knowledge of which class it belongs to is not certain.

The purpose of our work has been to develop a method that handles this kind of uncertainty. Our method is based on the decision tree learning method, which we modify with the use of belief functions. Belief functions are used to compute an uncertainty measure that will replace the concept of entropy used in ordinary decision tree learning methods. It was our belief that the introduction of belief functions would enable us to train the classifier on data with the kind of uncertainty mentioned above.

1.2 Reader's guide

This report is roughly divided in three parts. The first part introduces the important concepts related to the problem domain. This part is supposed to give the reader the necessary knowledge about the issues in question. It is to be regarded as an introduction to what is presented in the following parts. The second part presents the method we propose in our work and the theory it is based upon. The third part presents experiments we have performed in order to test our method.

The first part of the report starts with a chapter giving a short introduction to the domain of classification, and in particular to the domain of classifying with uncertain data. Chapter 3 provides an introduction to the decision tree learning method. Chapter 4 defines the concepts of the Dempster-Shafer theory of evidence, and in particular the

interpretation of this theory that constitutes the transferable belief model.

The second part consists of Chapter 5, which presents our ideas for a new method combining decision tree learning and the Dempster-Shafer theory of evidence. This chapter elaborates the mathematical foundation for our proposed method and gives a short description of our implementation of the method.

The third part presents experiments and results. In Chapter 6 we describe how our method has been tested on a real world classification problem and compared to an ordinary decision tree learning program. Chapter 7 analyses our results and gives an evaluation of our method as well as proposing further work that may be done to obtain more solid results and to improve our method.

Chapter 2

Classification of uncertain data

2.1 Introduction

A common problem in the context of machine learning is the task of classification, or diagnosis. Examples of "diagnostic systems" are medical systems, that use observed symptoms for a patient to assess whether some disease is present or not, or technical systems, that for instance can use the observed behaviour of some machine to assess whether this machine is in a fault state or not.

In order to build a reliable "diagnostic system", a good classifier has to be found. There exist several methods to produce classifiers, such as Neural networks, Decision trees, k-Nearest neighbour and Rough Set theory to mention some of them. A lot of work has been done to find new methods and to improve existing methods in order to increase the reliability of the classifiers the different methods produce.

Not all of the existing methods are able to handle adequately data that contain some sort of uncertainty. However, as most of the data used as a basis for classification that reflect real world situations contain uncertainty in one way or another, it is an important task to find a method that is able to handle uncertain data.

This chapter gives an overview of the classification task, and explains some of the problems involved in dealing with uncertain data. Some existing methods are mentioned, and a summary of our proposed method is given.

2.2 Classification

The classification process consists of deciding which *class* among two or more possible classes some instances or objects of a kind belong to. These instances or objects in

question are commonly gathered in *data sets*.

Each instance in a data set can be regarded as a vector with a given number of values. Each of the values represents an *attribute value* measured or chosen for the instance in question. The attributes that are considered for the set of instances are often called *conditional attributes*. The collection of possible attribute values can be *binary*, *discrete* or *continuous*.

In order to make a classifier, one needs to have a set of already classified instances to train the system. The set of already classified instances used for this purpose is called the *training set*. In addition to all the conditional attributes, these instances also have an attribute called the *decision attribute*, which represents the classification of each instance. The values of this attribute represent the possible classes the instances from this particular distribution can belong to. Thus, these values can either be *binary*, if there are only two classes, or *discrete*.

An example of a data set that can be used as a basis for learning a classification task is shown in Table 2.1. The example is taken from [1] and represents the task of learning how to choose whether to play tennis or not on a given day based on information about the weather conditions.

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Table 2.1: Example data set

When a classifier has been constructed, its performance can be tested by using a set of instances that have not yet been classified. The set of instances used to test the system's performance is called the *test set*. This set should be drawn from the same distribution as the training set.

If the classifier performs adequately, one can classify future instances for which one

does not know the class by running them through the classifier. For instance, if a classifier was built based on the data in Table 2.1, one can easily use this classifier to decide whether or not to play tennis one particular day given this day's weather conditions.

2.3 Data sets with uncertainty

Classification systems that are based on empirical data, such as technical systems that are designed to monitor the state of a machine, have to involve a method of handling uncertainty. The classifier in such systems are built based on data observed during a period of time, and if the classifier is to be reliable with respect to classifying states that will occur in the future, it has to take into consideration the uncertainty inherent in the collected data.

Uncertainty in data may have different causes. If the data used as basis are constructed from some expert's opinion, which in turn is based on this expert's experience and earlier observations, assumptions that this expert makes about some observation can introduce uncertainty in the data. Uncertainty also arises if there are several experts who each have their opinion, since these opinions may conflict.

Another source of uncertainty may simply be measurement uncertainty when data are collected. If the means of measuring or observing is not completely reliable, the measured data will consequently not be completely reliable.

It is also necessary to take into consideration the possibility that an event may occur given some conditions, but may not occur later given the same conditions. This induces an uncertainty in our experienced data, because the two observed situations, that the event did occur and that the event did not occur, produces a conflict in our statements.

Other causes of uncertainty may be imprecision in the representation method of the data or simply vague terminology, for instance if objects are classified as "small" or "large" without any definite limit between the two values.

Since we know that uncertainty may be contained in the training data we use to build our classifier, the classification results should also reflect this situation. It is desirable to have a classifier that not only classifies the instances as belonging to one of the classes, but is able to say something about the degree of reliability of this classification. It should be able to give us a clue about how much we should believe in the classification it produces.

2.4 Method proposed in our work

In order to use uncertain data in a profitable way, one has to find a method for representing and taking care of the uncertainty. There are several ways of doing this. Methods have for instance been developed based on the concept of probability, but other methods have also been developed.

The theories of fuzzy sets and rough sets address the problem of uncertainty in data, see for instance [2] and [3] for a description of fuzzy sets and rough sets. A method based on interval estimation of probabilities has been developed by Kurt Weichselberger and Sigrid Pöhlmann, see [4]. Thierry Denoeux has developed a method for pattern classification that uses the k-Nearest Neighbour method based on the Dempster-Shafer theory of evidence, see [5], [6] and [7].

Uncertainty may be present in many forms. In our work we are concerned with uncertainty in class labels, for instance if we do not know for certain which of the possible classes a given instance belong to. Our work has the purpose of finding a way to build reliable classifiers when the values of the *decision attribute* are uncertain, i.e., finding a way to train the system even though the training attributes do not have a crisp classification.

We propose a method that combines the decision tree learning method with the concepts of the Dempster-Shafer theory of evidence for representing this kind of uncertainty. The method of decision tree learning does not in itself address the problems of uncertain class labels in training data as described above, but we propose to extend this method with the use of belief functions in order to be able to work with uncertainty.

Our method will be fully described in Chapter 5. The next chapters give an overview of respectively the decision tree learning method and the Dempster-Shafer theory of evidence.

Chapter 3

Decision trees

3.1 Introduction

The decision tree learning method is one of the methods that are used for classification or diagnosis. As for many other machine learning methods, the learning in decision trees is done by using a data set of already classified instances to build a decision tree which will later be used as a classifier. The set of instances used to "train" the decision tree is called the training set.

When the decision tree has been built based on the training set, the testing of the system's performance is done by using a test set with instances that are not yet classified, but which are taken from the same distribution of instances as the training set. The test set instances are put through the resulting decision tree in order to get a classification for each of the test instances. The goal is to get a tree which, based on the instances in the training set, is able to classify the instances in the test set correctly.

Decision tree learning has several advantages. One of the advantages is that it gives a graphical representation of the classifier which makes it easier to understand. However, this is not the case for large trees, which tend to get over-complex and difficult to follow. Another advantage is that this method can handle missing attribute values in the training data.

This chapter gives a definition of some basic notions in the decision tree method, and gives a description of the common procedure used when building decision trees. An example is also provided, in order to illustrate the concepts. This chapter is to be regarded as an overview only. For a more thorough description of decision trees, refer to [8], [9] and [1].

3.2 Basic concepts

In order to understand how decision trees are used for classification purposes, some basic notions have to be defined.

Decision trees consist of *nodes* and *branches*. The nodes can either be ordinary nodes with other descendant nodes, or *leaf nodes* which do not have descendants. The *root node* is the first node on top of the tree. Figure 3.1 illustrates these concepts.

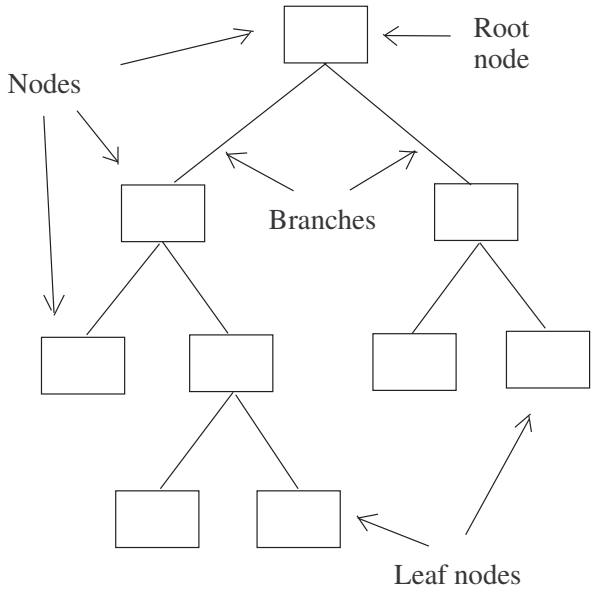


Figure 3.1: An example tree

Each of the ordinary nodes represents a test on one of the attributes that are measured for each object, and each of the leaf nodes represents a classification of the object. The branches descending from a node corresponds to the possible values for the attribute in question, and consequently there are as many branches from a node as there are attribute values for the attribute which is tested by the node. If the attribute values are continuous, there are usually two branches descending from the node, corresponding to a division of the objects according to some split $x \leq x_i$ and $x > x_i$, where x is the attribute value for the object tested at the node, and x_i is some defined split value for the attribute in question.

Another important concept in the decision tree method is the concept of *entropy*. The entropy is a value that is used when the tree is being built, and it reflects how well the objects in the training set are separated with respect to the different classes at some node. If there are an approximately equal amount of training objects from each class connected to the node, the entropy is large, and if there are only training objects from

one class connected to the node, the entropy is 0. The entropy concept is explained in detail later.

3.3 The decision tree algorithm

There are several procedures and methods for building decision trees, such as ID3, C4.5 and CART (see [9] and [8]). We have chosen to base our work on the ID3-algorithm, because this algorithm visualises well the general approach to building decision trees. As a consequence, this algorithm forms the basis of the description of decision trees in this chapter. The ID3-algorithm is shown in Algorithm 3.1.

The ID3-algorithm employs a top-down, greedy search through the possible trees, and it never backtracks to reconsider the choices it takes at each node.

To build a decision tree, one has to have a set of training objects on which to base the learning of the tree. These training objects have a known classification before the building of the tree starts.

The general concept of the building phase consists of finding the attribute that best separates the remaining objects in the training set, and then choose this attribute as the attribute to be represented in the node. The building thus starts with finding the best overall attribute that discerns the objects in the training set based on the decision classes they belong to. This attribute is chosen as the attribute for the root node. Then one adds as many branches to the root node as there are attribute values for the chosen attribute and separates the training set objects into groups corresponding to the different values the attribute may take. Each group of training objects follows its corresponding branch from the root node.

Then one builds the tree downwards by choosing for each node the one attribute out of the remaining attributes which best separates the objects from the training set that belong to this part of the tree.

If in a node there are no more attributes to check on, the node becomes a leaf node with a label corresponding to the decision class which has the highest number of representatives among the remaining training set objects.

If all the remaining training set objects in a node belong to the same class, the node becomes a leaf node with a label corresponding to this decision class, as there is no use in separating the objects further.

To find the attribute that best separates the training set objects, the ID3-algorithm uses a concept that is called *information gain*. Information gain is a measure on how well one can discern the remaining objects according to their classes by using a given attribute as a separator. Associated to the concept of information gain is the concept of *entropy*.

Algorithm 3.1 The ID3-algorithm

ID3$(Examples,\ Decision\ attribute,\ Attributes)$

Examples are the training objects. Decision attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given examples.

Create a $Root$ node for the tree.
if all $Examples$ are positive, $Return$ the single-node tree $Root$, with $label = +$
end
if all $Examples$ are negative, $Return$ the single-node tree $Root$, with $label = -$
end
if $Attributes$ is empty, $Return$ the single-node tree $Root$, with $label =$ most common value of $Decision\ attribute$ in $Examples$
end
Otherwise
begin
  $A \leftarrow$ the attribute from $Attributes$ that best* classifies $Examples$
  The decision attribute for $Root \leftarrow A$
  for each possible value, v_i, of A,
    Add a new tree branch below $Root$, corresponding to the test $A = v_i$
    Let $Examples_{v_i}$ be the subset of $Examples$ that have value v_i for A
    if $Examples_{v_i}$ is empty
      then below this new branch add a leaf node with $label =$ most common value of $Decision\ attribute$ in $Examples$
      else below this new branch add the subtree
        ID3$(Examples_{v_i},\ Decision\ attribute,\ Attributes - \{A\})$
      end
    end
  end
end
RETURN $Root$

* The best attribute is the one with the highest *information gain*.

Entropy is a measure of the "impurity" of the set of training objects.

The entropy is defined as

$$E(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i$$

where S is the set of training objects containing objects from c different classes, and p_i is the proportion of objects in the set S from class i. The entropy is 0 if all objects in S belong to the same decision class, and can be as large as $\log_2 c$.

For the special case where the objects are divided between only two decision classes, let us say a positive class and a negative class, the entropy reduces to

$$E(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

In this case, the entropy is between 0 and 1, and 1 if there are an equal number of objects from each of the two decision classes.

The information gain measure is based on the computed entropy for each attribute, and states the expected reduction in entropy if the training objects are separated by using the attribute in question. The information gain of an attribute A relative to a set of objects S is defined as

$$G(S, A) \equiv E(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} E(S_v)$$

where $Values(A)$ contains all possible values of the attribute A and S_v is the set of objects in S for which attribute A has value v. We see from the equation that the information gain is a measure on how much the entropy is expected to decrease if we partition the training set based on a given attribute. Because of this we choose as the "best" attribute the attribute which gives the highest information gain, since our goal is to decrease the entropy as we split the training objects.

When there are no more training objects, or there are no more attributes to split on, we have a final decision tree which can be used as a classifier. The performance of the tree can now be tested by running pre-classified test objects through the tree, and observe whether or not the tree classifies them correctly.

3.4 An example

To illustrate the procedure of the decision tree method explained in this chapter, an example is presented. The example objects are taken from [10], and modified slightly.

The example objects given in Table 3.1 will be used as a basis for building a decision tree.

The *objects* (rows) in this example are bottles of red wine. Different *attributes* (columns) are measured for each bottle. The measured attributes are *Wine district*, *Main grape variety*, *Vintage* and *Storage temperature*. For each bottle we have a classification that says whether the bottle is ready to be drunk now or if it should be kept in store.

	Wine district	Main grape variety	Vintage	Storage temp.	Decision
x_1	Bordeaux	Cabernet Sauvignon	1992	12-15	Drink
x_2	Rhône	Syrah	1992	<12	Hold
x_3	Chile	Cabernet Sauvignon	1995	12-15	Drink
x_4	Bordeaux	Merlot	1995	>15	Drink
x_5	Chile	Cabernet Sauvignon	1992	12-15	Hold
x_6	Rhône	Merlot	1992	12-15	Hold
x_7	Bordeaux	Merlot	1995	12-15	Drink
x_8	Chile	Merlot	1992	<12	Hold
x_9	Bordeaux	Merlot	1992	>15	Drink
x_{10}	Rhône	Syrah	1995	<12	Hold
x_{11}	Chile	Merlot	1992	12-15	Drink

Table 3.1: Example objects

The first step in building a decision tree, according to ID3, is to create a root node. Then we have to choose which attribute is to be tested on the root node. The best attribute is the one with the highest Information Gain, so we have to compute the Information Gain for each possible attribute, which at this first step are all the attributes.

At the root node we consider all the example objects, because we have not yet started to divide them into groups, so in this case the set of all training examples, S, contains all 11 objects from Table 3.1. We observe that we have 6 objects belonging to the decision class *Drink*, and 5 objects belonging to the decision class *Hold*. We are then able to compute the Entropy for this set.

$$E(S) = -p_{Drink} \log_2 p_{Drink} - p_{Hold} \log_2 p_{Hold}$$
$$E(S) = -\frac{6}{11} \log_2(\frac{6}{11}) - \frac{5}{11} \log_2(\frac{5}{11}) = 0.994$$

We then compute the information gain for the attribute *Wine district*. To simplify the notation, *Wine district* is represented by WD, *Bordeaux* is represented by B and so

on:

$$G(S, WD) = E(S) - \sum_{v \epsilon Values(A)} \frac{|S_v|}{|S|} E(S_v)$$

$$G(S, WD) = E(S) - \frac{|S_B|}{|S|} E(S_B) - \frac{|S_R|}{|S|} E(S_R) - \frac{|S_C|}{|S|} E(S_C)$$

$$E(S_B) = -\frac{4}{4} \log_2(\frac{4}{4}) = 0$$

$$E(S_R) = -\frac{3}{3} \log_2(\frac{3}{3}) = 0$$

$$E(S_C) = -\frac{2}{4} \log_2(\frac{2}{4}) - \frac{2}{4} \log_2(\frac{2}{4}) = 1$$

$$G(S, WD) = 0.994 - \frac{4}{11} * 0 - \frac{3}{11} * 0 - \frac{4}{11} * 1 = 0.6304$$

The entropy and the information gain for the remaining attributes are computed the same way and we get these information gain values for the four attributes at the root node level:

$$G(S, Wine\ district) = 0.6304$$
$$G(S, Main\ grape\ variety) = 0.2427$$
$$G(S, Vintage) = 0.0721$$
$$G(S, Storage\ temperature) = 0.4931$$

We see from these results that if we choose to split on attribute *Wine district* at the root node, we will get the highest information gain. This tells us that this split will divide the training objects in the best possible way concerning the two different classes. Our goal is to have as "pure" nodes as possible in the end, nodes that contain - if possible - objects from only one class. Thus, we choose to split the objects on the attribute *Wine district* at the root node.

We now have the tree shown in Figure 3.2. At the root node we divide the training objects into groups according to which value they have for the attribute *Wine district*, and create a child node for each possible attribute value.

The next step is to consider which attribute to split on for the child nodes we have created. For the child node corresponding to the attribute value *Bordeaux*, we have four training objects that have *Bordeaux* as value for the attribute *Wine district*. We observe that all these objects belong to the class *Drink*, and according to ID3, this means that the child node corresponding to *Bordeaux* can be a leaf node with decision *Drink*.

The same applies to the child node corresponding to the attribute value *Rhône*. For this node we have three training objects, which all belong to the class *Hold*. We can then let this child node be a leaf node with decision *Hold*.

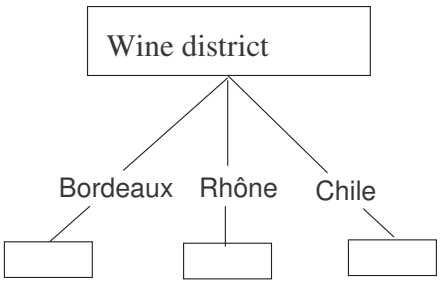


Figure 3.2: Our decision tree at the first stage

The third child node, the one corresponding to the attribute value *Chile*, has four training objects connected to it. Among these objects there are two objects that belong to the class *Drink* and two objects that belong to the class *Hold*. This means that we have to split these objects in two, based on one of the three remaining attributes, *Main grape variety*, *Vintage* or *Storage temperature*. We apply the same method as above and compute the information gain for each attribute. Note that now the starting point is the set with the four training objects that belong to this node, which we denote S_C. The computation results are as follows:

$$G(S_C, Main\ grape\ variety) = 0$$
$$G(S_C, Vintage) = 0.3113$$
$$G(S, Storage\ temperature) = 0.3113$$

The information gain values obtained by splitting on the attributes *Vintage* and *Storage temperature* are equal, so we can choose any one of them. We choose *Vintage* as the attribute to split on for the child node in question. We now have the tree shown in Figure 3.3.

If we continue like this, until the training objects are divided into groups with objects from only one class, we finally get the tree shown in Figure 3.4.

The tree is now ready to be used for classification. The classification is done by running objects, for which we do not know the class, through the decision tree. Let us say that we have another bottle of wine, and we want to find out whether it is better to keep this bottle in store or if it is ready to be drunk. This bottle is represented in Table 3.2.

	Wine district	Main grape variety	Vintage	Storage temp.	Decision
x_{12}	Chile	Merlot	1995	>15	?

Table 3.2: Object to be classified

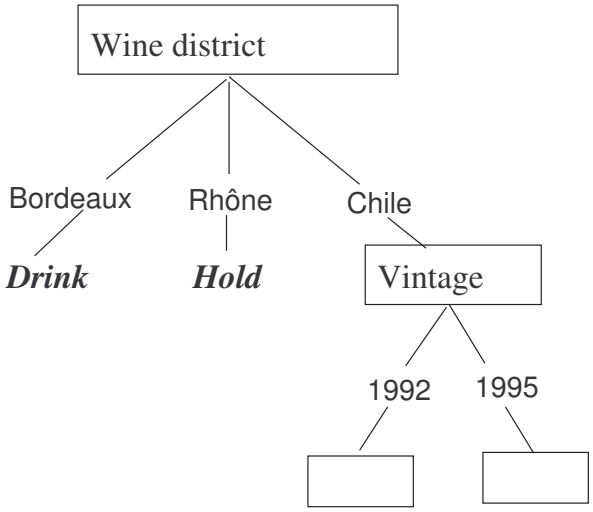


Figure 3.3: Our decision tree at the second stage

According to our tree in Figure 3.4 we should first check the attribute *Wine district* for this object to be classified. We observe that the object is from *Chile*, and according to our tree, the object should follow the branch down to the node labeled *Vintage*. We check the attribute value for *Vintage* for our object, and we observe that it is from *1995*. The object should follow the branch representing *1995*. The node connected to this branch is a leaf node with decision *Drink*. We can then conclude that our bottle is ready to be drunk.

3.5 The problems

Building a decision tree consists of several problems, for instance finding out how to consider which attribute is best for each node, and finding out when to stop building the tree, i.e. how deeply one should grow the tree.

The problem of finding out when to stop growing the tree is connected to a problem called overfitting the data. Overfitting means that if one grows a tree that classifies perfectly the objects in the training set, this tree may not classify other objects too well, because the tree is too specific. The result of growing too big a tree may accordingly be that the tree is too specifically trained to handle exactly the details of the objects in the training set, and poorly trained to handle anything else. The classification of objects outside the training set will consequently be worsened, which is not a desired behaviour.

Mitchell, [1], gives this definition of overfitting:

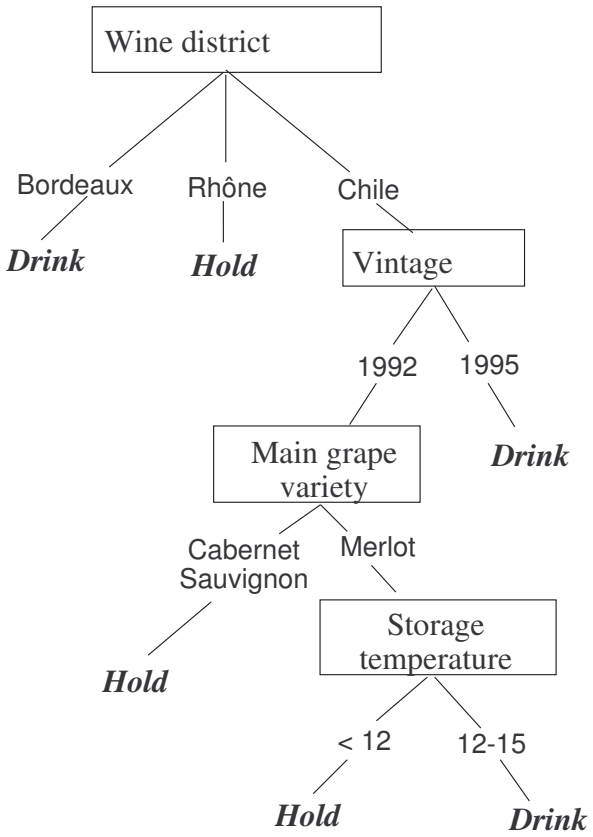


Figure 3.4: Our final decision tree

Definition: Given a hypothesis space, H, a hypothesis $h \in H$ is said to **overfit** the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

The problem of overfitting is particularly present in the cases where the training objects contain noise or errors. Then a full grown tree will be too sensitive with respect to the noise or errors in the data, and the result is that the tree classifier performs badly on other data sets.

Figure 3.5 is taken from [1] and illustrates the consequence of overfitting. The results in this figure were obtained by applying the ID3-algorithm to the task of learning which medical patients have a form of diabetes. The numbers along the horizontal axis represents the size of the tree as it is grown. As the figure shows, the accuracy over the training objects increases as the tree grows, because the tree learns gradually more from the training objects as it grows. However, the accuracy on the independent test objects increases from a certain size of the tree, which indicates that when the tree grows larger than this size, it gets too specific in relation to the test data.

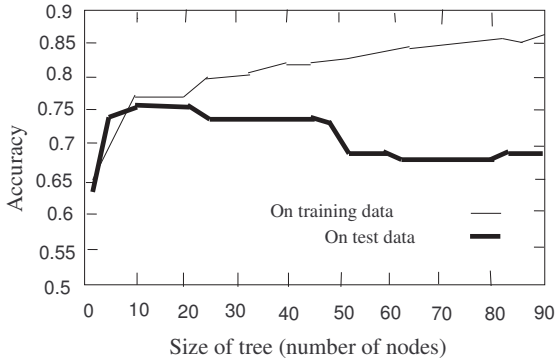


Figure 3.5: Overfitting in decision tree learning

The most common methods to avoid overfitting in decision trees are to either stop growing the tree before it is fully grown, or to grow a full tree and then prune [1] it. In either case the problem is to find a criterion to use as an indication of which sized tree will be the tree to perform best. It is not easy to know exactly when to stop in order to get the best tree. Too large a tree will use too much of the detailed information contained in the training set, while too small a tree may not make use of all the helpful information contained in the training set. There are several approaches to finding the right sized tree. The most common of them is to use a validation set of objects, which should be distinct from the training objects, to evaluate how much post-pruning a node will improve the tree classifier.

In order to decide which nodes to prune, by using the validation set, one can use the concept of *error rate estimation*. The validation set can be run down trees of different sizes, and for each tree one can use the classification results to compute an estimated error rate. For instance, if one first grows a full tree based on the training set, and then prune the full grown tree node by node and run the validation set down the tree each time a node has been pruned, one will get an error rate estimation for each possible tree size. By comparing the computed values, one can see that there are tree sizes that are better than others. The typical results will be that as one prunes the full grown tree, the error rate estimate will decrease, until one reaches a size where continued pruning will result in increasing error rate estimates.

The problem with this approach is that it requires a large amount of available data. If the data available is limited, it may not be desirable to hold out a set of objects for validation, because that would result in a smaller training set, which in turn would result in a less reliable decision tree than if all the available information was used to build the tree.

[1] To prune a node from a decision tree is to remove the subtree of that node, and to make the node a leaf node with decision value the most common value of the training objects connected to that node.

Another problem in decision tree building is to find a good way of choosing splits for continuous attribute values. The common way of splitting objects with continuous attribute values is to divide the objects in two groups, according to whether or not they have an attribute value x which is smaller than or greater than some value x_i. The split divides the objects in two groups with $x \leq x_i$ and $x > x_i$. The difficulty is to find the best split for the attribute in question, i.e., to decide the value x.

3.6 Summary

The decision tree learning method is widely used for classification purposes, and this chapter has presented the method and its concepts. Decision tree learning represents an understandable and reliable method for constructing classifiers. The tree structure makes it intuitively easy for humans to visualise the classifier.

There are however several problems connected to the use of decision trees as a classification method, and some of them were outlined here. Another problem, which we have been concerned with in our work, is that decision tree learning does not cover a good method for handling uncertainty in the training data. This problem will be further outlined in the remaining chapters.

Chapter 4

Dempster-Shafer Theory of Evidence

4.1 Introduction

The Dempster-Shafer theory of evidence is a mathematical theory concerning belief and uncertainty and was evolved by Arthur Dempster and Glenn Shafer [11]. The theory in large involves assigning a value between 0 and 1 to some hypothesis, regarding this value as the degree of belief in the hypothesis in question, based upon a given body of evidence.

The theory differs from probability-based methods in that it does not claim that the sum of the belief in a hypothesis and its negation has to be 1. If, for instance, the belief in a hypothesis is assigned the value 0, the negation of the hypothesis may also be assigned the value 0, which in short reflects a situation where there is no information available that helps in choosing between the two.

The Dempster-Shafer theory of evidence also involves the possibility of assigning belief values to sets of hypotheses, as well as combining distinct bodies of evidence in order to assign combined belief values to the different hypotheses.

This chapter presents an overview of some of the concepts of the Dempster-Shafer theory of evidence. There exist several interpretations of this theory, and we have based our work on the interpretation called the *transferable belief model* (TBM) that has been introduced by P. Smets ([12] and [13]). Consequently, the presentation in this chapter will be based on this model. Since it will be too extensive to give the reader full knowledge of the Dempster-Shafer theory, only the basic concepts and the concepts concerning our work will be outlined. For a full description of the Dempster-Shafer theory of evidence, see [11], [12] and [14].

4.2 Basic concepts

In the transferable belief model there are two levels at which uncertain information is processed, the *credal* level and the *pignistic* level[1]. At the credal level beliefs are quantified by belief functions, and a possible updating of our beliefs is done at this level. The pignistic level succeeds the credal level in time and appears when one is confronted with decision making based on one's beliefs. Our work is concerned with classification, where there is a decision to be made in the end, so we have to take into consideration both levels.

At the credal level, our beliefs are built. In order to build them we have to define what we will have beliefs about. This is what we in the last section called the "hypotheses", and they are represented as a set of possible situations, called the *Frame of Discernment*

$$\Theta = \{\theta_1,\ \theta_2,\ \ldots,\ \theta_n\}$$

where $\theta_1,\ \theta_2,\ \ldots,\ \theta_n$ are mutually exclusive and exhaustive. The power set of Θ, 2^Θ, denotes the set of all subsets of Θ.

A *basic belief assignment* is a function that assigns a value between 0 and 1 to each subset in the Frame of Discernment

$$m:\ 2^\Theta \rightarrow [0,1],$$

such that

$$m(\emptyset) = 0$$

$$\sum_{A \subseteq \Theta} m(A) = 1.$$

$m(A)$ is called A's *basic belief number*, and represents a mass of belief that is assigned to A. This belief is assigned on the basis of the given body of evidence concerning the hypotheses.

According to the statements above, no belief is assigned to $\emptyset$, since at least one of the subsets in the Frame of Discernment has to represent the true situation, if we adopt the closed-world assumption [12]. Also, for the same reason, the total belief in Θ is 1. The quantity $m(A)$ is the degree of belief that is assigned exactly to the set A, and it can not be subdivided in order to get the assigned values for the possible subsets of A.

[1]The word *pignistic* comes from the Latin word *pignus*, which means a bet.

The total belief in A, including the values for the subsets of A, can be obtained by

$$Bel(A) = \sum_{B \subseteq A} m(B).$$

The function $Bel : 2^{\Theta} \rightarrow [0, 1]$ is called a *belief function* over Θ. The belief function is called the *vacuous* belief function when $m(\Theta) = 1$ and $m(A) = 0$ for all $A \neq \Theta$, then $Bel(\Theta) = 1$ and $Bel(A) = 0$ for all $A \neq \Theta$. This represents a situation where one has no evidence.

A subset A of a Frame of Discernment Θ is called a *focal element* of a belief function Bel over Θ if $m(A) > 0$. The union of all the subsets that are focal is called the *core* of Θ.

Another function can be defined from the basic probability assignment, the *plausibility function*

$$Pl(A) = \sum_{B \cap A \neq \emptyset} m(B).$$

The plausibility function can also be given by

$$Pl(A) = 1 - Bel(\Theta \setminus A), \ \forall A \subseteq \Theta.$$

As we see, the plausibility function is another way of presenting our beliefs. Yet another way of stating beliefs is the commonality function defined by

$$Q(A) = \sum_{B \supseteq A} m(B).$$

The functions $m(A)$, $Bel(A)$, $Pl(A)$ and $Q(A)$ are all several ways of representing the same information and can all be obtained from the others.

Dempster's rule of combination provides the tool for combining several belief functions over the same Frame of Discernment. In order to use this rule, the different belief functions must be based upon distinct bodies of evidence. Using Dempster's rule of combination involves computing the orthogonal sum of the belief functions to be combined. Dempster's rule of combination is defined by

$$\begin{aligned} m(\emptyset) &= 0 \\ m_3(C) &= \frac{\sum_{A \cap B = C} m_1(A) m_2(B)}{1 - \sum_{A \cap B = \emptyset} m_1(A) m_2(B)} \quad C \neq \emptyset. \end{aligned}$$

This states that for two belief assignments m_1 and m_2, a combined belief assignment, m_3, can be computed. In order to get a better understanding of this rule, a geometrical representation is shown in Figure 4.1. The belief assignment $m_3(C)$ is computed by summing up all the combined belief assignments where $A_i \cap B_j = C$, and then normalise the result by the inverse sum of all the combined belief assignments where the intersections are empty.

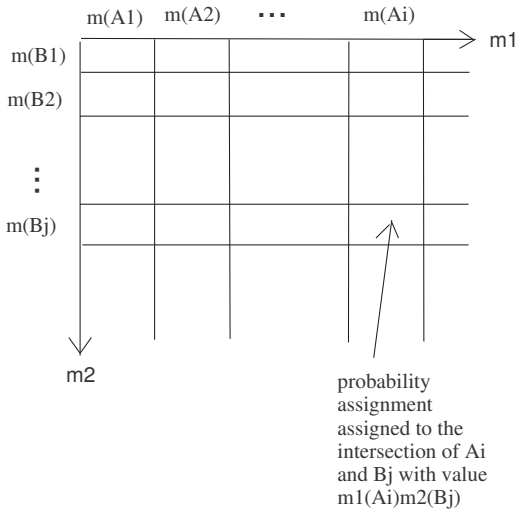


Figure 4.1: Dempster's rule of combination

As stated earlier, when one wants to use belief functions to make a decision, the attention is transferred to the pignistic level. The idea is that one can have beliefs prior to and independent of decision making, but when a decision has to be made, one has to base the decision on a probability distribution in order for it to be coherent. So a probability measure has to be made based on the belief that have been stated at the credal level, i.e., we have to make a pignistic transformation from belief functions to probability functions.

Smets [12] proposes a rule to construct a probability distribution, $BetP$, from a belief function

$$BetP(\theta) = \sum_{A \ni \theta} \frac{m(A)}{|A|}.$$

This rule represents the principle that, in lack of information, $m(A)$ should be distributed among the elements of A with an equal amount.

The notions mentioned above may be used to define two types of uncertainty, nonspecificity and discord, which have been defined in work done to extend the Dempster-Shafer theory of evidence [15]. Nonspecificity reflects an uncertainty based on lack

of information, while discord reflects an uncertainty based on conflict in the given information.

Nonspecificity is defined as

$$NS(m) = \sum_{A \subseteq \Theta} m(A) \log_2 |A|.$$

where $|A|$ denotes the cardinality of the set A and Θ signifies the frame of discernment.

The nonspecificity function measures the amount of total ignorance of whether a hypothesis is true or not. If there is little information to support either one of the possible hypotheses, some uncertainty is present, and the nonspecificity is a measure of this uncertainty. The range of the nonspecificity measure is

$$0 \leq N(m) \leq \log_2 |\Theta|$$

where $N(m) = 0$ represents full certainty ($m(\{\theta\}) = 1$ for some $\theta \in \Theta$), while $N(m) = log_2|\Theta|$ represents total ignorance ($m(\Theta) = 1$).

Discord is defined as

$$D(m) = -\sum_{A \subseteq \Theta} m(A) \log_2 BetP(A),$$

where $BetP(A)$ is the *pignistic probability distribution* as defined above.

The discord function measures the amount of conflict that is present in the information given. If there is information that gives reason to put belief in one of the hypotheses, and at the same time there is information present that gives reason to put belief in another of the possible hypotheses, this information introduces an uncertainty that is based on contradictory evidence. The discord is a measure of this type of uncertainty.

Klir [15] states that these two measures may both be present in a body of evidence. He proposes a total uncertainty measure inherent in the body of evidence as the sum of the nonspecificity and the discord:

$$U(m) = NS(m) + D(m).$$

In decision tree classification, entropy is used as a measure of "impurity" in each node. Instead, we propose to use the measure of uncertainty based on nonspecificity and discord to represent this impurity inherent in the body of evidence that is present, i.e., the training examples that are associated with each node. If we have many training examples of each class associated with a node, the discord would be high because of the contradictory "evidence". As the number of training examples decreases during the building of the tree, they will arouse less conflict and the discord will decrease.

However, less training examples will represent less information and consequently the nonspecificity will increase as the tree is built. We thus propose to use this measure as a means of computing what is in ordinary decision tree learning called “entropy”. This idea will be further explained in the following chapter.

4.3 An example

To illustrate the use of belief functions and their related concepts, we use the example introduced in Chapter 3 and shown in Table 2.1.

Now suppose that we do not know for certain if the 11 bottles we have can be drunk or should be held in store for some time, i.e., we do not know for certain whether the objects given belong to the class *Drink* or to the class *Hold*. Suppose we do not have the last column of the table.

The usual way of using belief functions is to first define the set of possible situations, the “hypotheses”, and then assign to each of the possible situations, i.e., each of the items in the Frame of Discernment, a mass of belief (a basic belief number). This assignment may for instance be based on certain given evidence, such as observed facts, or it may be done by experts in the field of interest.

In our example, the set of possible situations, the Frame of Discernment, is $\Theta = \{Drink, Hold, \}$. This means that our possible “guesses” about the objects will be either that the object belongs to class *Drink*, that it belongs to class *Hold* or that it belongs to one of the classes, but we don’t know for certain which one of them. All these possible outcomes must be assigned some basic belief number. Let us say that we contact a person which is an expert on red wines and present to this person the data about our 11 bottles. Assume that the expert gives us the answer shown in Table 4.1, with the values for $m(\{Drink\})$, $m(\{Hold\})$ and $m(\{\Theta\})$ given for each bottle. To simplify the notation, the class *Drink* is represented by a D and the class *Hold* is represented by an H.

From these values, one can compute the belief in each of the hypotheses, that the bottle in question belongs to class *Drink*, that the bottle belongs to class *Hold* or that it can belong to either one of the classes. When there are only two classes, as in this example, the computation of belief is simple:

$$
\begin{aligned}
Bel(A) &= \sum_{B \subseteq A} m(B) \\
Bel(\{Drink\}) &= m(\{Drink\}) \\
Bel(\{Hold\}) &= m(\{Hold\}) \\
Bel(\{\Theta\}) &= m(\{Drink\}) + m(\{Hold\}) + m(\{\Theta\}) = 1.
\end{aligned}
$$

	Wine dist.	Main gr. var.	Vint.	St. temp.	$m(\{D\})$	$m(\{H\})$	$m(\{\Theta\})$
x_1	Bordeaux	Cab. Sauv.	1992	12-15	0.6	0.3	0.1
x_2	Rhône	Syrah	1992	<12	0.1	0.8	0.1
x_3	Chile	Cab. Sauv.	1995	12-15	1	0	0
x_4	Bordeaux	Merlot	1995	>15	1	0	0
x_5	Chile	Cab. Sauv.	1992	12-15	0.3	0.4	0.3
x_6	Rhône	Merlot	1992	12-15	0.1	0.7	0.2
x_7	Bordeaux	Merlot	1995	12-15	0.6	0.1	0.3
x_8	Chile	Merlot	1992	<12	0.4	0.5	0.1
x_9	Bordeaux	Merlot	1992	>15	0.5	0.2	0.2
x_{10}	Rhône	Syrah	1995	<12	0.2	0.6	0.2
x_{11}	Chile	Merlot	1992	12-15	0.8	0.1	0.1

Table 4.1: Example objects with expert's opinion

These results can be used for each bottle to show how much belief we have in the bottle being ready to be drunk, or whether it should be kept in store.

If there are several distinct sets of evidence, for instance several experts to assign masses to the hypotheses, these assignments can be combined by Dempster's rule of combination to obtain a composite belief function. In our example this would correspond to a situation where we have the opinions of several experts on red wine. We could then, for each bottle, compute the combined belief in whether the bottle is ready to be drunk or not, based on these distinct statements.

Suppose now that we were to use this data set to train a classifier, as in Chapter 3. However, this time we are not able to use the decision tree learning method, because we do not have crisp labels for the training objects. We are not able to count how many objects we have from each class. Nevertheless, we should be able to use the belief functions given for each object to compute a joint belief function for the training set. We could then use this belief function to state what our belief would be of what class the next bottle of red wine we encounter will belong to. This belief function could be used to compute the nonspecificity and discord contained in the training set. Then we have a way of measuring the inherent uncertainty of the training set. This uncertainty measure would correspond to the entropy measure we have seen in the decision tree learning method. Using this uncertainty measure, we will thus be able to use the decision tree learning for building a classifier. How to do this will be elaborated in the next chapter.

4.4 Summary

Many interpretations of Dempster-Shafer theory of evidence have been presented. The general opinion is that this theory can be used to model one's degree of belief. This belief is connected to certain given hypotheses, and expresses how much we believe in the hypotheses, based on the evidence given.

The Dempster-Shafer theory of evidence can be a good means of representing uncertainty, as it allows you to express ignorance. This can in turn be used when one wishes to express several types of uncertainty, as nonspecificity and discord.

It is our belief that, combined with the decision tree method, the concepts of the Dempster-Shafer theory of evidence can be used as basis for a method that makes the classification task more robust to uncertainty - both in the training data and the test data. Our proposed method will be outlined in the next chapter.

Chapter 5

The method proposed in our work

5.1 Introduction

As stated in Chapter 2, one of the problems in diagnostic systems is finding a way to handle uncertainty in data. We have in our work been concerned with handling data with uncertain classification labels. This corresponds to training the system on objects that do not have a crisp classification, i.e., we are not able to state definitely which of the possible classes they belong to. Instead we may have for each object a belief function that tells, for each class, what our belief is that the object in question belongs to this class.

We propose in this chapter a method that introduces belief functions into the decision tree learning method. This will allow us to use as training data both objects with crisp labels and objects with uncertain labels. Our method builds a belief function from previously seen cases, i.e., the training data. As output, the method will give not only a classification label, but also a belief function that tells us how the classification is supported, i.e., states our belief in the classification. We will in this chapter first describe how we propose to use belief functions in the decision tree learning method. Then we will elaborate this and show how this can be used to build classifiers from training data with uncertain labels.

In order to test our method, we have implemented the method in MATLAB. A description of our program is given in this chapter.

5.2 Modification of the entropy computation

As stated earlier, in diagnostic systems the problem is to build a classifier that is able to classify an object or an instance as belonging to one of several possible classes. The

classifier is built by training the system on previously observed objects. We wish to be able to state, based on what we have seen earlier, what is the most likely class the new object should belong to.

In such a problem, our knowledge of the domain is not exhaustive, since the only representations we have of possible situations to be encountered are the objects we have already seen. According to Smets [13] and [16] this situation is one where the probability distribution of the possible outcomes among our objects is only partially known. We do not know the exact composition of the possible classes among our objects. Therefore, we are not able to build a probability distribution for our situation, but we can state some beliefs.

According to the transferable belief model, beliefs are quantified by belief functions. In our diagnostic problem, the objects can be seen as random events that occur according to an underlying probability function. This probability function is only partially known to us, but we have some knowledge of previous cases that can induce a belief function. When we build a classifier from our previously seen cases, we use this belief function to state our belief in how the future cases will behave.

To illustrate this situation, Smets [16] gives as an example an urn in which there are 100 balls which are either black, white or red. The amount of balls of each colour is not exactly known, but we know that there are between 30 and 40 black balls and between 10 and 50 white balls. We are interested in finding out what our belief is that a randomly selected ball will be black. If we select 50 balls at random with replacement and observe their colour, our belief will probably change, because now we have a wider experience.

We can regard a classification problem as a situation of this kind. We have observed some events, our previously observed objects, and from them we are interested in building a belief function for the possible classes our objects may belong to. If we accept the close world assumption that one and only one of the possible classes can be assigned to each object, i.e., we have a frame of discernment that is mutually exclusive and exhaustive, we can build a belief function on this frame of discernment. If the underlying probability function of our situation is fully known to us, our belief function should be equal to the probability function, according to the principle Smets refers to as the Hacking Frequency Principle. However, if this is not the case, as in our classification problem, we will need a means of computing our belief function based on the knowledge we have.

Smets proposes in [16] a way of computing this belief function for the case where there are only two elements in the frame of discernment, i.e., there are only two possible classes the objects can belong to. Smets calls his two classes *success*, S, and *failure*, F, and states that if you observe r successes and s failures on n independent experiments,

where $r + s \leq n$, the belief function on $\Theta = \{S, F\}$ will be

$$Bel_{\Theta}(S|r, s) = \frac{r}{r + s + 1}$$

$$Bel_{\Theta}(F|r, s) = \frac{s}{r + s + 1}$$

and

$$m_{\Theta}(S \cup F|r, s) = \frac{1}{r + s + 1}$$

The method we propose is based on this result. However, since Smets has given the result only for two-class problems, we will only look at classification problems with two classes. Suppose that we have two classes, class 1 and class 2, and that we have observed a number of objects from each class. These objects will constitute our training set. We can then compute our belief that the next observed object wil belong to class 1 or class 2. In the decision tree learning method, we can use this knowledge to build a belief function for each node, based upon the objects that have been observed and are associated with the node. When the object to be classified have been put through the tree and ended in a leaf node, we use the belief function associated with this node to state our belief of which class this new object belongs to.

Suppose we denote by n_1 the number of objects observed from class 1 and n_2 the number of objects observed from class 2. The belief function at each node regarding the two classes can be computed with Smets' results. In order to use the belief in our computations, we will work with the basic belief assignment, $m(A)$, instead of the belief function, $bel(A)$, since this is just another way of representing belief.[1] We get the basic belief assignment

$$m(\{1\}) = \frac{n_1}{n + 1}$$

$$m(\{2\}) = \frac{n_2}{n + 1}$$

$$m(\{1, 2\}) = \frac{1}{n + 1}$$

where n is the total number of objects.

[1]Remember from Chapter 4 that in the case where we have only two classes, the belief for the two classes, $Bel(\{1\})$ and $Bel(\{2\})$ equals the basic belief numbers, $m(\{1\})$ and $m(\{2\})$.

The belief function obtained for each node may be used to determine the splits to use for building the tree. In ordinary decision tree building, the entropy is used as a measure of the impurity of a node. The goal is to split the training objects such that the entropy in each leaf node is as small as possible, because then we have found the best way of discerning objects from the different classes. In the same way, our goal should now be to split the training objects in such a way that the uncertainty produced from the computed belief function is as small as possible.

Thus, we propose to substitute the entropy measure in decision tree learning with a measure of uncertainty based on belief functions, i.e., the entropy measure given in Chapter 3:

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i$$

will be substituted with the uncertainty measure proposed in Chapter 4

$$U(m) = NS(m) + D(m)$$

where NS is the nonspecificity measure and D is the discord measure.

The idea would now be to use the algorithm of the decision tree method to build a tree, but using the uncertainty measure stated above to find the best attribute on which to split for each node. For all the training objects associated to a node, we will compute a belief function, and from this belief function we will compute the measure of uncertainty. We will choose the attribute and the split that results in the smallest possible uncertainty in the child nodes.

The nonspecificity measure will thus be

$$\begin{aligned} NS(m) &= \sum_{A \subseteq \Theta} m(A) \log_2 |A| \\ NS(m) &= m(\{1\}) \log_2(1) + m(\{2\}) \log_2(1) + m(\{1,2\}) \log_2(2) \\ &= m(\{1,2\}) \end{aligned}$$

The discord measure will be

$$
\begin{aligned}
D(m) &= -\sum_{A\subseteq\Theta} m(A)\log_2 BetP(A) \\
D(m) &= -m(\{1\})\log_2(\frac{m(\{1\})}{1}+\frac{m(\{1,2\})}{2}) \\
&\quad -m(\{2\})\log_2(\frac{m(\{2\})}{1}+\frac{m(\{1,2\})}{2}) \\
&\quad -m(\{1,2\})\log_2(1) \\
D(m) &= -m(\{1\})\log_2(m(\{1\})+\frac{m(\{1,2\})}{2}) \\
&\quad -m(\{2\})\log_2(m(\{2\})+\frac{m(\{1,2\})}{2}).
\end{aligned}
$$

The nonspecificity measure will reflect the uncertainty that is represented by little information. If we have observed few objects from each class, the nonspecificity will be high because there is little information present to support our beliefs. The discord measure will reflect the uncertainty that is represented by the conflict that may arise from our information. When we have observed objects from different classes, they each provide "evidence" in favour of their class, and there would thus be conflicting evidence present.

We see from this that if we use these measures in the context of decision trees, the nonspecificity at the root node will be relatively small, since we have all the training objects associated with this node, i.e., we have quite a lot of information present. As the building of the tree proceeds, the training objects are divided into smaller sets. This will result in an increase of nonspecificity, since the objects remaining provide less information. The discord, however, will be relatively high at the root node, since we probably will have many objects from each class at this stage. During the building of the tree, the splitting of the training set results in more "pure" sets, i.e., sets with less conflicting information about the class associated with the remaining training objects. This will result in the discord decreasing as the tree is grown.

The behaviour of the two measures suggests that during the building of the decision tree, the uncertainty measure we propose will initially decrease, until a stage where the nonspecificity increases more than the discord decreases, and the uncertainty will start to increase. This suggests a criterion to use in order to decide when to stop building the decision tree. The tree is grown as long as the uncertainty decreases, but when the uncertainty starts to increase, the growth stops. This way we may avoid the problem of overfitting, since the nonspecificity will tell us when there are too few objects left to consider for reliable information, i.e., the information is too specific.

Another point of our approach is that by using belief functions to build the tree, we will take into account how reliable our previous information is. We will not only get

a classification, but we will also get a belief function telling what the classification is based on. It is obvious that if we have a large amount of training objects to build our tree from, the classifier will be more reliable than if we only had a few training objects. The belief functions obtained will reflect this difference. In the ordinary method of decision tree learning, the entropy is based on the concept of proportions, how many training objects there are from one class relative to how many training objects there are altogether. That is, the entropy is computed based on the ratio

$$p_i = \frac{n_i}{n}.$$

This ratio will be the same for instance in a case where we have 1 object from class i out of 10 objects altogether as in a case where we have 10 objects from class i out of 100 objects altogether.

In our method these two situations will be distinguished, because the belief functions will be different. Our method would, for two classes, for instance give the results:

Case 1:

$$m(\{1\}) = \frac{1}{11} = 0.09$$
$$m(\{2\}) = \frac{9}{11} = 0.81$$
$$m(\{1,2\}) = \frac{1}{11} = 0.1$$

Case 2:

$$m(\{1\}) = \frac{10}{101} = 0.099$$
$$m(\{2\}) = \frac{90}{101} = 0.891$$
$$m(\{1,2\}) = \frac{1}{101} = 0.0099$$

We see from this that the next object observed in both cases would be classified as belonging to class 2, but we will have a more reliable result in the second case than in the first case. So we see that the belief function can give additional and useful information.

5.3 Classification with uncertain labels

The method outlined above provides the basis for what we want to achieve with our work, to provide a method for building classifiers from training data with uncertainty

in their class labels. In other words, we would like to be able to handle situations where the training data do not have a crisp classification, but a basic belief assignment associated with them.

In order to understand how this can be done, a more thorough explanation of Smets' results [16] is needed. As stated above, Smets proposes a method of computing a belief function when the underlying probability function is only partially known. The background for his results is outlined here.

Smets' starting point is that we have a frame of discernment, $\Theta = \{S, F\}$, and a set of probability functions over Θ, $P_\Theta = [0, 1]$. He states that from this we can construct a new frame of discernment, $W = P_\Theta \times \Theta$, on which we can build a belief structure m_W. All the focal elements of m_W consists of a set of mutually exclusive and exhaustive intervals that are either in the domain of S or in the domain of F. Since W is not a finite space, the basic belief masses on it will be regarded as densities, and are called basic belief densities. Figure 5.1 shows an example of a basic belief density on W on the focal element $([a, 1], S) \cup ([0, a), F)$.

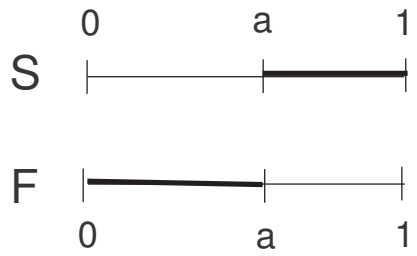


Figure 5.1: An example basic belief density on W

Smets chooses to use the commonality function in order to state our belief on W. The commonality function is chosen, because it has the property that it makes the use of Dempster's rule of combination easy.

Suppose that we know that the probability of success, $P(S)$, is somewhere between a and b, and $0 \leq a \leq b \leq 1$. The belief is obtained by integrating on the basic belief densities, and is in the form of the commonality function given by

$$Q([a, b]) = \int_0^a \int_b^1 m([x, y])\, dy\, dx.$$

The mass function can be obtained from

$$m([a, b]) = -\frac{\partial^2 Q([a, b])}{\partial a \partial b}$$

These formulae are generalisations of the corresponding commonality and mass formulae for the finite case given in Chapter 4.

If we perform independent experiments and observe the number of successes and failures, the commonality function induced by a success, a failure or by $S \cup F$ is

$$\begin{aligned} Q([a,b]|S) &= a \\ Q([a,b]|F) &= 1-b \\ Q([a,b]|S \cup F) &= 1. \end{aligned}$$

To obtain the commonality function induced by r successes and s failures, we use Dempster's rule of combination, which in case of the commonality function corresponds to multiplying all the obtained commonality functions, and we get

$$Q([a,b]|r,s) = a^r(1-b)^s$$

which after derivation and normalisation yields

$$m([a,b]|r,s) = \frac{\Gamma(r+s+1)}{\Gamma(r)\Gamma(s)} a^{r-1}(1-b)^{s-1}$$

where Γ is the gamma function.

If we want to compute the belief in the next outcome being a success given the previous observations of r successes and s failures, we get

$$\begin{aligned} Bel(S|r,s) &= \int_0^1 \int_a^1 Bel(S|P(S) \in [a,b]) m([a,b]|r,s)\, db\, da \\ &= \int_0^1 \int_a^1 a \frac{\Gamma(r+s+1)}{\Gamma(r)\Gamma(s)} a^{r-1}(1-b)^{s-1}\, db\, da \\ &= \frac{r}{r+s+1} \end{aligned}$$

which was stated in the previous section.

If for experiment i we have a belief function concerning the outcome, i.e., we have $m_i(S)$, $m_i(F)$ and $m_i(S \cup F)$, we will get

$$Q([a,b]|m_i) = m_i(S)a + m_i(F)(1-b) + m_i(S \cup F).$$

For n experiments we get

$$\begin{aligned} Q([a,b]|m_1, m_2, ..., m_n) &= \prod_{i=1}^{n} [m_i(S)a + m_i(F)(1-b) + m_i(S \cup F)] \\ &= \sum_{i+j \leq n} \alpha_{ij} a^i (1-b)^j \end{aligned}$$

where α_{ij} are constants.

We can then compute the belief that our next experiment will result in a success, based on the previous experiments

$$bel(S|m_1, m_2, ..., m_n) = \sum_{i,j|i+j\leq n} \alpha_{ij} a^i (1-b)^j \frac{i}{i+j+1}$$

For our classification situation, this will give us a way of computing a belief function based on all the previously observed cases' belief functions. Then we will be able to build a classifier with the method outlined in the previous section, but we build it based on training objects which have uncertain labels instead of crisp labels.

5.4 Implementation of our method

In order to test our method, we have built a program in MATLAB implementing the above ideas. The source code of the program is shown in Appendix A.

The program is developed to handle continuous attribute values only, since our purpose is to test our method and not, at this stage, to build a complete program. But the extension of the program to other attribute types would not involve much work.

The program includes two learning methods, one that is based on learning from examples with crisp labels, and one that is based on learning from examples with uncertain labels. The program is based upon the ID3-algorithm explained in Chapter 3 and is modified according to what is outlined in the previous sections. The program builds a decision tree from given training objects, and the uncertainty measure $U(m) = NS(m) + D(m)$ is used to decide which attribute and which split to choose at each node.

To find the best split for each attribute, two methods are used. One method finds the range of the attribute values, and proposes a split at certain given intervals for this range. The other method counts the number of objects, and proposes splits so that the objects are divided in equal groups. Then the uncertainty is computed for each possible split, and the best one is chosen, i.e., the one which yields least uncertainty. For the method that learns from crisp labels, the best split is first chosen according to the amount of objects, and then the split chosen is adjusted by searching in the area around the split by using both splitting methods and choosing the best one. For the method that learns from uncertain labels, the computation time is very large, so the split is chosen only using the method that splits according to the amount of objects and not adjusted afterwards.

The program takes as input a text file with one training object represented on one line, with the attribute values succeeding each other separated by blank space, and the decision attribute as the last attribute. The user can choose between learning from

crisp or uncertain labels, as well as choose to build a full tree or to use a stop criterion. The stop criterion is implemented as building the tree as long as the uncertainty is decreasing. As we will show in the next chapter, the uncertainty measure had to be adjusted with a parameter in order to demonstrate the expected behaviour of initially decreasing until a point is reached where it starts to increase. Our uncertainty measure is therefore implemented as

$$U_\lambda(m) = NS(m) + \lambda D(m)$$

The output of the program is a decision tree. At each node the uncertainty and either the number of objects from each class or the belief function are stored. The attribute to split on, the split, and the most common class are also stored at each node. The program also produces a text file with information from the building of the tree, such as the uncertainty computation at each node.

For testing the classification, the program takes as input a text file with test objects in the same format as the training objects. The output of the test classification is a text file that lists the test objects with their known classification and the produced classification of the decision tree. Both the belief function and the classification is shown. The error rate is also computed.

There are two methods of computing the classification error, according to which classification labels are used for testing. Since we have beief functions as output from the classifier, the error rate should not be a pure misclassification error rate, but an error measure stating the difference between the output and the belief that was assigned in advance.

If objects with crisp labels are used for testing, the error measure is a kind of misclassification error rate. In advance, a class is chosen for each object based on the object's belief function. The classifier produces a new belief function, and the class with highest belief is chosen as the class of the object. The error measure is thus computed based on the difference between the class indicated by the previous mass assignment and the class indicated by the classifier's output. This error is thus the percentage of wrongly classified objects given by

$$Error = \frac{wrongly\ classified\ objects}{total\ number\ of\ objects}$$

For testing objects with uncertain labels, it is not as useful to compute the error measure given above, since the belief functions given as output should be compared to the belief functions previously assigned to each object. It would give more information about the performance of the classifier to compute the disagreement between each object's two belief functions. So, to better assess the classification in these situations, an error

measure is computed based on the concept of pignistic probability:

$$Error = \frac{1}{n}\sum_{i=1}^{n}(1 - \widehat{BetP_i}(\widehat{m_i}))$$

where $\widehat{BetP_i}$ is the pignistic probability for object i induced by $\widehat{m_i}$, the belief function produced for object i. The pignistic probability for object i is given by

$$\widehat{BetP_i}(\widehat{m_i}) = \sum_{A \subseteq \Theta} m_i(A)\widehat{BetP}(A).$$

The interpretation of this error measure is that the error is small if the produced belief function does not differ much from the previously assigned belief function, but it does not consider as an error a classification where the mass has been "transferred" from uncertainty to one of the classes.

5.5 Summary

We have in this chapter outlined our proposed method and given its background and underlying ideas. We have introduced a method based on the decision tree learning method, combined with the notions of belief functions. Our method consists of modifying the entropy measure in the ordinary decision tree learning method to an uncertainty measure based on belief functions. Our method has two possible ways of building a classifier, one that concerns learning from objects with crisp labels, and one that concerns learning from objects with uncertain labels.

This method allows to classify objects with crisp labels at least as well as other methods, and in addition it offers a way of handling objects with uncertain labels. This means that one is able to make the most of the information that is in this kind of representation. We will in the next chapter show results obtained from experiments with our method.

Chapter 6

Experiments and results

6.1 Introduction

For verification of our method, we have performed a test on data from a real world problem. The results from running these data through our method are compared to a decision tree learning program developed by Quinlan, based on the ID3/C4.5 algorithm.

This chapter presents the data and a short description of the domain they are taken from. It also presents the C4.5 program with which our results were compared. A presentation of the results obtained is given in the section called Results. The results will be analysed in the next chapter.

6.2 The example data set

The chosen data set is taken from the domain of monitoring sleep stages. The problem stated is to detect different waveforms in the sleep electroencephalogram (EEG), and in particular detecting the transient EEG pattern of the K-complex wave. For a thorough presentation of this problem, see [17] and [18].

The activity of the human brain during sleep is classified into stages. The problem of detecting the K-complex is important in the assessment of sleep stages. It is a difficult task, because this transient signal has much in common with the patterns of a waveform that is called the delta waveform. Figure 6.1 shows examples of a K-complex waveform and a delta waveform.

The data used to test our classification method are EEG signals measured 64 times during 2-second intervals for one person during sleep. Each such 2-second interval

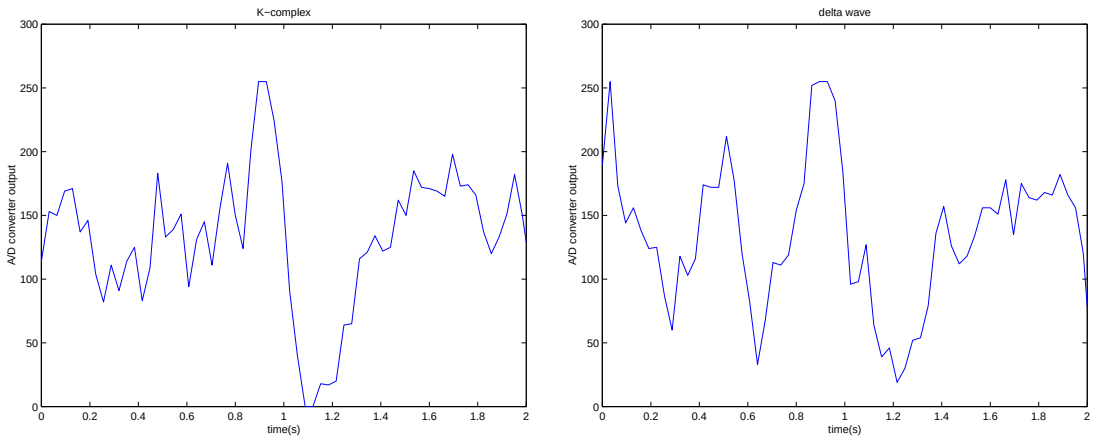


Figure 6.1: Examples of the K-complex and the delta wave

will be regarded as an object with 64 attributes containing values corresponding to the 64 measured values. Since the problem is to train a classifier to distinguish K-complex waveforms from delta waveforms, examples of these two waveforms have been extracted. We have in our data set 781 objects representing patterns of the K-complex waveform and 397 objects representing patterns of the delta waveform.

Since the K-complex pattern is difficult to detect, five experts on the domain have been asked to look at the 781 objects that represent the K-complex waveform and state for each object whether he believes that the pattern is in fact representing a K-complex signal or not. We have then for each of these objects five expert opinions of whether it is a K-complex object, in which case the expert has put the value “1”, or whether the object is representing some other waveform, in which case the expert has put the value “0”.

We will also adopt the closed-world assumption that there are only two classes of objects, *K-complex* (class 2) objects and *not K-complex* objects (class 1), in our case represented by the delta wave objects. The reason for this is that if an expert says that the object is not a K-complex pattern, that does not necessarily mean that it is a delta wave pattern, it may mean that he does not know what it is, or it may mean that he has recognised it as something else. However, since the only other waveform we have examples of in our data set is the delta waveform, this is the only other waveform that the classifier will be trained to recognise.

Figure 6.2 shows how the data are distributed according to the experts’ classification.Two attributes containing the most important information regarding the classes have been found by feature extraction, which makes us able to plot the data in two dimensions. The K-complex objects with consent from different number of experts are plotted together with the delta objects to see how they differ from them.

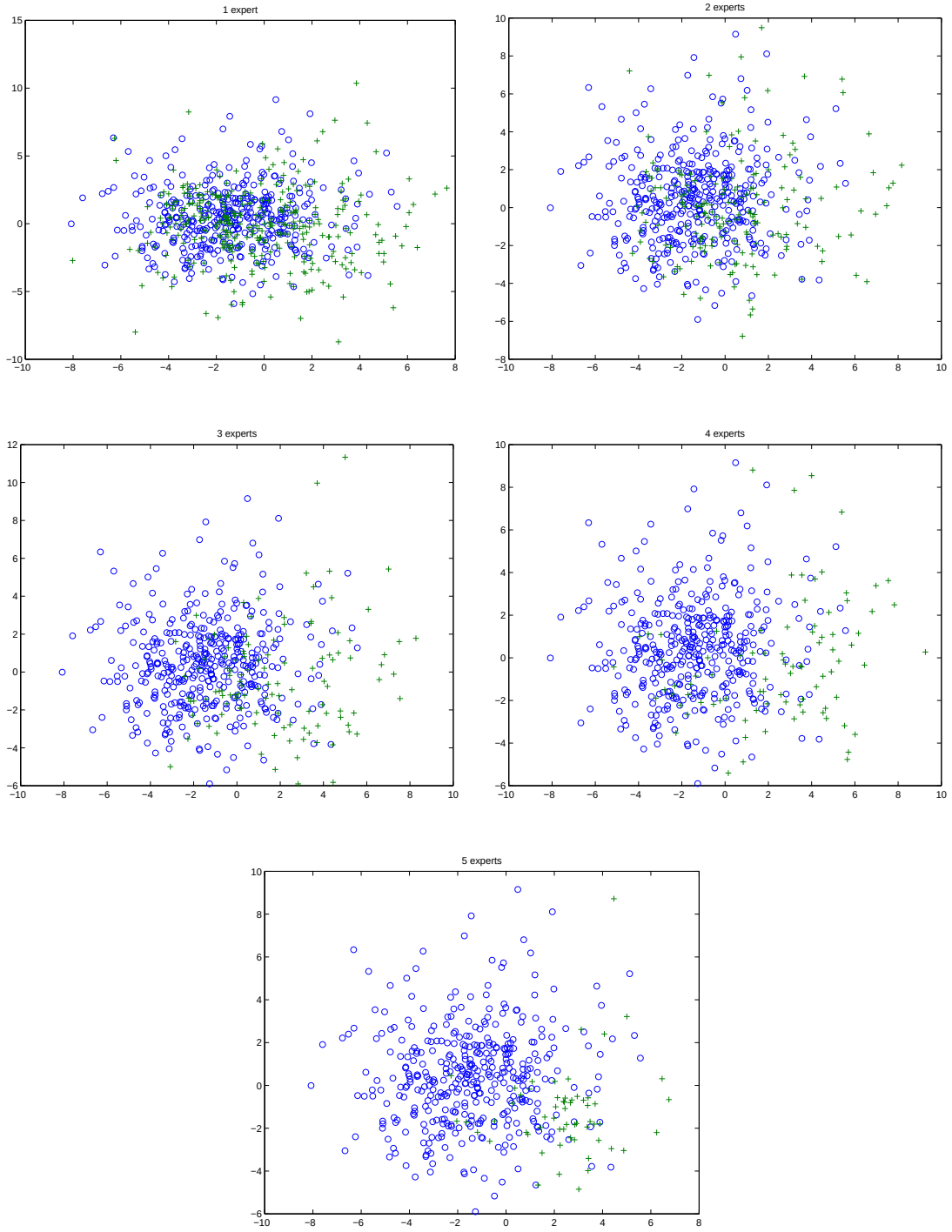


Figure 6.2: The distribution of the objects according to the experts' classification

We see from the figure that the objects for which all the experts agree that it is a K-

complex object are quite separate from the delta objects. As there is consent from fewer experts, the K-complex examples tend to move in the direction of the delta objects and be more uncertain.

We also observe from Figure 6.2 that the objects that have been classified as K-complex objects by only one expert are widely distributed in the entire space. This confirms our above statement that if the objects are not K-complex objects, they are not necessarily delta objects. The figure also confirms that the experts' classification is quite a realistic one.

For our method based on classification of objects with crisp labels, we have defined all the 781 objects representing patterns of the K-complex waveform as objects from class 2, and the 397 objects representing patterns of the delta waveform as objects from class 1.

To use our method for classification with uncertain labels, we have to build a belief function for each object. For the 781 K-complex objects this belief function is based on the 5 experts' opinions. There is no common way of assigning belief in a case like this, so we have suggested a heuristic that assigns belief according to the number of experts who have classified the object as a K-complex object. If only one expert is certain that the object is a K-complex object, the belief assigned to class 1 should be quite small, and the belief assigned to $1 \cup 2$ should be quite high. If all the experts have classified the objects as a K-complex object, the belief assigned to class 1 should be high. So, we have used the belief functions shown in Table 6.1 for our K-complex objects.

Number of experts	$m(\{1\})$	$m(\{2\})$	$m(\{1,2\})$
1	0	0.2	0.8
2	0	0.4	0.6
3	0	0.6	0.4
4	0	0.8	0.2
5	0	1	0

Table 6.1: Belief functions for uncertain K-complex objects

As shown in Figure 6.2, the objects with assent from a small number of experts can be anything, they are not necessarily delta objects. This is reflected in our assigned belief functions, since in these cases the mass of belief is not added to class 1, but to $1 \cup 2$.

For the objects that represent the delta wave pattern, the mass assignment will be

$$m(1) = 1$$
$$m(2) = 0$$
$$m(1,2) = 0$$

since we know for sure that these objects are examples of the delta waveform.

6.3 The C4.5 program

To evaluate the behaviour of our method, we will compare our results with the results obtained from a traditional decision tree learning method. We have chosen to use the C4.5 program developed by J. R. Quinlan. His program is fully described in [9].

Quinlan's C4.5 program is implemented in C for the UNIX environment. It has options both for building a decision tree and for creating a set of production rules, which is another way of representing the same information.

The program uses three files of data to build a tree and to test it. The attributes for the data set have to be defined in a file with the extension *.names*. This file starts with a list of all the possible values for the decision attribute, then a list of the conditional attributes and their type follows which states whether they are continuous or have certain predefined discrete values. The training data are listed in a file with the extension *.data*, with one line in the file representing one object. The test objects are listed in a file with the extension *.test*. All these three files must have the same stem. The program handles unknown attributes, and these are represented in the data files with a '?'.

The C4.5 program uses the algorithm stated in Chapter 3 to build a decision tree. The best attribute to split on is chosen by computing the information gain based on the entropy. The best tree is found by building a full tree and then pruning it upwards by looking at how the error rates will change if a subtree is replaced with a leaf node.

The output of the program is a listing which first states what data files and options have been used, then the produced decision tree is listed followed by a simplified decision tree, i.e., the pruned version of the tree, and ends with listing the classification results for both trees on the training data and the test data. The decision tree is shown as a set of rules testing on an attribute value according to a split. Each leaf node is followed by some numbers in parentheses, which indicate the number of training objects that are associated with this leaf node, and the number of misclassified training objects for this node.

The results are listed both for evaluation on the training data and for evaluation on the test data. For each case, both the results obtained from the full tree and from the pruned tree are shown. The results list the size of the tree, i.e., the number of nodes, the number of misclassified objects and a misclassification error rate given in percentage of the total number of objects. An estimated error rate is also given, computed from the upper limit of the confidence limits obtained from regarding the misclassification error rate as a probability distribution of error occurring. At the end of the output,

a confusion matrix is shown, giving the number of objects that have been classified correctly and wrongly for each class.

We will run our data set through the C4.5 program, and the output of this program will be compared to the results obtained from using our method. This way we will be better able to assess the performance of our method.

6.4 Results

Experiments have been made both with the method for building a classifier from training objects with crisp decision labels and with the method for building a classifier from training objects with uncertain decision labels. The experiments and their results are divided in two parts, according to the two methods.

6.4.1 Crisp labels

The first experiment we did was to see how our method worked on an ordinary data set, i.e., a data set with crisp labels for the decision attribute. We built a data set from our sleep data described above, with all the 1178 objects, 397 objects representing the delta wave pattern (class 1) and 781 objects representing the K-complex pattern (class 2).

With this data set, both the C4.5 program and our method performed poorly, with a misclassification error rate of around 30 percent. This was to be expected, since the data set consists of all the data available from the K-complex class, i.e., it contains objects which the majority of the experts have classified as not being an example of the K-complex pattern. Consequently, this set of objects contains uncertain information in which it will be difficult to find a pattern. The data are shown in Figure 6.3.

In order to get more distinct results, we removed the most uncertain objects from the data set. We thus built a data set consisting of the 397 objects from the “delta” class and the most certain objects from the “K-complex” class. From Figure 6.2 we see that the objects with assent from all the experts obviously can be regarded as certain K-complex objects, since they are quite separated from the delta objects. In addition, the objects with assent from 3 and 4 experts can also be regarded as relatively certain, because the objects do not yet become totally mingled with the delta objects. However, as the most certain objects we chose the objects for which at least 4 of the experts had classified it as an instance of the K-complex waveform, to be able to verify our method with quite certain objects. We thus had 147 certain K-complex objects.

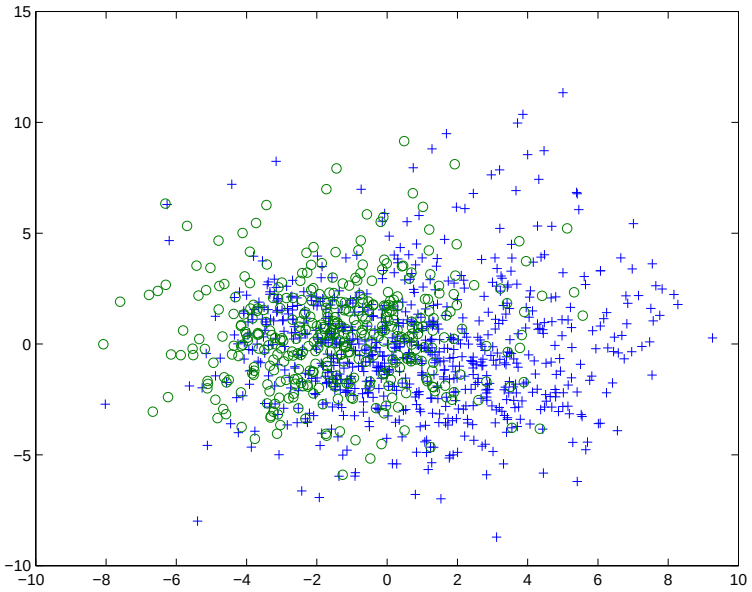


Figure 6.3: All the 1178 data objects. '+' are the objects from the K-complex class and 'o' are the objects from the delta class.

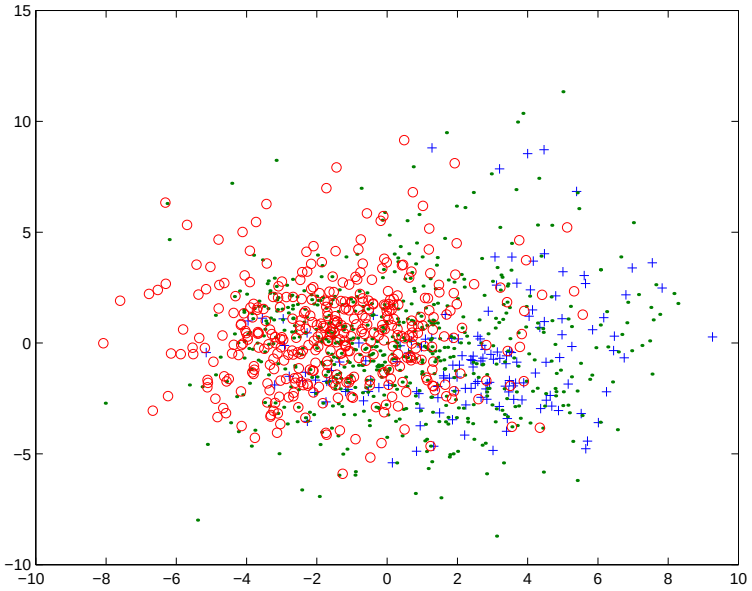


Figure 6.4: All the 1178 data objects. '+' are the most certain objects from the K-complex class, '.' are the uncertain objects from the K-complex class and 'o' are the objects from the delta class.

In Figure 6.4 the uncertain data are plotted as a separate class in order to illustrate the difficulty in classifying these objects. As the figure shows, the most uncertain data are not easy to distinguish from the certain ones. Figure 6.5 shows the situation without these uncertain objects. We can see from the figure that if we select the most certain objects, i.e., all the delta objects and the K-complex objects for which 4 or 5 experts have given their assent, the classification task is easier.

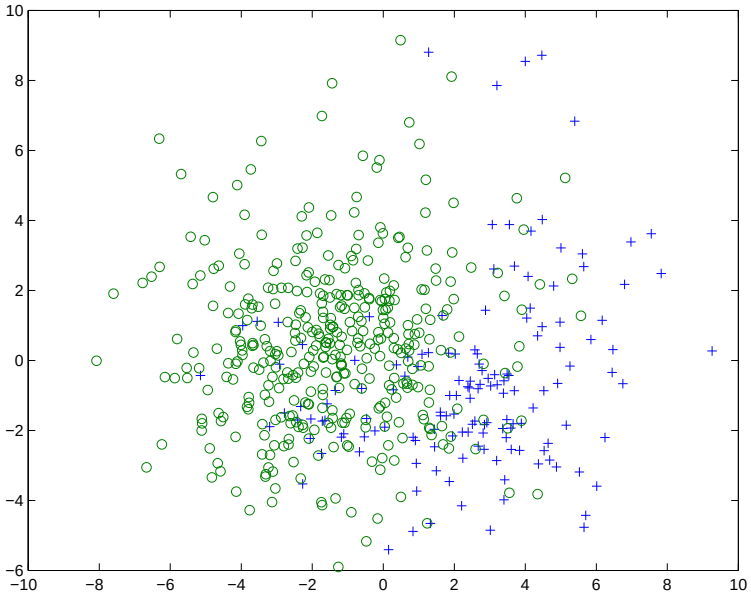


Figure 6.5: The 544 most certain objects. '+' are the most certain objects from the K-complex class and 'o' are the objects from the delta class.

The 544 objects of the data set were randomly mixed, in order to avoid any impact from the fact that all data were measured during the same sleep period, which means that they were ordered sequentially by time in our initial data files. Five different random sets were made, and they were each divided in a training set of 425 objects and a test set of 119 objects. The result from running these data through our program is shown in Table 6.2.

	Misclassification error rate					
	Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
Train	0.0024	0.00	0.0024	0.0024	0.0024	0.0019
Test	0.2437	0.2101	0.1513	0.1345	0.1513	**0.1782**

Table 6.2: Error rates for the 5 data sets.

With these data sets it appeared that even though the stop criterion of building the tree as long as the uncertainty measure was decreasing was used, an approximately full tree was built each time. It can be seen from the large difference in the results for classification on the training set and on the test set that the overfitting is high. This suggests that, as presumed, we should modify our uncertainty measure by inserting a parameter, λ, in order to be able to obtain the desired behaviour from the uncertainty measure. We then modified the uncertainty measure to be

$$U_\lambda(m) = NS(m) + \lambda D(m)$$

Inserting this parameter means that we have to run the data sets through the program for different values of λ in order to choose the best tree. This means that we will have to adjust the parameter for our data to find which λ value gives the most optimal tree. This is analogous to the pruning method in the C4.5 program, where the misclassification error rate is computed for each size of the tree in order to choose the tree that gives the lowest error rate.

We tried running the same 5 data sets as above through the program for different values of λ. The results from this experiment can be seen in Table 6.3.

		Misclassification error rate					
λ		Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
1	Train	0.0024	0.00	0.0024	0.0024	0.0024	0.0019
	Test	0.2437	0.2101	0.1513	0.1345	0.1513	**0.1782**
0.5	Train	0.0024	0.00	0.0024	0.0024	0.0024	0.0019
	Test	0.2269	0.1849	0.1596	0.1345	0.1513	**0.1714**
0.2	Train	0.0071	0.0047	0.0094	0.0047	0.0094	0.0071
	Test	0.2269	0.2017	0.1177	0.1261	0.1261	**0.1597**
0.15	Train	0.0071	0.0071	0.0142	0.0118	0.0212	0.0123
	Test	0.2269	0.1933	0.1261	0.1092	0.1345	**0.1580**
0.1	Train	0.0330	0.0566	0.0708	0.0212	0.0566	0.0476
	Test	0.1849	0.1681	0.1765	0.1177	0.1261	**0.1546**
0.075	Train	0.0660	0.0802	0.0943	0.0920	0.0778	0.0821
	Test	0.2017	0.1849	0.1429	0.1597	0.1345	**0.1647**
0.05	Train	0.10142	0.0991	0.0943	0.1274	0.1321	0.1109
	Test	0.1765	0.2101	0.1429	0.2017	0.1765	**0.1815**
C4.5	Train	1.7/2.1	1.2/1.2	1.9/3.5	1.7/3.3	2.1/2.1	1.72/2.44
	Test	19.3/19.3	20.2/20.2	13.4/14.3	16.0/19.3	19.3/19.3	**17.64/18.48**

Table 6.3: Error rates for different values of λ

The value $\lambda = 1$ obviously corresponds to the case above without the λ parameter. The

table also shows the results obtained by running the same data sets through the C4.5 program. The C4.5 program gives, as stated earlier, as output the misclassification error rate in percentage of the total number of objects. There are two numbers given for each case, the first number is the error rate for the full tree, and the second number is the error rate for the pruned tree.

We see from these results that for this data set the best tree will be obtained when $\lambda = 0.1$. We also see that the misclassification error rate is slightly better than before we introduced the λ parameter.

Table 6.4 shows the confusion matrices obtained for the test set of the 5 data sets, both for our method with $\lambda = 0.1$ and for the C4.5 program. The confusion matrices show how many objects have been correctly classified as belonging to class 1, how many objects have been wrongly classified as belonging to class 2 and so on. We see from the results that the two methods perform approximately equally, with a small advantage for our method.

Data set 1				
	Our		C4.5	
	Classified as			
	1	2	1	2
Real 1	63	17	71	9
2	12	27	14	25

Data set 2				
	Our		C4.5	
	Classified as			
	1	2	1	2
1	66	14	67	13
2	11	28	11	28

Data set 3				
	Our		C4.5	
	Classified as			
	1	2	1	2
1	70	10	71	9
2	8	31	8	31

Data set 4				
	Our		Quin	
	Classified as			
	1	2	1	2
Real 1	71	9	71	9
2	7	32	14	25

Data set 5				
	Our		Quin	
	Classified as			
	1	2	1	2
1	70	10	69	11
2	8	31	12	27

Table 6.4: Confusion matrices for the 5 data sets with 544 objects

6.4.2 Uncertain labels

Experiments were also done to test how our method would work for classification with data labeled with belief functions as decision labels instead of crisp labels. Because our implementation of the computation of belief functions when building a tree in this

case is very time consuming, we had to build smaller data sets. We also found from experiments that by using only two attributes found by feature extraction, we got as good results as when using all the 64 attributes. This also reduced the computation time. So we decided to use only two attributes for these experiments.

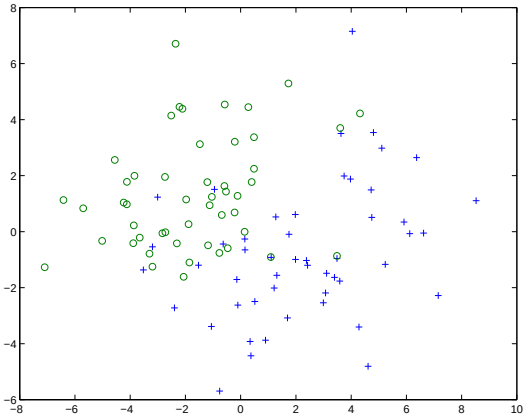
The data sets we used for these experiments were of two types. One data set contained delta objects and only the most certain K-complex objects, which we chose to be the objects for which 3,4 or 5 of the experts had classified them as being an example of the K-complex pattern. The other data set contained also the data selected as examples of the K-complex pattern, but which at least three of the experts classified as not being an example of such. The "certain" data set used for training contained 100 objects, i.e.,50 randomly drawn objects among the most strong cases of K-complex examples, and 50 randomly drawn objects among the examples of the delta wave. The "uncertain" data used for training contained 150 objects, i.e., the same 100 certain objects as above and in addition 50 randomly drawn objects from the uncertain K-complex examples.

These two data sets were used to train the system, both with crisp labels and with belief function labels. In order to evaluate the two methods' behaviour in comparison with each other, two validation sets were also built, consisting of different objects. These two sets followed the same structure as the training sets, i.e., we made a "certain" validation set containing 100 certain objects and an "uncertain" validation set containing in addition 50 uncertain objects. Figure 6.6 shows the two training sets and the two validation sets.

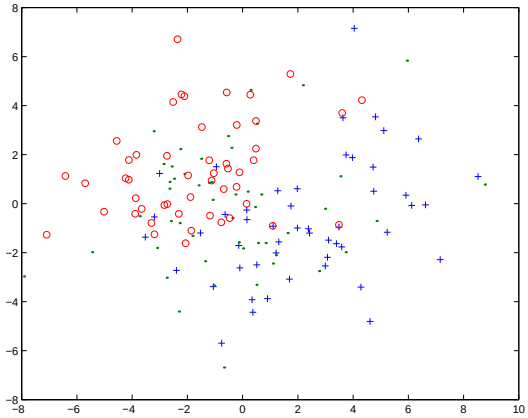
Normally in a classification problem, we have to deal with uncertain objects. We can then choose to train our classifier on all the objects we have observed, or we can choose to leave out the uncertain ones and use only the certain objects. To reflect this real world situation, the idea was to build trees both by using only the uncertain objects and by including the uncertain objects. The trees obtained should then be validated on the validation set containing uncertain objects, to see in what way including uncertain objects in the training set would affect the classifiers performance regarding the cassification of uncertain objects.

First, we used the certain training set to build trees for both methods. For the method using crisp labels, all the K-complex objects belong to class 2 and all the delta objects belong to class 1. For the method using uncertain labels, we assigned a belief function for each of the objects based on the experts' opinions, as explained earlier.

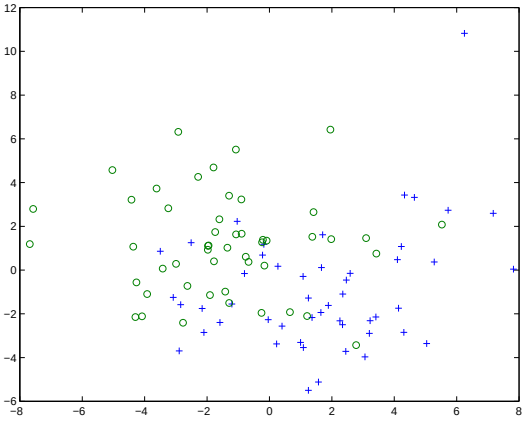
For the certain training set, we used crossvalidation in order to find the value of λ that provided the optimal tree. Crossvalidation consists in holding out a certain amount of objects as test objects chosen randomly from the data set using the remaining objects to build the tree, and doing this several times with different objects as test objects each time. We used a 5-fold crossvalidation, which means that we chose test objects randomly 5 times, so that we got 5 different data sets. Each time we extracted 25 objects as test data and the remaining 75 objects were used as training data.



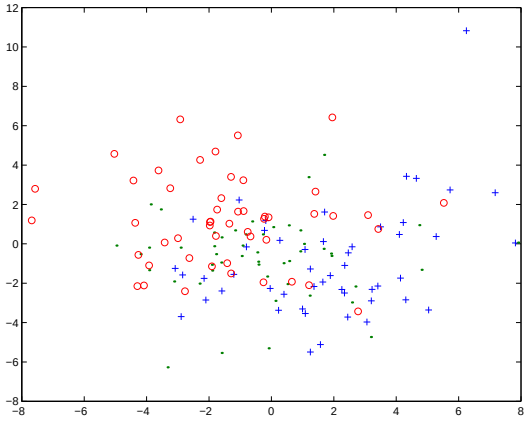
(a) Certain training set, 100 objects



(b) Uncertain training set, 150 objects



(c) Certain validation set, 100 objects



(d) Uncertain validation set, 150 objects

Figure 6.6: The different data sets used for classification. The data sets on the left are the certain data sets, and the data sets on the right are the uncertain data sets. The topmost data sets are the training sets, and the lower data sets are the validation sets. '+' are the most certain objects from the K-complex class, '.' are the uncertain objects from the K-complex class and 'o' are the objects from the delta class.

Then we ran the data through the method based on crisp labels, with different values for λ. The results of this experiment is shown in Table 6.5. As we see from these results, the best tree is obtained with $\lambda = 0.2$. Then a tree was built on all the 100 training objects with $\lambda = 0.2$. This tree was validated on the uncertain validation set, i.e., the set that contains 150, including 50 of the uncertain K-complex objects. For this experiment the misclassification error rate was **0.333**. The disagreement error measure

based on the pignistic probability was **0.21**.

		Misclassification error rate					
λ		Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
1	Train	0.0133	0.00	0.00	0.0133	0.0080	0.008
	Test	0.16	0.28	0.28	0.20	0.28	**0.24**
0.75	Train	0.0133	0.0133	0.0133	0.0267	0.0267	0.0187
	Test	0.32	0.24	0.32	0.20	0.28	**0.272**
0.5	Train	0.04	0.0267	0.0533	0.0133	0.0267	0.032
	Test	0.32	0.24	0.20	0.24	0.32	**0.264**
0.2	Train	0.12	0.1067	0.12	0.1067	0.08	0.1067
	Test	0.20	0.16	0.12	0.20	0.24	**0.184**
0.1	Train	0.12	0.12	0.1733	0.12	0.08	0.1227
	Test	0.20	0.20	0.28	0.20	0.24	**0.224**

Table 6.5: Crossvalidation for learning on certain data with crisp labels for different values of λ

		Disagreement measure					
λ		Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
1	Train	0.1508	0.1172	0.1312	0.1219	0.1203	0.1283
	Test	0.2713	0.2202	0.2193	0.1637	0.2872	**0.2323**
0.75	Train	0.1508	0.1172	0.1648	0.1303	0.1203	0.1367
	Test	0.2713	0.2202	0.2317	0.1902	0.2872	**0.2401**
0.5	Train	0.1644	0.1172	0.1648	0.1303	0.1203	0.1394
	Test	0.2839	0.2202	0.2317	0.1902	0.2872	**0.2426**
0.2	Train	0.1799	0.1372	0.1648	0.215	0.1311	0.1656
	Test	0.2889	0.1774	0.2317	0.2702	0.2911	**0.2519**

Table 6.6: Crossvalidation for learning on certain data with belief functions as labels for different values of λ

Then the same data were run through the method based on uncertain labels. For this method we got the results shown in Table 6.6. Since we now were using uncertain labels, the disagreement measure based on the pignistic probability was used as an error measure for the crossvalidation. The best tree was obtained with $\lambda = 1$, and a tree was built from all the 100 training data with $\lambda = 1$. The classifier was tested on the same uncertain validation set as above, and the misclassification rate for this

experiment was **0.3267**. The disagreement measure in this case was **0.2008**. We see from these results that the method using uncertain labels performs slightly better than the method using crisp labels.

The confusion matrices for the two classifiers are shown in Table 6.7. The confusion matrices show that we get a slightly better result regarding the objects that are not K-complex when using uncertain labels than when using crisp labels.

		Crisp labels			Unc. labels	
		Classified as			Classified as	
		1	2		1	2
Real	1	37	13	1	38	12
	2	37	63	2	37	63

Table 6.7: Confusion matrices for the uncertain validation data on trees built from certain training data with crisp labels and uncertain labels

Then we tried to build a classifier from the uncertain data set with 150 objects. For the method based on crisp labels, we put in class 2 both the 50 most certain K-complex examples and the 50 uncertain K-complex examples. The 50 delta examples were put in class 1. For the method based on uncertain labels, the belief functions for the new 50 uncertain objects were assigned based on the experts' statements, as explained earlier.

We did the same as for the certain training set, we used a 5-fold crossvalidation with 110 training objects and 40 test objects. The results for different values of λ for the 5 data sets using crisp labels are shown in Table 6.8. The best tree was obtained with $\lambda = 0.1$. The results for the method based on belief functions as labels are shown in Table 6.9. Here the best tree was obtained with $\lambda = 0.5$.

The uncertain validation set with 150 objects was also run through both of these classifiers. The misclassification error rate for the tree built from data with crisp labels was **0.3467**, and the disagreement measure was **0.2625**. The misclassification error rate for the tree built from data with belief functions as labels was **0.340**, and in this case the disagreement measure was **0.2247**. The confusion matrices for both trees are shown in Table 6.10.

We have now built four trees, two trees built on the certain data training set, one using crisp labels and one using belief functions as labels, and two trees built on the uncertain training data set, also in this case using both kinds of labels. All trees have been tested on the certain validation set of 100 objects. The decision boundaries obtained by the four trees are shown for the certain validation set in Figure 6.7.

		Misclassification error rate					
λ		Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
1	Train	0.00	0.00	0.00	0.00	0.00	0.00
	Test	0.35	0.325	0.425	0.30	0.325	**0.345**
0.75	Train	0.0091	0.00	0.00	0.0091	0.00	0.0036
	Test	0.325	0.325	0.425	0.325	0.325	**0.345**
0.5	Train	0.0364	0.1364	0.00	0.0182	0.0091	0.04
	Test	0.35	0.375	0.425	0.30	0.325	**0.355**
0.2	Train	0.2091	0.2182	0.0625	0.1818	0.20	0.1743
	Test	0.40	0.40	0.40	0.25	0.275	**0.345**
0.1	Train	0.2091	0.2455	0.0625	0.2546	0.30	0.2143
	Test	0.40	0.275	0.40	0.25	0.30	**0.325**

Table 6.8: Crossvalidation for learning on uncertain data with crisp labels for different values of λ

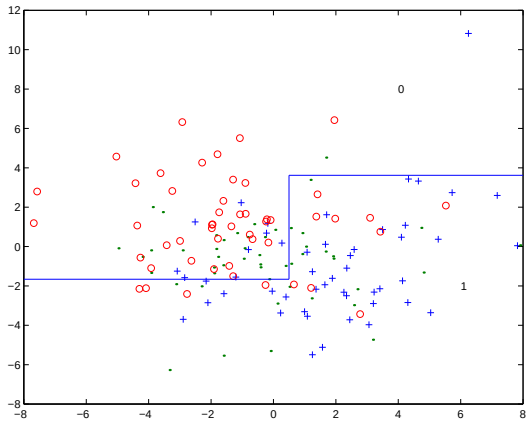
		Disagreement measure					
λ		Data set 1	Data set 2	Data set 3	Data set 4	Data set 5	**Average**
1	Train	0.1061	0.1196	0.1080	0.1250	0.1174	0.1152
	Test	0.2237	0.2729	0.1611	0.2143	0.2147	**0.2173**
0.5	Train	0.1435	0.1555	0.1597	0.1583	0.1740	0.1566
	Test	0.2348	0.2314	0.1557	0.2241	0.2117	**0.2115**
0.2	Train	0.1688	0.181	0.1859	0.1858	0.2081	0.1859
	Test	0.2459	0.2433	0.1701	0.2641	0.2209	**0.2289**

Table 6.9: Crossvalidation for learning on uncertain data with belief functions as labels for different values of λ

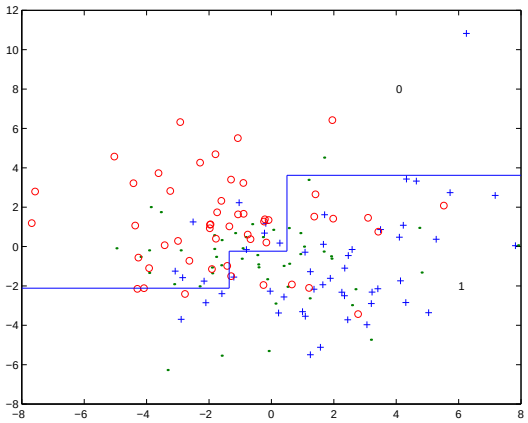
	Crisp labels			Unc. labels		
		Classified as			Classified as	
		1	2		1	2
Real	1	35	15	1	37	13
	2	37	63	2	38	62

Table 6.10: Confusion matrices for the uncertain validation data on trees built from uncertain training data with crisp labels and uncertain labels

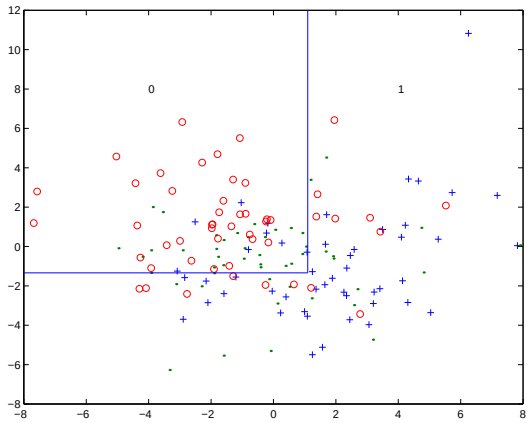
If we compare these results with the results obtained from the trees trained on certain labels, we see that it is slightly more difficult to classify the validation objects when the classifier has been trained on uncertain objects than when it has been trained only on certain objects. However, it seems in both cases that the method that uses belief functions as labels for the training data performs slightly better than the method that uses crisp labels for the training data. The confusion matrices show the results more detailed.



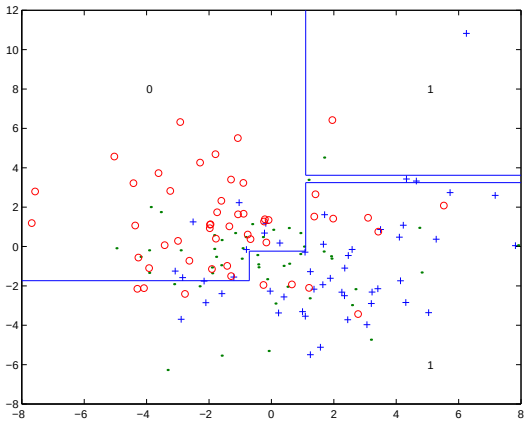
(a) Training on certain data with crisp labels



(b) Training on certain data with belief function labels



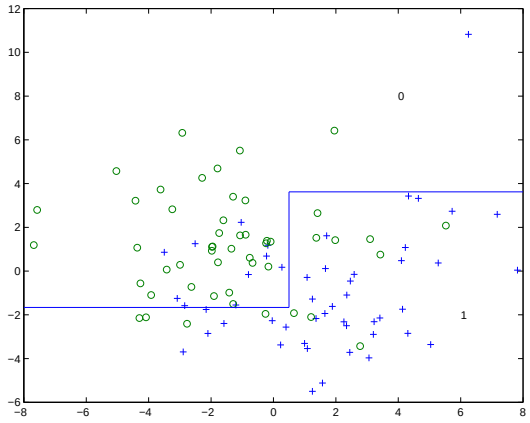
(c) Training on uncertain data with crisp labels



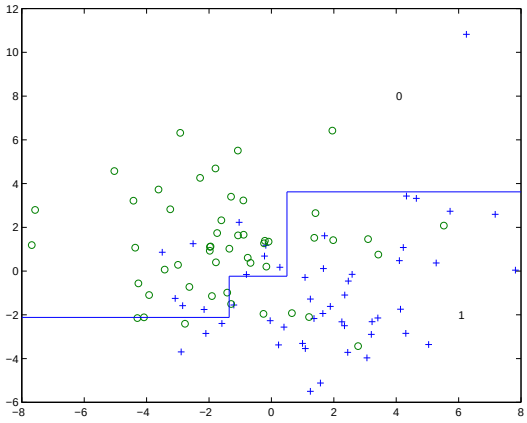
(d) Training on uncertain data with belief function labels

Figure 6.7: Decision boundaries for the uncertain validation set with 150 objects for the four different trees built. '+' are the most certain objects from the K-complex class, '.' are the uncertain objects from the K-complex class and 'o' are the objects from the delta class.

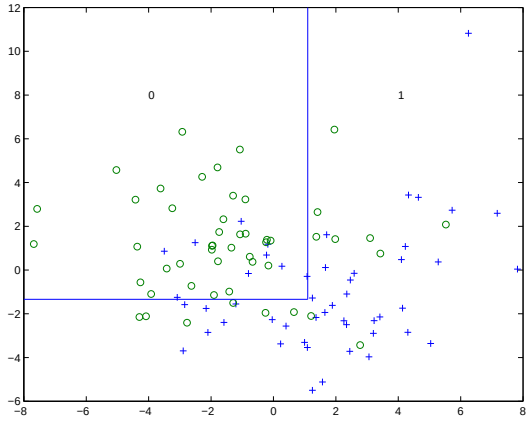
The four classifiers built were also tested on the certain validation set, i.e., the set that contains only the most certain objects of the K-complex examples. The decision boundaries for the certain validation set in each case are shown in Figure 6.8.



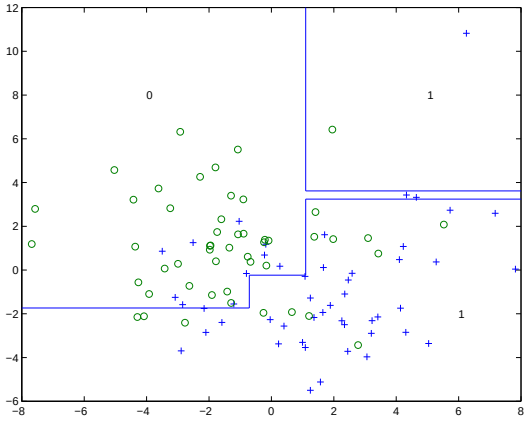
(a) Training on certain data with crisp labels



(b) Training on certain data with belief function labels



(c) Training on uncertain data with crisp labels



(d) Training on uncertain data with belief function labels

Figure 6.8: Decision boundaries for the certain validation set with 100 objects for the four different trees built. '+' are the most certain objects from the K-complex class and 'o' are the objects from the delta class.

When training on certain data using crisp labels, the misclassification error rate was **0.24**, and the disagreement measure was **0.2546**. For the method using uncertain labels, the misclassification error rate was **0.24**, and the disagreement measure was

0.2361. When training on uncertain data using crisp labels, the misclassification error rate was **0.24**, and the disagreement measure was **0.3440**. For the method using uncertain labels, the misclassification error rate was **0.24**, and the disagreement measure was **0.2712**. Again we observe that the method based on uncertain labels performs slightly better than the method based on crisp labels.

6.5 Summary

We have in this chapter presented experiments performed on data from a real world classification problem. Experiments have been done on both parts of our proposed method, the method using crisp labels as decision labels for the training set, and the method using belief functions as labels for the training set (uncertain labels). Experiments have been done both with data sets containing only the most certain objects and with data sets that contain an amount of uncertainty represented by objects for which the experts did not agree on the classification. The results obtained will be interpreted and analysed in the following chapter.

Chapter 7

Discussion

7.1 Introduction

In the previous chapter several experiments were described and the results of the experiments were presented. We will in this chapter look closer at the results in order to discuss the performance of the method we have proposed in this work. An analysis of these results will make us able to draw conclusions regarding our method.

In order to refine and improve the method, some further work should be done. In the last section of this chapter a suggestion of what can be done in the future to extend this work is given.

7.2 Analysis of the results

We chose to test our method on a real world classification problem, in order to get an impression of how our method performed. We chose the problem of discerning K-complex patterns from delta wave patterns in EEG's measured during sleep. This problem represented a good example of a problem that our method would be able to handle. The different objects which were supposed to be examples of the K-complex pattern were presented to some experts, and they were asked to give their opinion on the objects. The experts did not always agree on the classification, which introduced some uncertainty in the labeling of the objects. This is exactly the situation we wanted to be able to handle with our method.

As we saw it, the data set contained three types of objects. One type of objects consisted in the examples of the delta wave pattern, which had a certain classification. Then there was the set of objects that was supposed to be examples of the K-complex pattern, but we considered that this set of objects contained in fact two different sub-

classes. One subclass was the objects that were most certain examples of the K-complex pattern and the other type was the objects for which there was much doubt as to whether they were examples of the K-complex pattern or not. Obviously, the latter would be the hardest to detect. However, for a classifier using crisp labels, they would all belong to the same class and would consequently make it more difficult to classify the certain objects as well. We got this confirmed when we tried to build a classifier on the whole data set with crisp labels, which performed quite badly. This classifier had a misclassification error rate of around 0.3. This would indicate that a simple classifier that classified all of the objects as examples of the K-complex class would perform equally well, which is certainly not a desired result. It was our belief that a method that would distinguish between the certain and uncertain objects by using uncertain labels instead of crisp ones would be better able to detect the different types of objects.

7.2.1 Learning from crisp labels

First we introduced belief functions in the decision tree learning method, which makes us able to obtain a belief function instead of a crisp classification for each object to be classified. The belief function will contain more information than just a crisp classification, because it will tell us for which class we should have the highest belief concerning our object. This makes us able to make a decision afterwards about which class our object most probably belongs to. Our first concern was consequently to test how our proposed method of substituting the entropy measure in the decision tree learning algorithm with an uncertainty measure based on belief functions would perform.

We removed the most uncertain objects from the K-complex class, and tested our method on the remaining objects. This was done in order to get results that were not affected by the difficulty introduced by these objects. We built a classifier with the most certain objects labeled by crisp classes, and computed the misclassification error rate from the classification obtained from the belief functions.

The results shown in Table 6.3 show that our method performs quite well, and apparently slightly better than the C4.5 method regarding the misclassification error rate. However, this difference may not be significant, because this classification problem is not very difficult when the uncertain objects are removed. In addition, it is tested on a relatively small amount of data. Using crossvalidation also tends to give a slightly better result, since the average is drawn from different compositions of the same data. That way the training and test data will be more similar than if there was a totally different test set to validate the results as is the case when we run the C4.5 program. However, the results at least show that our method performs approximately as well as the ordinary method for decision tree learning, which is an encouraging result.

7.2.2 Learning from uncertain labels

Once we had assured that our method performed well, we could extend our experiments to the more interesting part of our method, the part of handling uncertain labels in the training data. In order to see how this method performed, we wanted to compare it to the method we had already introduced, i.e., the method that uses crisp labels. To do this, we had to test the two methods using the same data sets. The data sets we used for these experiments were quite small because of the data available and because of the computation time required for this method, so the results would not be as significant as desired, but they would still be able to give us an indication of how the method performed.

A summary of the results described in Chapter 6 is shown in Table 7.1. This table shows the results obtained with the four different classifiers built, both for the uncertain validation set and for the certain validation set. The misclassification error rate and the disagreement measure based on the pignistic probability is given in each case. Validation with uncertain objects (the topmost part of the table) is the most realistic situation. In a real world classification problem, the data to be classified would contain uncertain objects to some extent. This is therefore the situation that gives the most interesting results. Validation on certain data is included to obtain more information about the method.

Uncertain validation set

	Certain training		Uncertain training	
	Crisp labels	Unc. labels	Crisp labels	Unc. labels
Miscl. error	0.333	0.3267	0.3467	0.340
Disagr. measure	0.21	0.2008	0.2625	0.2247

Certain validation set

	Certain training		Uncertain training	
	Crisp labels	Unc. labels	Crisp labels	Unc. labels
Miscl. error	0.24	0.24	0.24	0.24
Disagr. measure	0.2546	0.2361	0.344	0.2712

Table 7.1: Summary of results for the uncertain validation set and the certain validation set, for both methods of presenting the error measure.

From the results shown in Table 7.1, we see that when validating on uncertain objects, the method using uncertain labels performs slightly better than the method using crisp

labels. This is especially the case when uncertain objects have been used for training. This indicates that using uncertain labels for classification improves the classifier when it is trained on uncertain objects. The method using uncertain labels attaches less importance to the uncertain objects than to the certain objects, and consequently the classifier is less affected by the uncertainty than the classifier built from crisp labels.

This is confirmed by the results obtained from validating on certain objects. The disagreement measure shows that the method using uncertain labels performs better than the method using crisp labels. Using uncertain labels is a more realistic way of labeling uncertain data, and our method takes advantage of this.

Regarding the difference between using uncertain objects for training and using only certain objects, we see that the error is slightly increased when uncertain objects are introduced. This was in a way to be expected, because introduction of the uncertain objects will to a certain extent "confuse" the classifier and make the classification task more difficult. However, the increase is not a great one, especially in the situation where we have validated the method on uncertain objects. Since this is the most realistic situation, we can conclude that the results in fact show that by including uncertain objects in the training set, it would be possible to obtain classifiers that perform about as well as when only certain objects are used. In a way this is a bit discouraging, because it would be desirable to find that since the uncertain objects contain additional information, the classifier would take advantage of this information and improve the result. However, using uncertain objects does not worsen the situation to a great extent, which means that we will be able to build classifiers with all the information we have and still get agreeable results. This agrees quite well with what we hoped to find, that our method makes us able to use all the objects we have previously observed to train a classifier.

Figure 6.7 shows the decision boundaries for the four classifiers that were built based on the certain training set and the uncertain training set. We see that the method using crisp labels creates quite simple classifiers, while the method using uncertain labels build more complex ones. This suggests that the method using uncertain labels is able to differentiate the objects a bit more than the method using crisp labels. However, this does not seem to improve the classification to any great extent.

We are able to conclude that the classification error is not worsened to any great extent by using uncertain K-complex objects in the training set. This result suggests that our method gives less importance to the uncertain objects than to the certain objects. With our method we will be able to use the information contained in the uncertain objects without attaching too great importance to them.

These results confirmed our belief that the method using training objects with uncertain labels results in better classifiers than the ones obtained from methods using crisp labels. Again it must be stated that the results are not significant because of the small amount of objects tested. There is only a small difference, which may be the result of

some coincidence. However, the results encourage further work.

7.3 Comparison to other methods

It is not easy to compare our method to other methods, since to our knowledge there are not many other methods available to handle uncertain classification labels.

T. Denoeux's method combining the k-Nearest-Neighbour approach with the Dempster-Shafer theory of evidence ([5]) have been tested on our example data set in an informal way, which gave approximately the same results as we obtained with our method.

What we are able to state is that our method performs at least as well as other decision tree learning methods, exemplified by Quinlan's C4.5 program. In addition, our method provides a way of handling uncertain labeled training objects, which the ordinary decision tree learning method can not do. We have thus found a method that meets a problem many other methods do not handle. It provides a way of using uncertain information as profitably as possible.

7.4 Further work

The work we have done shows some interesting results concerning the use of belief functions in decision tree learning. However, in order to be able to obtain more conclusive results, further and more extensive experiments should be performed. It is also possible to envisage certain modifications and extensions to the method outlined in this work. Some of the possible further work is presented below.

- The data on which we have tested our method seem to represent a fairly easy classification problem. This is suggested by the fact that the difference between classifiers trained on only certain objects and classifiers trained on objects that contain uncertainty is not very conspicuous. It would be interesting to test the method on a more difficult classification problem that would demonstrate more influence by the uncertain objects on the classification result. This could show to what extent our method can avoid the difficulties the uncertainty will introduce, and at the same time use the information contained in these uncertain objects in the best possible way. Other experiments would thus demonstrate to what extent the uncertain objects have an impact on the result, and whether or not our method is able to handle this impact in a satisfactory way. It would also be of use to test the classifiers with validation sets that contain a larger amount of objects, in order to get more reliable results.

- It would be interesting to perform experiments on other methods that meet the problem of training systems on objects for which there is uncertainty about their classification. Rough set theory provides such a method, and it would be interesting to perform experiments with this method to compare the results with the results obtained with our method.

- For the part of our method that uses uncertain labels for classification we have implemented a function that computes a belief function from a set of given belief functions that has been assigned to the training objects. This function uses an algorithm that is computationally of complexity n^3, which means that for a large amount of objects, it would be very time consuming. An improvement of the complexity of this algorithm would make it easier to perform experiments on larger sets of data, which would make our method more flexible concerning what data sets are used for training.

- An interesting extension of the method would be to make it able to involve classification problems with more than two classes. One way of doing this may be to divide the problem into several two-class problems. For instance, suppose there are 3 classes, a, b and c. This problem could be divided into three two-class problems, one that classifies whether the objects belong to class a or not, one that classifies whether the objects belong to class b or not and so on. The inconvenience of this method is that we would not get only one classifier, but we would get as many classifiers as there are classes. We would have to run our objects through all of these classifiers and combine the results in the end. Another way of including more than two classes would be to modify Smets' equations on how to compute a belief function based on a partial knowledge of the probability distribution to the case of more than two outcomes. This can possibly be done by approximation.

- It would not be too time consuming to extend the method to cover not only continuous attributes as it does now, but to make it able to handle binary and discrete attribute values as well. This would also make our method more flexible regarding what data sets it is feasible to use.

7.5 Summary

We have in this chapter given an analysis of the results obtained with our method. Even though our example data set may not represent a very difficult classification task, the results we obtained using this data set are promising. They show that our method performs equally well as the ordinary decision tree learning method.

In addition, our method provides a means of using uncertain labeled objects as training data. It is our belief that this will make the classifier able to extract more information

from the training data and produce more differentiated information about the objects to be classified.

The method should be tested further and refined in order to be more solid. Some suggestions of further testing and modifications to our method are outlined in this chapter. The results obtained in our work are suggestive and encourage this kind of further work.

Bibliography

[1] T. M. Mitchell. *Machine Learning*. McGraw-Hill Companies, Inc., New York, 1997.

[2] S. Russel and P. Norvig. *Artificial Intelligence - a modern approach*. Prentice-Hall International, Inc., New Jersey, 1995.

[3] L. Polkowski J. Komorowski, Z. Pawlak and A. Skowron. A rough set perspective on data and knowledge. In *Handbook of Data Mining and Knowledge Discovery*. Oxford University Press, 2000.

[4] S. Pöhlmann K. Weichselberger. *A Methodology for Uncertainty in Knowledge-Based Systems*. Springer-Verlag, Heidelberg, 1990.

[5] T. Denœux. A k-nearest neighbor classification rule based on Dempster-Shafer theory. *IEEE Transactions on Systems, Man and Cybernetics*, 25(05):804–813, 1995.

[6] T. Denœux. Analysis of evidence-theoretic decision rules for pattern classification. *Pattern Recognition*, 30(7):1095–1107, 1997.

[7] L. M. Zouhal and T. Denœux. An evidence-theoretic k-NN rule with parameter optimization. *IEEE Transactions on Systems, Man and Cybernetics C*, 28(2):263–271, 1998.

[8] R. A. Olshen L. Breiman, J. H. Friedman and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall, London, 1984.

[9] J. R. Quinlan. *Programs for Machine Learning*. Morgan Kaufmann Publishers, San Francisco, 1993.

[10] M. S. Bjanger. Vibration analysis in rotating machinery using rough set theory and rosetta. Project report, NTNU, Department of Computer and Information Science, April 1999.

[11] G. Shafer. *A mathematical theory of evidence*. Princeton University Press, Princeton, N.J., 1976.

[12] P. Smets and R. Kennes. The Transferable Belief Model. *Artificial Intelligence*, 66:191–243, 1994.

[13] P. Smets. The Transferable Belief Model for quantified belief representation. In D. M. Gabbay and P. Smets, editors, *Handbook of Defeasible reasoning and uncertainty management systems*, volume 1, pages 267–301. Kluwer Academic Publishers, Dordrecht, 1998.

[14] P. Smets. What is Dempster-Shafer's model ? In R. R. Yager, M. Fedrizzi, and J. Kacprzyk, editors, *Advances in the Dempster-Shafer theory of evidence*, pages 5–34. Wiley, New-York, 1994.

[15] G. J. Klir. Measures of uncertainty in the Dempster-Shafer theory of evidence. In R. R. Yager, M. Fedrizzi, and J. Kacprzyk, editors, *Advances in the Dempster-Shafer theory of evidence*, pages 35–49. John Wiley and Sons, New-York, 1994.

[16] P. Smets. Belief induced by the partial knowledge of the probabilities. In D. Heckerman, D. Poole, and R. Lopez de mantaras, editors, *Uncertainty in AI'94*, pages 523–530. Morgan Kaufmann, San Mateao, 1994.

[17] C. Richard. *Une méthodologie pour la détection à structure imposée. Applications au plan temps-fréquence*. PhD thesis, Université de Technologie de Compiègne, Compiègne, France, 1998.

[18] C. Richard and R. Lengellé. Data driven design and complexity control of time-frequency detectors. *Signal Processing*, 77:37–48, 1999.

Appendix A

Source code

The MATLAB implementation of our method is shown in this Appendix. The Appendix is divided in three parts. The first part shows the main functions of the program, i.e., the main file to run when building a tree, the main function that builds a tree and the main function that classifies test objects. The second part shows the functions used to compute the uncertainty measure (the entropy) and the belief function computation. The third part shows the auxiliary functions.

The tree data structure has been used to build the decision trees. MATLAB functions for this data structure were distributed free for use by a MATLAB programmer and are not included in this Appendix.

A.1 Main functions

A.1.1 buildtree.m

```
%%%
%%% Main file to run when building a decision tree
%%%

% Ask user for text file to use to build decision tree
[fname,pname] = uigetfile('*.dat','Load Training Data');

% Ask for number of attributes in text file
prompt  = {'Enter number of conditional attributes:'};
title   = 'Attributes';
answer  = inputdlg(prompt,title);

% Load text file and store objects in matrix a
numatts = str2num(answer{1});
num = [];
for i = 1:numatts
   num = [num '%g'];
end
```

```
file = strcat(pname,fname);

% Ask if the data are labelled with crisp or uncertain labels
button = questdlg('Which labels?', 'Labels', 'Crisp', 'Uncertain', 'Uncertain');
if strcmp(button,'Crisp')
   label = 0;
elseif strcmp(button,'Uncertain')
   label = 1;
end
fid = fopen(file);
if label == 0
   a = fscanf(fid, num, [(numatts+1) inf]);
elseif label == 1
   a = fscanf(fid, num, [(numatts+3) inf]);
end
a = a';
fclose(fid);

% Ask if the user wants to build a full tree or not
stop = 0;
button = questdlg('How to build tree?', 'Building tree', 'Use stop criterion', 'Build full tree',
         'Use stop criterion');
if strcmp(button,'Use stop criterion')
   stop = 1;
elseif strcmp(button,'Build full tree')
   stop = 0;
end

% Ask for parameter to use in entropy computation
prompt  = {'Enter parameter to use for uncertainty measure:'};
title   = 'Entropy Parameter';
answer  = inputdlg(prompt,title);
lambda = str2num(answer{1});

% Open file for writing results
fid = fopen('results.txt', 'w');
fprintf(fid, 'Entropy parameter: %4.3f\n\n', lambda);

% Initialize entropy value
[m,n] = size(a);
fprintf(fid, 'Root node! \n\nNumber of objects: %5u\n', m);

if label == 0
   % Check number of values for the decision attribute
   attval = attvallist(a, n, m);
   [y,x] = size(attval);

   % Count number of objects from each class
   % (n1=class 1, n2=class 2)
   n1 = 0;
   n2 = 0;
   if x == 2
      for i = 1:m
         if a(i,n) == 0
            n1 = n1 + 1;
         elseif a(i,n) == 1
            n2 = n2 + 1;
         end
      end
   elseif x == 1
      if a(1,n) == 0
         n1 = m;
      elseif a(1,n) == 1
         n2 = m;
      end
```

```
   end
   fprintf(fid, 'Number of objects from class 1: %5u\n', n1);
   fprintf(fid, 'Number of objects from class 2: %5u\n\n', n2);
   m1 = n1/(n1+n2+1);
   m2 = n2/(n1+n2+1);
   mOm = 1/(n1+n2+1);
   BetP1 = m1+mOm/2;
   BetP2 = m2+mOm/2;
   NS = mOm*log2(2);
   D = -(m1*log2(BetP1))-(m2*log2(BetP2));
   E = NS+(lambda*D);
elseif label == 1
   bfmat = a(:,n-2:n);
   [mlist,A] = labelcomb(bfmat);
   m1 = mlist(1);
   m2 = mlist(2);
   mOm = mlist(3);
   BetP1 = m1+mOm/2;
   BetP2 = m2+mOm/2;
   NS = mOm*log2(2);
   D = -(m1*log2(BetP1))-(m2*log2(BetP2));
   E = NS+(lambda*D);
end
fprintf(fid, 'Entropy: %9.4f\n\n', E);

% Build tree
if label == 0
   [tree, root] = DT(a, E, n1, n2, stop, fid, lambda, label);
elseif label == 1
   [tree, root] = DT(a, E, m1, m2, stop, fid, lambda, label);
end

% Prune leafnodes which have the same target value
tree = prunetree(tree, root);

% Draw decision tree on screen
DrawTree(tree, 'num2str(data(5))', 'num2str(data(6))');
save tree tree;

if label == 0
   % Find number of objects from each class at each node,
   % and the entropy
   NValues = ncont(tree, 1, [], 1);

   % Write results to file
   OutNValues = NValues';
   fprintf(fid, '\nNumber of objects at each node \n---------------------\n\n');
   fprintf(fid, 'Level: N1:\t N2:\t Entropy:\n');
   fprintf(fid, '%2u\t%5u\t%5u\t%9.4f\n', OutNValues);
end

% Ask if user wants to classify any objects

button = questdlg('Classify test objects?', 'Classify', 'Yes', 'No', 'No');
if strcmp(button,'No')
   classify = 0;
elseif strcmp(button,'Yes')
   classify = 1;
end
if (classify == 1)
   classifytest(tree, label);
end

% Ask if user wants to classify again
if (classify == 1)
```

```
   button = questdlg('Classify again?', 'Classify', 'Yes', 'No', 'No');
   while strcmp(button,'Yes')
      classifytest(tree, label);
      button = questdlg('Classify again?', 'Classify', 'Yes', 'No', 'No');
   end
end
fclose(fid);

%%%
%%% END
%%%
```

A.1.2 DT.m

```
function [newtree, newroot] = f(Examples, E, m1, m2, stop, fid, lambda, label)
%
% Function DT is the main function that builds a decision tree
%
% Examples is a two-dimensional matrix where each row
% contains an example with its attribute values and a
% decision value
%
% E is the value of the Entropy for this new node
%
% m1, m2 is either the number of objects from each class
% for this new node (if crisp labels are used) or the mass
% assignment for the two classes (if uncertain labels are used).
%
% stop is a parameter which tells whether or not to use
% the stop criterion. stop = 0 builds a full tree,
% stop = 1 stops when there is no more decrease in Entropy
%
% fid is the file identifier of the file to which the
% results are written
%
% lambda is a parameter to use when the entropy is computed
%
% label says what labels are used for building the tree
% (0 = crisp labels, 1 = uncertain labels)
%
% The data field of the nodes in the tree are built
% in this way:
%   [ [decision value] [entropy] [n1/m1] [n2/m2] [split attribute] [split value] ]

% Build a root node for a tree
[newtree, newroot] = NewTree(2,6);
leafnode = 0;

% Display on screen that a new node has been made
disp('New node...')
fprintf(fid, 'New node!\n---------\n\n');

% Check size of Examples-matrix
[m,n] = size(Examples);
fprintf(fid, 'Number of objects: %5u\n', m);

% Build list of attributes to be checked
atts = [];
if label == 0
   decatt = 1;
elseif label == 1
   decatt = 3;
```

```
end
for i = 1:(n-decatt)
   atts = [atts i];
end

if label == 0
   % Check number of values for the decision attribute
   attval = attvallist(Examples, n, m);
   [y,x] = size(attval);
   dec = findmostcommon(Examples, attval, m, n);

   % If all examples are of the same class, return node
   % with value as decision value for this class
   if x == 1
      data = [attval(1) E m1 m2 attval(1)];
      newtree = TreeNodeData(newtree, newroot, data);
      fprintf(fid, 'A leaf node with decision: %1u\n\n', attval(1));
      fprintf(fid, 'Data written to node: %1u\t%5.4f\t%4u\t%4u\t%1u\n\n', data);
      leafnode = 1;
   end
elseif label == 1
   equal = 1;
   i = 1;
   exbf = Examples(:,n-2:n-1);
   while (i<m)&(equal==1)
      if ~(exbf(i,:) == exbf(i+1,:))
         equal = 0;
      end
      i = i+1;
   end
   count1 = 0;
   count2 = 0;
   for i = 1:m
      if (exbf(i,1)>exbf(i,2))|(exbf(i,1)==exbf(i,2))
         count1 = count1 + 1;
      elseif exbf(i,1)<exbf(i,2)
         count2 = count2 + 1;
      end
   end
   if (count1>count2)|(count1==count2)
      dec = 0;
   elseif count1<count2
      dec = 1;
   end
   if (m < 5)|(equal == 1)
      data = [dec E m1 m2 dec];
      newtree = TreeNodeData(newtree, newroot, data);
      fprintf(fid, 'A leaf node with decision: %1u\n\n', dec);
      fprintf(fid, 'Data written to node:\n Dec: %1u\t E:%6.4f\t m1:%4.3f\t m2:%4.3f\t
                    Dec:%1u\n\n', data);
      leafnode = 1;
   end
end

if leafnode == 0
   % Initialize main storage matrix for entropy
   Entropy = [];

   % Count number of attributes (b = number of attributes)
   [a, b] = size(atts);

%%%% ENTROPY COMPUTATION

   % Do for every conditional attribute left
   for j = 1:b
```

```
   % Display on screen which attribute is checked
   out = strcat('Attribute ',num2str(j));
   disp(out)
   fprintf(fid, 'Attribute checked: %1u\n\n', j);

   % Find range for the attribute's values
   [min, max] = findrange(Examples, m, atts(j));

   % Compute interval for splits to test (10% of the range)
   int = (max-min)/10;

   if ~(min == max)
      % (If min = max, all the objects have the same
      % attribute value, and no splits can be made)

      % Entropy-computation with 10% of objects as step
      % for choosing splits
      Ent = compute_entropy_obj(Examples, m, n, atts(j),
                                lambda, label);
      OutEnt = [Ent(:,1) Ent(:,2)]';
      fprintf(fid, 'Computed entropy for suggested splits:\n');
      fprintf(fid, 'Split:\t Entropy:\n');
      fprintf(fid, '%9.4f\t%9.4f\n', OutEnt);

      % Find the split with the smallest entropy for this attribute
      EntVal = Ent(:,2);
      [Echeck,I] = min(EntVal);
      split = Ent(I,1);
      m1min = Ent(I,4);
      m2min = Ent(I,5);
      m1max = Ent(I,6);
      m2max = Ent(I,7);
      Emin = Ent(I,8);
      Emax = Ent(I,9);
      fprintf(fid, '\nSmallest entropy: %9.4f\n\n', Echeck);
      fprintf(fid, 'Corresponding split: %9.4f\n\n', split);
      if label == 0
         [u,v] = size(Ent);
         if (u>1)
            % Adjust the split in the interval around the split 10% of objects
            Entropylist = bestsplit(Examples, atts(j), split, Ent, I, fid, lambda);
            split = Entropylist(1);
            Echeck = Entropylist(2);
            m1min = Entropylist(3);
            m2min = Entropylist(4);
            m1max = Entropylist(5);
            m2max = Entropylist(6);
            Emin = Entropylist(7);
            Emax = Entropylist(8);
            fprintf(fid, '\nAdjusted split and entropy: %9.4f\t%9.4f\n\n', split, Echeck);
         end
      end

      % Store the attribute with the split and the entropy value
      Entropy = [Entropy; atts(j) split Echeck m1min m2min m1max m2max Emin Emax];
   else
      fprintf(fid, 'Not enough different attribute values to consider splits on this
                   attribute\n\n');
   end
end % for i = j:b    (ENTROPY COMPUTATION)

OutEntropy = Entropy';
fprintf(fid, 'Computed entropy for all attributes:\n');
if label == 0
   fprintf(fid, 'Att:\t Split:\t Entropy:\t n1min:\t n2min:\t n1max:\t n2max:\t
```

```
                Emin:\t Emax:\n');
elseif label == 1
   fprintf(fid, 'Att:\t Split:\t Entropy:\t m1min:\t m2min:\t m1max:\t m2max:\t
                 Emin:\t Emax:\n');
end
fprintf(fid,'%1u\t%9.4f\t%9.4f\t%3u\t%3u\t%3u\t%3u\t%9.4f\t%9.4f\n', OutEntropy);

if ~(isempty(Entropy))
   % Find the attribute with the smallest entropy
   EntropyVal = Entropy(:,3);
   [Echosen,I] = min(EntropyVal);
   j = Entropy(I,1);
   k = Entropy(I,2);
   m1min = Entropy(I,4);
   m2min = Entropy(I,5);
   m1max = Entropy(I,6);
   m2max = Entropy(I,7);
   Emin = Entropy(I,8);
   Emax = Entropy(I,9);

   fprintf(fid, '\nSmallest entropy: %9.4f\n\n', Echosen);
   fprintf(fid, 'Corresponding attribute: %1u\n\n', j);
   fprintf(fid, 'Corresponding split: %9.4f\n\n', k);

   % Build subtrees for this attribute

   % Initialise matrix for examples to build subtree
   newExamplesmin = [];
   newExamplesmax = [];

   % Fill matrix with examples that have this att-value
   for i = 1:m
      if (Examples(i,j) < k) | (Examples(i,j) == k)
         newExamplesmin = [newExamplesmin; Examples(i,:)];
      elseif (Examples(i,j) > k)
         newExamplesmax = [newExamplesmax; Examples(i,:)];
      end
   end

   % Check size of the two new Examples matrices
   [mchild1,nchild1] = size(newExamplesmin);
   [mchild2,nchild2] = size(newExamplesmax);

   if ((stop == 1) & ~(Echosen < E)) % Use stop criterion
      % Return a leaf node with the most common decision value
      data = [dec E m1 m2 dec];
      newtree = TreeNodeData(newtree, newroot, data);
      fprintf(fid, 'STOP! A leaf node with decision: %1u\n\n', dec);
      fprintf(fid, 'Data written to node: %1u\t%4.3f\t%4u\t%4u\t%1u\n\n', data);
   else
     % Add the first child (left branch) to the existing tree for
     % this attribute
      [newtree, child] = AddChild(newtree, newroot, 1, []);
      fprintf(fid, 'Adding first child node\n\n');
      fprintf(fid, 'Number of objects: %5u\n', mchild1);
      if label == 0
         fprintf(fid, 'Number of objects from class 1: %5u\n', m1min);
         fprintf(fid, 'Number of objects from class 2: %5u\n', m2min);
      end
      fprintf(fid, 'Entropy: %9.4f\n\n', Emin);

      % Merge the existing tree with a computed subtree at the child node
      newtree = GraftTrees(newtree, child, DT(newExamplesmin, Emin, m1min, m2min, stop, fid,
                           lambda, label));
      % Add splitting information to parent node of new child node
```

```
         parent = GetParent(newtree, child);
         data = [dec E m1 m2 j k];
         newtree = TreeNodeData(newtree, parent, data);
         fprintf(fid, 'Data written to parent node: %1u\t%4.3f\t%4u\t%4u\t%2u\t
                      %5.2f\n\n', data);
         % Add the second child (right branch) to the existing tree for this attribute
         [newtree, child] = AddChild(newtree, newroot, 2, []);
         fprintf(fid, 'Adding second child node\n\n');
         fprintf(fid, 'Number of objects: %5u\n', mchild2);
         if label == 0
            fprintf(fid, 'Number of objects from class 1: %5u\n', m1max);
            fprintf(fid, 'Number of objects from class 2: %5u\n', m2max);
         end
         fprintf(fid, 'Entropy: %9.4f\n\n', Emax);

         % Merge the existing tree with a computed subtree at the child node
         newtree = GraftTrees(newtree, child, DT(newExamplesmax, Emax, m1max, m2max, stop, fid,
                              lambda, label));
      end

   else  % (Entropy = [], no more attributes or possible splits to check)
         % Return leaf node with most common value
         data = [dec E m1 m2 dec];
         newtree = TreeNodeData(newtree, newroot, data);
         fprintf(fid, 'A leaf node with decision: %1u\n\n', dec);
         fprintf(fid, 'Data written to node: %1u\t%4.3f\t%4u\t%4u\t%1u\n\n', data);
   end
end
return
```

A.1.3 classifytest.m

```
function f(tree, treelabel)
%
% Function classifytest takes a tree and classifies objects
% with this tree.
%
% tree is the decision tree to be used
%
% treelabel says what labels have been used for building the tree
% (0 = crisp labels, 1 = uncertain labels)

% Open file for writing results
fid1 = fopen('classify.txt', 'w');

if isempty(tree)
   % Get file with decision tree
   [fname,pname] = uigetfile('*.mat','Load Tree');
   file = strcat(pname,fname);
   load (file);

   % Ask if the tree were built with crisp or uncertain labels
   button = questdlg('Tree built from which labels?', 'Labels', 'Crisp', 'Uncertain',
                     'Uncertain');
   if strcmp(button,'Crisp')
      treelabel = 0;
   elseif strcmp(button,'Uncertain')
      treelabel = 1;
   end
end

% Get text file with test data
```

```
[fname,pname] = uigetfile('*.dat','Load Test Data');

% Ask for number of attributes in text file
prompt  = {'Enter number of attributes:'};
title   = 'Attributes';
answer  = inputdlg(prompt,title);

% Ask if the test data are labelled with crisp or uncertain labels
button = questdlg('Which labels on test data?', 'Labels', 'Crisp', 'Uncertain', 'Uncertain');
if strcmp(button,'Crisp')
   classlabel = 0;
elseif strcmp(button,'Uncertain')
   classlabel = 1;
end

% Load file with objects, and store the objects in matrix a
numatts = str2num(answer{1});
num = [];
for i = 1:numatts
   num = [num '%g'];
end
file = strcat(pname,fname);
fid = fopen(file);
if classlabel == 0
   a = fscanf(fid, num, [(numatts+1) inf]);
elseif classlabel == 1
   a = fscanf(fid, num, [(numatts+3) inf]);
end
a = a';
fclose(fid);
[x,y] = size(a);

% Add empty columns at the end of the matrix, new matrix is objects
objects = [a zeros(x,1) zeros(x,1) zeros(x,1) zeros(x,1)];

% Do for each object to be classified
for i = 1:x
   % Extract object and empty last column for this object
   object = objects(i,:);

   % Classify object
   [cl, m1, m2] = classify_rek(tree, object, 1, treelabel);
   mOm = 1 - (m1+m2);

   % Store the classification value in the last column for this object
   object(y+1) = cl;
   object(y+2) = m1;
   object(y+3) = m2;
   object(y+4) = mOm;
   objects(i,:) = object;
end

% Build format for output to file
format = [];
for i = 1:numatts
   format = [format '%5.4f\t'];
end
if classlabel == 0
   format = [format '%1u\t %1u\t %5.3f\t %5.3f\t %5.3f\n'];
elseif classlabel == 1
   format = [format '%5.3f\t %5.3f\t %5.3f\t %1u\t %5.3f\t %5.3f\t %5.3f\n'];
end

% Print result to file
fprintf(fid1, '\nClassification of file %s:\n---------------\n',fname);
```

```
for i = 1:x
   fprintf(fid1, format, objects(i,:));
end

if classlabel == 0
   % Compute error rate and values for confusion matrix
   right0 = 0;
   right1 = 0;
   wrong1 = 0;
   wrong0 = 0;
   for i = 1:x
      if objects(i,y) == 1
         if objects(i,y+1) == 1
            right1 = right1 + 1;
         elseif objects(i,y+1) == 0
            wrong1 = wrong1 + 1;
         end
      elseif objects(i,y) == 0
         if objects(i,y+1) == 0
            right0 = right0 + 1;
         elseif objects(i,y+1) == 1
            wrong0 = wrong0 + 1;
         end
      end
   end

   fprintf(fid1, '\nConfusion matrix:\n-----------------\n');
   fprintf(fid1, '(0)\t(1)\t\t<- classified as\n---------------\n');
   fprintf(fid1, '%3u\t%3u\t\t(0)\n', right0,wrong0);
   fprintf(fid1, '%3u\t%3u\t\t(1)\n', wrong1,right1);

   right = right0 + right1;
   wrong = wrong0 + wrong1;
   Err = wrong/(right+wrong);
   fprintf(fid1, '\nError rate =  %6.4f\n\n', Err);

   % Print error rate to screen
   e = num2str(Err);
   error = strcat('Error rate for file ',fname);
   error = strcat(error,' = ');
   error = strcat(error, e);
   errordlg(error, 'Error rate');
elseif classlabel==1
   Error = [];
   for i = 1:x
      m1 = objects(i,y-2);
      m2 = objects(i,y-1);
      mOm = objects(i,y);
      m1hat = objects(i,y+2);
      m2hat = objects(i,y+3);
      mOmhat = objects(i,y+4);
      BetPm = (m1*(m1hat+mOmhat/2)) + (m2*(m2hat+mOmhat/2)) + mOm;
      Error = [Error; 1-BetPm];
   end
   err = sum(Error)/x;
   fprintf(fid1, '\nError rate =  %6.4f\n\n', err);

   % Print error rate to screen
   e = num2str(err);
   error = strcat('Error rate for file ',fname);
   error = strcat(error,' = ');
   error = strcat(error, e);
   errordlg(error, 'Error rate');
end
fclose(fid1);
```

```
return
```

A.2 Functions for computing the uncertainty

A.2.1 entropybel.m

```
function [Emin, Emax] = f(n1min, n2min, n1max, n2max, lambda)
%
% Function entropybel computes the entropy (belief functions method)
% for some given values of n1 and n2
%
% n1min/n2min are the number of objects from class 1 and class 2
% that have an attribute value smaller than a certain split
%
% n1max/n2max are the number of objects from class 1 and class 2
% that have an attribute value greater than a certain split
%
% lambda is a parameter to use when computing the entropy

nmin = n1min + n2min;
nmax = n1max + n2max;
n = nmin + nmax;

% Entropy for attribute values smaller than split
m1_min = n1min/(nmin + 1);
m2_min = n2min/(nmin + 1);
mOm_min = 1/(nmin + 1);

BetP1_min = m1_min + mOm_min/2;
BetP2_min = m2_min + mOm_min/2;

NS_min = mOm_min*log2(2);
D_min = -(m1_min*log2(BetP1_min)) - (m2_min*log2(BetP2_min));

% Entropy for attribute values greater than split
m1_max = n1max/(nmax + 1);
m2_max = n2max/(nmax + 1);
mOm_max = 1/(nmax + 1);

BetP1_max = m1_max + mOm_max/2;
BetP2_max = m2_max + mOm_max/2;

NS_max = mOm_max*log2(2);
D_max = -(m1_max*log2(BetP1_max)) - (m2_max*log2(BetP2_max));

% Total entropy
Emin = NS_min + lambda*D_min;
Emax = NS_max + lambda*D_max;

return
```

A.2.2 entropybelbf.m

```
function [Emin,Emax] = f(m1min, m2min, mOmmin, m1max, m2max, mOmmax, lambda)
%
% Function entropybelbf computes the uncertainty/entropy
```

```
% (belief functions method) for some given values of m1 and m2
%
% m1min/m2min/mOmmin is a belief function for the objects that
% have an attribute value smaller than a certain split
%
% m1max/m2max/mOmmax is a belief function for the objects that
% have an attribute value greater than a certain split
%
% lambda is a parameter to use when computing the entropy

% Entropy for attribute values smaller than split

BetP1min = m1min + mOmmin/2;
BetP2min = m2min + mOmmin/2;

NSmin = mOmmin*log2(2);
Dmin = -(m1min*log2(BetP1min)) - (m2min*log2(BetP2min));

% Entropy for attribute values greater than split

BetP1max = m1max + mOmmax/2;
BetP2max = m2max + mOmmax/2;

NSmax = mOmmax*log2(2);
Dmax = -(m1max*log2(BetP1max)) - (m2max*log2(BetP2max));

% Total entropy
Emin = NSmin + lambda*Dmin;
Emax = NSmax + lambda*Dmax;

return
```

A.2.3 labelcomb.m

```
function [mX,A] = labelcomb(m)
%
% Function labelcomb computes a belief function from a
% set of belief functions given in matrix m.

n=size(m,1);
mX=zeros(1,3);
A=zeros(n+1,n+1);

A(1,1)=m(1,3);
A(1,2)=m(1,2);
A(2,1)=m(1,1);
for k=2:n,
   B=zeros(n+1,n+1);
   for i=0:k,
      for j=0:k-i,
         i1=i+1;j1=j+1;
         B(i1,j1)=A(i1,j1)*m(k,3);
         if i>=1,B(i1,j1)=B(i1,j1)+A(i1-1,j1)*m(k,1);end;
         if j>=1,B(i1,j1)=B(i1,j1)+A(i1,j1-1)*m(k,2);end;
      end;
   end;
   A=B;
end;

for r=0:n,
   for s=0:n-r,
      mX=mX+A(r+1,s+1)*[r s 1]/(r+s+1);
```

```
   end;
end;
```

A.2.4 compute_entropy_obj.m

```
function Ent = f(Examples, m, n, j, lambda, label)
%
% Function compute_entropy_obj computes the entropy for a given attribute
% by using steps containing 10% of the objects to find possible splits
%
% Examples is a matrix that contains all the training objects
%
% m is the number of training objects
%
% n is the number of attributes, including the decision attribute
%
% j is the attribute for which the entropy is computed
%
% label says what labels are used for building the tree

% Initialize temporary entropy storage matrix
Ent = [];

Sorted = sortrows(Examples, j);
step = m/10;

i = 1;
while (i<m)
   count = 0;
   while (count<step) & (i<m)
      count = count + 1;
      i = i+1;
   end
   if ~(i>m)
      k = (Sorted(i-1,j) + Sorted(i,j))/2;
      if ~(ismember(k, Ent))
         if label == 0
            [m1min, m2min] = countmin(Examples, j, m, k, n);
            [m1max, m2max] = countmax(Examples, j, m, k, n);
            nmin = m1min + m2min;
            nmax = m1max + m2max;
            ntot = nmin + nmax;
            [Emin, Emax] = entropybel(m1min, m2min, m1max, m2max, lambda);
            newentropy = (nmin/ntot)*Emin + (nmax/ntot)*Emax;
         elseif label == 1
            [u,v] = size(Examples);
            Examplesmin = [];
            Examplesmax = [];
            for y = 1:u
               if (Examples(y,j) < k) | (Examples(y,j) == k)
                  Examplesmin = [Examplesmin; Examples(y,:)];
               elseif (Examples(y,j) > k)
                  Examplesmax = [Examplesmax; Examples(y,:)];
               end
            end
            [min,st] = size(Examplesmin);
            [max,st] = size(Examplesmax);
            if (min>0)&(max>0)
               bfmatmin = Examplesmin(:,v-2:v);
               bfmatmax = Examplesmax(:,v-2:v);
               [mlistmin,A] = labelcomb(bfmatmin);
               [mlistmax,A] = labelcomb(bfmatmax);
```

```
            m1min = mlistmin(1);
            m2min = mlistmin(2);
            mOmmin = mlistmin(3);
            m1max = mlistmax(1);
            m2max = mlistmax(2);
            mOmmax = mlistmax(3);
            [Emin,Emax] = entropybelbf(m1min, m2min, mOmmin,
                          m1max, m2max, mOmmax, lambda);
            newentropy = (min/u)*Emin + (max/u)*Emax;
         end
      end
      Ent = [Ent; k newentropy i m1min m2min m1max m2max Emin Emax];
   end
end
end
return
```

A.3 Auxiliary functions

A.3.1 attvallist.m

```
function attval = f(Examples, dec_att, nobj)
%
% Function attvallist takes a set of training examples
% and returns a list of the possible decision values
% for the examples in the given training set.
%
% Examples is the matrix containing the training examples
%
% dec_att is the column number of the decision attribute
%
% nobj is the number of objects in Examples

% Add the first decision value to a storage matrix
attval = [Examples(1,dec_att)];

%Do for every remaining example in Examples
for i = 2:nobj
   % Check if decision value is already represented in attval
   exist = ismember(Examples(i,dec_att), attval);
   if exist == 0   % (If new decision value)
      % Add new value to storage matrix attval
      attval = [attval Examples(i,dec_att)];
   end
end
return
```

A.3.2 bestsplit.m

```
function Entropylist = f(Examples, att, split, Ent, ind, fid, lambda)
%
% Function bestsplit takes a set of training examples, an attribute
% and a split, and adjusts the split for this attribute
% by computing the entropy for splits in an interval area around
% the given split
%
```

```
% Examples is the matrix with the training examples
%
% att is the attribute in question
%
% split is the chosen split
%
% Ent is the Entropy-matrix already computed for this attribute
%
% ind is an index that tells where in Ent the smallest Entropyvalue
% is
%
% fid is the file handle for the file to which the output is written
%
% lambda is a parameter used to compute the entropy

% Check the size of the matrix Examples
[m,n] = size(Examples);

% Sort the Examples matrix according to the given attribute
Sorted = sortrows(Examples, att);

% Initialize the storage matrix or the entropy
Entro = [];

% Set values to use for adjusting the split
[u,v] = size(Ent);
if ind == 1 %(if split is the first value in the list)
   kmin = Sorted(1,att);
   kmax = Ent(ind+1,1);
   imin = 1;
   imax = Ent(ind+1,3);
elseif ind == u %(if split is the last value in the list)
   kmin = Ent(ind-1,1);
   kmax = Sorted(m,att);
   imin = Ent(ind-1,3);
   imax = m;
else
   kmin = Ent(ind-1,1);
   kmax = Ent(ind+1,1);
   imin = Ent(ind-1,3);
   imax = Ent(ind+1,3);
end

step = (imax - imin)/10;
i = imin;

% Compute entropy for splits around given split
while (i<imax)
   count = 0;
   while (count<step) & (i<imax)
      count = count + 1;
      i = i+1;
   end
   if ~(i>imax)
      k = (Sorted(i-1, att) + Sorted(i,att))/2;
      [n1min, n2min] = countmin(Examples, att, m, k, n);
      [n1max, n2max] = countmax(Examples, att, m, k, n);
      nmin = n1min + n2min;
      nmax = n1max + n2max;
      ntot = nmin + nmax;
      [Emin,Emax] = entropybel(n1min, n2min, n1max, n2max, lambda);
      entropy = (nmin/ntot)*Emin + (nmax/ntot)*Emax;
      Entro = [Entro; k entropy n1min n2min n1max n2max Emin Emax];
   end
end
```

```
% Print results to file
OutEntro = [Entro(:,1) Entro(:,2)]';
fprintf(fid, '\tComputed adjusted entropy, related to number of objects:\n');
fprintf(fid, '\tSplit:\t Entropy:\n');
fprintf(fid, '\t%9.4f\t%9.4f\n', OutEntro);

% Find split with smallest entropy
EntVal = Entro(:,2);
[Entr1,I] = min(EntVal);
split1 = Entro(I,1);
n1min1 = Entro(I,3);
n2min1 = Entro(I,4);
n1max1 = Entro(I,5);
n2max1 = Entro(I,6);
Emin1 = Entro(I,7);
Emax1 = Entro(I,8);

% Check if method with intervals gives a different result
int = kmax - kmin;
Entropylist2 = bestsplit_int(Examples, att, split, int, fid, lambda);
Entr2 = Entropylist2(2);

% Choose the best result
Entr = min(Entr1,Entr2);
if Entr == Entr1
   split = split1;
   n1min = n1min1;
   n2min = n2min1;
   n1max = n1max1;
   n2max = n2max1;
   Emin = Emin1;
   Emax = Emax1;
elseif Entr == Entr2
   split = Entropylist2(1);
   n1min = Entropylist2(3);
   n2min = Entropylist2(4);
   n1max = Entropylist2(5);
   n2max = Entropylist2(6);
   Emin = Entropylist2(7);
   Emax = Entropylist2(8);
end

Entropylist = [split Entr n1min n2min n1max n2max Emin Emax];

return
```

A.3.3 bestsplit_int.m

```
function Entropylist = f(Examples, att, s, int, fid, lambda)
%
% Function bestsplit_int takes a set of training examples, an attribute,
% a split and an interval, and adjusts the split for this attribute
% by computing the entropy for splits in the interval area around
% the given split
%
% Examples is the matrix with the training examples
%
% att is the attribute in question
%
% s is the chosen split
%
```

```
% int is the interval that has been used to find the split
%
% fid is the file handle for the file to which the output is written
%
% lambda is a parameter used to compute the entropy

% Check the size of the matrix Examples, m=number of objects,
% n=number of attributes
[m,n] = size(Examples);

% Initialize the storage matrix of the entropy
Entro = [];
newint = int/5;

%Compute entropy for splits around the given split
for k = (s-int):newint:(s+int)
   [n1min, n2min] = countmin(Examples, att, m, k, n);
   [n1max, n2max] = countmax(Examples, att, m, k, n);
   nmin = n1min + n2min;
   nmax = n1max + n2max;
   ntot = nmin + nmax;
   [Emin, Emax] = entropybel(n1min, n2min, n1max, n2max, lambda);
   entropy = (nmin/ntot)*Emin + (nmax/ntot)*Emax;
   Entro = [Entro; k entropy n1min n2min n1max n2max Emin Emax];
end

% Print results to file
OutEntro = [Entro(:,1) Entro(:,2)]';
fprintf(fid, '\tComputed adjusted entropy, related to given intervals:\n');
fprintf(fid, '\tSplit:\t Entropy:\n');
fprintf(fid, '\t%9.4f\t%9.4f\n', OutEntro);

% Find split with smallest entropy
EntVal = Entro(:,2);
[Ent,I] = min(EntVal);
split = Entro(I,1);
n1min = Entro(I,3);
n2min = Entro(I,4);
n1max = Entro(I,5);
n2max = Entro(I,6);
Emin = Entro(I,7);
Emax = Entro(I,8);

Entropylist = [split Ent n1min n2min n1max n2max Emin Emax];

return
```

A.3.4 classify_rek.m

```
function [cl, m1, m2] = f(tree, obj, root, label)
%
% Function classify_rek is a recursive function that goes through a
% binary decision tree in order to classify a given object
%
% tree is the decision tree in which to search
%
% obj is the object to be classified
%
% root is the node in the tree from which to start the search
%
% label says what labels have been used for building the tree
% (0 = crisp labels, 1 = uncertain labels)
```

```
% Get the data stored at root

data = TreeNodeData(tree, root);

if IsLeaf(tree, root)
   if label == 0
      cl = data(1);
      n1 = data(3);
      n2 = data(4);
      m1 = n1/(n1+n2+1);
      m2 = n2/(n1+n2+1);
   elseif label == 1
      m1 = data(3);
      m2 = data(4);
      if (m1 > m2) | (m1 == m2)
         cl = 0;
      elseif m2 > m1
         cl = 1;
      end
   end
else
   % Find the attribute to check
   att = data(5);

   % Find the split to use
   split = data(6);

   % Find the children of root
   children = GetChildren(tree, root);

   % If the object's value for the given attribute is smaller than
   % or equal to the split, go down left branch of root
   if (obj(att) < split) | (obj(att) == split)
      root = children(1);

   % If the object's value for the given attribute is greater than
   % the split, go down right branch of root
   elseif (obj(att) > split)
      root = children(2);
   end
   [cl, m1, m2] = classify_rek(tree, obj, root, label);
end
return
```

A.3.5 countmax.m

```
function [n1max, n2max] = f(Examples, att, nobj, split, dec_att)
%
% Function countmax counts the number of objects from each
% decision class that has attribute value over a given split
% for a given attribute
%
% Examples is the example matrix
%
% att is the attribute in question
%
% nobj is number of objects in Examples
%
% split is the split for attribute att
%
% dec_att is the target attribute
```

```
% Initiallize values
n1max = 0;
n2max = 0;

% Do for each object in the examples matrix
for i = 1:nobj
   if (Examples(i,att) > split)
      if Examples(i,dec_att) == 0
         n1max = n1max + 1;
      elseif Examples(i,dec_att) == 1
         n2max = n2max + 1;
      end
   end
end
return
```

A.3.6 countmin.m

```
function [n1min, n2min] = f(Examples, att, nobj, split, dec_att)
%
% Function countmin counts the number of objects from each
% decision class that has attribute value under a given split
% for a given attribute
%
% Examples is the example matrix
%
% att is the attribute in question
%
% nobj is number of objects in Examples
%
% split is the split for attribute att
%
% dec_att is the target attribute

% Initialize the values
n1min = 0;
n2min = 0;

% Do for each object in the examples matrix
for i = 1:nobj
   if (Examples(i,att) < split) | (Examples(i,att) == split)
      if Examples(i,dec_att) == 0
         n1min = n1min + 1;
      elseif Examples(i,dec_att) == 1
         n2min = n2min + 1;
      end
   end
end
return
```

A.3.7 findmostcommon.m

```
function commonclass = f(Examples, attval, nobj, dec_att)
%
% Function findmostcommon finds the most common decision value
% among the objects in an examples matrix
%
```

```
% Examples is the example matrix
%
% attval is a vector containing all the values for the
% decision attribute
%
% nobj is number of objects in Examples
%
% dec_att is the decision attribute (last in Examples)

% Find number of values for decision attribute, ntarg=number of values
[n,ntarg] = size(attval);

% Initialize temporary count matrix
C = [];

% Do for each value of decision attribute
for j = 1:ntarg
   % Initialize counter
   count = 0;
   % Do for each object in the examples matrix
   for i = 1:nobj
      if Examples(i,dec_att) == attval(j)
         count = count + 1;
      end
   end
   % Store number for decision value in count matrix
   C = [C count];
end

% Find largest number in count matrix
countmax = max(C);

% Find decision class corresponding to this number
index = find(C == max(C));
commonclass = attval(index);
[x,y] = size(commonclass);
if y>1   % If there are an equal number of objects from several classes
   commonclass = commonclass(1);
end
return
```

A.3.8 findrange.m

```
function [min, max] = f(Examples, nobj, att)
%
% Function findrange finds the range of values for a given attribute
%
% Examples is the example matrix
%
% nobj is number of objects in Examples
%
% att is the attribute in question

% Initialize min and max values
min = 1000;
max = -1000;

% Do for each object in the examples matrix
for i = 1:nobj
   if Examples(i,att) < min
      min = Examples(i,att);
   end
```

```
   if Examples(i,att) > max
      max = Examples(i,att);
   end
end
return
```

A.3.9 ncont.m

```
function NValues = f(tree, root, N, level)
%
% Function ncont finds the number of objects and the entropy
% value stored at each node in a binary decision tree
%
% tree is the decision tree in which to search
%
% root is the root of the tree
%
% N is the matrix in which the found values are stored
%
% level says which level in the tree is reached

% Get data from root
data = TreeNodeData(tree, root);

Entropy = data(2);
n1 = data(3);
n2 = data(4);
NValues = [N; level n1 n2 Entropy];

if ~(IsLeaf(tree, root)) % If root is not leaf node
   % Get the child nodes and their values
   children = GetChildren(tree, root);
   level = level + 1;
   NValues = ncont(tree, children(1), NValues, level);
   NValues = ncont(tree, children(2), NValues, level);
end
return
```

A.3.10 prunetree.m

```
function [tree, root] = f(tree, root)
%
% Function prunetree prunes the leaf nodes in a binary
% decision tree if two descending leaf nodes from a node
% have the same decision value
%
% tree is the decision tree in which to search
%
% root is the root of the tree

% Get the children of root
children = GetChildren(tree, root);

% If not both of the children are leaf nodes
if ~(IsLeaf(tree, children(1)) & IsLeaf(tree, children(2)))
   % If child 1 is a leaf node
   if IsLeaf(tree, children(1))
      % Prune the subtree with child 2 as a root
```

```
      [tree, root] = prunetree(tree, children(2));

      % Find the parent of the new pruned subtree, and
      % its new children for further pruning
      root = GetParent(tree, root);
      children = GetChildren(tree, root);

   % If child 2 is a leaf node
   elseif IsLeaf(tree, children(2))
      % Prune the subtree with child 1 as root
      [tree, root] = prunetree(tree, children(1));

      % Find the parent of the new pruned subtree, and
      % its new children for further pruning
      root = GetParent(tree, root);
      children = GetChildren(tree, root);

   % If none of the children are leaf nodes, prune both subtrees
   else
      [tree, root] = prunetree(tree, children(1));
      [tree, root] = prunetree(tree, children(2));
   end
end

% If both of the children are leaf nodes
if IsLeaf(tree, children(1)) & IsLeaf(tree, children(2))
   % Get data from both of the leaf nodes
   data1 = TreeNodeData(tree, children(1));
   data2 = TreeNodeData(tree, children(2));

   % If the two leaf nodes have the same decision value
   if data1(1) == data2(1)
      % Remove child 1 and child 2
      [tree, root] = Prune(tree, children(1));
      [tree, root] = Prune(tree, children(2));

      % Add decision value to parent node of the
      % two pruned leaf nodes
      tree = TreeNodeData(tree, root, data2(1));
   end
end
return
```