

UNIVERSITÀ DI PISA



Department of Computer Science
Bachelor's Degree in Computer Science

A Python Implementation of a Categorical Multi-Split Decision Tree

Supervisor:
Prof. Riccardo Guidotti

Presented by:
Giuseppe Muschetta

Academic Year 2022/2023

English translation of the original Italian thesis

*To myself,
for believing in it again, whatever my age.*

Contents

1	Introduction	4
2	Background	7
2.1	Basic Concepts	7
2.2	Decision Tree Classifier	10
2.3	Hunt's Algorithm	14
2.4	Impurity Measures	16
2.4.1	Impurity Measures for a Single Node	17
2.4.2	Impurity of the Child Nodes after the Parent Split	18
2.5	Final Characteristics of Decision Trees	19
3	Implementation	22
3.1	The Inner Class Node and the Class MixedDecisionTree	24
3.2	Impurity and Gain Computation	26
3.3	The Core of Everything: the ID3 Algorithm	27
3.3.1	Stopping Criteria	27
3.3.2	Pre-Pruning Criteria	28
3.4	Handling Continuous and Categorical Attributes	30
3.4.1	Attribute Handling in the fit Method	30
3.4.2	Attribute Handling in the predict Method	31
3.5	Transforming Continuous Attributes into Categorical Ones	34
3.6	Visualizing the MDT	36
3.7	The Three Modes of the MixedDecisionTree	38

3.7.1	Classic Mode	39
3.7.2	Median Mode	40
3.7.3	Binned Mode	41
4	Experiments	43
4.1	Preliminary Information	43
4.2	The Datasets	44
4.3	Tests	45
4.3.1	Metric Considered: ACCURACY	47
4.3.2	Metric Considered: F1_SCORE	59
4.3.3	Measuring Time	66
5	Conclusions	68
A		70

Chapter 1

Introduction

To me, mathematics, computer science, and the arts are insanely related. They're all creative expressions. (Sebastian Thrun)

In this thesis, a decision tree was implemented from scratch in the Python programming language. Python was chosen because of its advanced libraries, such as NumPy and Pandas, which make it easier to develop software that manipulates arrays, lists and datasets. In the following paragraphs we will look more closely at the concept of a dataset and, in the third chapter, we will see how datasets are manipulated with the Pandas library.

Decision tree learning is a supervised approach used in Statistics, Data Mining and Machine Learning, the field that lets machines learn from data autonomously and without explicit instructions, since the data themselves define what is learned.

In the context of machine learning, a decision tree is a predictive model that performs very well, works well on tabular data and is built with learning techniques that start from the initial datasets. A dataset is usually split into a training set, used to build the structure of the tree, and a test set, used to evaluate the accuracy of the predictive model. The proportions of the split can be handled by a library function, presented in the third chapter.

Decision trees are appreciated because they are easy for humans to read and interpret, and because of their excellent computational cost. Once trained, a decision tree works with a time complexity that is linear in the depth of the tree itself.

The reason for building a decision tree from scratch is that the decision trees of the scikit-

learn library are designed mainly to work with numerical variables and do not support categorical variables *natively*. Categorical variables must be converted into numerical ones, using techniques such as one-hot encoding, before the tree can be trained.

A further gap in the library is the support for multi-way splits: the decision trees in scikit-learn only support binary splits. In other words, every root or internal node of the tree has exactly two children. This can be limiting for problems in which a multi-way split would be more appropriate or more efficient.

This thesis implements a decision tree that is **categorical** and supports **multi-way splits**, which gives better readability, more versatility and closer adherence to the decision tree literature. The decision tree provided by the well-known Python library scikit-learn, on the other hand, is computationally efficient but, as said above, has some important limitations: in particular, it does not support multi-way splits and is not designed to work natively with categorical variables. These restrictions can reduce the flexibility and the applicability of the scikit-learn approach in some contexts.

The decision tree proposed here was named **MixedDecisionTree**, because it reads both categorical values (natively) and continuous values (through changes made outside the tree), and it allows an arbitrary number of children for every node, unlike the scikit-learn DT.

We will compare the two trees using metrics such as *Accuracy* and *F1-score*, analysing how they change as several pre-pruning hyper-parameters vary, such as *max-depth*, *min-sample-split* and *min-sample-leaf*, which are described in more detail later on.

The code of the project is available at:

<https://github.com/giuseppemuschetta/Native-Categorical-Multiway-Split-Decision-Trees>

The second chapter presents decision trees and the theoretical foundations that govern how they work.

The third chapter focuses on the key elements of the implementation of the MDT and introduces two additional operating modes alongside the default one, called *classic*: the *median* and *binned* modes. To allow deeper and more meaningful comparisons in the fourth chapter, these modes are applied to the scikit-learn decision tree as well.

The fourth chapter carries out a series of tests aimed at producing tables, histograms and

overlaid plots, which highlight the most relevant comparisons between the MDT and the scikit-learn DT.

Finally, the fifth and last chapter assesses whether the goal of developing a versatile and easily interpretable DT, with native handling of categorical attributes and full support for multi-way splits, has been achieved.

Chapter 2

Background

Computer Science is no more about computers than astronomy is about telescopes.

(Edsger Dijkstra)

This chapter focuses mainly on the theory of decision trees: we will see how they work internally, study Hunt's algorithm, which is the basis of everything, and explain the mechanism that actually makes machine learning possible, that is, the reason why a decision tree classification model is really able to *learn* from data.

2.1 Basic Concepts

Decision trees are essentially classifiers, and by classification we mean assigning a (class) label to data that do not have one: a classifier gives each record a single class label.

A **classifier** is therefore the model that performs the classification. In the literature, *classifier* and *model* are very often used as synonyms.

A classifier is usually described in terms of a model built from a given set of instances, known as the training set, which contains attribute values and class labels for every instance. The systematic approach to learning a classification model from a training set is known as a learning algorithm.

*The process of using a learning algorithm to build a classification model from the training set is known as induction, or the inductive bias of the algorithm. **Induction is what makes***

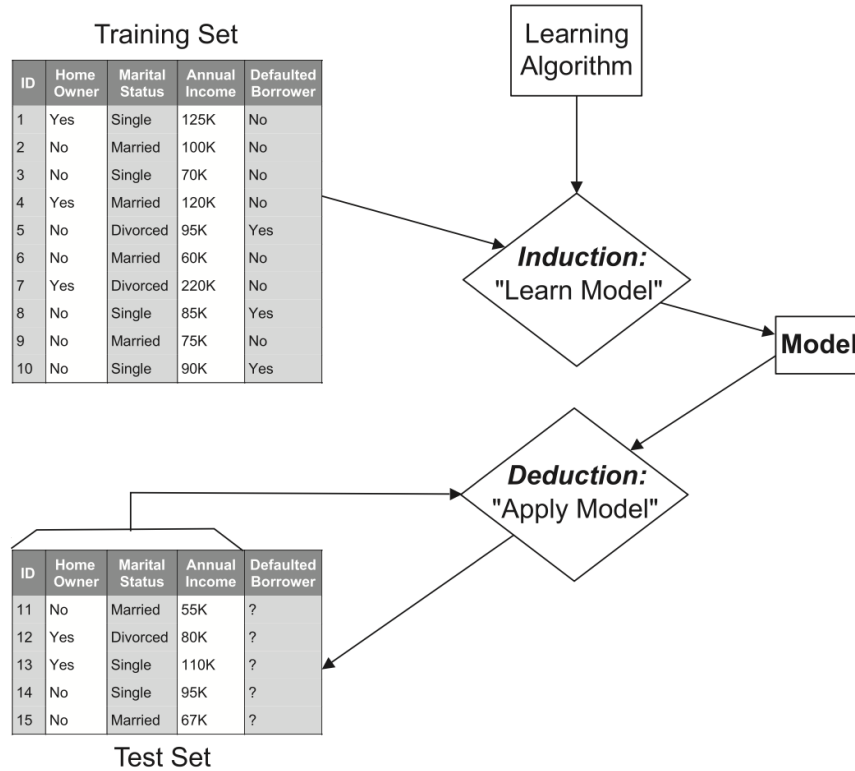


Figure 2.1: The role of deduction and induction [3]

learning possible. The process of applying a classification model to new test instances (never seen before) in order to predict their class labels is known as **deduction**. The classification process therefore involves two steps:

1. applying a learning algorithm to the training set so that the model "learns";
2. applying the trained model (which has already learned) to assign labels to instances that have none, that is, unlabeled instances.

We have mentioned induction and deduction: they express the key concepts for building a classification model that learns from the training data and is then able to predict the labels of data it has never seen before, or unseen data. Figure 2.1 clearly shows what has just been said: the induction and deduction steps must be carried out separately. *The training set and the test set must be independent of each other*, so that the induced model can accurately predict the class labels of instances it has never met before. Models that provide such predictive insight are said to have good generalization performance.

The performance of a model (classifier) can be evaluated by comparing the labels predicted by the tree with the true labels of the test instances. By true labels of the test instances we mean that, when the test is carried out, the true labels are already known; otherwise it would make no sense to call it a *test* set.

All the test instances are fed to the tree without their true labels, and we then count how many of them are predicted correctly. This information can be summarized in the following table, called the *confusion matrix*, for a binary classification problem:

		Predicted Class	
		Class = 1	Class = 0
Actual Class	Class = 1	f_{11}	f_{10}
	Class = 0	f_{01}	f_{00}

Figure 2.2: An example of a confusion matrix when there are only two classes [3]

Each entry f_{ij} is the number of instances of class i that are predicted to be of class j . For example, f_{01} is the number of instances of class 0 wrongly predicted as class 1. The number of correct predictions made by the model is $(f_{11} + f_{00})$ and the number of wrong predictions is $(f_{10} + f_{01})$.

Although a confusion matrix provides the information needed to determine how well a classification model works, summarizing this information in a single number makes it easier to compare the relative performance of different models. This can be done with evaluation metrics such as **Accuracy** and **F1-score**, which are computed as follows:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total Number of predictions}}$$

For binary classification problems, the accuracy of a model (classifier, DT, etc.) is given by:

$$\text{Accuracy} = \frac{f_{11} + f_{00}}{f_{11} + f_{10} + f_{01} + f_{00}}$$

Another very important metric, which the comparisons in the next chapters will also take into account, is the F1-score mentioned above. To define it properly, we first need to introduce the two metrics it is made of: *precision* and *recall*. Precision is defined as follows:

$$\text{precision} = \frac{\text{Number of True Positives}}{\text{Number of True Positives} + \text{Number of False Positives}}$$

Recall, instead, is defined as:

$$\text{recall} = \frac{\text{Number of True Positives}}{\text{Number of True Positives} + \text{Number of False Negatives}}$$

Now that we have defined the two fundamental components of the F1-score, we can compute it as follows:

$$\text{F1-score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Another interesting metric is the *Error-rate*; obviously, the lower it is, the better:

$$\text{Error-rate} = \frac{\text{Number of Wrong Predictions}}{\text{Total Number of Predictions}}$$

For most classification techniques, learning algorithms are designed to learn models that reach very high values of accuracy and F1-score and, of course, very low values of Error-rate when applied to the test set.

Now that concepts such as deduction, induction, classification and performance metrics have been clarified, we should finally be ready to define the nature and the workings of a Decision Tree Classifier.

2.2 Decision Tree Classifier

To illustrate how a decision tree works, consider the problem of telling days that are suitable for playing tennis from days that are not, using the dataset in Figure 2.3 on page 11.

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

Figure 2.3: PlayTennis dataset [2]

Given a generic day, how can we predict whether it is a suitable day or not? A first approach is to ask questions about the attributes of the dataset (that is, the column names). For example, we might ask whether the temperature will be mild or too hot: if it is too hot, and perhaps there is also too much wind (now the Wind attribute comes into play as well), it would be better not to play tennis that day.

It is therefore clear that *the classification process starts from a series of questions about the values of the attributes*: first we asked whether the day was hot or not, then whether the wind was strong or weak. In short, we first examined one attribute with one of its possible values (*Attribute: Temperature, Value: Mild*) and then another attribute (*Attribute: Wind, Value: Strong*), again with one of its possible values.

The previous example shows how a classification problem can be solved by asking a series of carefully crafted questions about the attributes. Each time an answer is received, another question can immediately be asked, and so on, until the class label can be decided once and for all (play tennis on that day: "YES" or "NO").

The series of questions and their possible answers can be organized in a hierarchical structure called a **decision tree**. Figure 2.4 on page 13 shows an example of the complete, final decision tree for the PlayTennis problem. A typical decision tree consists of three types of nodes:

- **A root node**, with no incoming links and zero or more outgoing links.
- **Internal nodes**, each of which has exactly one incoming link and two or more outgoing links.
- **Leaf** or terminal **nodes**, each of which has exactly one incoming link and no outgoing links.

Every leaf node in the decision tree is associated with a class label; in the case of PlayTennis there are only two class labels: "YES" or "NO". PlayTennis is therefore a classification problem with two classes, also called binary. The non-terminal nodes, which include the root and the internal nodes, contain attribute test conditions, usually defined on a single attribute. Each possible outcome of the attribute test condition is associated with exactly one child of that node. For example, the root node of the tree shown in Figure 2.4 uses the attribute Outlook (the weather forecast) to define an attribute test condition with three outcomes, sunny, overcast and rain, which results in three child nodes.

Given a decision tree, classifying an instance is simple: starting from the root node, the test condition is applied and, depending on its outcome, the appropriate branch is followed. It leads either to another internal node, where a new attribute test condition is applied, or to a leaf node. Once a leaf node is reached, it contains the label to assign to that test instance.

For example, looking again at Figure 2.4 on page 13, if the forecast says rain and the wind is also strong, we reach a leaf with the classification "NO", whereas if the forecast says sun, with normal humidity, the leaf we reach gives "YES". Now pay attention: if the weather forecast, the attribute chosen at the root, directly gives the value overcast, we immediately reach a leaf associated with the class label "YES".

From this example, the attentive reader will understand that if, with suitable criteria, *we choose the best attributes to ask questions about right from the root, the tree will be smaller and a classification will be reached more easily.*

We have explained, in simple terms, what the inductive bias is that allows the tree to *learn* from examples and to predict other examples it has never seen (unseen).

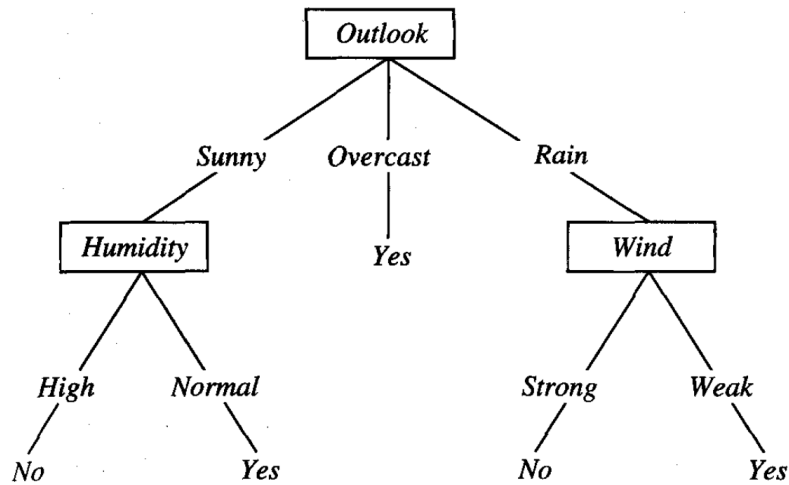


Figure 2.4: Decision tree for the PlayTennis dataset [2]

Mitchell defines the inductive bias as follows: "**The inductive bias of an algorithm is the set of assumptions that the classifier uses to predict the output given inputs that it has not encountered yet**". For a decision tree, the inductive bias used by the algorithm that generates it (we will see the algorithm later) is to prefer **shallower** trees to deeper ones.

For the tree-building algorithm to generate the shallowest of all the trees it could possibly generate, it must select, starting from the root, only the best attributes, that is, the attributes that have certain properties.

Trees that place the attributes with *the largest Gain in statistical impurity* as close as possible to the root are to be preferred to the others. This is why, in Figure 2.4, the attribute at the root of the tree is "Outlook".

2.3 Hunt's Algorithm

```
TreeGrowth (E, F)
1: if stopping_cond(E, F) = true then
2:   leaf = createNode().
3:   leaf.label = Classify(E).
4:   return leaf.
5: else
6:   root = createNode().
7:   root.test_cond = find_best_split(E, F).
8:   let V = {v | v is a possible outcome of root.test_cond }.
9:   for each v ∈ V do
10:    E_v = {e | root.test_cond(e) = v and e ∈ E}.
11:    child = TreeGrowth(E_v, F).
12:    add child as descendent of root and label the edge (root → child) as v.
13:   end for
14: end if
15: return root.
```

Figure 2.5: Hunt's algorithm for a Decision Tree [3]

Hunt's algorithm, in Figure 2.5 on page 14, uses a *greedy* strategy to grow the decision tree *top-down*, making a series of locally optimal choices about which attributes to use to partition the training data. It is also one of the most widely used algorithms and the basis of many real implementations of DTs.

The MixedDecisionTree uses the ID3 algorithm (Iterative Dichotomiser 3), but there are also other real implementations of Hunt's algorithm, such as C4.5 and CART, which will not be discussed further here.

Hunt's algorithm in Figure 2.5 on page 14 gives the pseudocode of the decision tree induction algorithm. Its input consists of two parameters: the first is a set of training examples E , the second is the set of all the attributes F . The algorithm works by *recursively selecting the best attribute to split the data* (as can be seen in line 7 of the pseudocode) and expanding the nodes of the tree (lines 11 and 12) until a *stopping criterion* is met (line 1). The stopping criteria are the following:

- If all the examples E are positive, stop, create a leaf node to return and assign it the classification +

- If all the examples E are negative, stop, create a leaf node to return and assign it the classification -
- If the set of attributes F becomes empty, stop, create a leaf node to return and assign it, as classification, the most frequent label value in E

The details of this algorithm are explained below:

1. The function **createNode()**, line 6, extends the decision tree by creating a new node. A root or internal node of the decision tree is implemented by storing in it a test condition, or split condition, e.g. `node.test_cond`, while a leaf stores a class label, e.g. `node.label`
2. The function **find_best_split(E,F)**, line 7, determines the attribute test condition used to partition the training instances associated with a node. The splitting attribute chosen depends on a given criterion, usually statistical impurity. Popular measures include entropy and the Gini index, which we will see in the next sections.
3. The function **Classify(E)**, line 3, determines the class label to assign to a leaf node.
4. The stopping function **stopping-cond(E,F)**, line 1, is used to end the growth of the tree. Since decision trees use a top-down recursive partitioning approach to build a model, the number of training instances associated with a node decreases as the depth of the tree increases. As a result, a leaf node may contain too few training instances to make a statistically significant decision about its class label. This is known as the data fragmentation problem. One way to avoid it is to forbid splitting a node when the number of instances associated with it falls below a certain threshold. With this in mind, it is worth presenting the pre-pruning hyper-parameters considered in this thesis (they are also found in the scikit-learn library), which stop the construction of the tree under certain conditions to avoid excessive specialization: **max_depth**, **min_sample_split** and **min_sample_leaf**. All three are integer threshold values. The first is an upper bound that prevents a new node from being created if the current depth of the tree is strictly greater than the threshold. The second makes sure that a node can no longer be split, becoming a leaf (classified with the most frequent label in the data), if the number of samples associated with it is strictly lower than the integer threshold. The last of the

three checks, before the new node is even generated, whether the potential split would result in a child node with fewer samples than the integer threshold requires. If that happens, the node becomes a leaf, classified with the most frequent label in the data.

In the next section we will finally see how the best attributes are chosen and placed as close as possible to the root, so that the inductive bias of Hunt's algorithm, the *absolute preference for shallower trees over deeper ones*, is satisfied. In short, we will understand that the function in line 7 of the pseudocode in Figure 2.5, **find_best_split(E,F)**, *selects the best attribute to split on the basis of the concept of statistical impurity*.

These measures try to give preference to the attributes that partition the training instances into purer subsets in the child nodes, where most instances share the same class label. Purer nodes are useful because a node whose training instances all belong to the same class does not need to be split further. On the contrary, an impure node, whose training instances mostly have different class values, is likely to require several further levels of node expansion, which considerably increases the depth of the tree and noticeably degrades its performance.

Larger trees are less desirable because they are more prone to overfitting, a condition that can degrade the ability of the tree to generalize (that is, to make correct predictions on data it has never seen, unseen data, that we want to classify). They are also hard to interpret and take more training and testing time than smaller trees, which are the more desirable ones.

Several ways of measuring *the impurity of a node and the collective impurity of its child nodes* will be presented; both are used to find the **best attribute to split**.

2.4 Impurity Measures

The best split attribute can be found through impurity measures known as Entropy, Gini Index and Misclassification Error (although the latter is mostly didactic and therefore not used in real implementations). We will first present the impurity measures for a single node and then the collective ones, which also involve the child nodes.

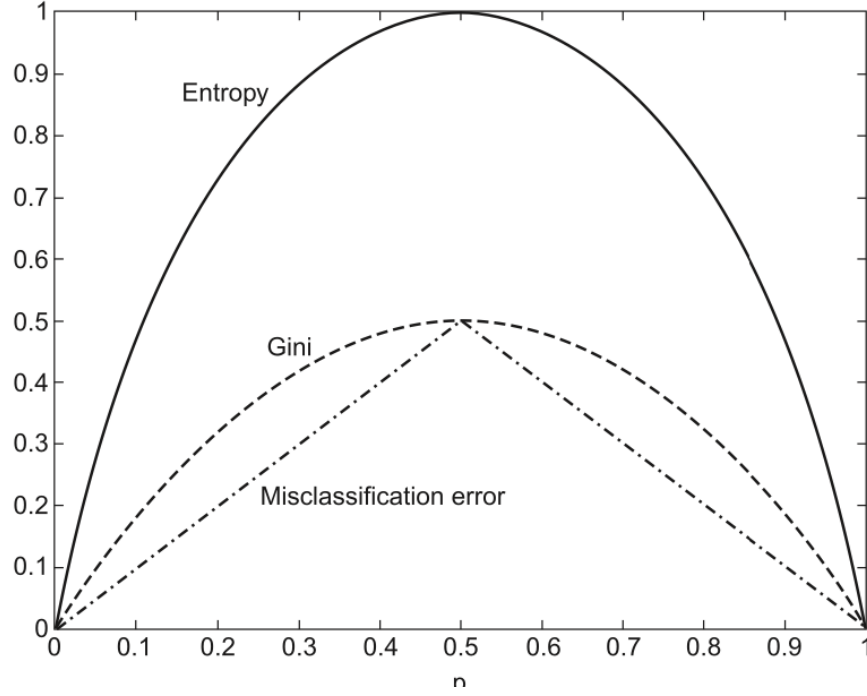


Figure 2.6: Comparison of the impurity measures for binary classification [3]

2.4.1 Impurity Measures for a Single Node

The impurity of a node measures how dissimilar the class labels are for the data instances that belong to a common node. The following are examples of measures that can be used to evaluate the impurity of a node t :

$$\begin{aligned} \text{Entropy} &= - \sum_{i=0}^{c-1} p_i(t) \log_2 p_i(t), \\ \text{Gini index} &= 1 - \sum_{i=0}^{c-1} p_i(t)^2, \\ \text{Classification error} &= 1 - \max_i [p_i(t)], \end{aligned}$$

Figure 2.7: The impurity measures of a single node [3]

Here $p_i(t)$ is the relative frequency of the training instances at node t that belong to class i , c is the total number of classes, and $0 \log_2(0)$ is set to zero when it appears in the computation

of the entropy. All three measures give an impurity of zero if a node contains only instances of a single class, and they reach their maximum impurity if the node contains an equal proportion of instances of several classes. Figure 2.6 on page 17 compares the relative magnitude of the impurity measures when they are applied to binary classification problems. Since there are only two classes, $p_0(t) + p_1(t) = 1$.

The horizontal axis p refers to the fraction of instances that belong to one of the two classes. The important thing to observe is that *all three measures reach their maximum when the class distribution is uniform* (that is, $p_0(t) = p_1(t) = 0.5$) and *their minimum value, 0, when all the instances belong to a single class* (that is, when $p_0(t)$ or $p_1(t)$ equals 1).

To make everything clearer, consider the following example:

Node N_1	Count	Gini = $1 - (0/6)^2 - (6/6)^2 = 0$
Class=0	0	Entropy = $-(0/6) \log_2(0/6) - (6/6) \log_2(6/6) = 0$
Class=1	6	Error = $1 - \max[0/6, 6/6] = 0$
Node N_2	Count	Gini = $1 - (1/6)^2 - (5/6)^2 = 0.278$
Class=0	1	Entropy = $-(1/6) \log_2(1/6) - (5/6) \log_2(5/6) = 0.650$
Class=1	5	Error = $1 - \max[1/6, 5/6] = 0.167$
Node N_3	Count	Gini = $1 - (3/6)^2 - (3/6)^2 = 0.5$
Class=0	3	Entropy = $-(3/6) \log_2(3/6) - (3/6) \log_2(3/6) = 1$
Class=1	3	Error = $1 - \max[3/6, 3/6] = 0.5$

Note that if all the instances in a node belong to the same class, the values of the entropy, the Gini Index and the Misclassification Error are at their lowest, that is 0. Zero is the lowest value because statistical impurity has no negative values.

If instead a node contains, for example, six instances with only two classes, three labeled "YES" and three labeled "NO", that node has the maximum degree of impurity: the Entropy reaches its maximum value of 1, while the Gini Index and the Misclassification Error both reach their maximum value of 0.5.

2.4.2 Impurity of the Child Nodes after the Parent Split

Consider an attribute test condition that splits a node containing N training instances into k children, $v_1, v_2, \dots, v_k$, where each child node is a partition of the data resulting from one of

the k outcomes of the attribute test condition. Let $N(v_j)$ be the number of training instances associated with a child node v_j , whose impurity value is $I(v_j)$. Since a training instance in the parent node reaches node v_j a fraction $N(v_j)/N$ of the times, *the collective impurity of the child nodes can be computed as a weighted average of the impurities of the child nodes*, as follows:

$$I(\text{children}) = \sum_{j=1}^k \frac{N(v_j)}{N} I(v_j),$$

At this point it is finally possible to identify the best attribute to split. To determine how good a split condition is, it is appropriate to compare the impurity of the parent node (before the split) with the weighted impurity of the child nodes (after the split). The larger their difference, the more attractive the choice of that particular attribute becomes. This difference, also known as Gain, the gain in purity, is what makes us prefer one attribute to another.

In a nutshell, the attribute with the maximum gain is selected among all the others and split, which nicely satisfies the inductive bias of the ID3 machine learning algorithm.

The impurity of the children is defined as the weighted average of their impurities; the gain in impurity is therefore given by the following equation:

$$\text{Gain} = \text{Impurity of the parent} - \text{Impurity of the children}$$

2.5 Final Characteristics of Decision Trees

At this point, the reader should have understood, at least in theory, how a Decision Tree classifier works.

Hunt's algorithm explains how it works better than a thousand words: at each recursive call, if no stopping condition holds, a new expansion node is always created, holding the best split attribute obtained through the greedy choice provided by the Gain (the gain in impurity). The attribute that has the maximum gain in impurity at that moment is selected, whichever criterion was chosen for the statistical impurity. The difference between Entropy and the Gini index lies entirely in their formula when the computation is carried out on a single node t . For further clarification, see Figure 2.7 on page 17 again.

The last interesting point is why DTs are so popular and so widely used compared with the other classification techniques available in machine learning. Here are some of the reasons:

- They are easy to interpret, especially when the resulting tree is not too deep.
- Their accuracy is comparable to that of other classification techniques on many simple datasets.
- They are robust to noisy data: DTs can easily handle redundant attributes (an attribute is called redundant when it is strongly correlated with another attribute in the data), because such attributes show very similar gains in impurity when considered for the split, but in the end only one of them (the one with the maximum gain) is selected by Hunt's algorithm as the best split attribute. They are also robust to irrelevant data (data are called irrelevant when they are only weakly associated with the target variable, the column of the dataset that contains only class labels), because, having a very small gain in impurity, they have a very low risk of being chosen by the algorithm as the best split attribute.
- They are inexpensive in terms of resources: building a tree costs relatively little.
- Once the tree has been built and trained well, it is always extremely fast at classifying new unseen data: *training a tree has a complexity of $O(M * N * \log N)$* , where M is the number of columns or attributes in the dataset and N the number of instances, objects or rows of the dataset, while *classifying a new record has a complexity of $O(w)$* , where w is the maximum depth of the tree.
- They also handle missing attribute values well. [1]

So do DTs only have strengths? In reality, as valid and excellent as they are, DTs are not free from flaws either:

- The memory needed by some trees can grow exponentially, which makes the search for the best DT almost hopeless, despite greedy approaches.
- They do not take into account some interactions that might exist between the attributes.

- Each decision involves only one attribute, which is a limitation.
- Data fragmentation: the number of instances shrinks at each level of the tree, so the number of instances that reach the leaf nodes can be too low to make a statistically relevant decision.
- Tree replication: some subtrees may appear identical in several branches.

Chapter 3

Implementation

Software and cathedrals are much the same: first we build them, then we pray.

(Sam Redwine)

This chapter explains the inner workings of the *MixedDecisionTree*, or MDT, with a strong emphasis on its implementation. For this reason, it is worth listing the libraries used in this project and their versions: **numpy** 1.21.5, **pandas** 1.4.4, **scikit-learn** 1.1.1 and **matplotlib** 3.5.2.

As anticipated in the introduction, the MDT accepts as input datasets in which categorical values are expressed as strings and continuous values as floats, that is, floating-point real numbers (any integer-valued columns are converted to float).

Essentially, the MDT takes as input the same datasets we find in textbooks, such as PlayTennis in Mitchell’s Machine Learning book [2], analysed in the previous chapter in Figure 2.3 on page 11, or other mixed datasets that contain categorical string values as well as integers, floats and so on.

If we tried to give the PlayTennis dataset as it is, without any change, to the scikit-learn decision tree, it would immediately raise a host of exceptions. This happens because the tree of the scikit-learn library directly accepts only continuous values, as integers or floats, and does not consider categorical attributes at all.

The MDT, instead, given the PlayTennis dataset as input, builds the tree immediately without raising any exception, and the resulting decision tree is exactly the one shown in Figure

2.4 on page 13. Note that the root node is the parent of as many as three children. Hence *splits are not limited to binary ones*.

At this point the question might be: "how does the MDT cope with fully continuous datasets such as the famous three-class Iris?" The answer is just as simple: it accepts it easily as input too, but the dataframe has to undergo a series of changes that take place outside the tree. With continuous attributes, the MDT performs binary splits.

After training, the MDT reaches practically the same accuracy and F1-score as the scikit-learn tree. The only tangible difference concerns time and space complexity: the MDT is much less efficient than the scikit-learn tree, as can be seen in the fourth chapter by looking at the table on page 66 and its description.

Going back to the MDT, it supports **three modes**. The first, called "*classic*", is the standard behaviour of the tree described so far. The second, called "*median*", uses the pandas.cut function together with mid: it works like the classic mode but *simplifies it enormously* without losing too much accuracy and F1-score (sometimes the accuracy even stays the same), while the computational saving in time and space is considerable. The third and last mode, called "*binned*", uses only pandas.cut with a variable number of bins, turning columns of continuous values into strings of intervals and leaving the columns of categorical values unchanged.

Did it make sense, then, to undertake this project and program a decision tree classifier from scratch? Here are some reasons in favour:

- Readability for the user: every root and internal node supports multi-way splits, so the tree is displayed exactly as in the literature. As a visual example, see again the DT obtained from the PlayTennis dataset on page 13.
- Datasets are more expressive and easier to read too: categorical attributes are expressed as strings and are read directly by the MDT.
- Three operating modes, *classic*, *median* and *binned*, all in the same package, make the MDT more versatile and suitable for a larger number of comparisons.

In the next section we will dive straight into the implementation of the MDT, starting from the main classes it is made of.

3.1 The Inner Class Node and the Class MixedDecisionTree

The class *Node.py* was placed inside the class of the tree itself, *MixedDecisionTree.py*, so that everything can be imported more easily into other programs.

```
1 class MixedDecisionTree:
2     class Node:
3         def __init__(self):
4             self.split_condition = None
5             self.gain: float = float()
6             self.split_value = None
7             self.isLeaf: bool = False
8             self.isRoot: bool = False
9             self.classification = None
10            self.parent = None
11            self.depth: int = int()
12            self.children = list()
13
14        def __init__(self, class_name='class', max_allowed_depth=50, min_samples_split=2, min_
            samples_leaf=1,
15                tree_mode='classic', number_of_bins=0, cut_threshold=10000, criterion='
                entropy'):
16            self.root = None
17            self.class_name = class_name
18            self.initial_depth = 0
19            self.max_allowed_depth = max_allowed_depth
20            self.min_samples_split = min_samples_split
21            self.min_samples_leaf = min_samples_leaf
22            self.tree_mode = tree_mode
23            self.number_of_bins = number_of_bins
24            self.cut_threshold = cut_threshold
25            self.criterion = criterion
26            self.most_common_value = None
27            # ...
```

Listing 3.1: MixedDecisionTree and the inner class Node

Looking at the inner class Node, in Listing 3.1, line 4 holds the test condition for the best attribute to split. As we know from the previous chapter, the best attribute to split is chosen according to the maximum gain (Gain = Impurity of the parent node - Impurity of the children), which can be computed using either Entropy or the Gini index as the impurity measure. Line

9 stores the classification value if the node is a leaf, so `self.classification = class label`. Lines 7 and 8 tell the user what kind of node it is, for example whether it is a root or a leaf node, line 11 stores the depth of the current node and, most importantly, line 12 shows that *the node holds a list of children*. This is a very convenient feature, because it is what gives the internal nodes (root included) their support for **multi-way splits**, that is, the possibility for every node to have an arbitrary number of children.

Besides the pre-pruning hyper-parameters, the class "MixedDecisionTree" also holds the configuration parameters to be set when the class is instantiated:

- **self.criterion** sets whether the tree will use entropy or the Gini index to find the best attribute to split
- **self.tree_mode** sets the mode of the tree; as said before, there are three modes: classic, median, binned
- **self.number_of_bins**, not used in classic mode but only in binned and median mode, sets the number of bins used by the `pandas.cut` function
- **self.cut_threshold** can only be used by the classic mode; it is an integer threshold that reduces the number of rows of a dataset. If the dataset has 7000 rows and the threshold is set to 3000, the resulting dataframe contains only the first 3000 rows; the remaining 4000 rows are not considered, because they are not in the dataframe.
- **self.max_allowed_depth** is a pre-pruning hyper-parameter that stops the growth of the tree once the preset maximum depth is exceeded.
- **self.min_samples_split** is a pre-pruning parameter that prevents a node from splitting if its samples are strictly fewer than the value of the parameter. If that happens, the node is not split but becomes a leaf, classified with the most frequent label in the data.
- **self.min_samples_leaf** is a pre-pruning parameter that checks, before the new node is even generated, whether the potential split would result in a child node with fewer samples than the integer threshold requires. If that happens, the node becomes a leaf, classified with the most frequent label in the data.

3.2 Impurity and Gain Computation

It is now time to show the method that computes the Gain in order to select the best split attribute. The computation uses entropy or the Gini index, depending on the user's choice when the MDT is instantiated. The chosen criterion is stored in the instance variable `self.criterion`.

```
1  def _get_impurity(self, attribute):
2      values, counts = np.unique(attribute, return_counts=True)
3      impurity_values = []
4      if self.criterion == 'entropy':
5          for i in range(len(values)):
6              impurity_values.append(-(counts[i] / np.sum(counts)) * (np.log2(counts[i] /
7                  np.sum(counts))))
8              impurity_measure = np.sum(impurity_values)
9      elif self.criterion == 'gini':
10         for i in range(len(values)):
11             impurity_values.append((counts[i] / np.sum(counts)) ** 2)
12             impurity_measure = 1.0 - np.sum(impurity_values)
13         else:
14             raise Exception("Wrong criterion, possible choices are: 'entropy' or 'gini'")
15         return impurity_measure
16
17 def _getGain(self, data, split_attribute_name):
18     I_parent = self._get_impurity(data[self.class_name])
19     assert (I_parent >= 0)
20     values, counts = np.unique(data[split_attribute_name], return_counts=True)
21     impurity_values = []
22     for i in range(len(values)):
23         impurity_values.append(
24             (counts[i] / np.sum(counts)) * self._get_impurity(
25                 data.where(data[split_attribute_name] == values[i]).dropna()[self.class_
26                     name]))
27     I_child = np.sum(impurity_values)
28     gain = I_parent - I_child
29     return gain
```

Listing 3.2: Gain computation

Line 16 of Listing 3.2 holds the helper method `_getGain`, while the helper method `_get_impurity`, at line 1, computes the impurity of a single attribute passed as input. If the criterion chosen when the tree was instantiated is 'entropy', the computation uses Entropy: lines 5 to 7 compute the entropy, whereas if the chosen criterion is 'gini', the computation uses the Gini index

and is exactly the one carried out in lines 9 to 11.

The method `_getGain` extends the computation to all the potential children until it finds the value of the gain for that attribute, as can be seen in line 26 of Listing 3.2:

$Gain = I_{parent} - I_{child}$, and returns it to the function in charge of choosing the best attribute. The computation of I_{child} takes place in lines 21 to 25 and can be seen as the weighted average of the impurity of the children after the split.

3.3 The Core of Everything: the ID3 Algorithm

It is finally time to go straight to the core of the `MixedDecisionTree`, the recursive function that builds the tree and from which practically all the calls to the other methods start. It is the method that implements the recursive ID3 algorithm (Iterative Dichotomiser 3). To follow the lines below more easily, it is advisable to read the pseudocode of Hunt's algorithm on page 14.

The first thing we meet are the stopping criteria, so the stopping conditions are checked: if they hold, the algorithm stops, and a leaf with the classification suited to that particular case is generated and returned.

In the version implemented in this thesis, the pre-pruning conditions were added to the stopping criteria; they have the same purpose of stopping the growth of the tree early.

From theory we know that the best decision trees are, for the same input data, their shallowest possible version (inductive bias).

If neither the stopping criteria nor the pre-pruning ones hold, the tree starts, and keeps on, expanding: at each recursive call it instantiates a new node, to which the best split attribute is assigned. *For each unique value of the split attribute we grow a branch. The recursion works on a set of samples that becomes smaller and smaller with respect to the initial one*, and it goes on until either a stopping criterion or a pre-pruning one ends the growth of the tree.

3.3.1 Stopping Criteria

The following snippet shows the stopping criteria of the tree, as provided by Hunt's algorithm; the ID3 version keeps the same ones.

```
1 def _stoppingCriteria(self, data, attributes):
```

```

2     node = MixedDecisionTree.Node()
3     node.isLeaf = True
4     # first stopping criterion
5     if len(np.unique(data[self.class_name])) <= 1:
6         node.classification = np.unique(data[self.class_name])[0]
7     # second stopping criterion
8     elif len(data) <= 0:
9         node.classification = self.most_common_value
10    # third stopping criterion
11    elif len(attributes) <= 0:
12        node.classification = self._getMostCommonValue(data)
13    else:
14        # If no stopping or pruning criterion holds, return
15        return None, False
16    return node, True

```

Listing 3.3: Stopping criteria of the MixedDecisionTree

These criteria are very easy to understand. With the **first stopping criterion** (line 5 of snippet 3.3), the tree stops if all the samples in the node have the same classification.

The **second stopping criterion** (line 8) states that if the number of samples in the node is zero, a leaf is returned with the classification "most common label value in the original dataset", that is, the statistical mode of the class column of the original dataframe; by original we mean the whole dataframe, before it started shrinking because of the recursive calls.

Line 11 holds the **third stopping criterion**: if the length of the list of attributes drops to zero, in other words, if the list of attributes still in play becomes empty, a leaf is returned with the classification given by the most frequent label value, not in the original dataset, as before, but in the current data, the data that keep shrinking at every recursive call.

3.3.2 Pre-Pruning Criteria

To complete the topic of "stopping criteria", we show the pre-pruning conditions of the tree, which have the same effect as the stopping criteria: stopping the growth of the tree early.

```

1     # PRUNING: MAX_DEPTH
2     if current_depth > self.max_allowed_depth:
3         node = MixedDecisionTree.Node()
4         node.isLeaf = True
5         node.classification = self._getMostCommonValue(data)

```

```

6     return node
7
8     # PRUNING: MIN_SAMPLES_SPLIT
9     if len(data) < self.min_samples_split:
10         node = MixedDecisionTree.Node()
11         node.isLeaf = True
12         node.classification = self._getMostCommonValue(data)
13         if current_depth == self.initial_depth:
14             return node
15
16     sub_data_pairs_list = list()
17     flag_leaf = False
18     # for each value of the split attribute we grow a branch:
19     for value in np.unique(data[best_split_attribute]):
20         split_attribute_value = value
21         # the subset of the dataset on which the recursion starts
22         sub_data = data.where(data[best_split_attribute] == split_attribute_value).
                dropna()
23         # PRUNING: MIN_SAMPLE_LEAF
24         # the samples of the sub-data must be strictly fewer than the hyper-parameter
25         if len(sub_data) < self.min_samples_leaf:
26             flag_leaf = True
27             break
28         sub_data_pairs_list.append((split_attribute_value, sub_data))
29
30     if flag_leaf:
31         # code that makes a leaf
32         node = MixedDecisionTree.Node()
33         node.isLeaf = True
34         node.classification = self._getMostCommonValue(data)
35         if current_depth == self.initial_depth:
36             node.isRoot = True
37         return node

```

Listing 3.4: Pre-pruning criteria of the MixedDecisionTree

The first two pre-pruning criteria are immediate and easy to understand. Line 2 of Listing 3.4 contains a simple condition: if the current depth of the tree exceeds the threshold given by the integer *max_depth*, the algorithm stops immediately and returns a leaf with the classification "most frequent label in the data up to that point".

Things are still very simple for *min_sample_split*: the condition in line 9 states that if the number of samples in the node is strictly lower than the integer threshold, that node will not

be split any further and becomes a leaf with the class label "most frequent class value in the current data".

min_sample_leaf is a little less intuitive at first (the condition is in line 25 of Listing 3.4): it is an integer threshold that checks, before the current node is even split, whether among the children that the split would create there is even one whose number of samples would be lower than the threshold. If so, the algorithm stops: the current node is not split, and a leaf is returned with the classification "most frequent label value in the current data".

3.4 Handling Continuous and Categorical Attributes

This section explains how continuous and categorical attributes are handled in various parts of the tree. The first part analysed is the one in the ID3 algorithm, implemented in the recursive method "_fit"; the second concerns another recursive method, "predict", whose purpose is to find out how well (or badly) the MDT generalizes.

Remember that a tree that "generalizes" well is a decision tree that correctly predicts and labels new instances, never seen before, thus keeping a good accuracy and F1-score.

Continuous and categorical attributes are also handled separately in the recursive method displayMDT, whose job is to print the MDT as text, level by level. Its implementation is given in Section 3.6 on page 36.

3.4.1 Attribute Handling in the fit Method

```
1  # ...
2  for pair in sub_data_pairs_list:
3      split_attribute_value = pair[0]
4      sub_data = pair[1]
5
6      # the recursive calls that grow the MDT in depth start here
7      child = self._fit(sub_data, remaining_features, current_depth + 1)
8
9      child.split_value = split_attribute_value
10     child.parent = root
11     child.depth = current_depth + 1
12     if is_numeric_dtype(split_attribute_value):
13         # the function is_numeric_dtype treats the value ''
```

```

14     # as a numeric attribute, not as a character or a string
15     if split_attribute_value != '?':
16         if split_attribute_value == 1.0:
17             root.children.insert(0, child) # insert as first child, the child in position
18                 0, left
19         elif split_attribute_value == 0.0:
20             root.children.insert(1, child) # insert as second child, the child in
21                 position 1, right
22         else:
23             raise Exception("fatal error... a non-numeric split attribute value arrived
24                 ...")
25     else:
26         raise Exception("Found a ? value that must be replaced with a valid value in the
27             .csv file")
28 else: # if instead the attribute is categorical, hence an object in Python
29     root.children.append(child)

```

Listing 3.5: Handling continuous and categorical attributes in the "fit" method

Listing 3.5 shows that the `MixedDecisionTree` was programmed to handle categorical attributes naturally (the opposite of `scikit-learn`, which naturally handles continuous attributes): a single line of code handles the child when the value of the attribute chosen as the best attribute to split is categorical.

Line 25 simply inserts the child node into the list of children of the root node, which now becomes the parent (of the child node) created earlier for the expansion. The remarkable thing is that if the root node, now the parent, has an attribute with 10 unique categorical values, its potential split consists of 10 children, which once again highlights the multi-way split capability of the MDT, which supports any number of children.

Lines 12 to 23 of the same Listing, instead, handle the value of a continuous attribute.

3.4.2 Attribute Handling in the predict Method

We have seen how continuous and categorical attributes are handled in the `_fit` method, but the `predict` method also has to handle the two kinds of attributes separately in order to work correctly. The purpose of the `predict` method is, given an unseen example as input, to return a class label for that example never seen before.

If the tree has been trained well, it will generalize just as well: it will make correct predic-

tions, keeping an excellent level of accuracy and F1-score.

```
1 def predict(self, X_test, y_test):
2     X_test[self.class_name] = y_test
3     examples = X_test.iloc[:, :-1].to_dict(orient="records")
4     predicted_list = list()
5     for i in range(len(X_test)):
6         predicted_list.append(self._predict(self.root, examples[i]))
7     return predicted_list
8 def _predict(self, root, example):
9     if root is None:
10        raise Exception("instantiate the tree and train it before calling the predict
11                           method")
12    if root.isLeaf:
13        return root.classification
14    if ' <= ' in root.split_condition:
15        temp_string = root.split_condition
16        # if there is a continuous split condition at the root:
17        whole_condition = re.split(' <= ', temp_string)
18        condition_name: str = str(whole_condition[0])
19        condition_number: float = float(whole_condition[1])
20        if len(root.children) > 1:
21            if float(example[condition_name]) <= condition_number:
22                return self._predict(root.children[0], example)
23            else:
24                return self._predict(root.children[1], example)
25        return self._predict(root.children[0], example)
26    else:
27        # if the split condition is categorical
28        example_value = example[root.split_condition]
29        for child in root.children:
30            if child.split_value == example_value:
31                return self._predict(child, example)
32        return self._predict(root.children[0], example)
```

Listing 3.6: Handling continuous and categorical attributes in the predict method

After training the tree we want to know how it generalizes; so the next step is to call the *predict* method, passing it unseen examples to classify.

Line 4 of Listing 3.6 creates a list that will contain the predictions made by the MDT. (To test the accuracy and the F1-score, we also need a test set, which by definition already has the labels with the "true" classifications, so that they can be compared with the predictions of our tree; obviously, if the MDT predicts every example correctly, we get 100% accuracy.) With a

for loop, in line 5, each example is passed to the recursive function `_predict`, which returns a prediction that is added to the list of predictions.

Let us see how `_predict`, defined in line 8 of Listing 3.6, works internally. As said several times, the MDT handles categorical attributes directly and naturally, because it was designed with exactly this purpose in mind; continuous attributes, instead, are not handled directly and must be made compatible, that is, the dataframe must undergo the transformation described in Section 3.5 on page 34.

Once the continuous attributes have been made compatible, they are treated as ordinary categorical attributes whose possible values can only be 0 and 1: in the implementation, they are treated as categorical attributes in which the lists of children held by the root and the internal nodes have only two children, the left child in position 0 of the list and the right child in position 1.

To recognize continuous attributes, it is enough to notice that they all have the structure: "original-column-name <= unique-continuous-value", so a condition like the one in line 13 of Listing 3.6 is sufficient.

Thanks to that condition, we know that the root node in question holds a continuous attribute which, through string transformations (lines 14 to 18), can be compared with the one of the unseen input examples: if the condition is true, the function `_predict` calls itself with more specific parameters, `root.children[0]` instead of `root`, so the recursion continues on the left child; otherwise, if the answer is negative, it continues on the right child, `root.children[1]`.

At the end of the recursion we inevitably reach a leaf node (the nodes that contain the classification) and, if the prediction is correct, we return the right classification; otherwise the tree has made a miss. To find out the accuracy, or the F1-score, it is enough to compare the list of all the predictions with the list of the exact ones.

Once the test has been carried out and the tree has shown an excellent accuracy and F1-score (so it generalizes well), it can finally be used for what it was really conceived for: **classifying**, that is, assigning a label to unseen examples for which we do **not** already have one.

The tree finally starts classifying instances, and it does so very quickly: once the tree has been trained, tested and found to work as it should, using it to classify examples never seen before (unseen) is a very fast and very efficient operation; its time complexity, as we already

know, is $O(w)$, where w is the maximum depth.

Since, and by now this should be set in stone, the inductive bias thanks to which a decision tree *learns* is precisely to keep the tree as shallow as possible, *w will be the minimum w for that tree.*

3.5 Transforming Continuous Attributes into Categorical Ones

How does the MDT change the dataframe to handle continuous attributes?

Suppose we have the **Iris** dataset, which consists of 5 columns: "Sepal-length", "Petal-length", "Sepal-width", "Petal-width", "class". The class column is the dependent variable Y , which contains all the class labels, so it is not considered in the transformation that follows.

Consider the first column, "Sepal-length", and, for simplicity, give it only 5 continuous values, for example 5.6, 4.8, 5.6, 6.4, 4.8. *The first step is to sort the values from the smallest to the largest, the second is to remove all the duplicates, if any, leaving only unique values in the starting column. After these operations, the column "Sepal-length" contains only 3 sorted unique values: 4.8, 5.6 and 6.4; the duplicate values are discarded.*

Then, for each unique value left in the column, an additional column is created with the name "*original-attribute-name* <= *unique-continuous-value*"; replacing the general expression with the names of the example we get "**Sepal-length <= 4.8**", and the values it contains are only 0 for no and 1 for yes, in this case 1 0 0; the next column is named "Sepal-length <= 5.6" and has the values 1 1 0, and the last column obviously contains all 1s: indeed, "Sepal-length <= 6.4" has "yes" for every answer, 1 1 1.

If a column has 40 continuous values and, after sorting them in ascending order, only 30 of them are unique, 30 additional columns are created following the rule that, by now, should be well understood.

Once the 30 columns have been created, *the original column from which the 30 unique values were derived is deleted*, and only the 30 new ones are kept (all with 0 and 1 as values, except the last one, which can safely be deleted since it contains only 1s), together with any columns of

categorical attributes, which are not touched and are left as they are. That is right: *categorical attributes are not changed*, because the MDT processes them directly, without any change to the dataframe made from outside.

The parts of the code that manipulate the dataframe to make the MDT compatible with continuous attributes are shown in the following Listing:

```
1  # start creating the columns
2  int_features = dataframe.select_dtypes(include=int)
3  float_features = dataframe.select_dtypes(include=float)
4  continuous_features = int_features.columns.tolist() + float_features.columns.tolist()
5  for continuous_feature in continuous_features:
6      X_train[continuous_feature] = X_train[continuous_feature].map(float)
7  cond_var_list = []
8  for feature in continuous_features:
9      training_classes = np.unique(X_train[feature])
10     for i in range(len(training_classes)):
11         attribute_name = feature + " <= " + str(training_classes[i])
12         cond_var_list.clear()
13         for value in feature_values_list:
14             if value <= training_classes[i]:
15                 cond_var_list.append(1.0)
16             else:
17                 cond_var_list.append(0.0)
18         to_be_added_df = pd.DataFrame({attribute_name: cond_var_list})
19         X_train = X_train.reset_index(drop=True).join(to_be_added_df.reset_index(drop=
                True), how='left')
20     del X_train[feature]
21  # we will use the training set as a non-fragmented copy of training_data
22  training_temp = X_train.copy(deep=True)
23  del X_train
24  X_train = training_temp
```

Listing 3.7: Code that interprets continuous attributes

Line 4 of Listing 3.7 creates a list containing the names, hence strings, of all the columns with continuous values, which lines 2 and 3 select among all the columns. Among all the columns means that the categorical columns are not selected and are therefore not part of the list.

The for loop in line 5 converts to float all the values of the columns concerned, whose names are in the list. This conversion ensures that there are *only two kinds of values in the dataframe*: **numeric** ones, native floats and integers converted to float, and **non-numeric**

ones, string values, which Python declares to be of type *object*. Every float value is interpreted as continuous by the MDT, every object value as categorical. Line 7 creates a list that will hold the zeros and ones that form the new columns built from the unique continuous values. For each unique continuous value there is an additional column.

The following lines finally contain the **code that creates the columns**: the for loop in line 8 runs over all the continuous attributes; in line 9, after sorting them in ascending order, all the unique values of the attribute of that iteration are collected. Line 11 creates the name of the new column from the original name of the attribute (the column) and its unique original value, and the innermost for loop, in line 13, fills the list created in line 7 with the correct values, 0 and 1. Line 18 uses the list of line 7 to create the actual column and add it to the dataframe; finally, the join in line 19 avoids the problem of memory fragmentation.

3.6 Visualizing the MDT

This part shows the text printout of the MDT. After training, the following code prints the trained tree, with all its depth levels, both on screen and to a file.

```
1 def displayMDT(self, root, file, width):
2     # set the number of spaces:
3     const = int(root.depth * width ** 1.5)
4     spaces = "-" * const
5     # if I am on the root node
6     if root.isRoot:
7         if root.isLeaf:
8             # I am the root of a tree that could not even be formed
9             root_string = "I am the root and at the same time a leaf, I can only give you
10                the classification " \
11                f"of the most common value, which is {root.classification}"
12             print(root_string)
13             file.write(root_string)
14         else:
15             # I am the root of a tree that is being formed, so I will be split
16             if self.criterion == 'entropy':
17                 root_string = f"Root at depth:{root.depth} with Split Condition: |{root.split
18                    _condition}| " \
19                    f"with info_gain = {root.gain}"
20                 print(root_string)
21                 file.write(root_string)
```

```

20     elif self.criterion == 'gini':
21         root_string = f"Root at depth: {root.depth} with Split Condition: |{root.
                split_condition}| " \
22                 f"with gini_gain = {root.gain}"
23         print(root_string)
24         file.write(root_string)
25     else:
26         raise Exception("Wrong criteria input, possible choices are: 'entropy' or '
                gini'")
27 else:
28     # if I am a leaf
29     if root.isLeaf:
30         if is_numeric_dtype(root.split_value):
31             if root.split_value == 1.0:
32                 word: str = "yes"
33             elif root.split_value == 0.0:
34                 word: str = "no"
35             else:
36                 raise Exception("valid values at this point can either be 1.0 or 0.0")
37         else:
38             word: str = str(root.split_value)
39
40         string_leaf = 'depth:', str(root.depth), f"|{spaces} if {root.parent.split_
                condition} == " \
41                 f"{word} --> Classification: {root.classification}"
42         print(string_leaf)
43         file.writelines(string_leaf)
44     # if I am an internal node
45     else:
46         if self.criterion == 'entropy':
47             string_not_leaf = 'depth:', str(root.depth), \
48                 f"|{spaces} Split Condition: |{root.split_condition}| " \
49                 f"with info_gain = {root.gain}"
50             print(string_not_leaf)
51             file.writelines(string_not_leaf)
52         elif self.criterion == 'gini':
53             string_not_leaf = 'depth:', str(root.depth), \
54                 f"|{spaces} Split Condition: |{root.split_condition}| " \
55                 f"with gini_gain = {root.gain}"
56             print(string_not_leaf)
57             file.writelines(string_not_leaf)
58         else:
59             raise Exception("Wrong criteria input, possible choices are: 'entropy' or '
                gini'")
60     if len(root.children) > 0:

```

```

61     for child in root.children:
62         assert (child.parent == root)
63         self.displayDT(child, file, width)

```

Listing 3.8: Code that prints the MDT to stdout and to a file

Lines 3 and 4 of Listing 3.8 create an offset, which depends on configurable parameters, to draw the depth levels of the tree. The function carefully distinguishes the cases in which the impurity is computed with Entropy or with the Gini index; in the same way, great care was taken to distinguish categorical attributes from continuous ones. The pandas function `"is_numeric_dtype()"` helps to achieve this.

When there is a continuous attribute (which is certainly a float, because even if it was originally an int, that int would have been converted to float, so that the tree has only two types: *numeric*, that is float, and *non-numeric*, that is object), the function `is_numeric_dtype()` returns true; for a categorical attribute it returns false.

Here is a printout of the MDT after running "fit" on the training data of the **Iris** dataset:

```

Root at depth:0 with Split Condition: |PetalLengthCm <= 1.9| with info_gain = 0.9149263727797274
depth:1|----- if PetalLengthCm <= 1.9 == yes --> Classification: Iris-setosa
depth:1|----- Split Condition: |PetalWidthCm <= 1.7| with info_gain = 0.6325253077661508
depth:2|----- Split Condition: |PetalLengthCm <= 5.1| with info_gain = 0.1952667279783059
depth:3|----- Split Condition: |SepalWidthCm <= 2.5| with info_gain = 0.10973196301060684
depth:4|----- Split Condition: |PetalLengthCm <= 4.4| with info_gain = 0.4464393446710155
depth:5|----- if PetalLengthCm <= 4.4 == yes --> Classification: Iris-versicolor
depth:5|----- Split Condition: |SepalLengthCm <= 6.0| with info_gain = 0.91829583405448
depth:6|----- if SepalLengthCm <= 6.0 == yes --> Classification: Iris-virginica
depth:6|----- if SepalLengthCm <= 6.0 == no --> Classification: Iris-versicolor
depth:4|----- if SepalWidthCm <= 2.5 == no --> Classification: Iris-versicolor
depth:3|----- if PetalLengthCm <= 5.1 == no --> Classification: Iris-virginica
depth:2|----- Split Condition: |PetalLengthCm <= 4.8| with info_gain = 0.11901271691308317
depth:3|----- Split Condition: |SepalLengthCm <= 5.9| with info_gain = 0.9182958340544896
depth:4|----- if SepalLengthCm <= 5.9 == yes --> Classification: Iris-versicolor
depth:4|----- if SepalLengthCm <= 5.9 == no --> Classification: Iris-virginica
depth:3|----- if PetalLengthCm <= 4.8 == no --> Classification: Iris-virginica
accuracy_score = 0.98
f1_score = 0.98

```

Figure 3.1: Example of the text printout of the MDT on the Iris dataset

3.7 The Three Modes of the MixedDecisionTree

As anticipated in the previous sections, the MDT has three operating modes: **classic**, **median** and **binned**. The user chooses the mode by assigning the string 'classic', 'median' or 'binned' to the parameter `self.tree_mode` when instantiating the tree. At that moment the user can also choose the number of bins, used only by the median and binned modes.

3.7.1 Classic Mode

The classic behaviour of the tree: it offers the highest possible precision for continuous attributes, since, given a column of continuous values, the dataframe is changed to create an additional column for every non-duplicate continuous value. Obviously, all this precision comes at a considerable cost in time and space.

```
1 def _classic_mode(self, dataframe):
2     dataframe.columns = dataframe.columns.str.replace(' ', '_')
3     dataframe[self.class_name] = dataframe[self.class_name].map(str)
4
5     # reduce the rows of the dataset if they exceed the cut_threshold
6     if len(dataframe) > self.cut_threshold:
7         dataframe = dataframe.drop(labels=range(self.cut_threshold, len(dataframe)), axis
8                                     =0)
9
10    y = dataframe[self.class_name].map(str)
11    columns = [c for c in dataframe.columns if c != self.class_name]
12    X = dataframe[columns]
13    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state
14                                                        =43, stratify=y)
15
16    # ... lines of code that create the columns...
17
18    X_train = X_train.reset_index(drop=True)
19    y_train = y_train.reset_index(drop=True)
20    return X_train, X_test, y_train, y_test
```

Listing 3.9: Code of the classic mode

In Listing 3.9, line 2 replaces every space between the words of an attribute name with an underscore; line 3 forces the conversion to string of the dependent variable Y (the class column, the one that contains the class labels).

Line 6 shows how the threshold parameter `cut_threshold` works; it can be used only in classic mode, the most expensive in terms of space and time, and its purpose is to reduce the total number of rows of the dataset.

Line 10 selects all the columns of the dataset except the class column, and assigns them to the variable X. The class column, instead, is stored in the variable Y, which, together with X, is fed to the scikit-learn function "train_test_split". This function returns all the components

needed to train the tree with the training set, `X_train` and `y_train`, and to test it with the test set, `X_test`, `y_test`.

3.7.2 Median Mode

The median mode has exactly the same code as the classic mode, except for the following lines:

```
1  # ....
2  y = dataframe[self.class_name].map(str)
3  columns = [c for c in dataframe.columns if c != self.class_name]
4  X = dataframe[columns]
5  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state
    =43, stratify=y)
6
7  # careful: cut and mid must be applied ONLY to the training set
8  for feature in continuous_features:
9      X_train[feature] = pd.cut(X_train[feature], self.number_of_bins)
10     X_train[feature] = X_train[feature].apply(lambda x: x.mid)
11
12  # ... lines of code that create the columns, omitted
13  # ... because they are in listing 3.7
14
15  # reset the indices of the training set with drop=True
16  X_train = X_train.reset_index(drop=True)
17  y_train = y_train.reset_index(drop=True)
18
19  return X_train, X_test, y_train, y_test
```

Listing 3.10: Code of the median mode

The difference between Listing 3.10, of the median mode, and Listing 3.9, of the classic mode, is immediately clear: there is no code concerning the threshold parameter `cut_threshold`, and, in line 9, the cut is applied with a given number of bins (the number of bins to use for the median and binned modes is also passed when the tree is instantiated) only to the training set, while in line 10, again on the training set only, the mid is applied through a lambda expression.

These lines of code considerably lighten the creation of the columns that follows, which uses exactly the same code as the classic mode; the only difference is that the number of bins puts an upper bound on how many unique values the column will have.

If, for example, the column of continuous values "Petal-length" has 150 floating-point val-

ues (of type float, which Python treats as continuous values), all different from each other, in classic mode we would have to create 150 columns, one for every unique value, whereas here, in median mode, the number of columns created equals the number of bins. This means that the number of unique values also equals the number of bins, and these values are computed to be evenly spread over "number-of-bins" midpoints.

This operation is applied only to the training set; the values of the test set stay unchanged. We can therefore see how much space and time this mode saves, without even losing many percentage points of Accuracy, or F1-score, compared with the very precise but very inefficient classic mode, which puts no upper bound on the unique values in each column.

3.7.3 Binned Mode

The binned mode also uses `pandas.cut`, but without applying the `mid` through the `lambda` expression seen in line 10 of Listing 3.10. The purpose of this mode is to turn a column of continuous values into a column of strings, where the strings are intervals of values. The precision of the intervals is proportional to the number of bins. This mode must be applied to the test set *as well*, otherwise there would be no match in the prediction phase.

As line 10 of snippet 3.11 below shows, the `cut` is applied to the whole dataframe, from which, only afterwards, the training set and the test set are obtained through the library function `"train_test_split()`".

```
1 def _binned_mode(self, dataframe):
2     dataframe.columns = dataframe.columns.str.replace(' ', '_')
3     dataframe[self.class_name] = dataframe[self.class_name].map(str)
4     int_features = dataframe.select_dtypes(include=int)
5     float_features = dataframe.select_dtypes(include=float)
6     continuous_features = int_features.columns.tolist() + float_features.columns.tolist()
7
8     # the whole core of the binned mode is here:
9     for feature in continuous_features:
10         dataframe[feature] = pd.cut(dataframe[feature], self.number_of_bins)
11
12     y = dataframe[self.class_name].map(str)
13     columns = [c for c in dataframe.columns if c != self.class_name]
14     X = dataframe[columns]
15     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state
        =43, stratify=y)
```

16

17 `return X_train, X_test, y_train, y_test`

Listing 3.11: Code of the binned mode

In Listing 3.11, the first lines work exactly like the other modes: all the columns with continuous values are selected, leaving the columns with categorical values unchanged, and *the cut is applied to both the training set and the test set*, as line 10 makes clear.

The result is a set of intervals of values that Python sees as intervals of strings (of type object), and the precision of these intervals depends on the number of bins with which the tree was instantiated.

In binned mode, therefore, the continuous has been turned directly into the categorical, without the need to create additional columns as in classic or median mode.

Needless to say, this is the cheapest mode in terms of resources such as time and space; however, unlike median, which is the perfect compromise between performance and computational cost (as will become clear in the next chapter), the binned mode suffers somewhat too much from losses in accuracy and F1-score.

If I had to train the MDT with unlimited time and space, I would use the classic mode without any doubt; but if time and space are limited, I would use the median mode without the slightest hesitation.

Chapter 4

Experiments

You can have data without information, but you cannot have information without data. (Daniel Keys Moran)

4.1 Preliminary Information

Besides writing the whole MDT with all its modes, other programs had to be written to carry out the tests presented in the rest of this chapter.

The first program, called **SKLEARN_verify.py**, tests the scikit-learn tree on all the datasets considered in this thesis and produces tables, saved in the *Results* folder, that contain important metrics such as accuracy, F1-score and execution time as parameters vary: *impurity criterion, tree mode, cut_threshold, number of bins, max_depth, min_sample_split and min_sample_leaf*. This program also **adapts the three modes** classic, median and binned to the scikit-learn tree, by suitably manipulating the dataframe *from outside*.

As for the purely "visual" side of the tree, the first program, besides printing the scikit-learn tree as text, also draws it graphically. This means that the internal nodes, the edges, the classification leaves and other useful information can be visualized; these printouts are saved in a dedicated folder called *Visual_sklearn*.

The second program is called **MDT_verify.py**; it runs the same kind of tests as the first one and generates the same tables, which are saved in the *Results* folder. The only difference is that the second program only produces text printouts, like the one in Figure 3.1 on page 38.

These printouts are saved in the *Visual_mixedDT* folder.

Other programs take as input the tables in .csv format produced by the first and the second test program and generate as output all the tables, histograms and overlaid plots analysed in this chapter.

4.2 The Datasets

Data are the real protagonists of everything that concerns Machine Learning, so it is time to get to know the datasets used in this thesis:

Dataset	Rows	Columns	Classes	Attribute type
Auto	392	7	3	mixed
Breast-Cancer	271	8	2	mixed
Carseats	279	9	2	mixed
Churn	500	13	2	mixed
Diabetes	393	8	2	continuous
HouseVotes	434	16	2	categorical
Iris	150	4	3	continuous
Parkinsons	100	22	2	continuous
Prostate	97	7	2	continuous
Sacramento	932	6	3	mixed

Figure 4.1: All the datasets considered in this thesis

The first dataset is called **Auto** and, as the name suggests, it is used to classify cars by weight, year, engine displacement, time to accelerate from 0 to 100 km/h, engine power in horsepower and so on. It is interesting above all because it has 3 classes rather than the usual two.

The second dataset, **Breast-Cancer**, uses many clinical data to predict whether the person carries the disease named by the dataset or not. Both this dataset and the previous one are

mixed datasets, that is, they contain at least one categorical column and at least one column of continuous data.

Carseats is the name of the third dataset: with our decision tree we predict the sales of car seats on the basis of many pieces of information, such as the advertising that was carried out, previous sales by area and so on.

Churn is another interesting dataset; it has 500 rows, as the table in Figure 4.1 shows, and we predict whether a former customer, who left a certain service provided by a given company, is willing to come back and become a customer again.

With the **Diabetes** dataset, as with "Breast-Cancer", we predict whether the person, with all the clinical data relevant to the case, could develop the disease described by the dataset or not.

Iris is practically the dataset par excellence, cited by many textbooks on our subject. It is interesting because it has 3 classes and not the usual two, and it is a fully continuous dataset: every column consists of continuous values, and we predict, from data such as sepal length or petal length, which of the three categories the iris belongs to.

Two other datasets that use medical measurements are **Parkinsons** and **Prostate**. With the first we predict whether the person suffers from the disease named by the dataset or not, with the second whether the person may have prostate cancer or not.

Finally, we consider a real estate dataset: **Sacramento**. Its data were taken from a file of transactions over a period of only 5 days. The transactions involve 932 houses (so it has 932 rows) and their sale prices in the city of Sacramento.

4.3 Tests

This section compares the MDT and the scikit-learn DT, supported by tables, histograms and overlaid plots that make them easier to read and understand. All the comparisons are based on varying some parameters while keeping others constant. The parameters in play are the pre-pruning ones we already know, together with the chosen impurity criterion (Entropy or Gini Index) and the number of bins (20 or 11).

First of all, we tested the ACCURACY reached by the two decision trees in the three modes

classic, median and binned, already presented in the third chapter, in particular **configurations of interest**. These configurations are listed in the table of Figure 4.4 on page 51.

Each configuration is determined by a particular choice of the *hyper-parameters*, which are *max-depth*, with values in the set $\{2, 3, 4, 5, 6, 7, 8, 25, 50\}$, *min-sample-split*, with values in the set $\{2, 3, 5, 10, 25, 50\}$, and *min-sample-leaf*, with values in the set $\{1, 3, 5, 10, 25, 50\}$, by a choice of the *impurity criterion* (Entropy or the Gini index) and by a choice of the *number of bins*.

We will also look in more detail at overlaid plots for each of the configurations considered; these plots are on pages 52 and 53.

In the first table, in Figure 4.2, each dataset has a privileged configuration in which we can study how the accuracy behaves as the parameters mentioned above vary. Histograms follow that make the tables themselves easier to read.

The second table, in Figure 4.6 on page 56, has no configurations of particular interest: it shows how the *maximum* ACCURACY varies for each dataset, regardless of particular parameter configurations. So, for each dataset and each mode, we record the peak ACCURACY value of the MDT and of the scikit-learn tree.

Then, in the same order, come the tables, histograms and overlaid plots for the second metric considered in this thesis: the F1_SCORE.

4.3.1 Metric Considered: ACCURACY

Dataset	Accuracy					
	Classic		Median		Binned	
	MDT	SKL	MDT	SKL	MDT	SKL
Auto	0.7385	0.7538	0.6923	0.7154	0.6231	0.7154
Breast-Cancer	0.6556	0.6778	0.6556	0.7000	0.6556	0.7222
Carseats	0.9355	0.9355	0.9032	0.8817	0.8710	0.8495
Churn	0.8424	0.8242	0.7939	0.7818	0.6061	0.7697
Diabetes	0.8077	0.8000	0.7000	0.7000	0.6615	0.7077
HouseVotes	0.9583	0.9583	0.9583	0.9583	0.9583	0.9583
Iris	0.9800	0.9800	0.9800	0.9800	0.9200	0.9200
Parkinsons	0.7576	0.7273	0.8182	0.6700	0.7576	0.6700
Prostate	0.8182	0.7879	0.7879	0.7879	0.7879	0.6970
Sacramento	0.9318	0.9351	0.9351	0.9351	0.9253	0.9318
Mean	0.8426	0.8380	0.8224	0.8110	0.7766	0.7942
Std	0.1074	0.1068	0.1175	0.1182	0.1357	0.1101
Wins	7	6	8	7	5	7

Figure 4.2: Comparisons on the Accuracy metric in all modes

The table in Figure 4.2 gives a fairly exhaustive overview of how the MDT and the scikit-learn tree perform in ACCURACY under the same configuration. These configurations can be found in the table on page 51. For example, for the dataset "Auto" the configuration of particular interest chosen is the following: Impurity = Gini Index, Number of Bins = 20, max_depth = 4,

`min_sample_leaf = 5`, while the pre-pruning hyper-parameter `min_sample_split` varies. Why `min-sample-split` varies will become clearer by looking at the overlaid plots on pages 52 and 53.

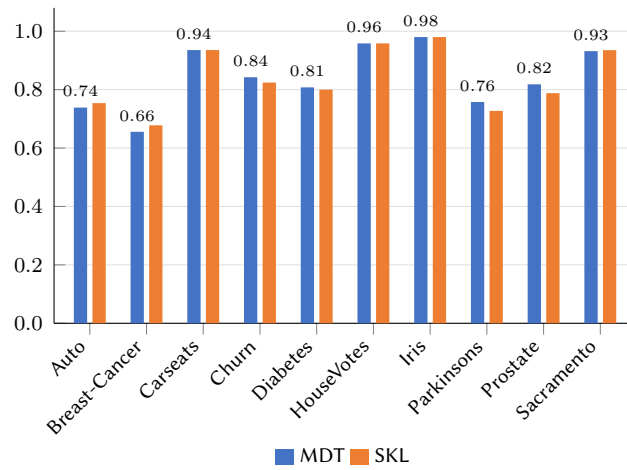
The cells in green mark a win of one tree over the other, or a tie (a tie gives a win to each tree). A tree "wins" over the other when, in the same mode, it performs better than the other on the metric being considered.

The last rows of the table contain the mean, the standard deviation and the wins of the MDT against the scikit-learn tree.

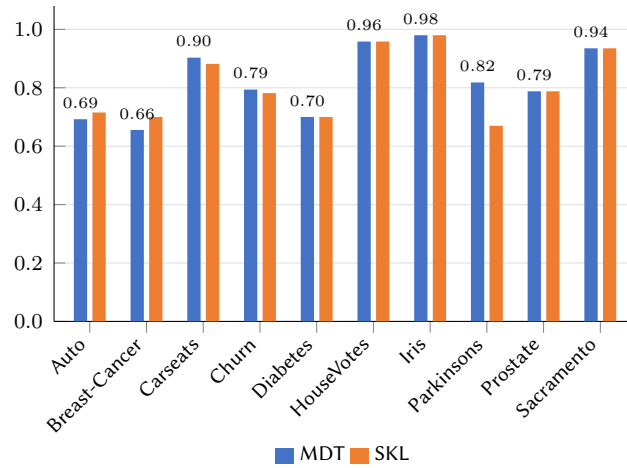
The classic mode, as all the comparisons will show, is the one implemented to deliver the highest possible ACCURACY, paid for dearly, however, in terms of space and time; the median mode, instead, can in some cases match the precision of the classic mode at a much lower computational cost. The binned mode is the coarsest: it is computationally even lighter than the median mode, but its performance is far lower.

To highlight the comparisons and see them more clearly, the following charts show the direct comparison between the two trees, under the same configuration and mode:

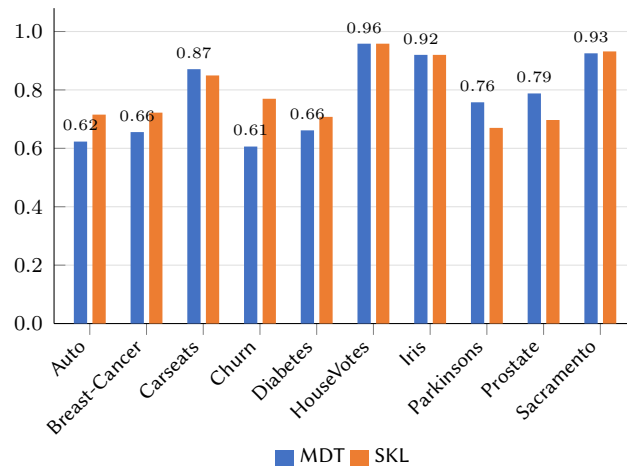
ACCURACY – CLASSIC mode



ACCURACY – MEDIAN mode



ACCURACY – BINNED mode



To sum up the total wins, the overall arithmetic mean and the total standard deviation,

here is the following small table:

	MDT	SKLEARN
Mean	0.8139	0.8144
Std	0.0143	0.0058
Wins	20	20

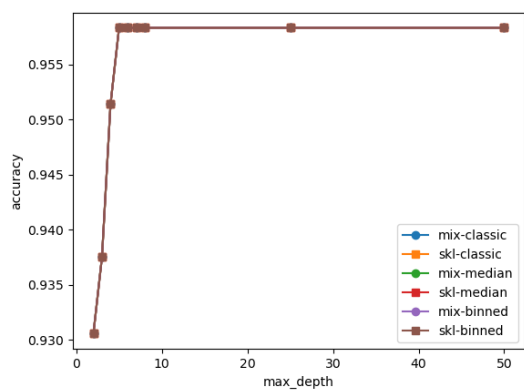
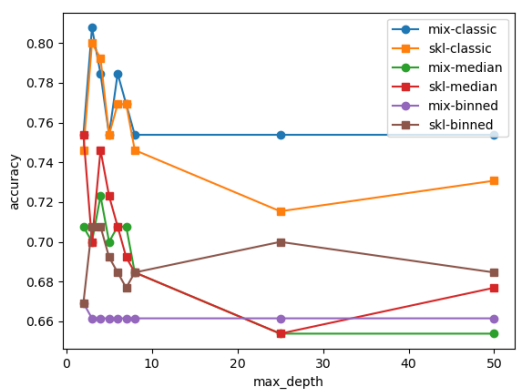
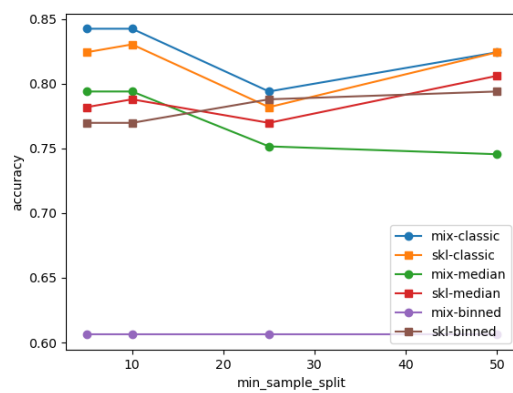
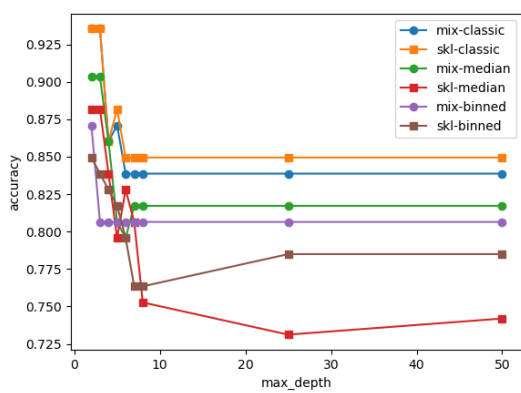
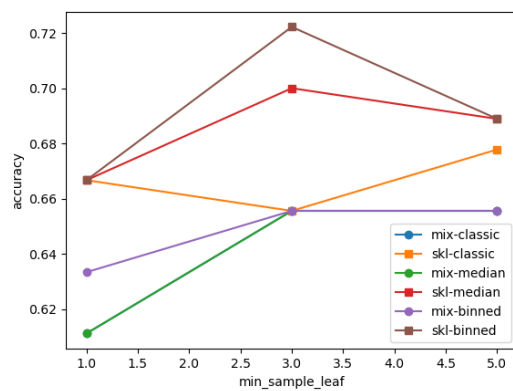
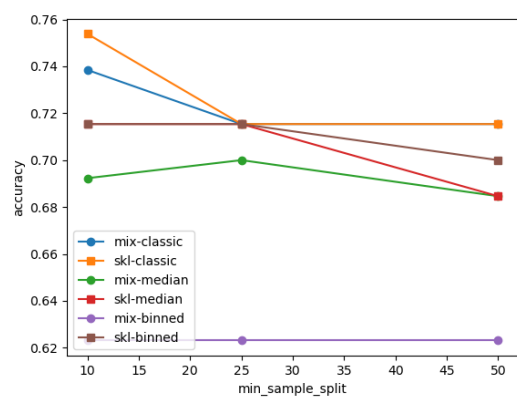
Figure 4.3: Final comparisons between the MDT and the scikit-learn tree

Table 4.3 shows that the total standard deviation is far larger for the MDT than for the scikit-learn DT. This is due to the poor ACCURACY results of the MDT in binned mode: indeed, the table in Figure 4.2 on page 47 shows, for every dataset, a rather disappointing ACCURACY value for the MDT in binned mode. These values tend to lie far from the arithmetic mean, which increases the standard deviation.

We said that each dataset has its own "privileged" configuration of hyper-parameters: two of them are fixed, while the third can vary, that is, take several values in a given range. The table below describes the various configurations:

Dataset	Accuracy				
	Impurity	Bins	Depth	MSS	MSL
Auto	Gini	20	4	varies	5
Breast-Cancer	Entropy	11	25	10	varies
Carseats	Entropy	11	varies	2	1
Churn	Entropy	11	6	varies	3
Diabetes	Gini	20	varies	2	1
HouseVotes	Gini	20	varies	5	3
Iris	Entropy	20	varies	2	1
Parkinsons	Gini	11	varies	3	1
Prostate	Gini	11	7	varies	3
Sacramento	Entropy	20	varies	3	1

Figure 4.4: The configurations adopted to test some ACCURACY and F1_SCORE scenarios ("varies" marks the hyper-parameter that varies; MSS = min_sample_split, MSL = min_sample_leaf)



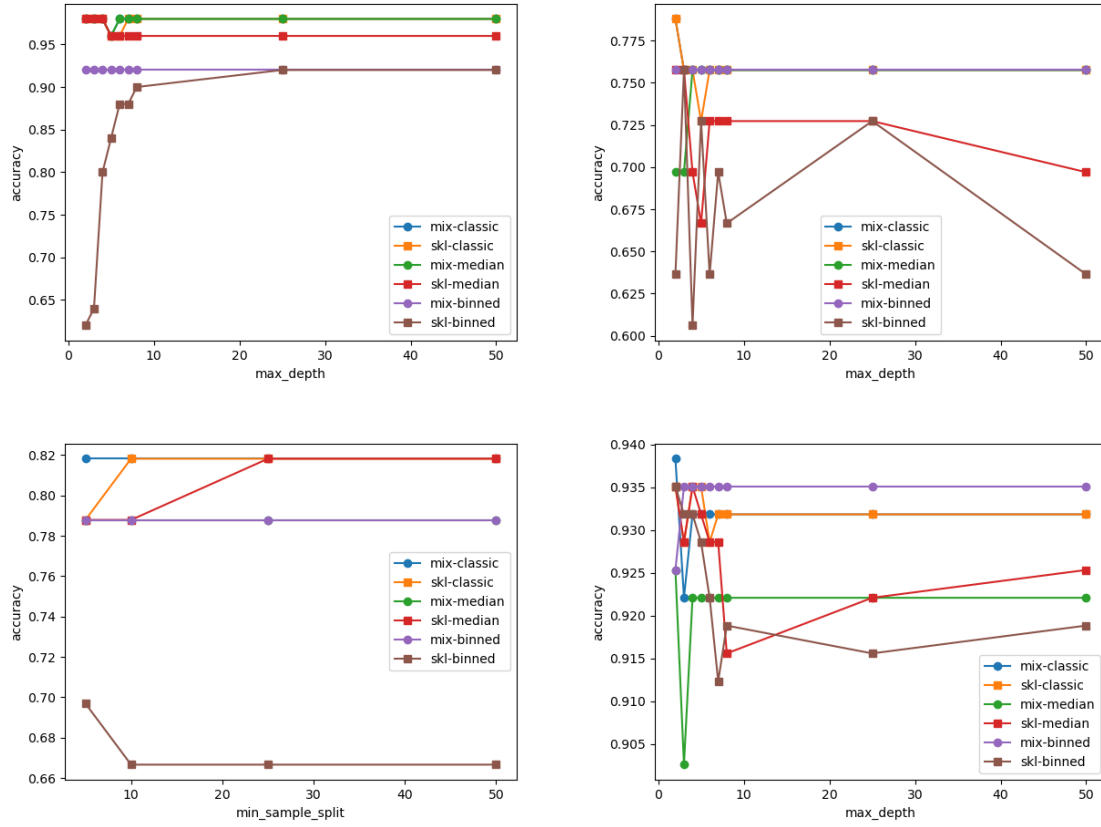


Figure 4.5: Each dataset, in the same order as the table in Figure 4.2 on page 47, is associated with a given configuration of hyper-parameters. The x-axis shows the parameter that varies, the y-axis the metric chosen for the comparisons

Consider the overlaid plots in Figure 4.5, where each plot represents a dataset, in the order of the table in Figure 4.1.

Each of them shows how the ACCURACY behaves as one of the pre-pruning parameters varies. Recall that they are the following: *max-depth*, *min-sample-split* and *min-sample-leaf*, with the particularity that every value of min-sample-split, for the model to make sense, must be strictly greater than the corresponding value of min-sample-leaf; that is, we cannot have $mss = 3$ and $m sl = 5$.

Starting from the first overlaid plot, for the dataset "Auto", it shows the ACCURACY values as the parameter min-sample-split varies, keeping constant all the other parameters on which our comparisons are based (pre-pruning parameters, impurity criterion and number of bins):

in this case max-depth is fixed at 4, min-sample-leaf at 5 and the number of Bins at 20. The impurity criterion chosen is the Gini Index.

For this plot, as said above, only the values of min-sample-split strictly greater than those of min-sample-leaf make sense: if msl is fixed at 5, min-sample-split can only take the values 10, 25 and 50. Indeed, looking at the first overlaid plot, each curve, one for each of the three modes of the two trees, has only three points, for a total of six curves, each associated with its own color.

Each of the six colors is associated with a mode of one of the two trees: for the MDT, light blue, green and purple stand for the classic, median and binned modes respectively; the scikit-learn DT is associated with orange, red and brown for the same modes.

Still looking at the first plot, it is clear that the binned mode is handled poorly by the MDT (the purple curve): it has the lowest accuracy of all, whereas the same mode applied to the scikit-learn DT performs fairly well, even beating the median mode of the MDT. As for the classic mode, both trees behave similarly and reach practically the same score.

The configuration chosen for the second plot, for the Breast-Cancer dataset, where min-sample-leaf varies, is also very special: it shows a crushing win of the binned mode, applied however to the scikit-learn DT (the brown curve), which even beats its own median and classic modes. In this case too there are only three points of variation.

The third plot, for the Carseats dataset, shows the situation when max-depth varies while mss and msl are fixed. In this case, varying max-depth with mss at 2 and msl at 1, the curves have nine points of variation. The peculiarity of the third plot is that the median mode of the scikit-learn DT performs worse than the binned mode of the MDT, which by now has turned out to be its weak point.

Quite curious is the situation in the sixth overlaid plot, for the HouseVotes dataset, where, as max-depth varies, all six curves follow the same trend. They clearly lie completely on top of each other.

The seventh plot, for the very famous Iris dataset, shown on page 53, shows the typical, predictable situation in which the median and classic modes coincide, resulting in high ACCURACY values, while the binned mode, as usual, performs worst for both the MDT and the scikit-learn DT.

The ninth plot, for the Prostate dataset, follows the same pattern as the seventh, with the binned modes trailing behind, this time especially the one of the scikit-learn DT, which reaches really low values, while the classic and median modes perform excellently for both decision trees.

The tenth plot, for the Sacramento dataset, shows a comeback of the MDT: its binned mode reaches the highest ACCURACY values, leaving aside the highest peak, reached at a single point by the classic mode, again of the MDT. The median and binned modes of the scikit-learn DT are the ones that perform worst, reaching the lowest values of the ACCURACY metric.

Now comes the part that may interest the reader most: given the same datasets, instead of choosing a specific configuration for each of them, we consider only the absolute maxima for each mode. The table in Figure 4.6, on the next page, shows the results of the comparison:

Dataset	Max accuracy					
	Classic		Median		Binned	
	MDT	SKL	MDT	SKL	MDT	SKL
Auto	0.8000	0.8077	0.7769	0.8000	0.7154	0.8231
Breast-Cancer	0.6700	0.7333	0.6700	0.7444	0.6700	0.7444
Carseats	0.9355	0.9355	0.9355	0.9032	0.8817	0.8710
Churn	0.8485	0.8545	0.8061	0.8545	0.7212	0.8485
Diabetes	0.8308	0.8231	0.7846	0.7923	0.7385	0.8231
HouseVotes	0.9583	0.9583	0.9583	0.9583	0.9583	0.9583
Iris	0.9800	1.0000	1.0000	1.0000	0.9600	0.9800
Parkinsons	0.7879	0.8182	0.9091	0.8788	0.7576	0.8182
Prostate	0.8485	0.8788	0.9091	0.9091	0.8182	0.9091
Sacramento	0.9416	0.9448	0.9416	0.9513	0.9351	0.9513
Mean	0.8601	0.8754	0.8691	0.8792	0.8156	0.8727
Std	0.0908	0.0788	0.0986	0.0775	0.1046	0.0716
Wins	3	9	5	8	2	9

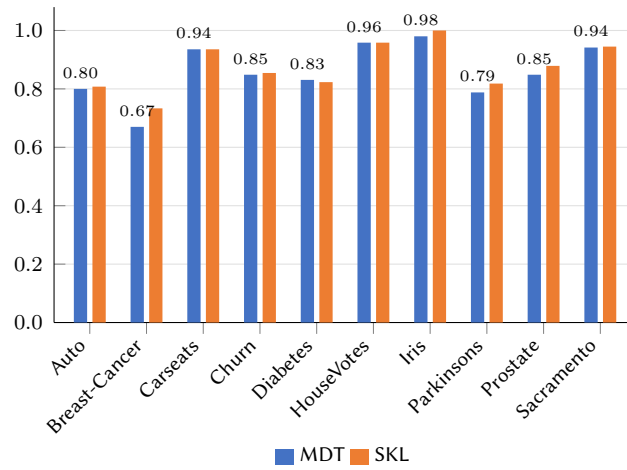
Figure 4.6: For each mode, only the maxima of the accuracy are considered

Note that, as in the previous table (on page 47), which considered the configurations, the green cells mean that tree x, in the same mode, has a higher accuracy than tree y. Needless to say, the MDT suffers a little in this table, which considers only the maxima; however, as the reader will immediately notice, in most cases the difference is really minimal.

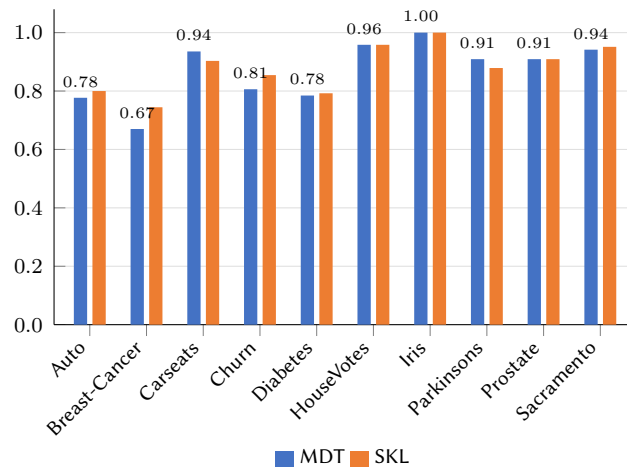
The histograms below make the direct comparison between the two decision trees easier,

measuring the maximum accuracy in the various modes:

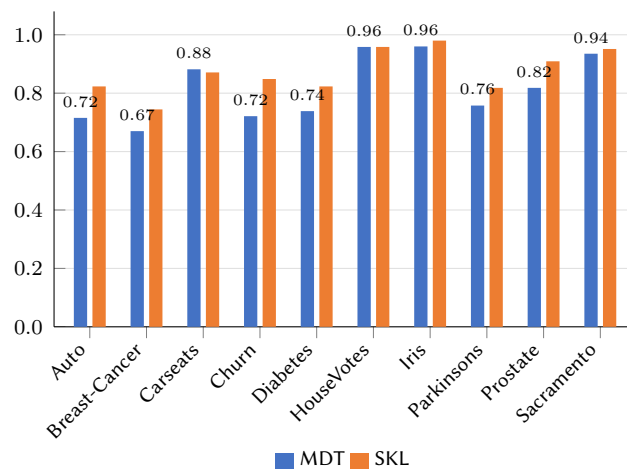
MAX ACCURACY – CLASSIC mode



MAX ACCURACY – MEDIAN mode



MAX ACCURACY – BINNED mode



The largest differences occur in the binned mode, as is easy to see from the table in Figure 4.6 and the histograms that follow it. In the other modes, instead, the differences are minimal and, in some cases, the MDT even beats the scikit-learn tree in ACCURACY: for example, on the "Carseats" dataset the median mode of the MDT exceeds the accuracy of the scikit-learn DT by as much as 0.0323, the first being 0.9355 and the second 0.9032.

Next comes the table in Figure 4.7, which reports the total number of wins of the MDT against the scikit-learn DT (and here the latter is clearly the winner, with 26 wins against only 10 for the MDT), together with the overall means and total standard deviations of the two decision trees.

When testing the MAX ACCURACY, the MDT again has a larger total standard deviation than the scikit-learn DT. In this case too, the blame can be put on the poorly performing binned mode of the MDT.

	MDT	SKLEARN
Mean	0.8483	0.8758
Std	0.0069	0.0038
Wins	10	26

Figure 4.7: Mean of the means and total standard deviations of the two trees in max accuracy

Let us now directly compare the histograms on the next page, which refer to the two main tables examined so far: the first in Figure 4.2 on page 47, where each dataset had its own configuration of parameters, and the second in Figure 4.6 on page 56, which shows only the maximum accuracy. For simplicity, only the most representative mode is chosen, that is, the classic mode for both.

The differences are clearly minimal; it follows that, whatever particular configurations are chosen, the accuracy values always tend to their peak, except in a few cases where the difference is not negligible: for example, on the "Auto" dataset the table with the configurations reaches a result about 0.06 lower than the table of the maxima for both trees.

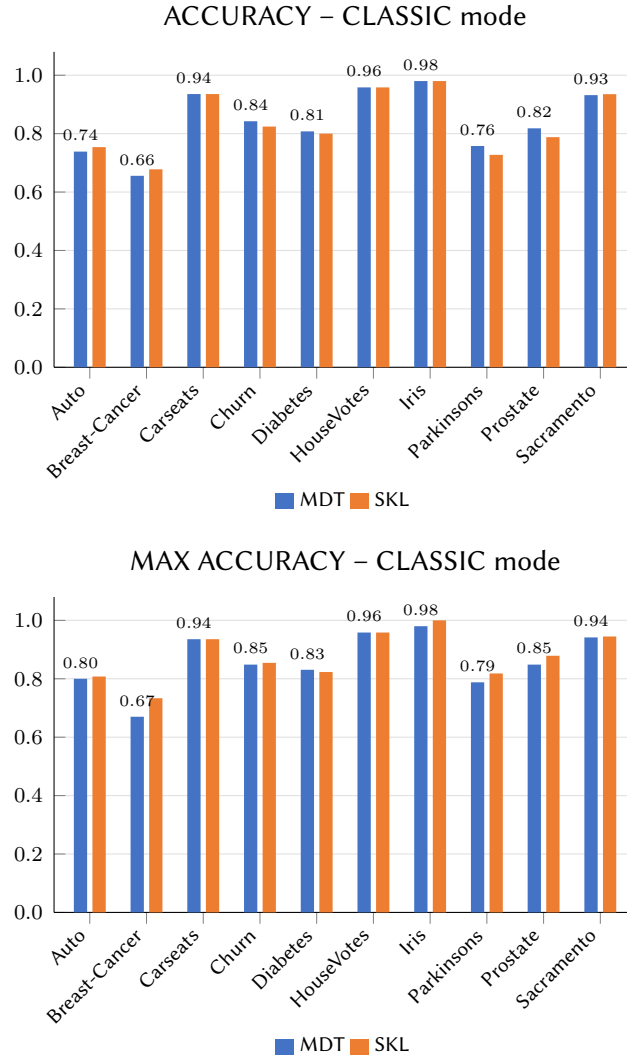


Figure 4.8: The histograms above show the direct comparison between the classic mode in which each dataset had a privileged configuration and the one in which only the maxima of each mode count

4.3.2 Metric Considered: F1_SCORE

Everything done so far with the ACCURACY metric is now repeated: the same configurations used before are used again, and we will end with the table that considers only the maximum F1_SCORE values.

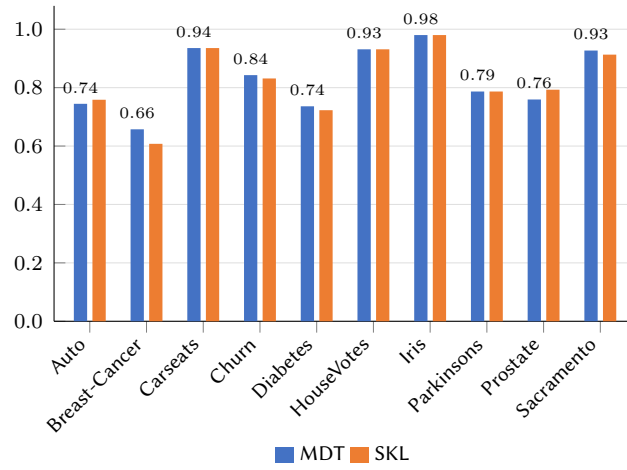
As can be seen by looking at all the tables and charts based on the F1_SCORE metric, they follow, with very small variations, the same trends we have already seen for the ACCURACY;

we therefore chose to spare the reader heavy and redundant descriptions, trusting that the tables, histograms and plots (all presented in the same way) are immediately understandable.

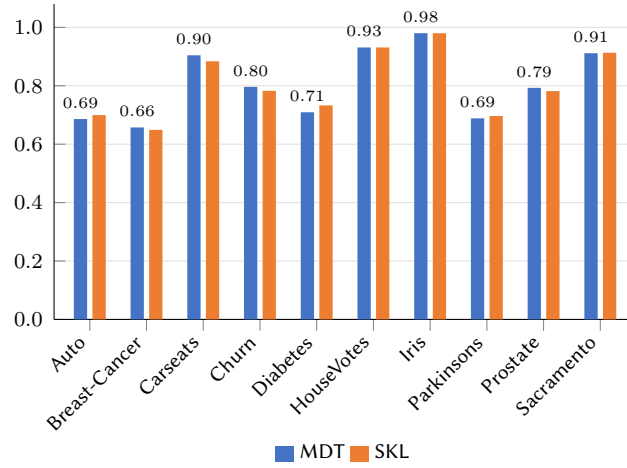
Dataset	F1 score					
	Classic		Median		Binned	
	MDT	SKL	MDT	SKL	MDT	SKL
Auto	0.7446	0.7585	0.6861	0.6995	0.6495	0.7296
Breast-Cancer	0.6573	0.6076	0.6573	0.6489	0.6573	0.6749
Carseats	0.9355	0.9355	0.9043	0.8838	0.8720	0.8710
Churn	0.8428	0.8314	0.7962	0.7828	0.6191	0.7666
Diabetes	0.7361	0.7227	0.7093	0.7328	0.6513	0.6789
HouseVotes	0.9312	0.9312	0.9312	0.9312	0.9312	0.9312
Iris	0.9800	0.9800	0.9800	0.9800	0.9211	0.9210
Parkinsons	0.7867	0.7867	0.6884	0.6964	0.7478	0.6323
Prostate	0.7594	0.7929	0.7929	0.7817	0.6944	0.6970
Sacramento	0.9270	0.9130	0.9114	0.9130	0.9196	0.9146
Mean	0.8301	0.8260	0.8057	0.8050	0.7663	0.7817
Std	0.1087	0.1154	0.1186	0.1145	0.1299	0.1163
Wins	8	6	6	6	5	6

Figure 4.9: Comparisons on the F1_SCORE metric on the same configurations seen in Figure 4.2

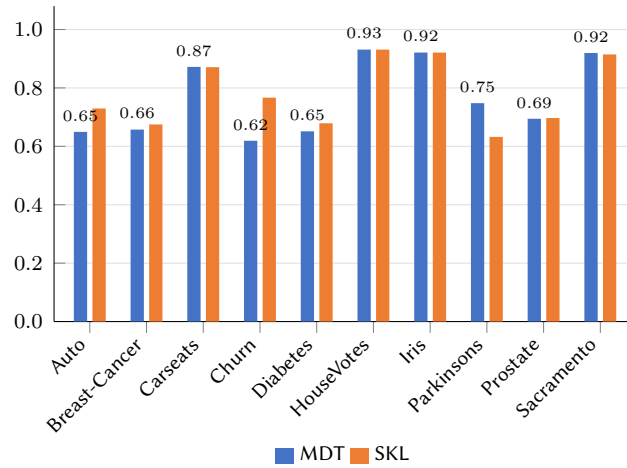
F1 SCORE – CLASSIC mode



F1 SCORE – MEDIAN mode

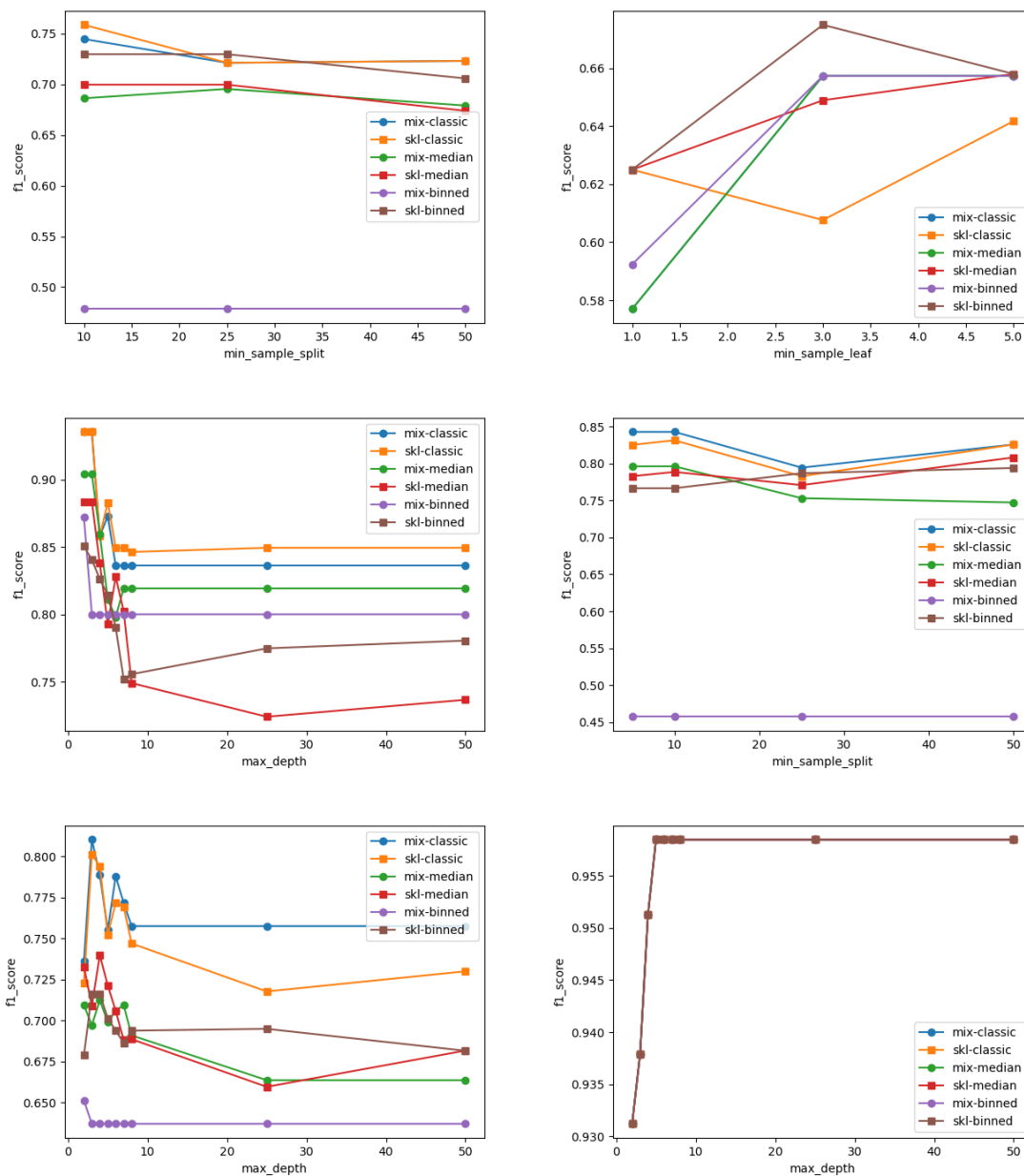


F1 SCORE – BINNED mode



The configurations chosen are the same as when the metric considered was the accuracy;

they can be found in the table of Figure 4.4 on page 51, and each has one hyper-parameter that varies while the other two are kept constant. The following plots show the situation:



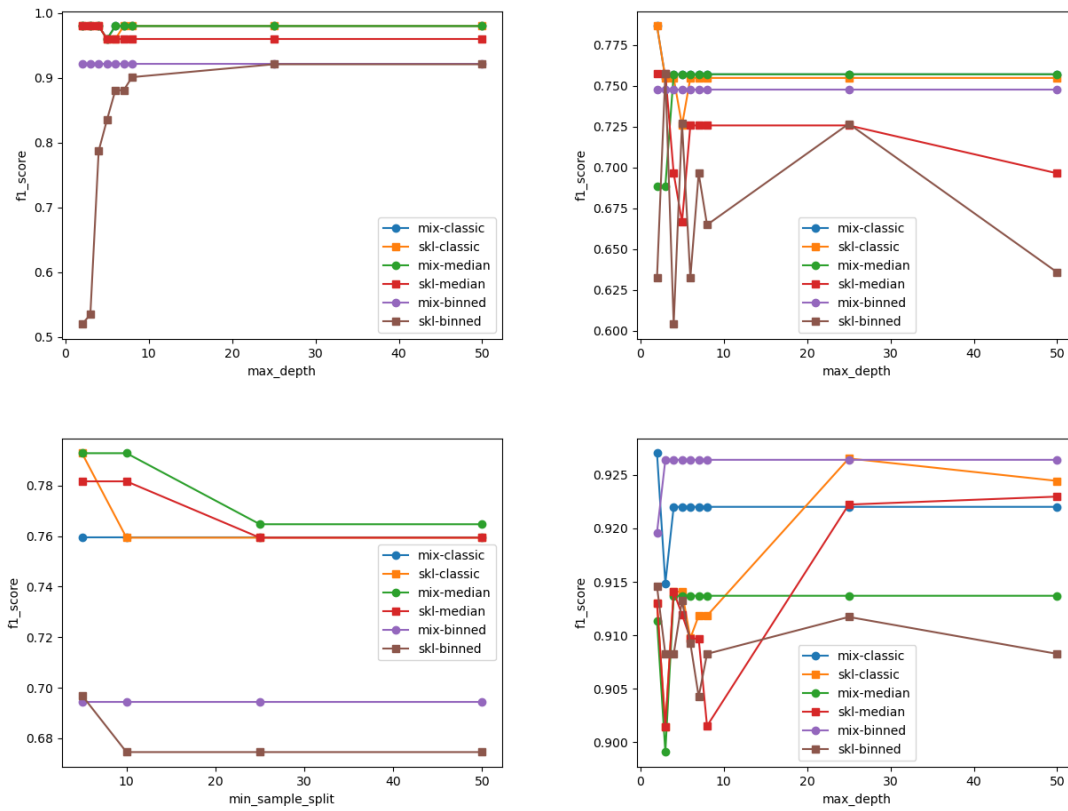


Figure 4.10: Each dataset, in the same order as the table in Figure 4.2, is associated with a given configuration of hyper-parameters. The x-axis shows the parameter that varies, the y-axis the metric on which all the comparisons are based

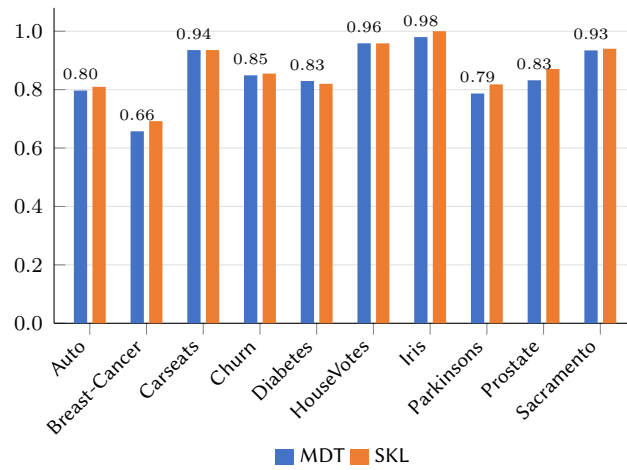
The following table considers only the peaks, that is, the absolute maxima of the F1_SCORE for each mode.

Dataset	Max F1 score					
	Classic		Median		Binned	
	MDT	SKL	MDT	SKL	MDT	SKL
Auto	0.7969	0.8096	0.7645	0.8128	0.6986	0.8233
Breast-Cancer	0.6573	0.6922	0.6573	0.7009	0.6573	0.7009
Carseats	0.9355	0.9355	0.9355	0.9043	0.8822	0.9043
Churn	0.8490	0.8549	0.8202	0.8552	0.6949	0.8478
Diabetes	0.8297	0.8201	0.7886	0.7940	0.7257	0.8171
HouseVotes	0.9585	0.9585	0.9585	0.9585	0.9585	0.9585
Iris	0.9800	1.0000	1.0000	1.0000	0.9606	0.9800
Parkinsons	0.7867	0.8180	0.9089	0.8786	0.7478	0.8786
Prostate	0.8320	0.8709	0.9065	0.9112	0.7594	0.8992
Sacramento	0.9343	0.9401	0.9301	0.9481	0.9264	0.9486
Mean	0.8560	0.8700	0.8670	0.8764	0.8011	0.8758
Std	0.0986	0.0910	0.1059	0.0890	0.1179	0.0829
Wins	4	9	4	8	1	10

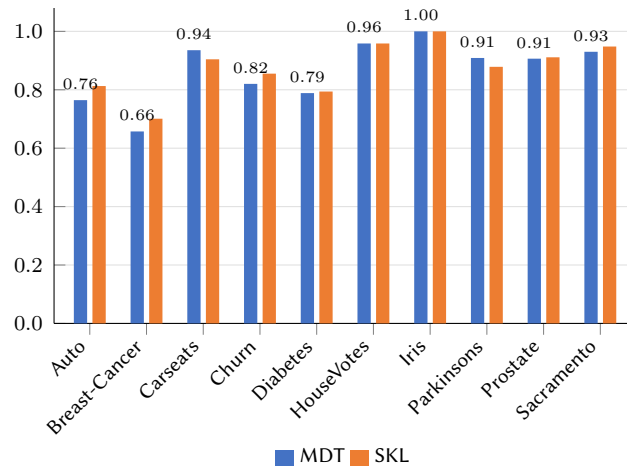
Figure 4.11: Comparisons on the Max F1_SCORE metric

For easier reading, here are the histograms of the three modes, for a more direct comparison:

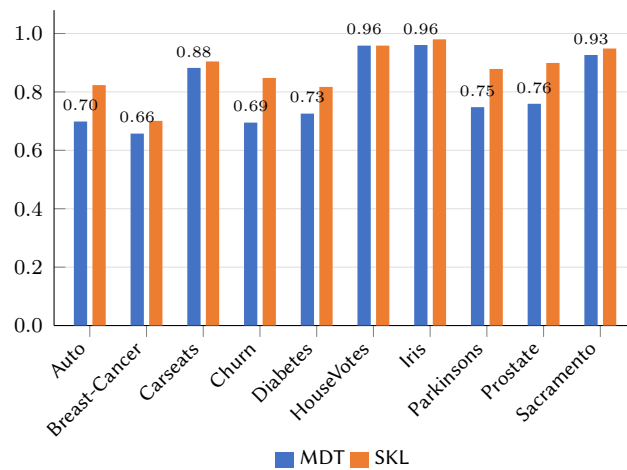
MAX F1 SCORE – CLASSIC mode



MAX F1 SCORE – MEDIAN mode



MAX F1 SCORE – BINNED mode



Here is the summary table of the total wins, means and standard deviations:

	MDT	SKLEARN
Mean	0.8414	0.8741
Std	0.0098	0.0042
Wins	9	27

4.3.3 Measuring Time

Dataset	Execution time in milliseconds					
	Classic		Median		Binned	
	MDT	SKL	MDT	SKL	MDT	SKL
Auto	16,666.0521	2.0182	758.7152	1.3468	572.0809	1.5361
Breast-Cancer	160.5158	1.6310	146.2131	1.3640	180.2812	0.9542
Carseats	19,999.5730	5.3451	1,562.3338	2.3971	863.3039	1.4908
Churn	201,342.5062	1.8492	2,487.1500	1.8623	2,657.8460	1.9426
Diabetes	27,856.9472	4.1931	1,413.8319	2.3460	1,453.5592	1.9462
HouseVotes	144.9978	1.3709	135.6590	1.0567	131.0821	0.8059
Iris	697.1960	4.6749	386.6210	1.9090	214.3610	1.3099
Parkinsons	69,797.8599	4.7150	4,095.7429	3.1629	2,081.2029	3.1097
Prostate	3,335.8209	4.3492	780.6048	2.9342	336.3130	2.4190
Sacramento	241,112.8542	5.7471	2,524.3368	2.7850	956.4180	2.3150
Mean	58,111.4323	3.5894	1,429.1208	2.1164	944.6448	1.7829
Std	84,404.9638	1.5931	1,213.4041	0.6878	823.4618	0.6717

Figure 4.12: The execution times, in milliseconds, were measured on the same machine, under the same conditions and on the same datasets.

Although the MDT is able, as we saw in the previous chapter, to match or, in some cases, even beat the performance of the scikit-learn tree, the latter was programmed by experts to be extremely efficient in time and space.

The table in Figure 4.12 shows the huge differences in execution time, in milliseconds, between the two trees on the same datasets and in the same modes. Above all, it shows the difference between the classic mode, which generates a column for every unique continuous value, and the median mode, which approximates the classic one with a preset number of bins and therefore generates far fewer columns, while sometimes keeping, as we saw in the previous charts and tables, the same accuracy and F1_SCORE as the classic mode. It follows that, if we wanted to declare an internal winner for the MDT, we could conclude that it is precisely the median mode.

If a particular context requires time and space to be taken into account, for example a dataset with many columns of continuous values and a great many rows, the first choice would be the median mode; but if the context does not impose particular limits on time or space, the classic mode is the only one able to deliver the maximum ACCURACY or F1_SCORE.

The MDT cannot compete at all with the scikit-learn tree in terms of complexity, to the point of being almost an order of magnitude worse. However, the native support for categorical attributes and, above all, the multi-way splits (every root and internal node can have an arbitrary number of children) still make this implementation worthwhile.

It is true that the MDT is inefficient in the modelling phase, but it is just as true that this inefficiency can easily be tolerated, considering that the model is trained *only once*, while, for all the remaining countless times, the tree is used only to label and classify unlabeled data, and in this it is just as efficient as the scikit-learn tree: in the worst case, the time complexity is linear in the maximum depth of the tree.

Chapter 5

Conclusions

The main goal of this thesis was to develop from scratch a decision tree characterized by remarkable *readability, expressiveness and versatility*. It is essential to stress that computational efficiency was not considered a priority. The fundamental characteristics mentioned above were obtained by creating a decision model of the decision tree type that is intrinsically categorical and multi-split.

The multi-split nature of the MDT significantly helps human understanding: when an internal or root node has more than two children, the user can see several edges leaving the node, each representing a unique categorical value of the attribute considered.

Moreover, the intrinsically categorical nature of the MDT makes it possible to use datasets commonly employed in the scientific literature.

Although the MDT does not favour efficiency, it is able to deliver results similar in accuracy and F1-score to those of the scikit-learn Decision Tree, which, on the contrary, strongly emphasizes computational efficiency. In fact, the only area in which the scikit-learn DT shows a clear superiority is efficiency in terms of time and space. However, efficiency was not among the main goals of this project, just as readability and expressiveness cannot be called distinctive features of the DT provided by the scikit-learn library.

It can therefore be said that the goal of the thesis has been achieved. In the *classic* and *median* modes, the MDT shows generally satisfactory Accuracy and F1-score and, in some cases, even better ones than the scikit-learn DT, as observed in the previous chapter.

It is important to stress that, although the MDT is less efficient in the modelling phase

(which happens only once), in the classification phase (that is, assigning labels to data not analysed before) it is just as efficient as the scikit-learn DT, since their computational complexity is linear in the depth of the tree.

In conclusion, the MDT is an extremely valid option, except when particularly large datasets have to be handled. Using the MDT in such contexts may not be advisable, since the training phase could require time and memory out of proportion to the size of the dataset itself. However, if in the future one wished to improve the efficiency of the MDT, there are several areas that could be explored:

- Code optimization: reviewing and optimizing the code to reduce its computational complexity and improve its efficiency in terms of execution time and memory usage.
- Parallelization: exploiting parallelization to distribute the work over several processors or cores, thus improving the efficiency of training and modelling.
- Pruning algorithms: implementing pruning algorithms to reduce the complexity of the tree, improving computational efficiency and reducing the risk of overfitting.
- Incremental learning techniques: exploring incremental learning methods to train the decision tree more efficiently on large datasets, updating the model as new data arrive.

Appendix A

The complete code of the thesis is available at the following address:

<https://github.com/giuseppemuschetta/Native-Categorical-Multiway-Split-Decision-Trees>.

The appendix provides a couple of code listings that describe some of the procedures used to create the tables and the overlaid plots. These listings produce the output from which all the tables and histograms of the fourth chapter were generated.

For clarity and simplicity, all the following snippets concern only the configuration in which **max_depth** varies while *min_sample_split* and *min_sample_leaf* are kept constant, since the code for the other two configurations is very similar.

```
1
2 def _max_depth_varies(tables, graphs, couple, tree_type, criterion, mode, bins, mss, msl
   ):
3
4     # read the names of the .csv files of the same dataset that I will analyse
5     mixed_csv = couple[0]
6     sklearn_csv = couple[1]
7
8     # dataset name:
9     temp_ds_name = re.split('/', couple[0])[1]
10    ds_name = re.split('_', temp_ds_name)[0]
11
12    # tell the two trees apart
13    if tree_type == 'MixedDecisionTree':
14        df = pd.read_csv(mixed_csv)
15    elif tree_type == 'SklearnDT':
16        df = pd.read_csv(sklearn_csv)
17    else:
18        raise Exception("Wrong Decision Tree name...")
19    df.insert(0, 'DATASET', ds_name)
20    df.insert(1, 'DT_TYPE', tree_type)
```

```

21
22 # the query starts here
23 if mode != 'classic':
24     df_queried = df[(df['Tree_Mode'] == mode) & (df['Criterion'] == criterion)
25                     & (df['Bins'] == bins) & (df['Min_sample_split'] == mss) &
26                     (df['Min_sample_leaf'] == msl)]
27 else:
28     if bins != 20:
29         return
30     else:
31         bins = 0
32         df_queried = df[(df['Tree_Mode'] == mode) & (df['Criterion'] == criterion)
33                         & (df['Bins'] == bins) & (df['Min_sample_split'] == mss) &
34                         (df['Min_sample_leaf'] == msl)]
35     # end_if
36 # end_if
37
38 # TABLES:
39 if tables:
40     # create the directories where the tables are saved:
41     dir_path = 'Tables/' + ds_name + '/'
42     try:
43         os.makedirs(dir_path, exist_ok=True)
44     except OSError as error:
45         print(error)
46         print("Path '%s' can not be created" % dir_path)
47
48     table_path = dir_path + 'max_depth_varies.csv'
49     df_queried.to_csv(table_path, index=False, header=True, mode='a')
50 # end_if
51
52 # PLOTS:
53 if graphs:
54     # create the directories where the plots are saved:
55     dir_path = 'Graphs/' + ds_name + '/' + 'max_depth_varies/'
56     try:
57         os.makedirs(dir_path, exist_ok=True)
58     except OSError as error:
59         print(error)
60         print("Path '%s' can not be created" % dir_path)
61
62     metrics_list = ['ACCURACY', 'F1_SCORE']
63     for metric in metrics_list:
64         graph_path = dir_path + tree_type + '_' + metric + '_' + criterion + '_' + mode
65                     + '_' \

```

```

65         + str(bins) + '_' + str(mss) + '_' + str(msl) + '.png'
66
67     x = df_queried['Max_Depth']
68     y = df_queried[metric]
69
70     plt.plot(x, y, label=tree_type, marker='o')
71     plt.xlabel('max_depth')
72     plt.ylabel(metric.lower())
73     plt.savefig(graph_path)
74     plt.close()
75 # end_if
76
77 return

```

Listing A.1: For each dataset we created tables and plots

From the tables and plots created by the previous snippet, further plots were generated, this time overlaid, to show how certain hyper-parameters vary for both of the decision trees considered in this thesis.

```

1
2 def graph_overlaps(couple, criterion, _bins, mss, msl):
3     # read the names of the .csv files of the same dataset that I will analyse
4     mixed_csv = couple[0]
5     sklearn_csv = couple[1]
6
7     # dataset name:
8     temp_ds_name = re.split('/', couple[0])[1]
9     ds_name = re.split('_', temp_ds_name)[0]
10
11     # tell the two trees apart:
12     df_mdt = pd.read_csv(mixed_csv)
13     df_sk = pd.read_csv(sklearn_csv)
14
15     # the queries start:
16     # we write several times on the same plot so that the lines of the plot overlap
17     tree_mode_list = ['classic', 'median', 'binned']
18     for mode in tree_mode_list:
19         if mode == 'classic':
20             bins = 0
21         else:
22             bins = _bins
23     # end_if
24     df_mix_queried = df_mdt[(df_mdt['Tree_Mode'] == mode) & (df_mdt['Criterion'] ==

```

```

criterion)
25         & (df_mdt['Bins'] == bins) & (df_mdt['Min_sample_split'] == mss) &
26         (df_mdt['Min_sample_leaf'] == msl)]
27
28     df_sk_queried = df_sk[(df_sk['Tree_Mode'] == mode) & (df_sk['Criterion'] ==
criterion)
29         & (df_sk['Bins'] == bins) & (df_sk['Min_sample_split'] == mss) &
30         (df_sk['Min_sample_leaf'] == msl)]
31
32     x = df_mix_queried['Max_Depth']
33     y = df_mix_queried['ACCURACY']
34     plt.plot(x, y, label='mix-%s' % mode, marker='o')
35
36     x = df_sk_queried['Max_Depth']
37     y = df_sk_queried['ACCURACY']
38     plt.plot(x, y, label='skl-%s' % mode, marker='s')
39 # end_for
40
41     dir_path = 'Overlaps_accuracy/' + ds_name + '/' + 'max_depth_varies/'
42     try:
43         os.makedirs(dir_path, exist_ok=True)
44     except OSError as error:
45         print(error)
46         print("Path '%s' can not be created" % dir_path)
47
48     graph_path = dir_path + 'ACCURACY' + '_' + criterion + '_' + str(_bins) \
49         + '_' + str(mss) + '_' + str(msl) + '.png'
50
51     # here the overlaid plots are finally created
52     plt.xlabel('max_depth')
53     plt.ylabel('ACCURACY'.lower())
54     plt.legend()
55     plt.savefig(graph_path)
56     plt.close()
57
58     return

```

Listing A.2: Procedure that creates the overlaid plots as max_depth varies

Bibliography

- [1] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.
- [2] Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.
- [3] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Data mining introduction. *People's Posts and Telecommunications Publishing House, Beijing*, 2006.