TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

NYYTI KINNUNEN

DECISION TREE LEARNING WITH HIERARCHICAL FEATURES

Master of Science thesis

Examiner: Prof. Tapio Elomaa
The examiner and topic of the thesis were approved on 1 Novevember 2017

ABSTRACT

NYYTI KINNUNEN: Decision tree learning with hierarchical features
Tampere University of Technology
Master of Science thesis, 57 pages
March 2018
Master's Degree Programme in Science and Engineering
Major: Theoretical computer science
Examiner: Prof. Tapio Elomaa
Keywords: machine learning, decision trees, missing data, hierarchical data

The performance of machine learning methods depends on the data they are given. Real life data sets can be incomplete and consist of various types of data. However, many methods are capable of handling only nominal and numerical features without any missing data, which causes loss of potentially useful information. Thus, this thesis had two research questions: can the information from hierarchical and interval values be utilized, and can the treating of missing values be integrated into a model handling untraditional data types.

In this work we developed a decision tree algorithm (DTHF) that uses hierarchical data to construct a model. The method requires that all data is preprocessed to a hierarchical form. In addition to nominal and numerical features, the method is capable of handling missing data, interval and hierarchical features, and several values for a single feature.

DTHF was tested using twelve data sets and the results were compared with results from CART and C4.5 decision tree algorithms. Tests were conducted using data sets without missing values as well as with sets with various rates of missing data. If data are not missing, there is no significant difference between DTHF, CART, and C4.5. However, if data are missing CART performs remarkably better than C4.5 and DTHF, which have similar performance.

More tests are needed to give a sufficient understanding of the method's performance. Especially, there is a need for tests utilizing the DTHF's capabilities. Further research topics are expanding the method into a random forest and studying how the transforming of data to a hierarchical form should be done. Other topics could be adding pruning and studying how the data set affects the performance when data are missing.

TIIVISTELMÄ

NYYTI KINNUNEN: Hierarkinen data päätöspuissa
Tampereen teknillinen yliopisto
Diplomityö, 57 sivua
maaliskuu 2018
Teknis-luonnontieteellinen koulutusohjelma
Pääaine: Ohjelmistotiede
Tarkastajat: Prof. Tapio Elomaa
Avainsanat: koneoppiminen, päätöspuut, puuttuva data, hierarkinen data

Koneoppimismenetelmien suorituskyky riippuu sekä koulutuksessa käytetyn datan määrästä että laadusta. Datasettien laatu ja sisältö voi kuitenkin vaihdella merkittävästi. Monet koneoppimismenetelmät kykenevät käsittelemään vain nominaalista ja numeerista dataa, jolloin mahdollisesti tärkeää informaatiota ei voida hyödyntää. Tämä työ pyrkii vastaamaan kahteen kysymykseen: voidaanko datan hierarkisuutta hyödyntää datan luokittelussa ja voitaisiinko puuttuvan datan käsittely integroida samaan menetelmään.

Työssä on kehitetty päätöspuumenetelmä (DTHF), joka muodostaa mallin käyttäen piirteiden hierarkioita kokonaisten piirteiden sijaan. Menetelmää varten data on esikäsiteltävä hierarkiseen muotoon. Nominaalisten ja numeeristen piirteiden lisäksi menetelmä kykenee käsittelemään esimerkiksi puuttuvaa dataa, intervallipiirteitä, hierarkisia piirteitä, sekä useita arvoja yhdelle piirteelle.

Kehitetty menetelmä testattiin käyttäen kahtatoista datasettiä. Testit tehtiin käyttäen sekä alkuperäisiä settejä, että settejä joista oli poistettu dataa. Testien tuloksia verrattiin CART- ja C4.5 päätöspuualgoritmeihin. Jos dataa ei puuttunut, kehitetty menetelmä suoriutui testeistä yhtä hyvin kuin verrokkimenetelmät. Mikäli tietoa puuttui, CART-algoritmi oli selkeästi paras, kun taas C4.5 ja DTHF suoriutuivat keskenään yhtä hyvin. Kehitettyä menetelmää olisi kuitenkin testattava datalla, joka toisi esiin DTHF:n vahvuudet, kuten datan hierarkisuuden.

Tärkein jatkotutkimusaihe olisi menetelmän laajentaminen satunnaismetsäksi, mikä luultavasti parantaisi menetelmän suorituskykyä. Toinen merkittävä jatkotutkimusaihe olisi tutkia, miten datan muuttaminen hierarkiseksi kannattaisi tehdä. Data muutetaan intervallipuita käyttäen, mutta itse puiden muodostamiseen on useita tapoja. Muita kehityskohtia olisivat muun muassa karsinnan lisääminen menetelmän toteutukseen sekä jakometriikan vaikutuksen tutkiminen suorituskykyyn.

PREFACE

This work has been done for a company called Vainu. I am thankful to the company for the opportunity to do my master's thesis while working there and especially for introducing me to machine learning. Because this work was done for a company, I was in the fortunate position of having two supervisors. I want to thank professor Tapio Elomaa for supervising and examining the work and giving me feedback during the process.

I am most grateful for PhD Lauri Häme, as he is the person who made it possible for me to write this thesis. Firstly, for accepting to supervise my work, and secondly for coming up with the idea for the developed method. He gave me an unbelievable amount of support throughout the process and I feel he believed in the project when I did not. His feedback was very valuable and he used his time to concretely help me with my problems. I feel like I cannot thank him enough, and I am grateful for being able to work with such a kind and talented person.

Tampere, 21.3.2018

Nyyti Kinnunen

CONTENTS

1. Introduction 1
2. Aspects of decision tree learning 3
    2.1 Data types 5
    2.2 Missing data 6
    2.3 Mixed data 9
    2.4 Hierarchical data 11
        2.4.1 Quinlan-encoding 12
    2.5 Decision Trees 13
        2.5.1 An example of split criteria: information gain 14
        2.5.2 Missing data in decision trees 16
        2.5.3 Strengths and weaknesses of decision trees 17
    2.6 Evaluation metrics for classifiers 20
    2.7 Known hierarchical models 23
3. Decision tree with hierarchical features 25
    3.1 Transforming data 26
        3.1.1 Flat nominal features 27
        3.1.2 Hierarchical nominal features 28
        3.1.3 Numerical features 28
        3.1.4 Interval features 30
        3.1.5 Handling features with multiple values 32
    3.2 DTHF Algorithm 34
        3.2.1 DTHF with complete data 36
        3.2.2 DTHF with missing data 39
4. Empirical results and analysis 43
    4.1 Description of the used datasets 43
    4.2 Performance 44
        4.2.1 Performance with complete data 45

4.2.2 Performance with missing data 46

5. Conclusions 50

Bibliography 52

LIST OF FIGURES

2.1 Transforming nominal feature *color* into a numerical feature using one-hot encoding. 10

2.2 Feature `home city` which has a hierarchical structure. 11

2.3 A decision tree constructed using the iris data set. The tree predicts the type of an iris flower. 14

2.4 Entropy for a set with two possible values with probabilities p and $(1-p)$. 15

2.5 Two decision trees trained on the same data [36]. The decision tree presented by the blue line is able to somewhat generalize the data but the green overfits to the training data. 20

2.6 A confusion matrix. 21

3.1 An example of a hierarchical nominal feature. Feature in the figure is `marital status` and it has seven possible nominal values (colored nodes). 28

3.2 A graph for transforming a numerical value to hierarchical. In the training data feature's values were integers between 7 and 43 and the used split limit was 10. 29

3.3 An example of an interval DAG that is used for transforming interval features into a hierarchical form. The first part on a node is the interval and the second value the identifier of the node. The colors indicate the state the node was added to the graph. 31

3.4 Forming the intervals based on the training data for the graph used for transforming interval features. 33

3.5 An example of a decision tree constructed by DTHF. The tree was constructed using data without missing values. 35

3.6 An example illustrating how adding new split rules affects $\mathcal{X}_p$. 36

3.7 The hierarchy for feature `work class`. This tree is used to transform a value of the feature to a hierarchical form. 38

3.8 An example of a hierarhichical decision tree 39

3.9 An example of a decision tree constructed by DTHF when there are missing data. 40

4.1 Average of F1-scores for methods for each missingness rate. 48

LIST OF TABLES

1.1 Data set containing information about a company. The data demonstrates the need for this work as it has many imperfections like missing data and unusual data types. 2

2.1 Data set containing used cars 3

2.2 Characteristics of different learning methods [20]. * represents the worst and **** the best performance. 18

3.1 An example of data where DTHF performs well. 41

4.1 The data sets used in testing. The table lists some of their qualities to illustrate the differences between the data sets. 44

4.2 10-fold cross validated F1-scores and standard deviations of unpruned DTHF, CART, and C4.5 algorithms using split limit 20 on various data sets. Green color indicates the best performance and red the worst. 45

4.3 F1-scores of DTHF, CART, and C4.5 using 10-fold cross-validation on data sets where data has randomly been removed. 47

5.1 Average F1-scores from all tests for each method on both complete and missing data. 51

LIST OF ABBREVIATIONS AND SYMBOLS

CART	Classification and regression tree
CCA	Complete case analysis
DAG	Directed acyclic graph
DTHF	Decision tree with hierarchical features
ISIC	The International Standard Industrial Classification of All Economic Activities
MAR	Missing at random
MCAR	Missing completely at random
NMAR	Not missing at random

d	data instance
D	data set
f	feature
h	hierarchy level
l	label

1. INTRODUCTION

Collecting, storing, and sharing data is easier and cheaper than ever before. This has made data, and especially large amounts of data, more accessible. Hence, data is being utilized in applications more than ever. Machine learning is one of the fields which have substantially benefited from the amount of available data. Many problems previously thought to be unattainable, like a computer winning humans on Go [39], have been solved using efficient machine learning methods and large quantities of data.

Usually, a machine learning algorithm is given a data set which it uses to construct a model. A *data set* D is simply a collection of similar data instances and each *data instance* $d \in D$ is a collection of values describing a single object. A data instance can be for example, a picture, a medical record, or measurements of a flower. Since all data instances in a data set represent the same object type, a data set has a domain it can be applied to. For example, a data set consisting of medical records can be used for predicting whether a person has cancer but not for predicting tomorrow's weather. The values in a data instance are called features. A *feature* f can be any aspect of the object the data instance represents. It can be, for instance, a single pixel of a picture, gender of a patient, or length of a petal.

When constructing a model using a machine learning algorithm, the size of the data set is important but that alone is not enough as the quality of the data is also of great importance. Unfortunately, many of the available data sets are imperfect in some way. This is especially the case when the data set is combined from several sources. Table 1.1 represents such data. In this case, each data instance consists of four features that describe a company: `founded`, `staff number`, `industry code` and `technologies`. Feature `founded` is a simple numerical feature representing the founding year of the company. The second feature is the staff number of the company. Some sources report an exact number for the feature while others give an interval. Industry code is stored as ISIC (The International Standard Industrial Classification of All Economic Activities) code which is a hierarchical value where the industry classification gets more specific with each number. Technologies is a list of technologies the company uses on their website. The data is also missing a

Table 1.1 *Data set containing information about a company. The data demonstrates the need for this work as it has many imperfections like missing data and unusual data types.*

features	`founded`	`staff number`	`industry code`	`technologies`
data line 1	2016	234	992	[javascript, php]
data line 2		100-300	93	[javascript, node]

value.

The main motivation for this thesis comes from the type of cases described above. Most commonly data sets consist of numerical and nominal data which is why most methods can only those data types. However, as seen earlier they are not the only possible types for data. These diverse data types can contain information that would be lost if they were simply converted to nominal or numerical values or just discarded from the data set. Many data sets are also missing data and dealing with such data is an integral part of data analysis. The research question in this thesis is two-folded. Firstly, we want to know if the information from hierarchical and interval features can be utilized in a way that it would bring new insight to the classification. The scientific background for using hierarchies in classification exists, even though it seems to be a relatively under-researched area. Secondly, we want to know if missingness can be integrated to the model so that the missing values would not have to be treated separately.

The main goal in this thesis is to develop a decision tree classification algorithm which is capable of handling the type of data presented in Table 1.1. This is done by transforming all features into a hierarchical form and using the hierarchies to form a decision tree. One of the main questions is, does the hierarchical approach have any advantages compared to normal decision tree algorithms. This is tested by comparing the performance of the developed method with existing decision tree algorithms.

The rest of the thesis is divided into four chapters. Chapter 2 describes the previously outlined problems like missing data and different data types in more depth and presents existing solutions to the problems related to them. The chapter also gives necessary background information about decision trees. After reading the chapter, the reader should be able to understand the developed method presented in Chapter 3. The method itself consists of preprocessing the data and constructing the model. In Chapter 4 the method is tested using public data sets and the results are compared with existing well-known methods. Finally, Chapter 5 presents the conclusions of this work.

2. ASPECTS OF DECISION TREE LEARNING

Machine learning algorithms use data sets which are collections of data instances $D = (d_1, d_2, \ldots, d_n)$ where n is the number of instances in the data set. All data instances in a data set have the same form and each data instance d_i consists of features $d_i = (f_{i,1}, f_{i,2}, \ldots, f_{i,m})$ where $f_{i,j}$ is the value of the jth feature on the ith data instance. The jth feature is denoted by f_j. There is no limit for the number of features in a data instance but in practise it varies from few to hundreds of thousands. Each instance is also given a *label* l_i which is the feature we are interested in predicting. A label can be for instance the age of a person in a picture, cancer diagnosis, or species of a flower.

Table 2.1 gives an example of a data set from Kaggle Datasets [21] fulfilling the above characterization.The data consists of information about used cars, and the goal is to predict their prices. In the table, each column represents a feature, whereas each row is a single data instance. The data set in question has three data instances d_1, d_2, and d_3 and each instance represents a particular car. Each instance consists of three features: `type`, `kilometres`, and `registration year`. The fourth column `price` is the label.

Table 2.1 *Data set containing used cars*

	f_1	f_2	f_3	l
	type	kilometres	registration year	price
d_1	Golf 3 1.6	150 000	1993	480
d_2	Mazda 3 1.6 Sport	150 000	2004	2000
d_3	Renault Clio 1.4	125 000	1999	590

Machine learning methods can be divided into two groups: supervised and unsupervised methods. With *unsupervised* methods the program is given a set of data containing no labels for data instances and it has to find the characteristics without further information. An example of this could be giving the program a set of photos of animals and the problem is to group the pictures by species. In *supervised* methods the program is first given a training data set and the program is told which animal is in which photo. The intention is to teach the program the characteristics of the problem. Using this data, the program constructs a model which is then tested using a separate labelled data set the program has not seen before. Here an example could be detecting diabetes from patient's medical records. The algorithm is given the medical records of both people who are known to have diabetes and people who are known not to have it. Using this data the program is supposed to deduce whether a new person has diabetes or not. This kind of learning needs good training data, since the model can only be as good as the training data it is given. From now on in this thesis we will only concentrate on supervised classifiers. [32, pp. 4–5]

Constructing a model using a supervised method requires *training data*. It is data for which the labels of the data instances are known. These labels are stored in a list $L = [l_1, l_2, \cdots, l_n]$, where each label corresponds to an instance in the same index in D. In addition to training data, many algorithms require *test data*. While training data is used to construct the model, test data is used to test the performance of the constructed model. Test data should be completely separate from the training data and it should not be used in training.

Training and test data sets are formed by splitting the original data set D in two: $D_{train} = (d_1, d_2, \cdots, d_k)$ and $D_{test} = (d_{k+1}, d_{k+2}, \cdots, d_n)$. The size of the test set varies but it is often around 30% of the data instances [15, p. 370]. The original data set D might be shuffled before making the split in order to make both sets represent the original data as well as possible. Sometimes *validation data* is used in addition to test data to enhance the performance of the model constructed on training stage. The performance can be enhanced for example by tuning the parameters. However, even when using validation data, test data is still used only to test the performance of the model.

Above we discussed making predictions using a trained model. There are two types of predictions that can be made: class labels and numerical values. In this thesis we are concentrating on predicting class labels which is called *classifying*. There are no limitations for the labels themselves but the set of labels has to be finite and discrete. Further, since supervised algorithms use training data there has to be samples from each label in the training data. The classifier is not able to predict a label it has not been shown enough samples of. In addition to predicting labels, it is possible to make numerical predictions. This is called *regression*. Here the labels are numbers and the range of possible labels is continuous and the label can be for example the price of a car, the income of a person, or the revenue of a company. It is not necessary to have an instance of each possible value, and this might even be impossible. [15, pp. 327–330]

In this chapter we present the necessary background information for this work. The next section goes through the most common data types found in data sets. Missing data is discussed in Section 2.2. The section covers the types of missingness as well as methods of handling missing data. Section 2.3 discusses mixed data: how it can be a problem and what kind of solutions are developed for it. Hierarchical data and how it can be used is covered in Section 2.4 and Section 2.5 introduces a machine learning method called decision trees. The section explains how they can be used, how they are constructed and what are their strengths and weaknesses. The basic evaluation methods for machine learning methods are defined in Section 2.6. Finally, the existing methods utilizing hierarchical data are discussed in Section 2.7.

2.1 Data types

Data can be collected from various phenomena and hence the data can be different by nature, for example a data set can represent customer satisfaction or a scientific measurement. Even in a single data set the features can be of different types and hence they can have very different properties. This might cause problems when fitting a model since not all methods can handle mixed data types. Han et al. [15, pp. 40–44] divide data types into four groups: nominal, ordinal, interval, and ratio.

Nominal features are categorical features which often are symbols or names of things, for example, features sex, eye color, and nationality are nominal values. They are not quantitative and they cannot be meaningfully ordered. Consequently, nominal features cannot be added or divided by each other and, for example, mean is not defined. Instead the most common value, mode, can be defined. but this does not mean that a number cannot represent a nominal value. If a nominal feature has number as a value it is treated as a symbol instead of a number.

Ordinal features are nominal values which have an order among them. However, they are not quantitative so the magnitude between the values cannot be determined. Suppose a feature measuring customer satisfaction which has values *satisfied*, *neutral*, and *dissatisfied*. The values have an intuitive order but the exact difference between *satisfied* and *dissatisfied* cannot be defined.

Numerical features can be divided into two categories: interval-scaled and ratio-scaled. *Interval-scaled features* have numerical values which have an order and the exact difference between values can be determined. Their weakness is they do not have a true zero-point, meaning that value zero is arbitrary and, consequently, ratios between interval-scaled values cannot be calculated. An example of an interval-scaled feature is temperature in degrees Celsius. The exact difference between two temperatures can be calculated for any two values but if temperature is zero, it does not mean there is no heat since $0°C$ is defined as the freezing point of water. *Ratio-scaled features* are interval-scaled features which have a true zero-point. This makes it possible to calculate ratios between values and hence mathematical functions are defined for them. Salary, age, and height are examples of ratio-scaled features.

There are also data types which are not covered by the previous groups. One type are *interval features* which have intervals as value. Cormen et al. [9, p. 348] define an interval as a set of numbers presented by an ordered set of numbers $[t_1, t_2]$ where $t_1 \leq t_2$. The interval is a set $\{x \mid t_1 \leq x \leq t_2\}$ where each number between t_1 and t_2 belongs to the set. The endpoints t_1 and t_2 of an interval might or might not be included in the set. The previously presented interval is closed since it includes both of its endpoints. If neither of the endpoints belong to the interval it is an open interval (t_1, t_2) and if only one of the endpoints belongs to the interval it is half-open $[t_1, t_2)$ or $(t_1, t_2]$. The interval can consist of real numbers $[t_1, t_2] = \{x \mid t_1 \leq x \leq t_2, x \in \mathbb{R}\}$ or integers $[t_1, t_2] = \{x \mid t_1 \leq x \leq t_2, x \in \mathbb{Z}\}$. The length of an interval is the difference between endpoints $length_{[t_1,t_2]} = t_2 - t_1$. Because intervals cover all values between endpoints, inclusion for closed intervals can be checked by comparing the endpoints, $[t, s] \subset [x, y]$ if $t \leq x$ and $s \leq y$. Dynamic sets of intervals can be organized and handled using interval trees [9, pp. 348].

2.2 Missing data

Missing data is a frequent problem in real world data sets where there are instances which do not have a value for all of the features in the data set. There are several reasons why data might be missing, for example, there was a malfunction while collecting the data, the data was erroneous, or the respondents did not answer to all questions [4]. Missing data lowers the quality of a data set and can thus lower the performance of a classifier trained with that data. Several methods have been developed for handling the problem. When choosing a method for a data set, in addition to the rate of missingness, it is important to take into account the reason why data is missing from the data set, as it can affect the results significantly. The common taxonomy in literature for data missingness was suggested in 1987 by Rubin and Little [24]. They divided missingness in three classes:

1. *Missing completely at random* (MCAR): Missingness depends on neither observed or unobserved data and each value has an equal chance of missing. An

example of this could be tossing a coin on each value whether to remove it or not.

2. *Missing at random* (MAR): Missingness depends on observed data but not on the unobserved data. In other words, data are missing conditionally at random where we can control the condition. For example, on a health survey a child might not give a phone number because she does not have one. This does not affect her health and the missingness of phone number depends on the age.

3. *Not missing at random* (NMAR): When the missingness is not explained by MCAR or MAR, it is NMAR. In this case the missingness depends somehow on unobserved data. It might depend on unobserved predictions or even the missing value itself. For example, persons who's yearly income is under $20,000 might not want to answer a question about their financial situation.

Over the years, several methods have been developed for handling missing data and they can be divided into five categories [23, pp. 19–20] [34, p. 1627]. The categories presented below do not exclude each other meaning a method can belong to several of them.

1. *Acquire missing values*: Sometimes it is possible to get the missing information, for example by buying it from a third party or by conducting further analysis. Usually this comes with an additional cost. Acquiring data should be used if possible when no data treatment method is able to handle the missingness with an acceptable accuracy or they are not possible to conduct.

2. *Procedures Based on Completely Recorded Units*: Perhaps the simplest method of handling missingness is to use only the complete instances. This means discarding the instances with missing values and conducting a normal analysis using the remaining data. In order to work properly, the method requires missingness at random, otherwise the resulting data set might be biased. Another issue with the method is the amount of data since it is possible to end up with too little data for conducting any further analysis.

3. *Weighting Procedures*: After the incomplete instances have been removed, it is possible to weight the remaining instances to make the data represent better the distribution of the original data. Weights are derived from the probabilities of missingness. An example of a weighting method is inverse probability weighting [38].

4. *Imputation-Based Procedures*: Instead of discarding data with missing feature values, these methods keep all of the data instances and fill in the missing values using the known values of the data. After filling in the values, standard methods for processing the data can be used. There are several methods for estimating the missing values, for example, using existing values, using the mean of existing values, and estimating the missing values using regression.

5. *Model-Based Procedures*: For some problems it is possible to construct a model of the problem using likelihoods and distribution of the data. Such methods include maximum likelihood and multiple imputation.

There are also methods which do not fall under the previous categories. An example of such method is to treat missingness as any other value in the data. This was proposed by Quinlan [29, pp. 97–98] but he concluded that the approach is not a good solution to the problem. Nowadays, many scientific studies handle missing data using *complete case analysis* (CCA) [11] which simply excludes all instances for which any data are missing. This is a simple approach to data missingness which explains its popularity despite its weaknesses. Gelman and Hill [14, p. 531] give two main problems with CCA the main problem being that the data has to be MCAR. Otherwise, the remaining data set would not represent the original data but a slightly different data set since if the missing data is dependent on the known data, excluding instances would cause a bias to the remaining data set. Unfortunately, often when the method is used in scientific studies the reason for data missingness is completely ignored [11]. Another significant problem with the method is the amount of data since excluding instances can lead to an insufficiently small data set. Especially, if instances have many features it is more likely for at least one of the values to be missing. Additionally, data can be missing only during the training phase since the model is constructed using only complete data. The resulting model assumes the all data is available for each instance resulting that if the test data is not complete, CCA cannot be used.

Another traditional method is single imputation. Unlike CCA, imputation does not discard data instances but instead it fills in the missing values. Single imputation methods treat the imputed values as real values and do not take the uncertainty of the imputation in account. There are several methods for filling a missing value, for example using the mean of the values, using regression, or matching similar instances. *Mean/mode imputation* is one of the simplest ways for performing a single value imputation. The missing values are replaced with the mean of the feature if the feature is numerical and with mode if the feature is nominal. This does not remove data from the set but lowers the variability of the data and the method often causes biased estimates [11]. Eekhout et al. [12] compared different missing data treatment methods and came to the conclusion that mean imputation results in highly biased data and suggest that any form of mean imputation should not be used.

Baraldi and Enders [3] argue that traditionally used complete case analysis and single imputation are often are not adequate approaches. Instead, multiple imputation and maximum likelihood should be used which were described as the "state of the art" methods. One of their strengths is that they produce unbiased estimates with MCAR and MAR data. *Multiple imputation* takes into account the uncertainty of imputation by imputing each value several times and thus forming multiple complete data sets. Multiple imputation is done in three stages: imputation, analysis, and pooling. In imputation stage the missing values are filled in. This is done k times for each missing value and hence k complete data sets are formed. There are several methods for imputing the values but data-augmentation is the most popular method for normally distributed data. The imputed data sets are treated as complete sets and normal analysis is conducted on each of the k data sets. Pooling combines the results of the k separate analysis into one final result.

Maximum likelihood has a completely different approach to the problem as it does not try to impute the missing values. Instead, it tries to define the most probable parameters to produce the given data. Once the parameters are defined, the missing values can be filled using the parameters. The parameters are estimated using log likelihood which represents the standardized distance between the value and the parameters. The log likelihood value is calculated for every value in the data set and the values are summed. These sums are then compared and the parameters that produced the highest value are chosen. The method uses all available data while testing the possible parameters.

2.3 Mixed data

In real life data sets can consist of several types of features, and all types can bring important insight for the problem. Data in which several types of features are present are called *mixed data*. For example, a data set describing cities can have a nominal feature `country` and a numerical feature `population`. This poses a problem since many machine learning algorithms can handle only one type of data.

One solution for the problem is transforming the incompatible data into a proper form. There are several methods for encoding nominal data to numerical but perhaps the most intuitive way to encode the nominal values is to simply assign each value a number. This is called *ordinal coding* and while it does not grow the dimensionality of the feature, it does propose the values have an ordering among them which they in reality do not necessarily have. However, it is possible that not all values of the nominal feature are equal. For example, not all directors are as recognized and hence not as valuable when rating a film and in such case it might be beneficial to weigh the values. One could also use *binary encoding* in which the values are first encoded as ordinal and these values are then transformed into a binary number. The digits from the resulting number are split into separate features. Another methods include for example, sum encoding, polynomial encoding, backward difference, and Helmert encoding [28].

A more detailed description is given for *one-hot encoding* [37]. In one-hot encoding a new feature is created for each value of the nominal feature. Each created feature has the value of either 1 or 0. Value 1 indicates that the encoded value is the same the feature represents and 0 that it is something else. Figure 2.1 gives an example of nominal feature `color` which has three possible values: *red*, *yellow*, and *green*. After applying one-hot encoding, the feature is transformed into three features, one for each value. The problem of one-hot encoding is the number of features it might produce. Creating a new feature for every nominal value can cause the dimensionality of the data grow remarkably. Also, some algorithms assume the data to be normally distributed which one-hot encoded data is not.

$$\text{color} = [\text{red, yellow, green}] \quad \Rightarrow \quad \begin{aligned} \text{red} &= [1,\ 0,\ 0] \\ \text{yellow} &= [0,\ 1,\ 0] \\ \text{green} &= [0,\ 0,\ 1] \end{aligned}$$

Figure 2.1 *Transforming nominal feature color into a numerical feature using one-hot encoding.*

Conversion can also be done the other way around, from numerical to nominal. However, here the problem is loss of data since the accuracy of values is reduced. The idea is to divide the range of values into separate segments. Each segment represents a value in the nominal feature. The original numerical value is transformed to the label of the segment the value falls into. For example, feature `weight` could be transformed to nominal using groups *underweight*, *normal*, and *overweight*. There are several ways for defining the groups. This can be done for example by an expert, or by using binning, clustering or decision trees [15, pp. 115–117].

2.4 Hierarchical data

Many methods assume the input data to be flat, i.e., a list of independent values and consequently the potential inner structure of the values is not taken in account in the model. Of course, not all features would even have a meaningful inner structure, for instance `eye color` and `blood type` could be such features. However, many features do have a potentially useful inner structure or hierarchy. Hierarchies can be found for example in geography, biology, and numerical intervals. Figure 2.2 presents feature `home city` which is an example of a feature that has a useful hierarchical structure. For example, the cities can be categorized according to the continent and the country they are located in. This gives additional information about the similarities of the cities which could not be deduced using only the names of the cities. For example, cities *Milan* and *Beijing* are probably more different than *Milan* and *Madrid* since *Beijing* is in *China* while *Milan* and *Madrid* are both in *Europe*.

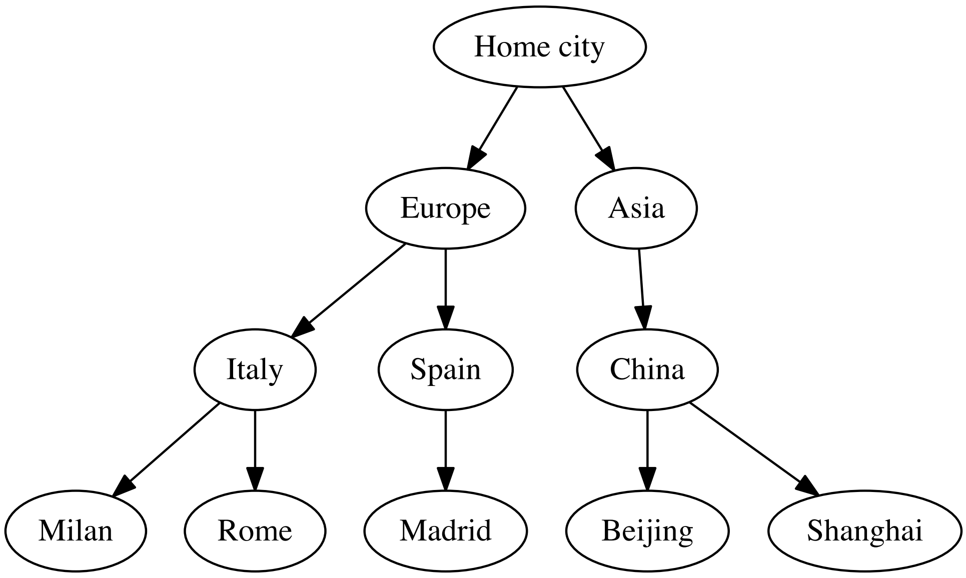


Figure 2.2 *Feature `home city` which has a hierarchical structure.*

Hierarchical features are features which have categorical, i.e. nominal, values organized in a hierarchical structure [16]. The hierarchical structure is defined as $(C, \leq_h)$, where C is a set of categories and $\leq_h$ is a partial order representing the supercategory relationship (for all $c_1, c_2 \in C : c_1 \leq_h c_2$ if and only if c_1 is a supercategory of c_2) [7]. The actual values of the feature can belong to any level on the hierarchy. In other words, they do not have to be at the leaves nor do the categories on the internal nodes have to be actual values of the feature. The hierarchy can for form a tree, DAG, or a general graph. The previous example with feature home town is an example of a hierarchical feature with a tree structure.

Han and Lam [16] discuss the possible use cases for hierarchical features. The hierarchy represents the a priori knowledge about the feature and how the possible values relate to each other. This knowledge can be used to tackle problems with the quality of data. It is possible that all data are not recorded with equal accuracy or a value is only known to belong to a set of values which might be caused by errors or the expense of collecting the data. The hierarchical structure can be used to give a reliable estimate instead of a uncertain precise value. However, the problem with hierarchical structures is that the number of categories can grow high.

2.4.1 Quinlan-encoding

Most machine learning algorithms cannot handle hierarchical features by default. Therefore, the hierarchical values have to be encoded into a form the standard algorithms can handle. When doing this, there are two aspects to take care of. Firstly, no information about the hierarchies should be lost during the encoding. Secondly, most machine learning algorithms assume that the data instances they receive have similar form. This means each instance should have the same features in the same order and each instance should have an equal number of features. However, a hierarchy tree might not be balanced which means that all values do not have an equal amount of hierarchy levels.

Almuallim et al. [2] describe a method for encoding hierarchical values developed by Quinlan [2, p. 14]. He suggests encoding a hierarchical nominal value f_j by creating a new nominal feature f_j^i for each level i of the hierarchy tree, where $1 \leq i \leq h$ and h is the depth of the hierarchy tree. This means that a single encoded hierarchical feature consists of h separate nominal features. The possible values for each created feature f_j^i are the values on the ith level of the hierarchy tree. For example, the hierarchy introduced in Figure 2.2 has three levels. The possible values on the first level are {*Europe, Asia*}, on the second level {*Italy, Spain, China*}, and on the third level the leaves of the tree. Naturally, not all combinations are possible, only the combinations formed by the paths from the root of the tree to the value. An encoded value for value *Madrid* would be $f_j = [f_j^1, f_j^2, f_j^3] =$ [*Europe, Spain, Madrid*]. If some path does not have h levels, it can be made longer by adding duplicate nodes. The position where the duplicates should be added depends on the hierarchy and the problem.

2.5 Decision Trees

Decision trees are a supervised machine learning method that can be used for both classification and regression. The basic idea behind decision trees is simple: instances belonging to different classes have at least one different value on one of their features [20]. This is why the method works by sorting the instances by the values of the features. Sorting is done by a systematically arranged series of questions so that each question queries a feature and branches based on the value of the attribute [43, p. 2]. In addition to machine learning, decision trees are used for example in data mining and operations research [32, pp. 5–8].

The tree in Figure 2.3 is an example of a decision tree. Its purpose is to predict which type of iris plant a data instance represents: setosa, versicolour or virginica. Each instance in the data has the following form: $d = [\texttt{sepal length}, \texttt{sepal width}, \texttt{petal length}, \texttt{petal width}]$ where each value is given in centimeters. In the figure, the first line on root node and decision nodes is the rule used to split the data called the *split rule*. Instances for which the rule is true are passed to the left child of the node and the rest to the right child. Entropy indicates how similar the instances in the node are, samples tell how many instances there are in the node and value shows the distribution of the instances between possible labels. The last line, class, is the label the node predicts. A new data instance is classified by following the rules on the nodes from the root to a leaf node. The instance is predicted to have the same label as the leaf . For example, $d = [6.3, 2.3, 4.4, 1.3]$ would be classified as versicolour.

Decision trees are directed trees. This means they are directed graphs with no cycles and satisfy the following properties: the graph has a single root node, the root node does not have any edges entering it and every other node has exactly one entering edge, and path from the root to a leaf is unique [35, p. 2]. The graph consists of *root*, *decision nodes* (internal nodes), and *leaves* [31, p. 5]. Root is the first node in the graph and it contains all of the data. A decision node is a rule used to split the data and a leaf node indicates the label to predict. In Figure 2.3 the type of a node is indicated by the color of the node.

Decision trees are constructed recursively. The process starts from the root node and continues until there are no nodes to split further. On each node a *stopping condition* is checked. If it is not met, the node is a decision node. In this case a split rule for the node is searched and data is split to the new nodes according to the rule. If a stopping condition is met, the node will not be split further and it becomes a leaf node. The stopping condition depends on the implementation but it

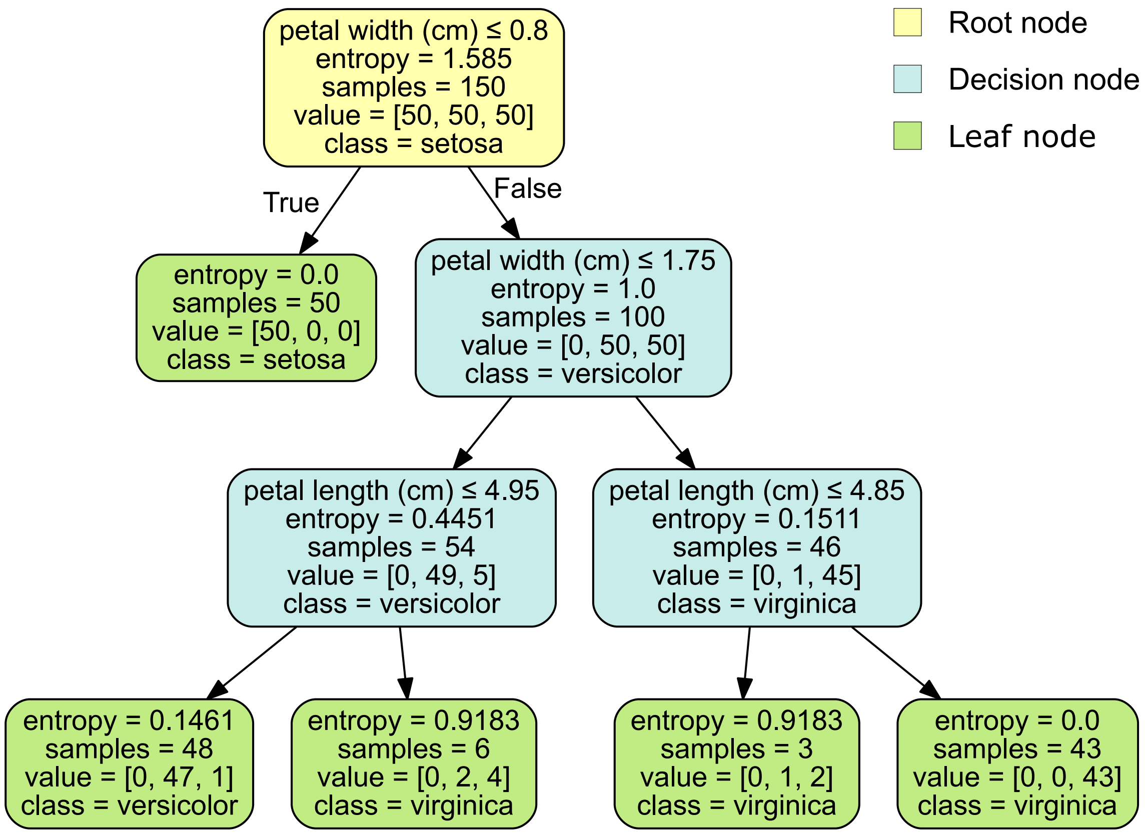


Figure 2.3 *A decision tree constructed using the iris data set. The tree predicts the type of an iris flower.*

can be for example, a *split limit* which is a requirement of the number of instances in order to make a split, a limitation to the depth of the tree, or to stop when all of the instances in a node have the same label. Finding good split rules is of essence on decision tree construction since they define the distribution of data and hence the predictions the model can make. Unfortunately, it was shown in 1973 by Hyafil and Rivest [18] that constructing an optimal binary tree from decision tables is NP-complete. Since, more conditions have been proven under which the construction of an optimal decision tree is NP-complete [13, p. 11]. Therefore a heuristic has to be used to select the split rules. Often the choice of the heuristic is greedy, which means making the locally optimal choice. Here it means that a rule that divides the data as well as possible on a single node is chosen. A greedy heuristic does not necessarily give the optimal solution.

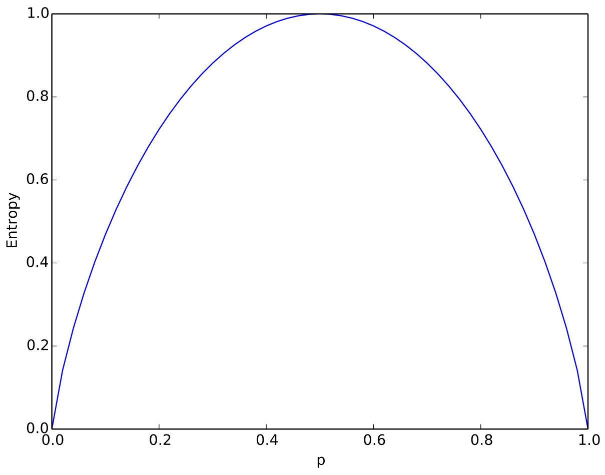
2.5.1 An example of split criteria: information gain

As explained, decision trees are constructed by partitioning the data using split rules. Usually each node has several potential split rules and the problem is to find the rule resulting in the best possible split. There are several criteria for determining the best split. Few of the most common univariate criteria in literature are impurity based, information gain, and gini index [32, pp. 53–55]. Each criterion has a different mathematical background and all of them have their strengths and weaknesses. However, majority of studies have come to the conclusion that there is no significant difference between the criteria [27].

Information gain is a splitting criterion which used for example in Quinlan's C4.5 algorithm [31]. It defines the goodness of a split rule by the decrease of entropy caused by the resulting partition. In information theory *entropy* is a measure of impurity of the data. With decision trees it is used to measure the homogeneity of a node. The more homogeneous the instances are in a node, the lower the entropy. Entropy is defined as

$$E(S) = -\sum p_i \log_2 p_i,$$

where S is a data set and p_i is the probability for value i. Figure 2.4 visualizes entropy for a set which has two values occurring with probabilities p and $(1-p)$. Entropy $E(S)$ is plotted in Figure 2.4 (a) for such set. The possible values for entropy in this case are between 0 and 1. The cases where entropy achieves minimum and maximum value are visualized in Figure 2.4 (b). The maximum entropy is achieved when the probability for a value is 0.5, meaning the sample consists evenly of two values and the homogeneity is at its lowest. When the sample consists of only a single class, entropy is 0 because the set is completely homogeneous.



(a) The entropy function plotted for all values of p.

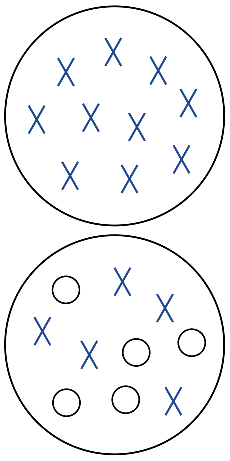

(b) The upper set has entropy of 0 and the lower entropy of 1.

Figure 2.4 *Entropy for a set with two possible values with probabilities p and $(1-p)$.*

Information gain for a split is defined as the difference between the entropy of a parent and the weighted average entropy of its children. When choosing the best split rule, each possibility is tested by calculating the information gain for all possible rules. The entropy is calculated for each child with the assumption that feature X was used to make the split. The weighted entropy for children is defined as

$$E(S|X) = \sum_{c \in X} P(c)E(c),$$

where c is a value of feature X, $P(c)$ is the probability for value c, and $E(c)$ is the entropy for c. Using the given definitions, information gain is defined as

$$Gain(T, X) = E(T) - E(T|X)$$

The feature which gives the highest information gain is chosen as the split rule.

2.5.2 Missing data in decision trees

Quinlan [30] brings up three problems that arise when decision trees are used with incomplete data. The first problem occurs while finding the best split rule. The rate of missingness on each feature might differ greatly. How should this be considered while choosing the split rule? The second problem appears after the split rule has been decided on. How should we treat instances which are missing the value used as split rule? The third problem is related to the second problem. After the tree is constructed and we are classifying a new instance, how can we classify an instance that is missing data?

There are several methods for handling missing data with decision trees. The general missing data treatments described in Section 2.2 like CCA and imputation can easily be used with decision trees. These methods do not encounter the second and third problem described above since they work with complete data. Aljuaid and Sasi [1] compared imputation methods for decision trees and concluded that imputing using expectation-maximization works well if the values are numerical and hot-deck imputation if the data is nominal or mixed. In addition to general missing data methods, it is possible to integrate handling missing data in the model. The simplest methods include treating missingness as a separate nominal value, and adding a rule for missing data to each node. MIA (missingness incorporated in attributes) is an example of such method. It handles the instances with missing values as a single group and tries to find split that works well even the missing values are assigned to one of the groups [42]. Treating missingness as a value can be effective if the missingness is not happening at random because in that case a missing value tells something about the value itself.

One way to handle missing data with decision trees is to use *probabilistic split*. The method is used for example in Quinlan's C4.5 algorithm [31]. In a probabilistic split the split rule is determined using normal heuristics but using only the instances which are not missing the value for the feature being tested as split rule. This is not CCA since an instance can be left out when testing one feature but used in another test depending on which values are missing from the instance. Once a split rule is found, the instances are split according to the rule. There are now instances which do not have the value for the split rule and cannot be split normally. Therefore in probabilistic split each instance is associated with a weight. The weight is the probability for the instance to belong in the group it is assigned. The weight is updated after every split by multiplying the old weight with the new probability. Instances with known value are assigned weight of 1 since we know for sure where the instance belongs. Instances that are missing the value for split rule cannot be assigned to either of the groups with certainty. This is why they are added to both groups. For these instances, the weight is the distribution of the instances defined by the instances for which the split can be made. The weight is the sum of weights in the parent divided by the sum of weights in the group.

Another well-known decision tree algorithm CART [40] (classification and regression tree) uses *surrogate splits*. In this method each node has several split rules. If the primary rule cannot be evaluated, a surrogate rule is used instead. If none of the rules cannot be used, the instance is forwarded to the node with the most instances. This is called the base rule. The best split is defined as normal using only the instances with required data. After the best split is found, the surrogate splits are defined by finding splits that produce the most similar splits as the primary split rule. The surrogate rules must produce a better match than just forwarding all instances to the larger node. If no such rules can be found, the base rule is used. [43, pp. 189–190]

2.5.3 Strengths and weaknesses of decision trees

Like every machine learning method, decision trees have their strengths and weaknesses. Kotsiantis [20] compared different classification methods and the results are collected to Table 2.2 as presented in his article with the exception that rule learners are excluded from the table. In the table there is a list of characteristics which are evaluated for each method in the table. In the table * means the worst performance and **** the best performance. The table shows how different classification methods can be and how important it can be to choose the right method for the problem. This requires being familiar with methods and knowing the problem and the data. For example, kNN does not train a model but instead it finds the nearest instance from the training data. This causes it to be extremely vulnerable to missing data and makes the classification slow.

Table 2.2 *Characteristics of different learning methods [20]. * represents the worst and **** the best performance.*

	Decision Trees	Neural Networks	Naive Bayes	kNN	SVM
Accuracy in general	**	***	*	**	****
Speed of learning with respect to number of attributes and the number of instances	***	*	****	****	*
Speed of classification	****	****	****	*	****
Tolerance to missing values	***	*	****	*	**
Tolerance to irrelevant features	***	*	**	**	****
Tolerance to redundant features	**	**	*	**	***
Tolerance to highly interdependent attributes	**	***	*	*	***
Dealing with discrete/binary/ continuous features	****	*** (not disc)	*** (not con)	*** (not directly disc)	** (not disc)
Tolerance to noise	**	**	***	*	**
Dealing with danger of overfitting	**	*	***	***	**
Attempts for incremental learning	**	***	****	****	**
Explanation ability/transparency of knowledge/classifications	****	*	****	**	*
Model parameter handling	***	*	****	***	*

Perhaps the most dominant feature of decision trees is their transparency which is a big factor in their popularity. The method uses simple yes-no or smaller than rules to classify an instance. These rules and their significance to the problem can be understood by a human and it is possible to go through the chain of reasoning behind a prediction. Also, as seen in Figure 2.3, it is possible to visualize a decision tree. Another significant strength of decision trees is the ability to handle diverse data. They can handle several labels associated to a problem and are good at handing mixed data. Both of these attributes are explained by the basic idea of the method. Decision trees also tolerate missing data quite well and several missing data treatments have been developed for decision trees as discussed in the previous subsection. Once the model has been trained, it is fast to classify an instance using the created tree.

Even though decision trees are successfully applied to many problems, they do have limitations. The most notable limitation might be their prediction power. While it might be adequate for some problems, there are methods which can reach better results in general such as neural networks and SVM (support vector machine). Decision tree is a high-variance method which means it can create an arbitrarily complex model of the data. This makes it susceptible to *overfitting*. The problem is visualized in Figure 2.5. The green line presents a larger decision tree meaning it has more nodes and hence more rules for splitting the data. However, although it represents the test data better than the blue line, it does not generalize the data as well as the blue line. The larger decision tree overfitted to the test data and learned individual instances (outliers) of the data instead of the sin curve. Therefore why smaller decision trees are often favoured over larger ones. Methods like pruning can be used to avoid overfitting. Another significant problem with decision trees lies with the data. They have a natural instability which can cause small changes in D_{train} to cause big changes in the formed model [22]. Also, although the data does not require much reprocessing, the data should be an even representation of the labels. Otherwise, the results might get biased in favor of the dominating labels of the training data.

Classifying instances is relatively fast with decision trees. The speed of the classification depends only on the constructed tree, and the time taken to classify an instance is the length of the path from root to a leaf the instance belongs to. However, there are significant differences in the time taken to construct a decision tree. Martin and Hirschberg [25] proved the time complexity for algorithms using top-down induction, which include for example C4.5, to be $O(m \cdot n^2)$ where n is the number of features and m is the size of D_{train}. The amount of available data has increased and as the used data sets become larger and larger, the time complexity becomes an issue and hence, faster methods are needed for constructing decision trees. However, the speed of construction alone is not enough and the performance of the new faster methods have to be approximately as good as the old methods'. Over the years, several methods have been developed to answer this problem. For example, Su and Zhang

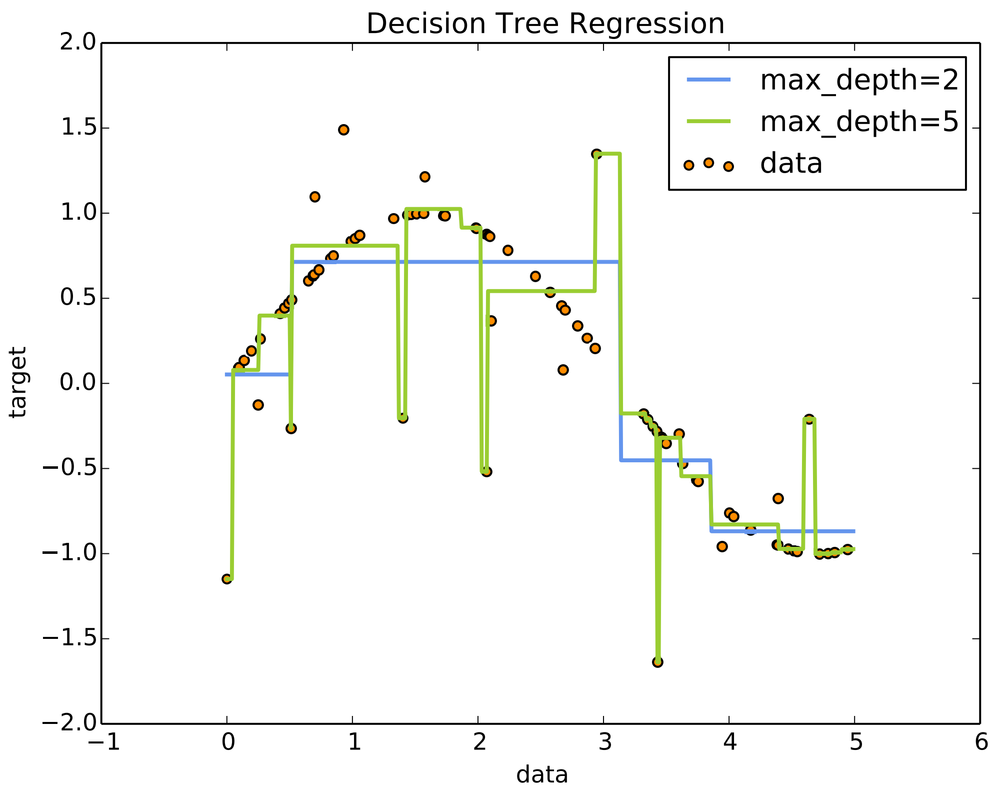


Figure 2.5 *Two decision trees trained on the same data [36]. The decision tree presented by the blue line is able to somewhat generalize the data but the green overfits to the training data.*

[41] have developed a method based on conditional independence assumption with time complexity of $O(m \cdot n)$. This is significantly faster than top-down induction can provide. They also reported a competitive accuracy with the C4.5 method.

2.6 Evaluation metrics for classifiers

Once a model is created, the next step is testing it. This often requires test data but more importantly a measure of goodness. What is considered good, however, depends on the problem itself. The goal might be to predict a class for an instance, in which case it is clear whether the classifier made the right prediction and it is simple to make calculations based on the number of instances classified correctly and incorrectly. This is not the case for regression problems since the prediction cannot be labelled as naively to be right or wrong. A completely different type of problem are classifiers recommending new products to a user. The correctness of an outcome is not so clear since the goodness depends individually on the users. The mathematically the performance of the system might be good but the users might still feel the system does not give them good recommendations [26].

This section uses Han et al. [15, pp. 364–371] as a reference. In this work we concentrate on the metrics used with classification methods. For simplicity, the metrics presented in this section are defined for cases when there are only two classes. However, the metrics can be simply generalized to several classes. This can be done for example by calculating a value for each class and taking the mean of the values.

Often the amount of data is limited while constructing a model. In addition to training the classifier, testing the classifier requires data. Using the same data for training and testing does not provide reliable results about how the model performs when it is given new data. There are several ways of using the data in a way that produces reliable evaluation results. Perhaps the simplest way is the *holdout* method where the data are split into train and test sets. Often around third of the data is reserved for testing. The problem with this is that the results depend on how the data are split. One method which does not have this problem is *k-fold cross-validation*. The data set is split into k separate subsets of approximately equal size $D = [D_1, D_2, \ldots, D_k]$. The evaluation is done k times. In each iteration subset D_i, where $1 \leq i \leq k$, is reserved for testing and the rest are used for training the classifier. The performance estimate is the average of the iterations.

Confusion matrix is a useful tool for understanding what happens inside the classifier. It does not give a value presenting the goodness of a classifier but instead it shows all classifications made by the classifier and it can be used for example, to see what kind of mistakes the classifier made. The matrix is a $n \times n$ matrix where n is the number of classes. Each row and column is labelled with one of the classes. A value $v_{j,i}$ on the matrix tells how many instances of class i are predicted to belong to class j. Therefore, the columns represent the predicted classes and the rows the actual classes. An example of a confusion matrix is in Figure 2.6. The data has two possible classes: class1 and class2. The values on the diagonal are the number of correct classifications so here 51 out of 68 instances are classified correctly. The matrix also shows that instances of class2 are distinguished well from class1 but not the other way around.

$$\begin{array}{r|cc} & class1 & class2 \\ class1 & 20 & 14 \\ class2 & 3 & 31 \end{array}$$

Figure 2.6 *A confusion matrix.*

Many evaluation metrics use key figures which are based on positive and negative class. *Positive class* is the main class of interest and *negative class* represents all other classes. There are four commonly used key figures. T_p is the number of instances of the positive class that were correctly classified and false positives F_p the amount of instances of positive class that were incorrectly classified. Figures true negatives T_n and false negatives F_n are similarly defined but for the negative class. These figures can easily be seen from the confusion matrix. Let *class1* be the positive class in Figure 2.6. The key figures are $T_p = 20$, $F_p = 14$, $T_n = 31$, and $F_n = 3$.

Perhaps the most intuitive way of measuring the goodness of a model is *accuracy* A. It is the ratio of right classifications and the number of instances as presented in Equation 2.1. For example, the accuracy for the model in Figure 2.6 is $51/68 = 0.75$. Accuracy is useful when the data is balanced and gives a starting point for the evaluation of the model. However, if the data is unbalanced it is favorable to predict only a single class. For example, a data set could consist of 80% persons who have cancer and 20% persons who do not have cancer. In this case the classifier might learn that it is beneficial to always predict that the person does not have cancer. This would give accuracy of 80% but the model has no predictive power. This is called the accuracy paradox.

Other basic metrics include precision, recall, and F1-score. Their formulas are presented in equation 2.1. *Precision* tells how many percent of the positive predictions are correct, and it can be thought as the exactness of the classifier. *Recall* on the other hand is the percentage of the positive class that is correctly classified. It can be thought as the completeness of the classifier. Recall and precision are combined in the *F1-score* which is the harmonic mean of the two metrics. Therefore, it can be thought of as a value that tries to find the balance between recall and precision and can be used for example when recall and precision are equally important. For the example in Figure 2.6 precision is $20/34 = 0.59$, recall is $20/23 = 0.87$, and F1-score is 0.70.

$$
\begin{aligned}
\text{accuracy } A &= \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \\
\text{precision } P &= \frac{T_p}{T_p + F_p} \\
\text{recall } R &= \frac{T_p}{T_p + F_n} \\
\text{F1-score } F1 &= 2 \cdot \frac{P \times R}{P + R}
\end{aligned}
\tag{2.1}
$$

As stated before, the problem defines how the performance should be evaluated.

This is true even if we are only concentrating on classification problems. For example, the distribution of classes in a data set affects the choice of the metric to use, it might be important that all made classifications are correct, or that all instances of a single class are caught. In addition to accuracy based metrics, there are several other aspects, like speed, robustness, scalability, and interpretability, that can be considered. For example, the speed of the classifier is crucial on real-time applications but for a medical application it could be important for the user to understand why a certain prediction was made.

2.7 Known hierarchical models

Hierarchies have been used a lot in machine learning and they have been applied to several existing machine learning methods. Usually the hierarchies are utilized either in the labels or in the features. If the labels are hierarchical, the possible values form a hierarchical structure which can be for example a graph or a DAG. An example of such label was shown in Figure 2.2. Using the graph, it is possible to predict a label from any level of the hierarchy. This can preserve the classifier from making an uncertain prediction. However, the labels on the higher levels are not as accurate and hence usually not as useful as the labels at the bottom of the hierarchy. It is established in several studies that label hierarchies improve the classification results [5].

In *hierarchical single-label* problems an instance has only a single label but the possible labels are hierarchical. Some developed methods use the possibility of not predicting a leaf, but some methods require that. An example of a method utilizing the ability to exchange accuracy with reliability is developed by Chen et al. [8]. They developed a method for constructing a decision tree using data with hierarchical class labels and their method proved to be superior to C4.5. In the literature the approaches for dealing with the hierarchy of the label are top-down (local) and one-shot (global). In local approach the hierarchy is processed level by level. On each level new classifiers are created using only the data in that level and the method produces a tree of classifiers. In comparison the global approach creates a classifier which handles the class hierarchy as a whole. [7]

The research of using hierarchical labels has been extended to having multiple labels. In normal *multi-label classification* an instance can have several labels associated with it. However, in *hierarchical multi-label classification (HMC)* an instance can be associated with several labels which belong to several paths on the label hierarchy. This is a more complex problem than predicting just a single flat class. Nevertheless, the problem has a lot of applications which is why a lot of research has been done on HMC. It is particularly common for classifying genes but is has been applied for example also in text classification [33].

Another approach to utilizing hierarchy in machine learning is to have hierarchical features. However, this seems to be a significantly less studied area. In this approach each feature can have its own hierarchies but the class label does not have to be hierarchical. For example, Han and Lam [17] have developed a framework which utilizes hierarchical features with good results. The hierarchies have been also used when the values are not equally accurate and the different accuracies are just levels on the hierarchy. Furthermore, Zhang and Honavar have developed a method that uses attribute value taxonomies to guide the decision tree construction [44].

3. DECISION TREE WITH HIERARCHICAL FEATURES

The goal of this work was to create a classification method capable to handle hierarchical and interval data while being able to handle missing data. This implies that the method has to be able to handle mixed and missing data. This is why we chose decision tree as the base method. It has naturally both qualities as discussed in Section 2.5.3. One major factor for choosing decision tree was that it is also relatively simple to implement from scratch and it does not require massive amounts of data to work.

One of the strengths of decision trees is that they can endure all kinds of data. However, there are still cases which decision trees are unable to handle. An example of such case is a single feature with several data types. This could occur when all values of a feature are not equally accurate. An example could be company's staff number. Some companies report the precise number of workers while others report only a range. How should this kind of situation be handled without losing information?

In this work the solution to the problem of several data types in a feature is are hierarchies. Perhaps somewhat surprisingly, the most common data types can be transformed to a similar hierarchical form and hence the values can be treated similarly. This makes it possible to have several different data types in a single feature. For example, real values and intervals can be combined easily to the same hierarchy since intervals consist of real values. Using hierarchies has also other benefits. They can contain information that is otherwise unattainable. Hierarchical data was discussed more in depth in Section 2.4.

Hierarchical data must be preprocessed before they can be used with decision trees. One known method for this is Quinlan-encoding which was explained in Section 2.4.1. However, this method can have serious drawbacks since the dimensionality of the data can grow large. This might make it very slow or even impossible to construct the model. It also requires that every value has the same number of levels in the hierarchy which is an artificial requirement. However, it is possible to use hierarchical data with decision trees in a way that gives control of the dimensionality to the user and even makes computing the model faster.

Most machine learning methods assume each instance to have only one value for a feature. However, in real life this often is not the case. For example, a company can be associated with several industry codes or a person can have several employers simultaneously. Of course, not all instances have several values for the same feature. One instance might have four values for a feature, another has only one and third is missing it completely. How should this kind of situations be handled? The developed method solves this by using a set of values instead of a list of values as input data.

The other goal of the work was to integrate handling missing data as part of the developed As outlined in Section 2.5.2, there are several existing methods for handling missing data with decision trees. The developed method uses a similar idea to what CART algorithm uses for handling missing data. The idea is that if the best split cannot be used to split the data, we should use the second best. The goal is to be able to handle missing data without making assumptions and to use all of the available data to classify the instances with missing data.

In this thesis we have modified the decision tree method to cope better with the problems discussed above. The method developed in this thesis is referred to as *decision tree with hierarchical features* (DTHF). It is a classification tree that uses data in a hierarchical form. The data do not have to be hierarchical by nature but the presentation must be. For example, numerical values have a natural hierarchy but nominal values necessarily do not. In that case a nominal feature can only be presented as hierarchical. The method assumes all input data are in a predefined hierarchical form. The process of transforming the data to the accepted form is described in Section 3.1. The algorithm for constructing the hierarchical decision tree is described in Section 3.2.

3.1 Transforming data

DTHF constructs a model using data that is in a specified hierarchical form. This is why all input data have to be preprocessed before constructing a model. The developed algorithm can handle an instance having several values for a single feature. This makes the mathematical notation for the data remarkably more complicated. Hence, in this chapter the used notation is for the case where single feature does not have multiple values. The case for handling multiple values is explained in Section 3.1.5.

The preprocessing of the data is done in two parts. First, each feature in the data is given an identifier id_i to distinguish it from other features. The identifiers are unique strings, for example a number, a letter, or a word describing the feature. The identifier is used to link a value to a feature. Secondly, each value $f_{i,j}$ in d_i is transformed to a hierarchical form $\bar{f}_{i,j} = (h^1_{i,j}, h^2_{i,j}, \dots, h^{n(i,j)}_{i,j})$. Here $h^m_{i,j}$ is the mth hierarchic level of ith data instance's jth value and $n(i,j)$ is the number of hierarchic levels on ith data instance's jth value. For the hierarchy levels it holds that for all $k > m$, $h^k_{i,j}$ is more specific than $h^m_{i,j}$. Value *(Animal, Mammal, Dog, Poodle)* is an example of such hierarchical value. The number of levels in a transformed feature depends on the value itself. This means that it is possible for two transformed values of the same feature have a different number of levels. Each data instance $d_i = [f_{i,1}, f_{i,2} \dots, f_{i,n_i}]$ is transformed to the following form:

$$\begin{aligned}\bar{d}_i &= [\bar{f}_{i,1}, \bar{f}_{i,2}, \dots, \bar{f}_{i,n_i}] \\ &= [(id_1, h^1_{i,1}, h^2_{i,1}, \dots, h^{n(i,1)}_{i,1}), (id_2, h^1_{i,2}, h^2_{i,2}, \dots, h^{n(i,2)}_{i,2}), \dots, \\ &\quad (id_{n_i}, h^1_{i,n_i}, h^2_{i,n_i}, \dots, h^{n(i,n_i)}_{i,n_i})]\end{aligned}$$

In addition to an identifier, each feature must be assigned a data type. The type defines how the value is transformed to a hierarchical form. Depending on the data type, some types require also additional parameters. These parameters are explained in the following subsections. The data types are defined using a list having a type for each feature in D_{train}. Each data type has a unique identifier which must be used when defining the data types for the features. DTHF accepts the following types: flat nominal (N), hierarchical nominal (HN), real valued (R), and interval valued (I). The value in parentheses after a type is the identifier for the data type. For instance $data\_types = [R, R, N, HN]$ would be a valid definition for feature types.

3.1.1 Flat nominal features

A flat nominal feature consists of discrete categorical values which can be symbols or names of things as specified in Section 2.1. In the developed method, flat nominal features are identified using *N*. This type does not assume an inner structure between the values and they are treated as completely separate. Values are thought as a hierarchical feature which has a single level consisting of the value itself. Hence, the transformation only presents the value in the right form and does not modify it. Nominal values are transformed to the form $\bar{f}_{i,j} = (id_i, value)$. For example, feature sex that has two possible values *female* and *male* would be transformed to *(sex, female)* and *(sex, male)*.

3.1.2 Hierarchical nominal features

Nominal features whose values have an inner hierarchical structure are hierarchical nominal features. Since the values are nominal, they are not quantitative and they cannot be ordered. Instead of an ordering, the values have a hierarchy where values can be generalized to higher concepts. Hierarchical values were defined in Section 2.4. The hierarchy might not be given with the data set in which case the user must define the hierarchy. The hierarchies must form a tree where the values have common parents. The identifier for a hierarchical nominal feature is *HN*. An example of a hierarchical nominal feature is presented in Figure 3.1. The feature in the figure is marital status, which has seven possible values. The original values (at leaves) are marked with grey color and the white nodes are the higher concepts that group the original values together.

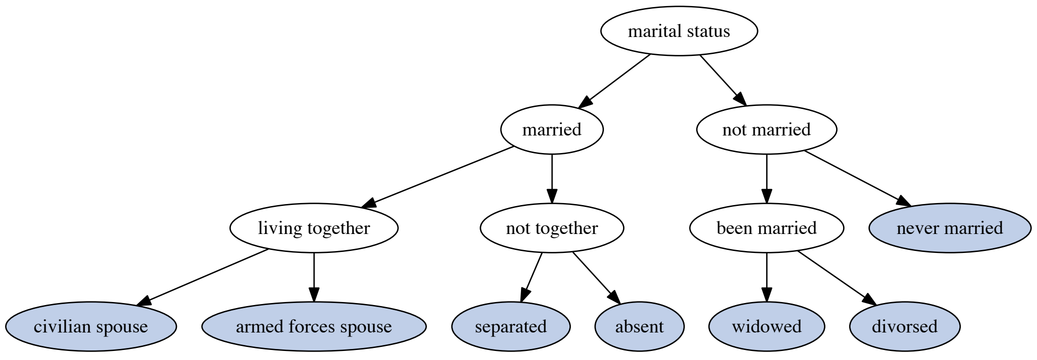


Figure 3.1 *An example of a hierarchical nominal feature. Feature in the figure is* `marital status` *and it has seven possible nominal values (colored nodes).*

The transformed value for a hierarchical nominal feature is the path from the root to the value being transformed. Hierarchical nominal features are transformed to $\bar{f}_{i,j} = (id_i, v_2, v_3, ..., v_k)$, where v_i is the ith value on the path and k is the length of the path. Root of the tree is left out from the transformation as it has no information. For example, value *widowed* would be transformed to *(marital, not married, been married, widowed).*

3.1.3 Numerical features

Numerical features have integer $\mathbb{Z}$ or real $\mathbb{R}$ values and they can be either interval-scaled or ratio-scaled. Those are defined in Section 2.1. The identifier for a numerical feature is R. Numerical values are transformed to hierarchical values using an interval binary tree. The transformed value is the path from root to the leaf including the value. The tree is constructed by first finding the minimum and maximum value of the feature in D_{train}. Those values form the interval in the root of the graph. New nodes are added by splitting each interval in half. The smaller half becomes the left child and the larger half becomes the right child. This is continued until the formed interval is smaller than the split limit. Figure 3.2 is an example of such tree. Here the range of values for the feature was between 7 and 43 and the used split limit was 10.

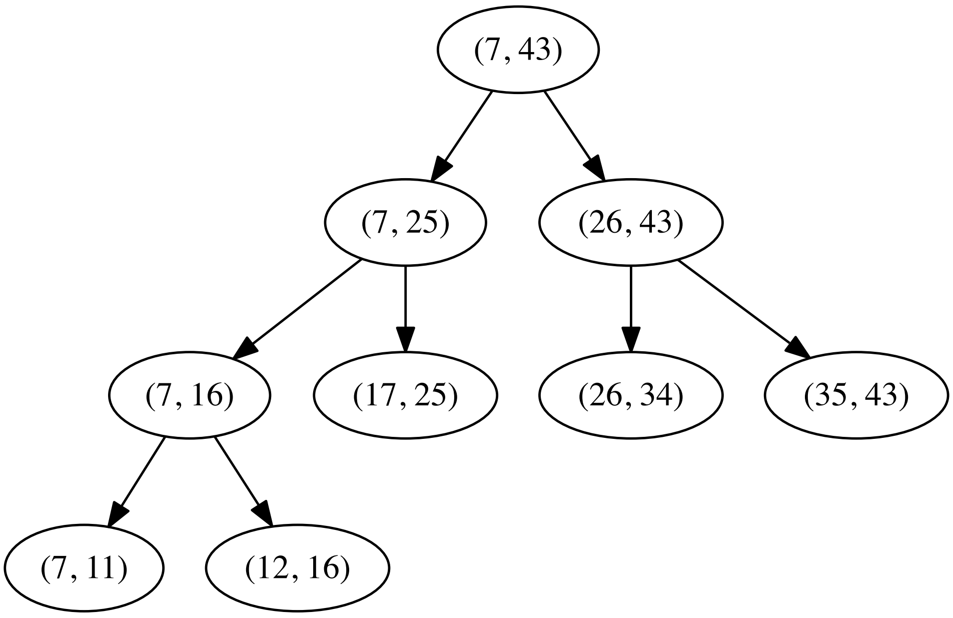


Figure 3.2 *A graph for transforming a numerical value to hierarchical. In the training data feature's values were integers between 7 and 43 and the used split limit was 10.*

Numerical feature f_i is transformed to $\bar{f}_{i,j} = (id_i, v_2, v_3, \ldots, v_k)$, where v_i is the ith node on the path and k is the length of the path. The transformed value consists of letters "L" (left) and "R" (right). The transformation is done by starting from the root of the graph and comparing the new value to the left child of the current node. If the value is included in the interval of the left child, it is made the current node and letter L is added to the transformed value. Otherwise the value must belong to the right child which means it is made the current node and letter R is added. This is continued until a leaf is hit. For example, using the tree in Figure 3.2, value 13 would be transformed to *(L, L, R)*. If the new value is greater than the maximum value in the tree, it is always directed to the right child. Accordingly, if the new value is smaller than the minimum of the tree, it is always directed left. For example, value 53 would be transformed to *(R, R)* and 3 to *(L, L, L)*.

In practice, the tree does not have to be completely constructed at any point of the transformation. Instead, on the training state the root interval of the graph, i.e. the minimum and maximum values of D_{train}, are saved. When a value is transformed, it is compared to the middle point of the interval and assigned L if it is smaller and R if it is bigger. This is continued until the interval is smaller than the split limit. The user should define a split limit separately for each numerical feature. A default value can be used but the sensible value depends on the feature. For example, petal width and company's revenue are in completely different orders of magnitude. The split limit controls the number of levels in a transformed value. The smaller the limit is, the longer the transformed hierarchies are.

3.1.4 Interval features

Features that can have both numerical and interval values are called interval features which are identified using I. Interval features can consists completely of intervals, completely of numerical values, or both. However, if the instances in D_{train} have only numerical values for a interval feature, the transformation is identical to the transformation done to a numerical feature. The transformation of features with interval values is done using an *interval DAG* which is a DAG consisting of intervals. The intervals are arranged such that an interval on a parent node includes all of the intervals on its children. A node can have an arbitrary number of children. Since DAGs are used instead of directed trees, two nodes can have several paths between them. Therefore, a single value can have multiple transformations. The paths in the graph are indicated by assigning each node a unique identifier. An example of an interval DAG is given is Figure 3.3 where each node has two values: the interval the node represents and the integer value which is the identifier of the node.

The transformation of value $f_{i,j}$ is the path from root to interval r which is the shortest interval for which $f_{i,j} \in r$. The transformed value is $\bar{f}_{i,j} = (id_i, v_2, v_3, \ldots, v_k)$ where v_i is the identifier of the ith node on the path and k is the length of the path. Root node is not included in the value since it is on all paths. For example, in Figure 3.3 the shortest interval including value $(20, 25)$ is $(18, 26)$ so the transformed value is $\bar{f}_{i,j} = (id_i, 0, 1, 9)$. It is important to note that the path does not have to end to a leaf. This is the case for interval $(16, 21)$ for which the transformed value is $\bar{f}_{i,j} = (id_i, 0, 1)$. In a case where several paths can be formed for a single value, a transformed value is added for each path. Interval $(27, 28)$ is an example of such situation because the smallest node including it is $(27, 29)$. There are two paths leading to it, meaning there are two transformed values: $\bar{f}_{i,j} = (id_i, 0, 3, 10)$ and $\bar{f}_{i,j} = (id_i, 0, 1, 10)$.

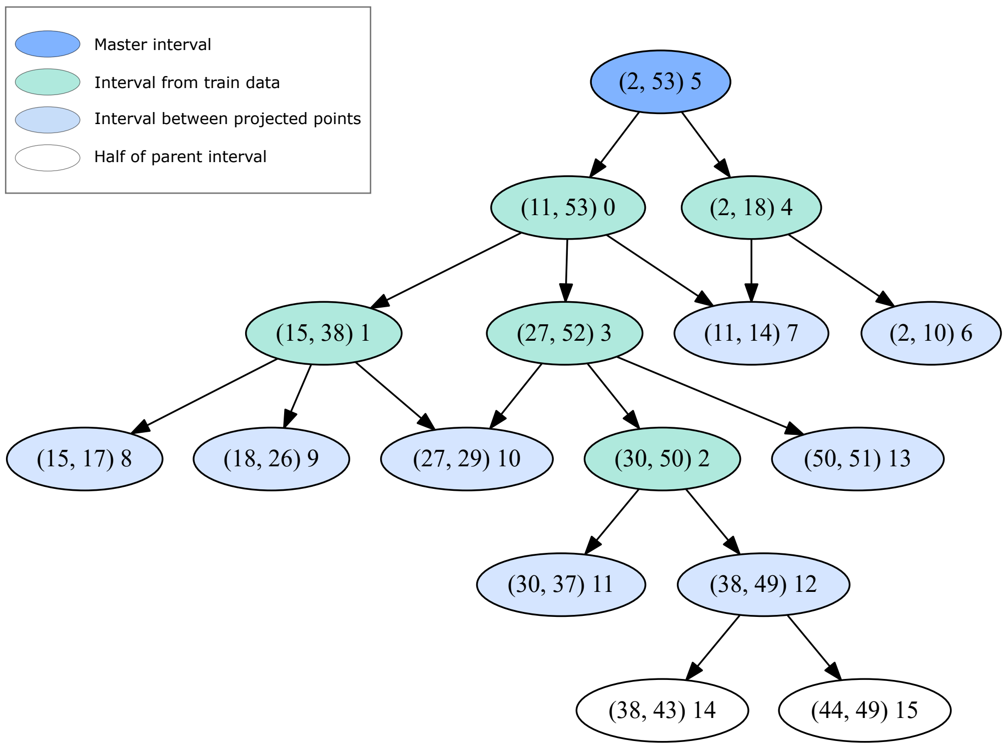


Figure 3.3 *An example of an interval DAG that is used for transforming interval features into a hierarchical form. The first part on a node is the interval and the second value the identifier of the node. The colors indicate the state the node was added to the graph.*

The interval graphs are constructed in four stages and each stage adds nodes to the graph using different rules. Many of the stages add nodes to the graph depending on the values the interval feature has in D_{train}, this set is denoted by $V_{f_j} = \{f_{i,j} \mid \forall i : d_i \in D_{train}\}$. The graph in Figure 3.3 is constructed using $V_{f_j} = \{(11, 53), (2, 18), (30, 50), (15, 38), (27, 52), 26, 3\}$. The stages for constructing a graph used for transforming interval features are described below.

1. *Root of the graph*: Constructing the graph is started from the root of the graph. The root has to include all values of the feature and therefore the root interval consists of the minimum and maximum value of the feature, $root = (\min(V_{f_j}), \max(V_{f_j}))$. This is called the *master interval*. In the figure the master interval is $(2, 53)$.

2. *Add intervals in V_{f_j}*: Each interval in V_{f_j} is added to the graph. The assumption is that the intervals in the training data also occur in the data to be

classified. In figure the added intervals are $(11, 53)$, $(2, 18)$, $(30, 50)$, $(15, 38)$ and $(27, 52)$.

3. *Fill the gaps between the intervals in the graph*: At this point the graph consists of the master interval and the intervals in the training data. It is possible for a value to fall between intervals added in the second stage i.e. the value would belong only to the root interval. Another possible problem is that a value belongs to several nodes. Therefore, the range of values defined by master interval is divided so that the leaf nodes are separate but cover the whole master interval. This is done by creating a set of the start and end points of each interval in V_{f_j} and master node. The list is sorted and a interval is added to the graph between every two consequent points if no such interval already exists. For example, in the figure the two smallest values are 2 and 11. There is no interval between those values so an interval (2, 10) is added to the graph.

4. *Split leaves:* The leaves of the graph are now completely separate of each other, meaning they do not overlap with each other. The length of the intervals in the leaves can differ greatly. At the last stage the leaf nodes are split further until the length of a produced interval is smaller than a threshold called *split limit*. A value for split limit has to be defined for each interval feature. In the figure the split limit is 10. After the third stage, the graph has only one interval whose length is greater than 10. Interval $(38, 49)$ is split in half from the middle and the nodes are added as its children. Both of the created nodes have length less than 10 so they are not split.

The process of adding intervals to the graph is illustrated in Figure 3.4. In the figure each horizontal line represents an interval in the graph. The red line (A, H) is the master interval. Intervals from the training data are drawn using a solid black line: $(B, D), (C, E)$, and (F, G). The projected start and end points of intervals are marked with letters $A - H$ on the master interval. Using the marked points, we add an interval between every pair of consequent points where an interval does not already exist. These intervals are drawn with dashed line: $(A, B), (B, C), (C, D), (D, E), (E, F)$, and (G, H). Note that interval (F, G) was in the original data set so it was not added on third stage. The fourth stage is not shown in the figure, since it would just split the existing intervals into smaller pieces.

3.1.5 Handling features with multiple values

In various cases, a single data instance may have several different values defined for a single feature. For example, a company may be associated with several different

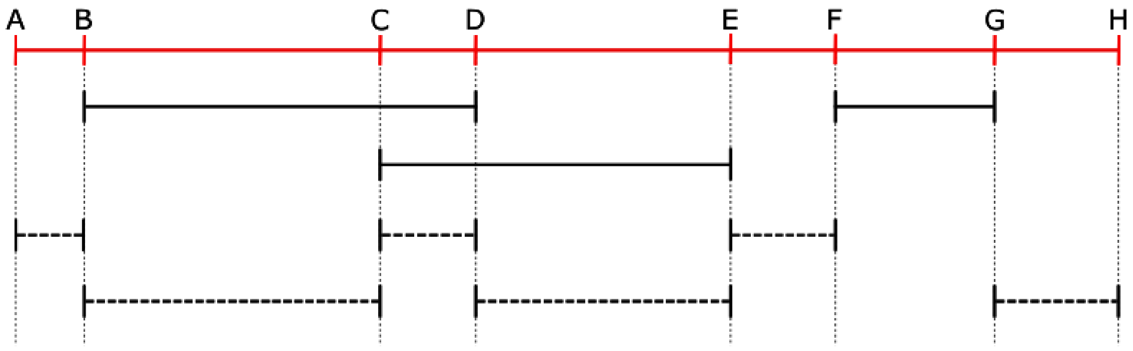


Figure 3.4 *Forming the intervals based on the training data for the graph used for transforming interval features.*

industry codes. However, not all instances necessarily have multiple values for a feature and some instances might not have the data for that feature at all. In this case we cannot assume the input data to be in a form where every instance is similar. Instead, the data can be presented as a set of values where each value is associated with a feature. For example, letting f_i denote the industry code feature and d_c denote a data instance representing company c with industry codes 4624 and 4623, the data instance d_c is defined by the list $d_c = [(f_i, 4624), (f_i, 4623)]$.

Generally, in order to handle multiple values for a single feature, a data instance d_i is redefined as a list of feature-value pairs $d_i = [(f_{i(1)}, v_{i,1}), (f_{i(2)}, v_{i,2}), \ldots, (f_{i(n(i))}, v_{i,n(i)})]$ where $f_{i(k)}$ denotes a feature with index $i(k)$, $v_{i,k}$ denotes the corresponding feature value and $n(i)$ denotes the amount of values in d_i. Note that feature-value pairs $(f_{i(k)}, v_{i,k})$ and $(f_{i(l)}, v_{i,l})$, where $k \neq l$ and $i(k) = i(l)$, represent two values $v_{i,k}$ and $v_{i,l}$ of the same feature $f_{i(k)} = f_{i(l)}$. The feature-value pairs are transformed into hierarchical form similarly as in the case with only one value per feature.

3.2 DTHF Algorithm

Many machine learning algorithms handle data instances d_i as described in the beginning of Chapter 2. The approach has its strengths, for example it does not necessarily require preprocessing and it is easy to understand, but it also has disadvantages which can create limitations for the data. For instance, there can be only one value for each feature, some data types cannot be used, and presenting hierarchical data can be problematic as discussed in Section 2.4. The machine learning method described in this thesis is a modified version of the classification tree described in Section 2.5. The modifications aim to construct a decision tree which addresses the previously discussed problems. The developed method has four main differences compared to the standard decision tree method:

1. Input data is transformed into a hierarchical form.
2. Split rules are hierarchic levels $h_{i,j}^k$ instead of complete feature values.
3. Data instances can have different numbers of values for a feature.
4. Missing data creates a new branch where the missing feature cannot be used to make a split.

Usually decision trees handle data instances d_i as a list of features. Each d_i has a similar form, meaning they have the same features in the same order. This convention is not used with DTHF. Instead, the input is a set of hierarchical values. The form for the hierarchical values was introduced in Section 3.1. Since the input is a set of values, an identifier is used instead of a position in the input to link a value to a feature. This modification makes it possible for an instance to have several values for a feature. However, this is not mandatory and each instance can have an arbitrary number of values associated with them.

Hierarchical data is especially useful with decision trees since the hierarchies can be used to make splits. Instead of the whole feature, in DTHF hierarchical levels are used as split rules. The hierarchy levels must be used from general to specific, for example hierarchy level *Poodle* cannot be used before *Dog* is used. This allows to first make a rough division of the data and then fine tune deeper in the tree if necessary. In a standard decision tree, each available value is tested as a split rule. With DTHF the possibility of values to test is smaller because the hierarchies group values together. Therefore, constructing a decision tree is faster.

Handling missing data is integrated in DTHF. If there are instances for which the outcome of split rule cannot be defined, a third node is created where the missing feature cannot be used. So, if there are missing data, the method uses the second best split rule in the third branch. This is similar, but not the same, approach as CART algorithm uses. This causes the missing data to be handled accurately on different hierarchy levels, which is a major advantage.

Figure 3.5 visualizes a tree constructed using DTHF. The tree is trained using only complete data instances and it cannot handle missing data. However, it would have been possible to construct a tree which could handle also missing data using the same D_{train} as for the tree given in the figure (see Section 3.2.2). Classifying instances is done identically as with a standard decision tree. However, Figures 3.5 and 2.3 do not present information similarly. Previously, each decision node displayed the split rule that was used to split the data in that node. Here each node has the split rules that have been followed to get to that node from the root. Therefore, each instance in a decision node fulfils the split rules listed on that node.

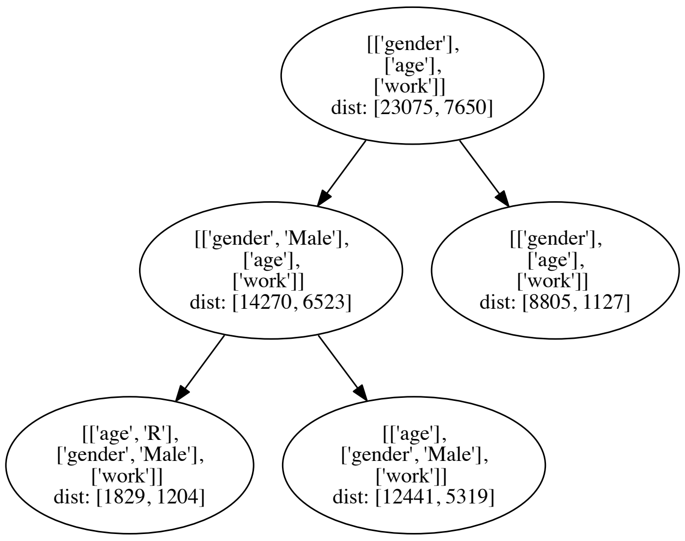

[['gender'], ['age'], ['work']] dist: [23075, 7650]
[['gender', 'Male'], ['age'], ['work']] dist: [14270, 6523]
[['gender'], ['age'], ['work']] dist: [8805, 1127]
[['age', 'R'], ['gender', 'Male'], ['work']] dist: [1829, 1204]
[['age'], ['gender', 'Male'], ['work']] dist: [12441, 5319]

Figure 3.5 *An example of a decision tree constructed by DTHF. The tree was constructed using data without missing values.*

Each node has two lists: $\mathcal{X}_p$ and *dist*. The first list on a node is $\mathcal{X}_p$ and it is used for keeping track of the used split rules. The split rules are sorted by the features they belong to and therefore $\mathcal{X}_p$ has a list for each feature in D_{train}. Once a hierarchy level is used as a split rule, it is added to the list of the corresponding feature. The newest split rule is always the last value of first element of $\mathcal{X}_p$ (see Figure 3.6). Initially $\mathcal{X}_p$ consists of the identifiers of all possible features in D_{train} so $\mathcal{X}_p = \{id_i \mid f_i \in D_{train}\}$.

For example, in Figure 3.5 initially $\mathcal{X}_p = [[gender], [age], [work]]$. Figure 3.6 gives an example of how $\mathcal{X}_p$'s value changes as new split rules are added.

$$\mathcal{X}_p = [[\textit{Work, Is working}], [\textit{Hometown, Africa}], [\textit{Gender}]]$$
$$\Downarrow$$
$$\text{New split rule; feature: } \textit{Hometown}\text{, value: } \textit{Nigeria}$$
$$\Downarrow$$
$$\mathcal{X}_p = [[\textit{Hometown, Africa, Nigeria}], [\textit{Work, Is working}], [\textit{Gender}]]$$
$$\Downarrow$$
$$\text{New split rule; feature: } \textit{Gender}\text{, value: } \textit{Female}$$
$$\Downarrow$$
$$\mathcal{X}_p = [[\textit{Gender, Female}], [\textit{Hometown, Africa, Nigeria}], [\textit{Work, Is working}]]$$

Figure 3.6 *An example illustrating how adding new split rules affects $\mathcal{X}_p$.*

The second list *dist* is the class distribution of instances in that node. The root node contains all instances of D_{train}. When there are no missing data, the number of instances is the same on every level of the tree. In Figure 3.5, there are 30,725 instances in total, and 23,075 belong to the first class and 7,650 to the second class. The first split divides the data so that there are 20,793 instances on the left child and 9,932 instances on the right child.

3.2.1 DTHF with complete data

The pseudo-code for constructing the tree introduced above is presented in Algorithm 1. In this version of the algorithm only complete instances of the train data are used to construct a tree and it can be used to classify instances with no missing data. The main idea of the algorithm is similar to constructing a standard decision tree. The algorithm starts by checking whether the current node is a leaf node using stopping condition. If it is a leaf node, no further processing is done for that node. The split rules on the path from the root to the current node are stored in $\mathcal{X}_p$. Initially the list has a list containing an identifier for each feature in the data $\mathcal{X}_p = [[id_1], [id_2], \ldots, [id_n]]$. Firstly, the algorithm identifies the possible split rules $\mathcal{X}_c$ which are the next unused hierarchical levels on each value. Once the possible split rules are found, each rule is tested using a heuristic for determining the best split rule. These heuristics were discussed briefly in Section 2.5. Once an optimal rule is found, the data is split according to the rule and new nodes are created by making a recursive call to the `grow` function for each child.

An instance can have several values for a feature. In such a case it may be ambiguous whether or not an instance fulfils the split rule since an instance can have a value matching and differing the split rule. In these cases, the instance is always thought to fulfil the split rule. Hence, it is sorted to the left child.

Algorithm 1: Tree growing recursion with hierarchical attributes, `grow`$(\mathcal{X}_p, S)$.

```
while stopping condition do
    Define candidate features:
      $\mathcal{X}_c \leftarrow \{h^k_{i,j} \mid d_i \in S \wedge h^k_{i,j} \in d_i \wedge h^{k-1}_{i,j} \in \mathcal{X}_p[id_j] \wedge h^k_{i,j} \notin \mathcal{X}_p[id_j]\}$
    Find optimal splitting feature $h^k_{i,j}$ from $\mathcal{X}_c$
    Split data according to split rule.
        $S_o = \{\bar{d}_i \mid \bar{d}_i$ fulfils split rule$\}$
        $S_d = S \setminus S_o$
    Continue recursion:
        Optimal:
            $\mathcal{X}_o = \mathcal{X}_p$
            add $h^k_{i,j}$ to list corresponding to $id_j$ in $\mathcal{X}_o$ and raise that list as the
             first element of $\mathcal{X}_o$.
            grow($S_o$, $\mathcal{X}_o$)
        Different:
            grow($S_d$, $\mathcal{X}_p$)
end
```

An example of constructing a hierarchical decision tree is provided using UCI's adult data set [19]. The goal of the data set is to predict whether a person's yearly income exceeds $50,000. Here the data set itself is not significant and the details are ignored. For demonstration purposes we take three different types of features from the data set: `age`, `work class`, and `gender`. Gender is a nominal feature which has values *Male* and *Female*, age is a numerical feature ranging between 17 and 90, and *work class* is a hierarchical nominal feature for which the hierarchy is presented in Figure 3.7. Algorithm 1 handles the case when no data are missing. Therefore only complete instances from the adult data set are used for this example.

The top of the hierarchical decision tree is visualized in Figure 3.8. Bottom of the constructed tree is left out because it is constructed in a similar manner as the top of the tree. The tree was constructed using 30,725 instances and there are two possible classes for each instance: over or under $50,000 per year. Constructing the tree starts by creating the root node. Because no splits have been made, it has all of the data instances. In the definition of the algorithm it is stated that initially $\mathcal{X}_p$ consists of the first levels of the hierarchies. These values are the identifiers for the features. In the constructed tree the features are `gender`, `age`, and `work`. The `grow` function described Algorithm 1 is called for this node. The possible split rules for the node are the values on the first level of the features: *male*, *female*, *has worked*,

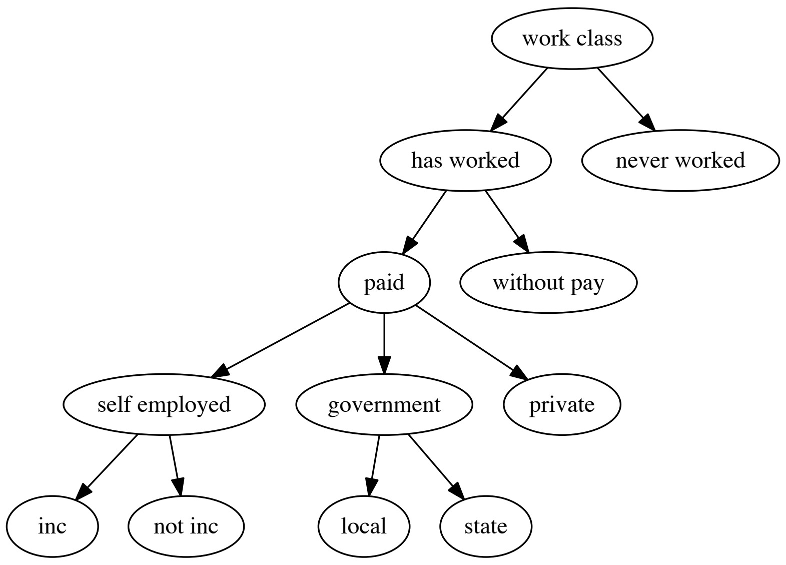


Figure 3.7 *The hierarchy for feature* `work class`*. This tree is used to transform a value of the feature to a hierarchical form.*

never worked, *L*, and *R*. Note that the features that are transformed using a tree do not use the root as a split limit since it is same for all the values and using it would not split the data.

On the root node, the best split was determined to be `gender` feature's value *Male*. This can be seen from root's left child where the feature on the top of the node and the last value was chosen as the split rule on the parent. Root's left child now includes all the instances for which the feature `gender` has value *Male*. The right child has all the instances for which the value of `gender` is something else, in this case *Female*. Constructing the tree continues by finding split rules for the new nodes. For the root's left child, the possible split rules are *has worked*, *never worked*, *L*, and *R*. From these, the best split rule was feature `age` and value *R* which means the person is older than 54. For the right child of the root, possible split rules are same as they were for the root. However, all the instances have now value *Female*, so the `gender` feature will not be chosen as split rule. Instead `work` feature's value *has worked* gets chosen. Again, all the instances which have something else as value of the feature `work` go to the right child. This node has only two instances and they both belong to the same class so the node is defined to be a leaf node and it is not split further. The rest of the tree is grown in same manner.

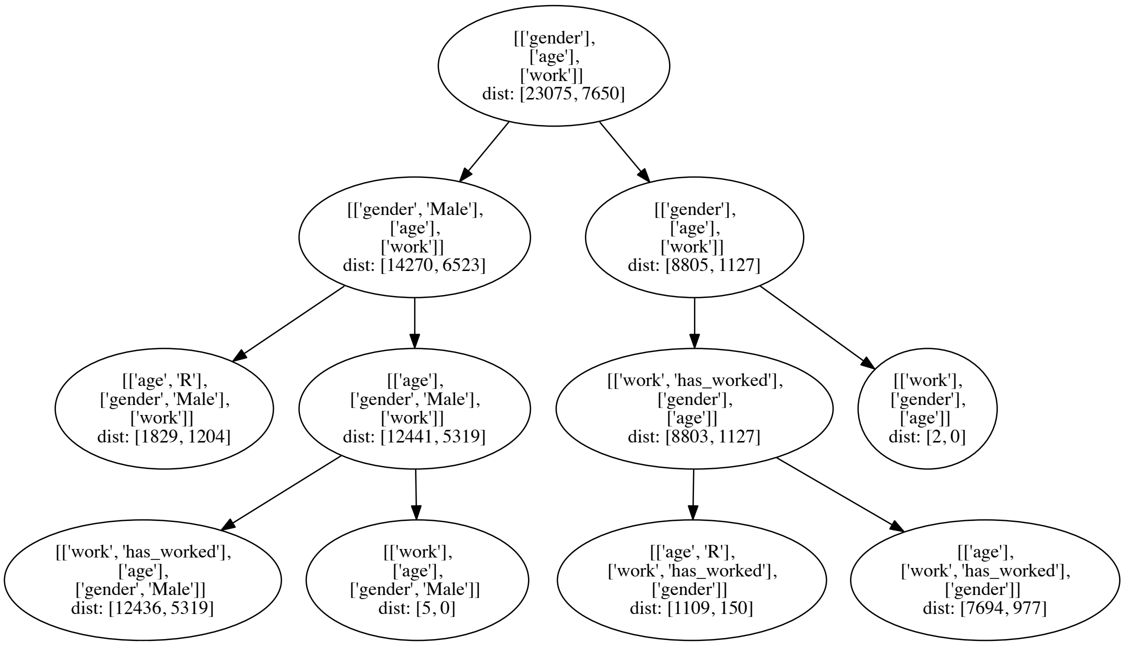

[['gender'], ['age'], ['work']] dist: [23075, 7650]
[['gender', 'Male'], ['age'], ['work']] dist: [14270, 6523]
[['gender'], ['age'], ['work']] dist: [8805, 1127]
[['age', 'R'], ['gender', 'Male'], ['work']] dist: [1829, 1204]
[['age'], ['gender', 'Male'], ['work']] dist: [12441, 5319]
[['work', 'has_worked'], ['gender'], ['age']] dist: [8803, 1127]
[['work'], ['gender'], ['age']] dist: [2, 0]
[['work', 'has_worked'], ['age'], ['gender', 'Male']] dist: [12436, 5319]
[['work'], ['age'], ['gender', 'Male']] dist: [5, 0]
[['age', 'R'], ['work', 'has_worked'], ['gender']] dist: [1109, 150]
[['age'], ['work', 'has_worked'], ['gender']] dist: [7694, 977]

Figure 3.8 *An example of a hierarhichical decision tree*

3.2.2 DTHF with missing data

As previously stated, DTHF is capable of handling missing data. Figure 3.9 illustrates a case when there are missing values in D_{train}. The only difference to the case when data are not missing is that nodes can have three children. The two first children are similar as previously explained. The data instances for which the split rule is true are passed to the left child and instances for which it is false are passed to the right/middle child. If there are instances for which the truth value cannot be determined, i.e. the data is missing, a third child is added. All the parent's data is copied to the third node and the feature used in the parent's split rule can no longer be used. This is indicated by adding value "END" to $\mathcal{X}_p$. For example, instance $\bar{d}_i$ = [['gender', 'Male'],['age', L, R], ['work']] would be classified to the rightmost leaf of the tree in the figure.

The pseudo-code for the case when data are missing is presented in Algorithm 2.The pseudo-code is very similar with the pseudo-code of Algorithm 1. The only exception is adding the node for missing data. Figure 3.9 gives an example of a hierarchical decision tree when data are missing. Only the top of a tree is presented but the rest of the tree is constructed in a similar way. On the root the split rule was defined to be `gender` feature's value *Female*. In the data there exists 876 instances which are missing the `gender` feature so a third node is created for instances missing the feature. The second node is similarly as with complete data. For the missing data node `work` feature's value *has_worked* is chosen as split rule. There are 1,375

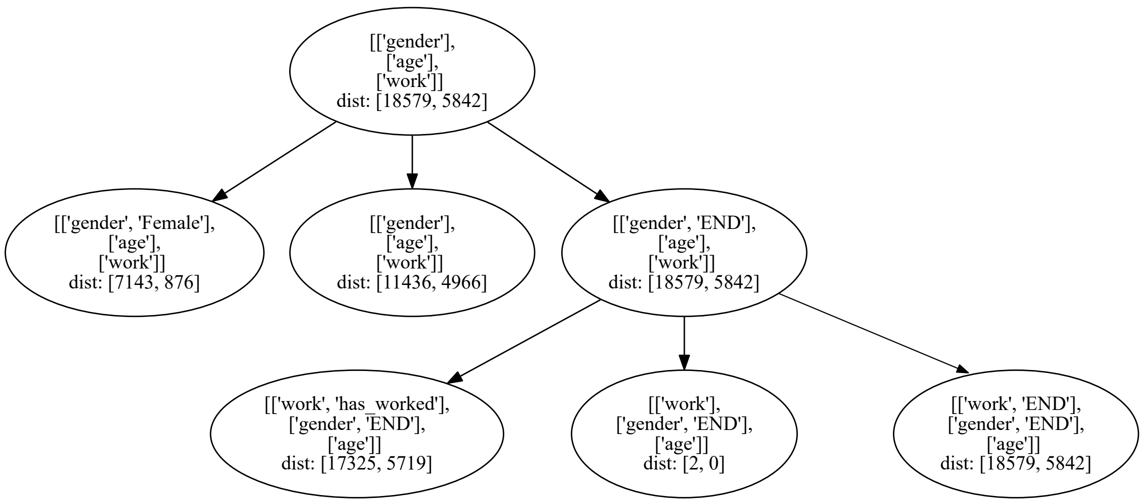

[['gender'], ['age'], ['work']] dist: [18579, 5842]
[['gender', 'Female'], ['age'], ['work']] dist: [7143, 876]
[['gender'], ['age'], ['work']] dist: [11436, 4966]
[['gender', 'END'], ['age'], ['work']] dist: [18579, 5842]
[['work', 'has_worked'], ['gender', 'END'], ['age']] dist: [17325, 5719]
[['work'], ['gender', 'END'], ['age']] dist: [2, 0]
[['work', 'END'], ['gender', 'END'], ['age']] dist: [18579, 5842]

Figure 3.9 *An example of a decision tree constructed by DTHF when there are missing data.*

instances which do not have a value for that feature. The two first nodes are created normally, but in addition a third node is added. This node consists of all 24,421 instances of the parent. Because the parent node's split rule used feature `work`, it cannot be used on the child. This is marked by adding 'END' at the end of that feature's list in $\mathcal{X}_p$.

It is worth noting that, unlike on the complete case, the number of instances in the children of a node is not necessarily the same as the number of instances on the parent. This is caused by the third node. The two first nodes cover the instances for which the value for the split rule can be determined, meaning they do not cover the instances with missing values. Instead of just the remaining instances for which the value cannot be determined, the third node has all the parent's data. So, the complete cases are there twice but the instances with missing value are there only once. Note that the proposed method for handling missing data is similar to the reduced-models approach which in some cases outperforms other missing data handling methods with a large margin [34].

Missing data handling example

In the following we compare Algorithm 2 to five common methods for handling missing data: *probabilistic split*, *CCA*, *reduced model*, *mode imputation*, and *missing data as value*. These methods were described in Sections 2.2 and 2.5.2. The example data are presented in the first two columns of Table 3.1. There are eight values in the example data set. Each value consists of a feature with two hierarchy levels and a class label A, B, or C. Missing data are marked using "?". Each method is first

Algorithm 2: Tree growing recursion with hierarchical attributes, `grow`($\mathcal{X}_p, D$).

```
while stopping condition do
    Define candidate features:
        $\mathcal{X}_c \leftarrow \{h^k_{i,j} \mid d_i \in S \wedge h^k_{i,j} \in d_i \wedge h^{k-1}_{i,j} \in \mathcal{X}_p[id_j] \wedge h^k_{i,j} \notin \mathcal{X}_p[id_j]\}$
    Find optimal splitting feature $h^k_{i,j}$ from $\mathcal{X}_c$
    Split data according to split rule:
        $S_o = \{\bar{d}_i \mid \bar{d}_i$ fulfils split rule$\}$
        $S_d = S \setminus S_o$
    Continue recursion:
        Optimal:
            $\mathcal{X}_o = \mathcal{X}_p$
            add $h^k_{i,j}$ to list corresponding to $id_j$ in $\mathcal{X}_o$ and raise that list as the
              first element of $\mathcal{X}_o$.
            grow($S_o$, $\mathcal{X}_o$)
        Different:
            grow($S_d$, $\mathcal{X}_p$)
        Missing:
            $\mathcal{X}_m = \mathcal{X}_p$
            add "END" to list corresponding to $id_j$ in $\mathcal{X}_m$ and raise that list as
              the first element of $\mathcal{X}_m$.
            grow($S$, $\mathcal{X}_m$)
end
```

used to train the decision tree and then each tree is tested with the training data. The results of the decision tree classification for the input data is shown in Table 3.1.

Table 3.1 *An example of data where DTHF performs well.*

Hierarchical feature	Class	probabilistic split	CCA	Reduced model	Mode imputation	Missing data as a value	DTHF
1, ?	A	A	-	A	A	A	A
1, ?	A	A	-	A	A	A	A
1, ?	A	A	-	A	A	A	A
1, 4	C	A	C	A	A	C	C
1, 4	C	A	C	A	A	C	C
2, 3	B	B	B	C	B	B	B
2, 4	C	C	C	C	C	C	C
2, 4	C	C	C	C	C	C	C
Classification score (out of 8)		6	5	5	6	8	8

As can be seen from the Table 3.1, most of the mistakes made were caused by the similarity between instances missing data and the instances $(1, 4)$ with label C. CCA is not fooled by the similarity since it does not care about the instances with missing values and the method which uses missing data as a value is able to predict all instances correctly since the instances with missingness have all class A. Mode imputation and probabilistic split however are fooled by the similarity and classify the 4-5th instances as A instead of C. There is not enough data with class B for the reduced model and therefore it is not capable of predicting class B at all. DTHF can predict all the instances correctly and there are situations where DTHF is better than the comparison methods.

4. EMPIRICAL RESULTS AND ANALYSIS

In this chapter we present the results from testing DTHF. The tests were divided into two parts: testing the method using data with no missing values and using data with varying rates of missingness. The results were compared to CART and C4.5 decision tree algorithms which both can handle missing data, and nominal and numerical features. In Section 4.1 we describe the data sets used for the tests. The performance of the algorithm is discussed in Section 4.2 which is divided into testing without missing data and with missing data. Results obtained from running the algorithms on complete data sets are presented in Section 4.2.1 and in Section 4.2.2 the algorithms are tested with incomplete data sets.

4.1 Description of the used datasets

The tests were made using 12 publicly available data sets from UCI Machine Learning Repository [10]. The sets were to selected to have somewhat different qualities by choosing sets with different types of features and varying number of instances. Table 4.1 summarizes the used data sets and tries to give the reader a better idea of how the data sets differ from each other. From each data are presented set name, the number of instances, the number of complete instances, the number of features in the data set, the number of classes in the data set, and the types of the features.

None of the data sets are especially large or have a large number of features. However, there are a few data sets with thousands of instances and there is some variation between the number of potential classes. The class distribution of the data sets is not presented in the table but it varies between data sets and some of the data sets like segment are perfectly balanced while others are not. In addition to the size of the data sets, none of the data sets have any hierarchical features and hence, only the numerical hierarchies are tested.

Table 4.1 *The data sets used in testing. The table lists some of their qualities to illustrate the differences between the data sets.*

data set	instances	complete	features	classes	data types
autos	205	159	26	7	nominal and numerical
balance	625	625	4	3	nominal
breast-cancer	699	683	10	6	nominal
german	1000	1000	20	2	nominal and numerical
glass	214	214	10	7	numerical
hepatitis	155	80	20	2	nominal
iris	150	150	4	3	numerical
letter	20000	20000	16	26	numerical
mushroom	8124	5644	22	3	nominal
segment	2310	210	19	7	numerical
sonar	208	208	60	2	numerical
soybean	307	266	35	19	nominal

4.2 Performance

The performance of DTHF was evaluated by running tests using the previously presented data sets. The results were compared to CART and C4.5 decision tree algorithms by running the same tests with same parameters to all the algorithms. These two methods were chosen as the comparison methods because they are both well-known decision tree algorithms, they can naturally handle both numerical and nominal data, and they have an integrated way of handling missing data as discussed in Section 2.5.2.

The tests were run on each data set using each algorithm and all combinations were 10-fold cross-validated. All three algorithms were run without pruning and the required minimum number for instances in a node for a split was 20 for all tests. The results are reported using F1-score, since there is no preference over precision or recall and some other performance metric could have been used as well. However, accuracy was not used because of the problems that unbalanced label distribution might cause as discussed in Section 2.6. For complete data we reported also the standard deviation of the cross-validation to give the reader an idea of the stability of the method.

The DTHF method was implemented using Python programming language. Information gain, which was discussed in Subection 2.5.1, was used as the split metric. The results were calculated using `sklearn.metrics` library's `f1_score` function because it supports calculating the score also for cases when there are more than two possible labels. The standard deviation was calculated using `numpy` package's `std` function.

Both CART and C4.5 algorithms were run on R programming language. CART was tested using `rpart` library. The library does not prune the tree on default but it was given parameter `xval`=1 to prevent it from doing cross-validation during model construction. C4.5 algorithm was tested using `RWeka` library's J48-method. The method was given parameter `U`=TRUE to prevent it from pruning the tree. Only the predictions were done using R, and the numerical results were calculated using the same Python-functions as with DTHF.

4.2.1 Performance with complete data

The data used for the tests was conducted by taking only the complete instances from each data set. The results of the tests are presented in Table 4.2 which consists of 10-fold cross-validated F1-scores and standard mean deviations of the cross-validations. The best F1-score for each data set is highlighted using green and the worst using red color. The standard deviation of the F1-score demonstrates the stability of the model although, this is not the most informative measure but it was simple to calculate and gives the reader some insight to the matter.

Table 4.2 *10-fold cross validated F1-scores and standard deviations of unpruned DTHF, CART, and C4.5 algorithms using split limit 20 on various data sets. Green color indicates the best performance and red the worst.*

data set	DTHF	CART	C4.5
autos	0.644 ± 0.146	0.728 ± 0.087	0.577 ± 0.179
balance	0.761 ± 0.044	0.796 ± 0.057	0.741 ± 0.070
breast-cancer	0.913 ± 0.022	0.938 ± 0.030	0.940 ± 0.031
german	0.69 ± 0.037	0.751 ± 0.044	0.718 ± 0.046
glass	0.709 ± 0.102	0.709 ± 0.048	0.586 ± 0.077
hepatitis	0.789 ± 0.120	0.809 ± 0.174	0.852 ± 0.108
iris	0.872 ± 0.097	0.922 ± 0.056	0.953 ± 0.052
letter	0.524 ± 0.027	0.510 ± 0.021	0.791 ± 0.007
mushroom	1.000 ± 0.00	0.997 ± 0.002	1.000 ± 0.000
segment	0.815 ± 0.026	0.831 ± 0.079	0.828 ± 0.104
sonar	0.75 ± 0.085	0.718 ± 0.127	0.705 ± 0.108
soybean	0.799 ± 0.117	0.788 ± 0.084	0.671 ± 0.087

The average F1-score of all data sets for CART was 0.79, while it was 0.78 for C4.5 and 0.77 for DTHF, indicating that the algorithms do not have a significant difference in overall performance. However, this was not surprising since, as stated in Section 2.5.1, studies have shown that the split criteria does not cause a significant difference to the performance of decision trees. Even the three algorithms did not have a significant difference in overall performance, CART seemed to be the most robust algorithm. It had the worst performance in only two of the data sets while both C4.5 and DTHF were the worst five times.

C4.5 and DTHF use the same split metric so in theory the algorithms should work similarly when the data set is complete and all the features are nominal. However, DTHF's results seem to be closer to CART than C4.5, and in some cases, like for the soybean data set, the results differ remarkably between C4.5 and DTHF. This is likely caused by the differences in the implementation of the algorithms, but a closer analysis on the matter would be needed to confirm this.

The different aspects of the data sets did not seem to correlate with the reciprocal performances of the algorithms. Each algorithm performed well on both smaller and larger data sets, and the number of features did not seem to be a significant factor. Even the distribution of labels did not seem to affect the results significantly. A better analysis would probably need some domain knowledge on the data sets. The standard deviations are quite similar for all algorithms, and they seem to tell more about the complexity of the data set rather than about the algorithm.

In conclusion, results for DTHF when using complete data is comparable to existing decision tree algorithms. The conducted tests did not include data sets with hierarchies, and more tests are required to see whether the method is able to benefit from them. If this would not be the case, using the method might not be sensible since the method requires a lot preprocessing for the data. The implementation of DTHF was also significantly slower than for comparison algorithms but this was expected as the implementation of DTHF was not optimized in any way whereas the implementations of comparison methods were from publicly available packages.

4.2.2 Performance with missing data

The incomplete data sets were created by randomly removing values from the original data sets. The rate of missingness varied between 10% and 90% every 10 percent. Each data set was created independently of the previous data sets, meaning that a data set missing 10% of its values might have missed completely different values than a set missing 20% of its values. All tests were made using 10-fold cross-validation and performance was measured using F1-score. For each rate of missingness the data set was created only once and all of the algorithms were given the same data on each missingness rate. Each algorithm was tested on all data sets and on all nine missingess rates. The results from the tests are collected to Table 4.3.

The results have some cases where algorithms give better results with less data,

Table 4.3 *F1-scores of DTHF, CART, and C4.5 using 10-fold cross-validation on data sets where data has randomly been removed.*

data set	method	percentage of removed data								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
autos	DTHF	0.15	0.06	0.05	0.06	0.06	0.05	0.05	0.06	0.05
	CART	0.59	0.55	0.55	0.54	0.47	0.44	0.41	0.43	0.45
	C4.5	0.38	0.44	0.03	0.03	0.03	0.03	0.03	0.03	0.03
balance	DTHF	0.66	0.61	0.62	0.58	0.50	0.49	0.48	0.45	0.47
	CART	0.77	0.74	0.74	0.71	0.63	0.60	0.61	0.60	0.60
	C4.5	0.76	0.74	0.65	0.67	0.59	0.63	0.20	0.19	0.14
breast-cancer	DTHF	0.49	0.84	0.82	0.78	0.707	0.74	0.71	0.64	0.73
	CART	0.95	0.94	0.93	0.92	0.92	0.90	0.90	0.86	0.80
	C4.5	0.89	0.85	0.79	0.79	0.79	0.79	0.79	0.79	0.79
german	DTHF	0.58	0.57	0.58	0.58	0.57	0.58	0.57	0.62	0.58
	CART	0.73	0.72	0.75	0.78	0.76	0.75	0.78	0.78	0.81
	C4.5	0.69	0.82	0.82	0.82	0.82	0.82	0.82	0.82	0.82
glass	DTHF	0.45	0.32	0.41	0.37	0.33	0.25	0.22	0.25	0.25
	CART	0.70	0.66	0.57	0.64	0.53	0.52	0.46	0.41	0.46
	C4.5	0.61	0.52	0.52	0.49	0.48	0.48	0.49	0.49	0.48
hepatitis	DTHF	0.76	0.79	0.75	0.74	0.77	0.67	0.69	0.71	0.71
	CART	0.84	0.80	0.86	0.77	0.82	0.82	0.80	0.83	0.88
	C4.5	0.88	0.34	0.33	0.32	0.32	0.33	0.33	0.33	0.33
iris	DTHF	0.93	0.84	0.83	0.80	0.69	0.59	0.58	0.39	0.29
	CART	0.96	0.91	0.86	0.83	0.81	0.67	0.64	0.58	0.43
	C4.5	0.906	0.647	0.47	0.49	0.50	0.48	0.48	0.49	0.49
letter	DTHF	0.05	0.05	0.04	0.04	0.03	0.02	0.02	0.02	0.01
	CART	0.45	0.38	0.29	0.25	0.20	0.15	0.09	0.07	0.07
	C4.5	0.56	0.30	0.10	0.07	0.08	0.08	0.08	0.08	0.08
mush-room	DTHF	0.85	0.81	0.76	0.71	0.66	0.61	0.54	0.47	0.36
	CART	0.99	0.97	0.95	0.93	0.91	0.89	0.84	0.78	0.70
	C4.5	0.82	0.68	0.68	0.65	0.65	0.65	0.66	0.66	0.65
segment	DTHF	0.47	0.43	0.31	0.27	0.34	0.31	0.25	0.20	0.22
	CART	0.82	0.77	0.67	0.59	0.54	0.44	0.36	0.29	0.18
	C4.5	0.20	0.25	0.23	0.24	0.24	0.24	0.24	0.25	0.25
sonar	DTHF	0.52	0.43	0.40	0.54	0.53	0.41	0.38	0.49	0.50
	CART	0.73	0.73	0.63	0.62	0.61	0.63	0.53	0.53	0.61
	C4.5	0.58	0.62	0.62	0.63	0.63	0.655	0.62	0.62	0.60
soybean	DTHF	0.26	0.20	0.13	0.14	0.08	0.12	0.02	0.12	0.09
	CART	0.66	0.67	0.51	0.43	0.45	0.35	0.31	0.23	0.22
	C4.5	0.15	0.06	0.05	0.06	0.06	0.05	0.05	0.06	0.05

for example, breast-cancer set for DTHF, and german for C4.5 and CART. This demonstrates how sensitive to changes in the data decision trees are, and even small differences in the training data set can produce a very different tree as discussed in Section 2.5. As the tests were conducted using data sets from which data were removed randomly, the results depended heavily on the generated data set. This could have been compensated by cross-validating each missingness rate. Several data sets could have been created for each rate and the cross-validation could have been conducted for each set. The final result would have been the mean of the cross-validation results.

Statistics calculated from the results would probably have no significance because of the way the tests were conducted. However, Figure 4.1 could give some insight to the performance of the algorithms. The figure illustrates the average performance of each of the algorithms on all rates of missingness. The percentage of missing data is on the x-axis and the average F1-score for all data sets on a certain rate of missingness is on the y-axis. The figure reveals that even all of the algorithms have almost the same starting point, as more data are missing CART performs significantly better than DTHF and C4.5. The performance of C4.5 and DTHF drops quickly but the decline stabilizes after the first 20% whereas CART has a linear drop on performance.

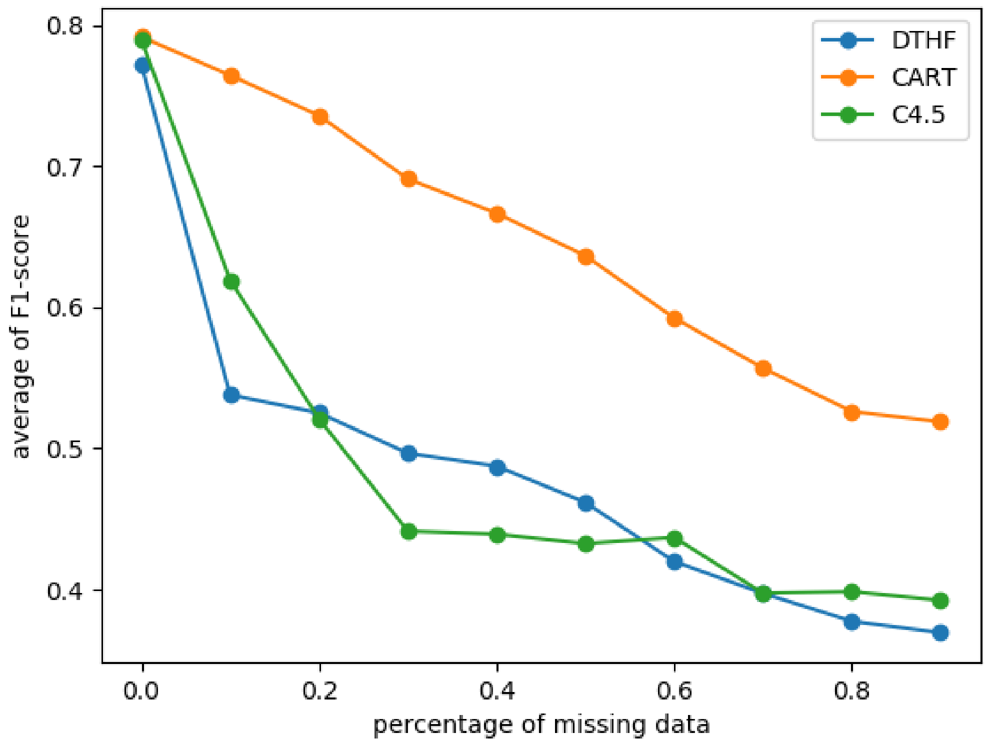


Figure 4.1 *Average of F1-scores for methods for each missingness rate.*

DTHF's performance is not consistent, and the drop on performance seems to depend on the data set. For some data sets, like the german and hepatitis data sets, the loss of information did not affect the performance remarkably. This could indicate that the data sets are easy, which was the case for hepatitis, but the german data set was not as easy to any of the algorithms. However, on the letter and soybean data set the performance drops dramatically as soon as data are missing. This is noteworthy since DTHF had the best performance on the soybean data set when data was not missing. The reasons why the algorithm behaves this way are unclear, and a more in-depth study would be needed on the matter.

The test results propose that the missing data handling in DTHF is not efficient compared to CART. This was a somewhat surprising result since DTHF and CART both create an alternative split rule that is used when data are missing. Instead, DTHF seems to perform similarly to C4.5 algorithm which has a clearly different kind of approach for handling missingness. The similarity in results could however be explained by the similar split heuristics. One factor in CART's superiority could be that data were missing completely random. Calculating a surrogate split that creates as similar split as complete data would make works well on completely missing data since it does not assume anything on the randomness. However, DTHF creates a completely separate branch if data are missing and thus the algorithm assumes that even missingness has information. C4.5 has a probability based approach which passes instances with missing values to both branches. Methods that CART and C4.5 use for handling missing data were explained in more detail in Section 2.5.2.

5. CONCLUSIONS

The goal in this thesis was to create a classification method capable of handling several different data types in addition to traditional nominal and numerical types, such as intervals and hierarchical values. This was done by developing a decision tree classification algorithm, DTHF, which uses hierarchic features to construct a model. Decision tree was selected as the base method because it is easy to implement and it is naturally good with mixed and missing data. The developed method is capable of handling numerical, nominal, hierarchical nominal and interval features as well as missing data.

Before constructing a model, DTHF transforms all data into a hierarchic form. Each feature is transformed to the same form, which allows a single feature to have several data types associated with it. The hierarchies are also utilized in the split rules as DTHF uses hierarchic levels as split rules instead of the whole feature. The levels are used from general to more specific and this makes constructing the tree faster since there are less possible split rules at a time. This also makes it possible to make first a rough split, and then go into more detail deeper in the tree if necessary. For example, instead of asking in the root whether the home town is *Madrid*, the first question could be whether the home town is in *Europe*.

Using hierarchies has also negative effects. All input data must be preprocessed, which can take a lot of resources, especially if there is a lot of data. The transformed data also takes more space than the original data. Because new data must be transformed in the same way as the training data, the parameters for the transformation have to be saved on creation and loaded every time new data is being evaluated. In addition, using hierarchies from generic to more specific as split rules might cause a larger decision tree than just using the complete feature would. In total, there are more potential split rules since every level of a hierarchy is a possibility. It is also possible that values which have common higher levels are not similar in the aspect of the problem. For example, even though *Madrid* and *Tampere* are both in *Europe*, their populations are in different order of magnitude.

The performance of DTHF was evaluated by testing with twelve UCI Machine Learning Repository's data sets and the results were compared with CART and C4.5 decision tree algorithms. The performance was tested with both complete data and data with missing values. DTHF method was implemented using Python, and the other methods were tested using R. All results were however computed using the same Python function.

The main results of the work are presented in Table 5.1 and two main points can be seen from it. Firstly, when data are not missing, DTHF is comparable to existing decision tree algorithms. Even though CART seems to be the most robust method, the differences between the methods are not significant. The other main result is that when data are missing, CART is remarkably better than DTHF and C4.5. The average F1-scores behave similarly for DTHF and C4.5 which is visible in Figure 4.1. This was unexpected since both DTHF and CART create a surrogate split rule while C4.5 has a probabilistic approach. Another interesting point in the results was how dependent the on the data set the drop of DTHF's performance was. One future research topic could be to find out what are the factors behind method's performance on missing data.

Table 5.1 *Average F1-scores from all tests for each method on both complete and missing data.*

	complete data	missing data
DTHF	0.77	0.48
C4.5	0.78	0.49
CART	0.79	0.65

The main limitation of the work was the testing of the developed algorithm. DTHF would need to be tested with data sets utilizing the possibilities of the method. Such data set could have naturally hierarchical features, several values for a feature, or intervals. However, such data sets could not been publicly found and thus such tests could not be conducted. Also, the tests with missing data should be improved by cross-validating the data sets which would make the results less dependent on the generated data set. Another limitation of the work is the lack of justification for the algorithm, we have argued why the algorithm has taken the approaches it does but the method was not studied in a more analytical way.

DTHF is a new method and there is still room for improvement in both the algorithm and the implementation. The performance could be boosted by adding pruning or applying the hierarchical construction of decision trees into an ensemble method like random forest. For standard decision trees this is known to enhance the performance of the classifier and therefore it is probable it would do the same for DTHF [6]. The method itself has also aspects that should be studied more. Such things would be for example, how should the transforming of data to a hierarchical form be done, and does the choice of split metric have an impact on the results.

BIBLIOGRAPHY

[1] T. Aljuaid and S. Sasi, "Proper imputation techniques for missing values in data sets," *Proceedings of the 2016 International Conference on Data Science and Engineering, ICDSE 2016*, 2017.

[2] H. Almuallim, Y. Akiba, and S. Kaneda, "On handling tree-structured attributes in decision tree learning," in *Proc. 12th Int. Conf. on Machine Learning*, 1995, pp. 12–20. [Online]. Available: https://tinyurl.com/zlmynq8

[3] A. N. Baraldi and C. K. Enders, "An introduction to modern missing data analyses," *Journal of School Psychology*, vol. 48, no. 1, pp. 5–37, 2010. [Online]. Available: http://wiki1.math.yorku.ca/images/6/6d/Enders_jofschoolpsyc.pdf

[4] G. Batista and M. C. Monard, "An analysis of four missing data treatment methods for supervised learning," *Applied Artificial Intelligence*, vol. 17, no. 5-6, pp. 519–533, 2003.

[5] W. Bi and J. T. Kwok, "Bayes-Optimal Hierarchical Multilabel Classification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 11, pp. 2907–2918, 2015.

[6] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[7] R. Cerri and G. L. Pappa, "An Extensive Evaluation of Decision Tree-Based Hierarchical Multilabel Classification Methods and Performance Measures," no. May 2013, 2014.

[8] Y. L. Chen, H. W. Hu, and K. Tang, "Constructing a decision tree from data with hierarchical class labels," *Expert Systems with Applications*, vol. 36, no. 3 PART 1, pp. 4838–4847, 2009.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition.* The MIT Press, 31.7. 2009. [Online]. Available: http://common.books24x7.com/book/id_49924/book.asp

[10] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[11] I. Eekhout, R. M. de Boer, J. W. Twisk, H. C. de Vet, and M. W. Heymans, "Missing data: a systematic review of how they are reported and handled," *Epidemiology*, vol. 23, no. 5, pp. 729–732, 2012.

[12] I. Eekhout, H. C. de Vet, J. W. Twisk, J. P. Brand, M. R. de Boer, and M. W. Heymans, "Missing data in a multi-item instrument were best handled by multiple imputation at the item score level," *Journal of Clinical Epidemiology*, vol. 67, no. 3, pp. 335–342, mar 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0895435613003879

[13] T. Elomaa, "Tools and techniques for decision tree learning," 1996.

[14] A. Gelman and J. Hill, "Missing-data imputation," pp. 529–543, 2007.

[15] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques.* Elsevier, 2011.

[16] Y. Han and W. Lam, "Utilizing hierarchical feature domain values for prediction," *Data and Knowledge Engineering*, vol. 61, no. 3, pp. 540–553, 2007.

[17] Y. Han and L. Wai, "Exploiting Hierarchical Domain Values in Classification Learning," *Proceedings of the Fourth SIAM International*, pp. 467–471, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.1863&rep=rep1&type=pdf

[18] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976.

[19] R. Kohavi and B. Becker. UCI Machine Learning Repository: Adult Data Set. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/adult

[20] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," *Informatica*, vol. 31, pp. 249–268, 2007. [Online]. Available: https://books.google.fi/books?id=vLiTXDHr_sYC&lpg=PA3&ots=CYkysxYDjr&dq=Supervised%20Machine%20Learning%3A%20A%20Review%20of%20Classification%20Technique&lr&hl=fi&pg=PA3#v=onepage&q=Supervised%20Machine%20Learning:%20A%20Review%20of%20Classification%20Technique&f=false

[21] O. Leka. (2017) Used cars database. [Online]. Available: https://www.kaggle.com/orgesleka/used-cars-database

[22] R.-H. Li and G. G. Belford, "Instability of decision tree classification algorithms," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '02*, 2002, p. 570. [Online]. Available: http://ai2-s2-pdfs.s3.amazonaws.com/c0b6/689d3d5ae6a73893b3780bfaae873ae3e830.pdfhttp://portal.acm.org/citation.cfm?doid=775047.775131

[23] R. J. A. Little and D. B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed. Wiley-Interscience, 2002.

[24] R. J. Little and D. B. Rubin, "Statistical analysis with missing data," 1987.

[25] J. K. Martin and D. S. Hirschberg, *The time complexity of decision tree induction*, 1995.

[26] S. M. McNee, J. Riedl, and J. a. Konstan, "Being accurate is not enough: how accuracy metrics have hurt recommender systems," *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, p. 1101, 2006. [Online]. Available: http://delivery.acm.org.libproxy.tut.fi/10.1145/1130000/1125659/p1097-mcnee.pdf?ip=130.230.162.80&id=1125659&acc=ACTIVESERVICE&key=74A0E95D84AAE420.8CAACA10A81E6DB1.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=800660889&CFTOKEN=89545176&__acm__=1503473

[27] S. K. Murthy, "Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey," *Data Mining and Knowledge Discovery*, vol. 2, pp. 345–389, 1998. [Online]. Available: https://pdfs.semanticscholar.org/2bb4/965aeb6a6cf2a27df0456d02bc616c89a572.pdf

[28] J. Perktold, S. Seabold, and J. Taylor. (2017) Patsy: Contrast coding systems for categorical variables. [Online]. Available: http://www.statsmodels.org/devel/contrasts.html

[29] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[30] J. Quinlan, "Unknown attribute values in induction," *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 164–168, 1989. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.5943&rep=rep1&type=pdf

[31] ——, *C4.5: Programs for Machine Learning*, ser. Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, 1993. [Online]. Available: https://books.google.fi/books?id=HExncpjbYroC

[32] L. Rokach and O. Maimon, *Data Mining with Decision Trees: Theory and Applications*, ser. Series in Machine Perception and Artificial Intelligence. World Scientific, oct 2014, vol. 81. [Online]. Available: https://books.google.fi/books?id=OVYCCwAAQBAJ&lpg=PR6&ots=tJgc91-hUR&dq=Data%20mining%20with%20decision%20trees%3A%20theory%20and%20applications&lr&hl=

fi&pg=PR14#v=onepage&q=Data%20mining%20with%20decision%20trees:%20theory%20and%20applications&f=false

[33] J. Rousu, C. Saunders, S. Szedmak, and J. Shawe-Taylor, "Kernel-based learning of hierarchical multilabel classification models," *Journal of Machine Learning Research*, vol. 7, no. Jul, pp. 1601–1626, 2006.

[34] M. Saar-Tsechansky and F. Provost, "Handling Missing Values when Applying Classification Models," *Journal of Machine Learning Research*, vol. 8, pp. 1625–1657, 2007.

[35] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[36] scikit-learn developers. (2017) 1.10. decision trees. [Online]. Available: http://scikit-learn.org/stable/modules/tree.html

[37] ——. (2017) 4.3.5. encoding categorical features. [Online]. Available: http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-categorical-features

[38] S. R. Seaman and I. R. White, "Review of inverse probability weighting for dealing with missing data," *Statistical methods in medical research*, vol. 22, no. 3, pp. 278–295, 2013.

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[40] C. J. Stone, "Classification and regression trees," *Wadsworth International Group*, vol. 8, pp. 452–456, 1984.

[41] J. Su and H. Zhang, "A Fast Decision Tree Learning Algorithm," *21st National Conference on Artificial Intelligence - Volume 1*, vol. 5, no. Quinlan 1993, pp. 500–505, 2006.

[42] B. E. T. H. Twala, M. C. Jones, and D. J. Hand, "Good methods for coping with missing data in decision trees," *Pattern Recognition Letters*, vol. 29, no. 7, pp. 950–956, 2008.

[43] X. Wu and V. Kumar, *The Top Ten Algorithms in Data Mining*, 2009. [Online]. Available: https://xa.yimg.com/kq/groups/81025504/1871639475/name/The_Top_Ten_Algorithms.pdf#page=184

[44] J. Zhang and V. Honavar, "Learning Decision Tree Classifiers from Attribute Value Taxonomies and Partially Specified Data," *Proceedings of the International Conference on Machine Learning*, 2003.