

École polytechnique de Louvain

Neurosymbolic AI: Combining Deep Learning and Rule Learning

Author: **Mathieu JACOB**
Supervisors: **Lucile DIERCKX, Siegfried NIJSSEN**
Reader: **John AOGA**
Academic year 2023–2024
Master [120] in Data Sciences Engineering

Abstract

In the context of classification, neural networks are generally considered as state-of-the-art in terms of performance. Yet, these black-box models suffer from a lack of interpretability. This can however be alleviated through a careful design of the network’s architecture. Such an approach has been followed in recent years to create the Decision Rules Network (DRNet) model. This neural network handles binary data and can be interpreted as a simple set of decision rules. In this thesis, we develop extensions of the DRNet model, with the aim of seamlessly handling datasets of various types and properties. This is achieved by adding a new layer to the network, which integrates the data binarisation closely with the rest of the learning process. At the same time, we wish to maintain a high degree of accuracy in our predictions, and favour rule sets that are composed of few, short rules. These are conflicting objectives, as high accuracy levels generally require more intricate decision rules, and conversely. Through our experiments, we find that the accuracy of our extended model remains high, and identify ways in which to further improve its simplicity.

Acknowledgements

I want to thank my supervisor Prof. Siegfried Nijssen for guiding me through this thesis via his ideas and advice, as well as for giving me motivation and reassurance.

Special thanks are also due to Lucile Dierckx for her technical expertise and her availability.

I am also grateful to John Aoga for enthusiastically accepting to read my thesis.

I also take this opportunity to acknowledge the help of ChatGPT in writing basic snippets of code and navigating the intricacies of some Python, Java and $\text{\LaTeX}$ modules, as well as DeepL for identifying more appealing sentence structures.

Last but not least, my heartfelt thanks go to my family and my friends for their everlasting support, enthusiasm and understanding through my academic journey, as well as to my cats for being a calming presence through stressful times.

Contents

1	Introduction	1
2	Theoretical Background	5
2.1	Classification	5
2.1.1	Principle	5
2.1.2	A Simple Classifier	6
2.1.3	Evaluation	6
2.2	Deep Learning	8
2.2.1	Principle	8
2.2.2	Layer	8
2.2.3	Training	10
2.2.4	Discussion	12
2.3	Rule Learning	14
2.3.1	Principle	14
2.3.2	Approaches	15
3	DRNet	19
3.1	Context	19
3.2	Architecture	20
3.3	Training Process	21
3.4	Points of Interest	22
3.5	Conclusion	23
4	Model Enhancement	25
4.1	Thresholds Layer	25
4.1.1	Initialisation	25
4.1.2	Implementation	26
4.1.3	Categorical Data	29
4.1.4	Training Process	32
4.2	Post-processing	32
4.2.1	Uninformative Rules	32
4.2.2	Interactions between Thresholds	33

4.2.3	Clipping	34
4.3	Parameters Initialisation	34
4.4	Conclusion	35
5	Performance	37
5.1	Datasets	37
5.1.1	Adult	38
5.1.2	Magic	38
5.1.3	House	38
5.1.4	Heloc	38
5.1.5	Drugs	38
5.1.6	Occupancy	39
5.1.7	Rockburst	40
5.2	Other Models	40
5.2.1	Original DRNet	40
5.2.2	RIPPER	41
5.3	Procedure	41
5.4	Results	42
5.4.1	Maximum Number of Thresholds	42
5.4.2	Models Comparison	45
5.4.3	Post-processing Effectiveness	47
5.4.4	Temporal Data	49
5.4.5	Imbalanced Data	50
5.5	Conclusion	51
6	Conclusion	53

Chapter 1

Introduction

The field of machine learning (ML), and in particular deep learning, has had an undeniable impact on society in recent years. As a case in point, large language models such as GPT [1] have become part of everyday language, even far beyond the scientific community. But this represents only one aspect of a much broader subject.

One of the main tasks for which machine learning models are used is to perform *classification*, which consists in correctly labelling some data given as input to the model. When confronted with such a task, a student learning about machine learning for the first time will primarily concern themselves with the *accuracy* of their model – or related metrics such as precision and recall – as well as its ability to *generalise* to new data and its *efficiency*. However, some additional issues may arise when considering concrete applications of ML models and artificial intelligence (AI) in general.

For example, many attempts have been made to utilise AI in the context of *health-care*, including the prediction of bloodstream infections, the diagnosis of tumours and even autism spectrum disorders [2]. The goal is then for these new technologies to be a reliable tool assisting healthcare professionals in finding patterns and making accurate decisions.

Another common example arises from the field of *transportation* and in particular, *autonomous vehicles*, where AI models play a key part regarding the tasks of (*local*) *route planning* and *decision making*, among others. In this case, the primary objective is to vastly reduce the number of accidents, deemed to mostly be caused by human error [3, 4].

These two examples were selected because they are both *critical*: the output of the models that are used could have direct, and potentially threatening, impacts on human lives. In light of this, it is then evident that some models are required to be *interpretable*: a human should be able to understand their decision, and the criteria upon which it was based. This is also important in many applications that are not as critical as the ones mentioned above. Indeed, if a user is presented with a result for which the model does not provide a clear explanation, they may be sceptical of it, and lack trust in it. This shows that interpretable AI plays a key role in what we refer to as *user adoption* [5].

Our goal in this thesis is to build an interpretable ML model to perform classification. One possible approach would be to utilise *rule learning* models. Their task is to identify rules that, when satisfied, indicate that an input belongs to a specific class. Consequently, these models are easily interpretable, as the rationale behind the model's classification is explicitly conveyed by the rule.

Another possibility is to employ *neural networks*. Thanks to its high empirical performance, advanced training techniques and its efficient use of modern hardware, this model has become a popular choice for classification tasks. But one of its drawbacks is its lack of interpretability. The intricate nature of a neural network's architecture often renders it challenging to understand the exact reasons behind the classes it outputs, making it a *black-box* model.

One might then wonder whether it would be possible to combine the advantages of rule learning models and neural networks, in order to create highly performant and interpretable models. In the literature, we find that this has been endeavoured through the *Decision Rules Network (DRNet)* model [6]. It is a neural network, and can thus benefit from all the advanced techniques used to train those models. But it is also built in a manner that makes it interpretable as a set of rules. This addresses the primary drawback of neural networks, making DRNet a compelling choice in the context of interpretable classification.

In practice, DRNet has been shown to yield positive results [6]. Nevertheless, there are still some shortcomings that we wish to address and improve upon in the present work. According to us, the principal inconvenient of DRNet is that it only handles *binary* data. However, in practice, datasets contain different types of data. To accommodate these, DRNet relies on additional processing of the data before injecting them into the network. We want to modify this step to incorporate it into the learning process.

Furthermore, while DRNet promotes the sparsity of the final rule set through regularisation, we believe that there is often an opportunity to make it even sparser, without modifying the model’s predictions. This is a worthwhile avenue for investigation, as a rule set that contains fewer and shorter rules is more interpretable than a more intricate one.

In this thesis, we thus tackle both of these issues. First, we modify DRNet’s architecture by adding a new component dedicated to the binarisation process. It will be able to handle numerical and categorical data in the exact same manner, and to evolve with the rest of the network. Moreover, in order to increase the sparsity of our rule set, we incorporate post-processing techniques into our model. Through careful experiments, we will then study the quality of our enhancements with regard to various metrics such as accuracy, simplicity and adaptability.

This work is divided into six chapters. In Chapter 2, we will familiarise ourselves with all the different theoretical concepts that will be relevant for the following parts. Then, Chapter 3 is dedicated to discovering the DRNet model and the opportunities it presents for improvement. We will subsequently present the way in which we build our own model in Chapter 4, by precisely detailing its conception and functioning. Following this, we will conduct a series of experiments to test this model in Chapter 5. After discussing and interpreting the results of these experiments, we will conclude this manuscript in Chapter 6.

Chapter 2

Theoretical Background

In this chapter, we introduce some concepts that will be relevant to the following parts of this work. As the goal of our model is to perform classification, we start by describing what this task precisely consists in. Afterwards, we detail in turn the techniques of deep learning and rule learning, which we will combine later on.

2.1 Classification

2.1.1 Principle

The goal of classification [7] is to correctly assign a class to a given input. Usually, the input x would be a vector containing features represented by real numbers. But in general, the input could take many different forms: categorical, sequential, audio... Then, using a classification model, we assign a class c to x , chosen among a finite set C of classes.

Obviously, this task is only interesting in the case where the correct class c^* is not known in advance by the model. Typically, the model is trained based on some examples for which the associations between inputs and classes are given. Afterwards, it is evaluated on new input data, on which it then actually performs a prediction $\hat{c}$, which can be compared to the correct class c^* . This type of learning process is discussed in Section 2.2.3.

2.1.2 A Simple Classifier

A basic example of a classifier is given by the *naive Bayes classifier*, which emits the following prediction:

$$\begin{aligned}
 \hat{c} &\equiv \arg \max_{c \in C} P(c|x) \\
 &= \arg \max_{c \in C} \frac{P(x|c)P(c)}{P(x)} \\
 &= \arg \max_{c \in C} P(x|c)P(c) \\
 &\approx \arg \max_{c \in C} \hat{P}(x|c)\hat{P}(c).
 \end{aligned}$$

This classifier's principle is thus to choose the class c which maximises the *posterior* probability $P(c|x)$ of the class being c given that we observe input x . In order to compute it, we make use of the well-known Bayes formula which, up to a scaling factor $P(x)$, allows expressing this quantity in terms of a *prior* probability $P(c)$ of observing class c , and a *likelihood* $P(x|c)$ of observing our input x given that the class is c . As these quantities are typically not known in practice, they are then approximated respectively as $\hat{P}(c)$ and $\hat{P}(x|c)$, based on the input data and the information we have regarding the task at hand, i.e., domain expertise.

2.1.3 Evaluation

Once the input has been assigned a class $\hat{c}$, be it through a naive Bayes classifier or any other model, we can compare it to the expected output c^* . In this thesis, we choose to focus on *binary* classification tasks. That is, the case in which $|C| = 2$. One of the classes will then be considered *positive* (P), and the other one negative (N). The choice of which class is which is generally apparent from the domain of application studied. As an example, if the classification task can be expressed as “Is this mushroom safe to eat?” then the positive class would correspond to answering “yes”, and the negative to answering “no”. Given these considerations, we can define the following quantities:

- TP = True Positive. A positive example correctly labelled as positive.
- FP = False Positive. A negative example incorrectly labelled as positive.
- TN = True Negative. A negative example correctly labelled as negative.
- FN = False Negative. A positive example incorrectly labelled as negative.

In the following, these abbreviations are also used to denote the *number* of examples belonging to each of these categories. Therefore, TP will also denote the amount of positive examples which have been classified as positive; and similarly

for the other categories.

These quantities can be used to define the evaluation metrics used in the context of binary classification. First off, the *accuracy*:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

This is a measure of the proportion of examples that have been correctly classified. A high accuracy can be a good indicator of the quality of a classification model, but this requires the numbers of examples to be relatively well-balanced between the two classes. Otherwise, the following argument can be made against the use of this metric. If we imagine a dataset in which 90% of the examples are positive, and the remaining 10% are negative, then a trivial classifier which predicts the positive class in every case has an accuracy of 90%. In that case, the accuracy metric is thus very large, even though the classifier is poor.

In some cases, it happens that one gives more importance to false positives (resp. false negatives), meaning that more emphasis is placed on avoiding to incorrectly classify negative (resp. positive) examples. In the first case, it is useful to consider the *precision* metric:

$$Precision = \frac{TP}{TP + FP},$$

which can be intuitively understood as favouring quality over quantity: examples classified as positive should actually be positive.

On the other hand, *recall* is computed as

$$Recall = \frac{TP}{TP + FN},$$

and corresponds to favouring quantity over quality: positive examples should actually be classified as such.

Finally, the *F1 score* is a compromise between the previous situations, and is computed as the harmonic mean of the associated metrics:

$$F1 = 2 \frac{Precision \times Recall}{Precision + Recall}.$$

More advanced metrics such as the area under curve (AUC) can be used in practice but are not detailed here, as they are outside the scope of this work.

2.2 Deep Learning

2.2.1 Principle

Originally, neural networks, as their name suggests, have taken inspiration from the functioning of neurons in the human brain [8]. These ML models are organised as layers of *neurons*, linked together by *synapses*, as detailed in Section 2.2.2.

A neural network is nothing more than a (usually quite complex) function mapping inputs to outputs. The input layer is the entry point to the network. In our general context of binary classification, it will contain the input to be classified. As for the output layer, it will represent the class predicted by the network. Between those, we typically find some hidden layers. The network is considered “deep” if there are at least two of them.

2.2.2 Layer

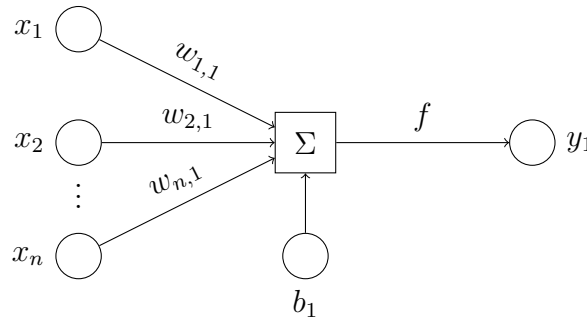


Figure 2.1: One layer of a neural network. Adapted from [9].

The representation of the typical architecture of a single layer of a neural network is found on Figure 2.1. The inputs of the layer are denoted $x_1, \dots, x_n$, and may very well be the outputs of a previous layer. Each input x_i is multiplied by the associated weight $w_{i,j}$. These products are summed together, and a bias b_j is then added, to constitute what is commonly known as the *activation*. We note here that while each index i refers to a specific input, the index j refers to a specific output. Indeed, although a single output is represented on Figure 2.1 for clarity, each layer can (and very often does, if it is the input layer or a hidden layer) have several outputs. Each input x_i can then be used several times, with different weights $w_{i,j}$ to produce each associated output y_j . Together, the weights and biases constitute the learnable parameters of the model. The manner in which their values evolve is described in Section 2.2.3.

The activations are not to be mistaken with the *activation function*, denoted f on Figure 2.1, which is applied to the activations. Up to this point, the operations performed in the layer have been linear. In order to ensure a high expressiveness of the model, it is then customary to use *non-linear* activation functions, which may vary from one layer to another. Popular choices include:

- the *sigmoid* – or more precisely, the *logistic* – function:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

whose output lies in $[0, 1]$. This function is such that $\lim_{x \rightarrow -\infty} \sigma(x) = 0$, and $\lim_{x \rightarrow +\infty} \sigma(x) = 1$. Graphically, the function increases smoothly between these two values, unlike a *step function*

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

- the *hyperbolic tangent*:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

whose output lies in $[-1, 1]$. While the sigmoid could be seen as a smoothed version of a step function from 0 to 1, the hyperbolic tangent can be interpreted as a smoothed version of a *sign function*

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}.$$

Both of these functions are thus quite similar conceptually, but are used in different situations, depending on the output range required¹.

- the rectified linear unit (ReLU), first used in [10]:

$$\text{relu}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} = \max(0, x).$$

The main advantage of the ReLU function is the fact that on the entirety of $\mathbb{R}^+$, it has a constant derivative equal to 1, whereas the derivatives of the previous two functions rapidly take very small values when we move away from the origin. This helps to mitigate the *vanishing gradient* issues described in Section 2.2.4.

¹A scaling factor can of course be applied to either function to shrink or expand its range. The main difference between them is whether negative values should also be part of the output range or not.

- the *softmax* function [11]:

$$\text{softmax}(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}},$$

takes as input a vector $\mathbf{x} \in \mathbb{R}^K$, and normalises it between 0 and 1 using the exponential function. The components of the resulting vector sum to 1. Because of these two properties, the softmax is generally used in the output layer of the network, in order for the outputs to be interpretable as probabilities. In the context of classification, these would then correspond to the estimated probabilities of the input belonging to each class.

2.2.3 Training

Given an initial dataset, it is customary to split it into a *training set* and a *test set* before starting the training process. This allows to assess, afterwards, how the model performs on unseen data. This is referred to as *cross-validation*. Within the training set, it is also possible to further define a *validation set*, used to fix some hyperparameters of the model, the parameters themselves being of course fixed during the training process.

In our case, we will use *K-fold cross-validation*, in which the dataset is randomly split into K *folds* (i.e., subsets) of equal size. Among these, $(K - 1)$ are used for training, while the last fold constitutes the test set. This process is repeated K times, in order for each of the K folds to act as a test set exactly once. Each time, we will compute some metric to evaluate the model's classification performance. In the end, the average of these metric values is computed to obtain one final estimation of the model's performance.

We can now discuss the training process of a neural network. It is organised into *epochs*, each consisting of forward and backward *passes* of the entire training set through the network. In order to make the process more efficient, the data are typically split into smaller *batches*.

Forward Pass. During this phase, the parameters of the network are fixed. Each batch is thus simply *forwarded* through the network. That is, the output $\hat{y}$ of the model is computed based on these inputs. It is compared to the expected output y in a *loss function* $L(\hat{y}, y)$ which must be minimised. In the case of our own binary classification model, discussed later, we will use the *binary cross-entropy (BCE)* loss, combined with a regularisation term. The BCE is expressed as:

$$l(\hat{y}, y)_i = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)].$$

Note that here, we have set our two classes to correspond to the values 0 and 1. This means that exactly one term among y_i and $(1 - y_i)$ will be equal to 1, and the other one to 0. If the prediction $\hat{y}_i$ corresponds perfectly to y_i , then $l(\hat{y}, y)_i = 0$. In the opposite case, where $\hat{y}_i$ predicts the opposite class, $l(\hat{y}, y)_i$ tends to $+\infty$. Intermediate values could also be obtained if the model assigns a non-zero probability for the input to belong to each of the two classes.

As for the regularisation term, it is used to prevent the model from becoming too complex. Let Θ denote the set of all the parameters in the model. Adding the regularisation term, the loss function can finally be expressed as

$$L(\hat{y}, y) = l(\hat{y}, y) + \lambda \|\Theta\|,$$

where λ is a meta-parameter, and we use $\|\Theta\|$ as a general expression of the complexity of the model. For example, we could choose $\|\Theta\| = \sum_{\theta \in \Theta} \theta^2$. By including such a regularisation term in the loss function that we are minimising, we can force some parameters to zero, hence favouring simpler models. The λ meta-parameter then helps to find a balance between simplicity and expressiveness.

Backward Pass. Once the loss $L(\hat{y}, y)$ based on the current batch has been extracted at the end of the network, it is used to *update* the parameters of the model. While many variants exist, they mostly revolve around the concept of *gradient descent*, which ultimately aims at minimising the loss, and works as follows [12]. In order to update any given parameter θ , which was used to compute the prediction $\hat{y}$, we compute $\frac{\partial}{\partial \theta} L(\hat{y}, y)$, the partial derivative of the loss with respect to θ ². This parameter is then updated in the following way:

$$\theta \leftarrow \theta - \eta \frac{\partial}{\partial \theta} L(\hat{y}, y),$$

where η denotes the *learning rate*. A more advanced update rule is discussed later in this section. But first, we take a look at how to compute $\frac{\partial}{\partial \theta} L(\hat{y}, y)$.

The computation of the gradient of the loss with respect to the output(s) of the network is immediate from the definition of the loss function. Afterwards, we rely on the principle of *backpropagation* [13], which is based on the well-known *chain rule*. Indeed, neural networks intrinsically have a hierarchical structure, with the outputs of one layer serving as inputs for the next one. Therefore, the gradient of the loss with respect to the parameters and inputs of a layer can be computed from those of the next layer, through simple multiplication by the gradient of the

²As the derivatives are computed based on a subset of the data (the batch), we sometimes refer to this approach as *mini-batch gradient descent* [12]. This method is itself a variant of the popular *stochastic gradient descent* (SGD), in which examples are considered one at a time.

function that relates them. This way, the gradient of the loss performs a *backward pass* through the network, updating every parameter of the model on its way by using the above rule.

We finish our presentation of the training process by discussing another update rule for the parameters, namely using the *Adam* optimiser [12, 14] which is the one that will be used in our own model. This optimiser operates in time steps, looping until convergence. At each time step t , the vector of parameters is updated as follows:

$$\Theta_t \leftarrow \Theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}.$$

In this expression, ϵ is a fixed parameter (typically $\epsilon = 10^{-8}$), while $\hat{m}_t$ and $\hat{v}_t$ are computed from g_t , the gradient at time t . They are respectively estimates of the mean and uncentered variance of g_t , and have both been corrected to not be biased towards zero. The Adam optimiser is widely used for its efficiency and scalability.

2.2.4 Discussion

Working with neural networks involves several *challenges* which historically, have slowed down the development and adoption of these models.

Data Availability. When training any machine learning model, the amount of available data is often a limiting factor. Several elements contribute to this issue. First off, the *curse of dimensionality*, first introduced in [15] in the context of *dynamic programming*, describes the fact that the amount of data required is large for models that have high-dimensional input spaces. In particular, this quantity grows *exponentially* with the input space size [16].

Furthermore, models that are more complex also require larger amounts of data. This is very relevant to deep neural networks, as these models are typically very complex: they often involve very large numbers of parameters and display complex architectures [17].

Two obvious approaches that help mitigate this problem are to either feed large amounts of data to the model or to limit its complexity. The former has recently been enabled by the *Big Data* phenomenon, while the latter can be achieved by privileging simpler architectures or, e.g., using the aforementioned *regularisation* principle. Another interesting possibility is the use of *transfer learning*, which re-uses (parts of) a neural network trained on a given task, to apply it to another task. The remaining training process is then supposed to be simpler, as the model needs not be retrained from scratch [18].

Training Time. As described above, deep networks are complex models that have to handle large quantities of parameters and data, through many epochs of training. Obviously, this means that training times are longer than with simpler models. A neural network requiring more than a couple of hours or days to be trained will generally be prohibitive. Reasonable training times are thus parts of the requirements for these models.

A key factor for speeding up the training process has been the use of *graphics processing units* (GPU's). Initially intended for video games and image processing, their high performance in repetitive and parallel tasks have made them an asset in the field of deep learning [19].

Vanishing/Exploding Gradient. This is a very common issue when training deep networks through gradient-based methods [20]. As we know, the gradient of the loss with respect to the different parameters is computed using the principle of backpropagation, based on the chain rule [13]. This means that when we reach “deeper” parts of the network, many partial derivatives have been multiplied together: if a single one of them is equal or close to 0, the entire product will be very small, hence leading to a small update of the associated parameters, and thus slow learning. This is the *vanishing gradient* problem. It is particularly prevalent in deep networks with very large numbers of layers, such as *recurrent neural networks* (RNN's) [20, 21].

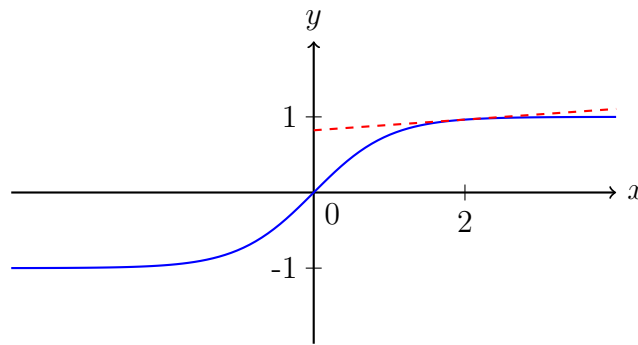


Figure 2.2: The hyperbolic tangent function (in blue) and the tangent line at the $x = 2$ abscissa (in red, dashed). The slope of the tangent line being almost horizontal illustrates the fact that this function has a derivative that is almost zero on a large part of its domain.

Small derivatives often arise in practice with the usual activation functions. For example, the *hyperbolic tangent* function has a derivative that quickly tends towards zero as we move away from the origin. This is depicted on Figure 2.2 in the case where $x = 2$.

The opposite situation, coined *exploding gradient* problem, may also arise. In that case, large quantities are multiplied together during backpropagation, leading to large parameter updates, and a lack of convergence of the training process, which will explore the parameters space with steps that are too large.

Considering both of these issues, the advantage of the ReLU activation function is twofold. Since its derivative is constant and equal to 1 over half of its domain, this makes backpropagation faster, as well as helping mitigate potential problems of vanishing/exploding gradient. Other techniques have also helped to alleviate these issues, such as the development of *long short-term memory* (LSTM) [21] as an evolution of the aforementioned RNN's.

2.3 Rule Learning

2.3.1 Principle

Rule learning is a broad term which covers many problems and their associated methods. Among those, one can identify two types of tasks: those related to *predictive rule learning*, and those that pertain to *descriptive rule discovery* [22]. The latter aims at identifying relationships and patterns between the different features of a given dataset. For instance, when performing *association rule discovery*, we seek to identify subsets of features that can successfully determine the presence of other subsets of features. In the literature, this is often denoted as $X \Rightarrow Y$, where X and Y are subsets of the input features, or *itemsets*. A typical application of this would be to identify items that are commonly bought together in supermarkets, in order to subsequently make more profitable decisions regarding the management of these items [23].

In the present work – which focuses on binary classification – whenever we mention “rule learning”, we implicitly reference the task of *predictive* rule learning. Unlike its aforementioned counterpart, this task aims at extracting rules that can be used to accurately *classify* previously unseen data. To this end, predictive rule learning algorithms build *lists* or *sets* of rules. The main difference being that lists of rules have an intrinsic order in which they must be tested, whereas sets obviously do not.

In the context of binary classification, the possibility we will opt for is to learn *rule sets* comprising all rules used to predict one class, typically the positive one. Implicitly, all examples to which none of the rules apply are then classified as belonging to the other class. This approach has the advantage of avoiding ambiguous situations where several contradictory rules would apply simultaneously [22].

2.3.2 Approaches

We briefly present a couple of approaches for predictive rule learning, before pondering on the similarities that they bear. This will allow us to highlight the way in which the paradigm presented in Chapter 3 differs from these approaches.

Association Rules

Although we have presented descriptive rule discovery and predictive rule learning as distinct tasks, some of the methods used to undertake those are related. Indeed, we consider again association rule learning, which was briefly described above. Even though association rule learning is not a supervised task, it can still be leveraged in the context of predictive rule learning. A particular form of association rule would be $X \Rightarrow \{y\}$, where the right-hand side of the rule has been reduced to a single attribute y . It is then clear that, when y corresponds to the target variable of our classification problem, such association rules can directly be used to build a rule set [22]. This is the approach taken, e.g., in the *CBA* (*Classification Based on Associations*) rule learning algorithm [24]. Seeing that association rule learning is thus relevant to our problem, it is worthwhile to take a step back and examine the way in which such association rules can be learned.

In [23], the problem is divided into two sub-problems:

1. *Frequent itemset mining*. This consists in identifying itemsets (subsets of the input features) that appear a large number of times in the dataset. Typically, what is considered “large” is determined by a threshold on the *support* of an itemset, which is the fraction of the dataset entries in which this itemset is present.
2. Based on each frequent itemset I that was obtained in the previous step, splitting it into rules of the form $X \Rightarrow \{y\}$ where $\{y\} = I \setminus X$, and keeping the rules for which the *confidence ratio* $\frac{\text{support}(I)}{\text{support}(X)}$ is deemed sufficiently large. Those are finally gathered to form a set of association rules.

The first sub-problem is the most challenging of the two, and numerous algorithms have been developed to tackle it, the first of which was *Apriori* [25] which operates in a level-wise fashion, progressively considering larger itemsets. Without diving into the details of frequent itemset mining algorithms, we can already observe that the general structure of an approach such as CBA to compute rule sets requires quite a lot of work before obtaining any rule, and that the rule set is built in an incremental manner. We will discuss this further at the end of this section.

RIPPER

Several rule learning algorithms build rule sets by using a *separate-and-conquer* learning approach [26]. The goal is to build rules that are able to correctly classify positive examples. To this end, such algorithms build each rule by repeatedly adding *literals* (an input feature or its negation) to the rule until it does not apply to any negative example of the training set. Such rules are added to the rule set until each positive example is *covered*, that is, at least one of the rules applies to it. However, as one may rapidly notice, such an approach easily leads to overfitting, as the rule set will have learned every specificity of the dataset it was derived from, and may not generalise well to new data at all.

Therefore, it is common to apply *pruning* techniques during the learning process, in order for the rules to fit the training dataset less perfectly but generalise better. *Pre-pruning techniques* stop the learning process early, while *post-pruning* methods remove (parts of) rules after they have all been learned. Both of those ideas are integrated in the *I-REP* algorithm [26], in which *incremental reduced error pruning* allows to prune clauses from each rule right after it has been generated. This is unlike previous approaches, which only performed pruning after the entire rule set had been generated.

Further improvements to I-REP, including an additional post-processing phase and a modified stopping criterion, have led to the well-known *RIPPER* (*Repeated Incremental Pruning to Produce Error Reduction*) algorithm [27]. As this approach remains quite performant to this day, we will use it as a reference to compare our own algorithm in Chapter 5.

Discussion

Although the rule learning algorithms presented heretofore are quite different in the way they identify rules, they bear similarities. Indeed, the rules are typically constructed *sequentially*: the algorithms create them one after the other. Moreover, those algorithms often operate in a *greedy* fashion, by choosing individual predicates and rules that maximise a metric such as confidence. Depending on the approach, they may also be modified or pruned at a later stage, in order to alleviate overfitting issues.

This is in stark contrast to the approach that will be employed in this work in order to extract rules. Indeed, these will be constructed using a specific *neural network* architecture, and will be directly related to the network's parameters. This signifies that rules will be selected based on how much they can decrease the value of the global *loss function*, namely the binary cross-entropy loss presented

earlier. Overfitting issues will then be addressed by evaluating this loss function on a test set that is distinct from the training set on which the network is trained, hence directly allowing to observe the model's performance on unseen data. Finally, within this new framework, rules are not constructed in a sequential fashion anymore. Rather, the entire set of rules is constructed at the same time. More precisely, at any given point, the parameters of the network can be interpreted as a rule set. But this rule set will certainly not constitute an accurate predictive model until it has undergone proper training allowing to refine it.

Chapter 3

DRNet

Our contributions in the interpretable neural network domain are based on an existing work, described in this chapter. After explaining the main ideas behind this model, its architecture and training process, we identify the elements that will guide our own research.

3.1 Context

While the ambition to develop interpretable machine learning models is not new, the particular idea of creating simple neural networks whose parameters can be interpreted as decision rule sets is fairly recent. It has been introduced in [6], and is the starting point for this thesis. In their paper, the authors present a *decision rules network*, commonly abbreviated as *DRNet*. The model is based on a simple rationale: combining the advantages of using a deep learning framework, as well as producing understandable results.

Indeed, although neural networks have been observed to produce highly accurate results in many classification applications, they are generally regarded as *black-box models*. As detailed in [6], those still have several advantages: state-of-the-art training approaches, regularisation techniques for a reduced model complexity, use of powerful computing hardware (GPU's) [19]. DRNet's architecture, detailed hereafter, addresses this issue of black-box model, by constructing the network in a way that allows its parameters to be directly interpreted as a set of rules. Moreover, experiments show that this model is mostly competitive with black-box models such as random forests in terms of accuracy.

3.2 Architecture

As indicated by its name, the decision rules network is a neural network used to produce a decision rule set. This is achieved by using a two-layer fully-connected architecture, of which a simplified representation is found on Figure 3.1.

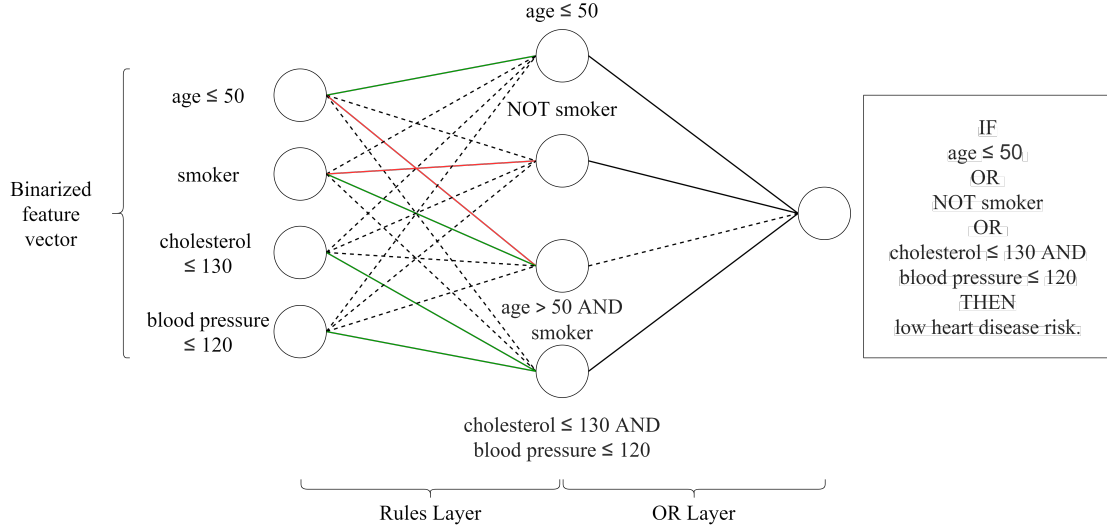


Figure 3.1: A simple example of the DRNet architecture, shown in [6].

The network takes as input a dataset with *binarised* features. This input format allows for the features to be interpreted as logical *literals* (or *predicates*), i.e., with a truth value. Moreover, as explained hereafter, DRNet’s architecture allows avoiding the need to represent both the positive and the negative form of the same input feature. As an example, if the feature “ $age \leq 50$ ” is present, then there is no need to also explicitly include its negation “NOT ($age \leq 50$)”, i.e., “ $age > 50$ ”.

The first layer that the input is passed through, is called the *rules layer*, or *AND layer*. It is a fully-connected linear layer, which includes a bias term. The intermediate outputs (one for each rule) of this layer are then passed through a binary activation function.¹ Thanks to a smart choice of the bias, the weights learned in the AND layer are directly interpretable: they indicate whether an input feature is associated positively, negatively, or not at all with the output. In particular, in case of association, whether it is positive or negative is determined by the sign of the weight, hence obviating the need to model both positive and negative expressions of the same feature in the input space.

¹In practice, in order for the gradient with respect to the weights to be defined, DRNet’s implementation uses *masked weights*. The weights themselves are not binary, but are multiplied by a *mask* when performing a forward pass. The backward pass uses a *straight-through estimator*.

As a second step, the output of the AND layer serves as the input for the *OR layer*. Again, this is a fully-connected layer – in this case, with a single output neuron – which computes dot products between its inputs and its weights, while also including a bias term. The resulting outputs are then passed through a sigmoid activation function, which finally yields a single *binary output* for the entire network. The way in which the forward pass is performed through the OR layer allows its weights to also be directly interpretable: they indicate whether the associated rule is part of the final rule set.

In summary:

- The AND layer handles features that can be interpreted as predicates, and uses them, in their positive or negative form, to create logical rules.
- The OR layer picks, among those logical rules, those that will be part of the final rule set.

This means that our model’s weights are interpretable as a decision rule set in *disjunctive normal form (DNF)*, i.e., a disjunction of conjunctions of predicates. For R rules, if we denote the predicates by $A_{k,i}$, with $i \in \{1, \dots, n_k\}$ for rule $k \in \{1, \dots, R\}$, this can be written as:

$$\{(A_{1,1} \wedge \dots \wedge A_{1,n_1}) \vee \dots \vee (A_{R,1} \wedge \dots \wedge A_{R,n_R})\}.$$

As this is a *set* of rules, there is no notion of order between them. If an input matches *any* of these rules, the output of the network will be positive; otherwise, it will be negative.

3.3 Training Process

In order to conclude our description of the DRNet model, it is worthwhile to note a couple of peculiarities about its training process.

As we know, DRNet uses the Adam optimiser [14] for training, which operates in time steps. This would typically mean that at each time step, every parameter of the model will be updated. However, instead, DRNet opts for a *two-phase learning strategy* [6]. As one might guess, these phases are dedicated respectively to the AND layer and the OR layer, and are repeated until convergence. During one phase, the parameters of the AND layer are updated, while those of the OR layer maintain fixed values. During the next one, the opposite applies: the AND layer is “frozen”, while the parameters of the OR layer are updated. There are thus two different Adam optimisers, each used in every other phase.

This particular approach has been chosen as it has been observed to lead to efficient training in practice. It also allows for finer tuning of the regularisation coefficients of these two layers [6]. Intuitively, we can also justify this strategy through the structure of our network, which inherently involves two processes: the selection of the predicates, and that of the rules containing these predicates.

Another particularity relates to the computation of the gradient in the AND layer. As it involves a binary *step* activation function, it is not differentiable, which signifies that the gradient must be estimated through other means. The chosen approach was that of the *straight-through estimator* (STE) [28]. Let a non-differentiable activation function f be applied to activation a_i in order to produce h_i . The gradient of the loss with respect to a_i is then simply estimated as that of the loss with respect to h_i . That is, we back-propagate through f as if it was the identity function [28].

In [6], this approach is enhanced by applying some *clipping* to the estimated gradient. This is done to improve performance, as well as to prevent large activations from growing even further. Indeed, the activation of the AND layer does not need to grow larger than 1, as it will be binarised anyway. Therefore, the gradient of the loss with respect to a_i is set to 0 in the following cases:

- The activation a_i is < 0 . This mimics the behaviour of the ReLU activation function presented in Section 2.2.2 [6].
- The activation a_i is ≥ 1 and the gradient with respect to h_i (and then a_i after applying the STE) is < 0 .

This STE idea will be revisited later when we build our extension of the DRNet model.

3.4 Points of Interest

Now that the default architecture and training process have been made clear, we can set some objectives that we wish to achieve, in order to improve on this implementation. As in the original work, it remains desirable to achieve a good *accuracy-simplicity* trade-off. That is, obtaining a sufficiently large accuracy, while favouring the simplicity of the learned rules, regarding both the amount of rules and the number of predicates inside each of them, for easy interpretability. Furthermore, we may be interested in ways to make our network more *versatile*, that is, simplifying as much as possible the process of obtaining classification results given an initial, unprocessed dataset containing different data types. Finally, we also seek methods allowing for training the model *faster*, and in a way that is more *robust* with respect to meta-parameters and initialisations.

Several elements of DRNet’s initial implementation – as described in [6] – have piqued our interest, as they present opportunities to modify the model in order to reach our objectives.

First, the network’s input must be *binary*. But of course, the vast majority of datasets that are handled in practice do not exclusively display binary features. Therefore, the authors apply some pre-processing, in order to accommodate for numerical and categorical data.

In practice, *one-hot encoding* is applied to categorical features, which yields a new binary feature for each value that this feature could take. As for numerical features, in a method named *quantile discretisation*, they are binarised by being compared to a certain number n of thresholds: $t_1, \dots, t_n$, with $t_i \leq t_{i+1}$. This yields n binary features, where the inputs are encoded as 1 or 0 depending on whether they are smaller than the associated threshold. By default, the value of n is set to 9, or less if a certain feature does not contain as many as 9 different values. Note that these new binary features replacing the numerical ones are not mutually exclusive, as $x \leq t_i \Rightarrow x \leq t_j, \quad \forall j \geq i$.

However, this method of binarisation is isolated from the rest of the network. One can then wonder if there exists a smarter way of setting these threshold values, by taking the rest of the model into account. This is explored in Section 4.1, as we add a *thresholds layer* to the network.

Furthermore, in DRNet, the rules provided by the network are directly used to perform the classification of new examples. This approach is fairly naive, especially in a context in which we seek rules that are as *sparse* as possible. Even though a rule set is regarded as an interpretable model, an excessively intricate rule set involving many conditions is not as interpretable as a simpler one. We can thus consider applying *post-processing* methods to the rule set provided by DRNet. The goal being to prune some predicates within a rule or even entire rules, while maintaining a model that remains equivalent to the “raw” one. This question is addressed in Section 4.2, in which we study several such post-processing techniques.

3.5 Conclusion

By taking a look at the constitution of the DRNet model, we have discovered an architecture split into two parts. The AND layer is responsible for selecting the predicates that will be part of the different rules, while the OR layer is tasked with choosing, among those rules, those that will constitute the final rule set. This particular architecture for a neural network, which would usually be considered as a black-box model, makes it easily interpretable. Afterwards, analysing

the specificities of this model's training process has further highlighted the two distinct steps that make up the rule set's constitution.

Having gained a comprehensive understanding of the DRNet model, we have identified several areas for improvement, as well as the objectives that will guide us in extending this model. Namely, we want to modify its architecture in order to enable the model to seamlessly handle data of different types and properties. Integrating this process into the network will make it able to account for the specificities of each dataset it encounters. At the same time, we want to maintain high performance in terms of accuracy and simplicity. The latter point further motivates the development of post-processing techniques to prune portions of the final rule set.

Chapter 4

Model Enhancement

It is time to dig into the details of how we have modified and enhanced the original DRNet model, in the quest of reaching our objectives. This part describes the elements that make this model our own, the motivation behind them and the concrete ways in which they have been implemented.

4.1 Thresholds Layer

A dedicated *thresholds layer* has been prepended to DRNet’s original implementation, with the objective of obtaining a binarisation of numerical features which is more flexible and integrated with the rest of the model than what we would have with a fixed quantile discretisation approach. Indeed, in our new approach, each threshold has become a learnable parameter. The maximum number of thresholds per feature will be denoted n throughout, and will determine the size of this thresholds layer.

4.1.1 Initialisation

Attempting to apply threshold learning to unprocessed numerical features leads to a problem of magnitude, as it can often happen in practice that different numerical features within the same dataset have very different ranges of values. Therefore, we prefer to work with input data that have been *normalised* between 0 and 1. In particular, we perform *min-max normalisation* [29]: for a given feature i , each input value x_i is mapped to $\frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)} \in [0, 1]$, where $\min(x_i)$ and $\max(x_i)$ respectively denote the minimum value and the maximum value observed for feature i in the training dataset. Note that in the context of train-test splitting, this normalisation is performed after the split, in order to not “leak” any information from the test set into the training set.

Afterwards, arises the question of which *initial values* to assign to the thresholds. One possibility, for n thresholds per feature, would be to initialise them in an equidistant fashion in the $[0, 1]$ interval, yielding: $\left[\frac{1}{n+1}, \dots, \frac{n}{n+1}\right]$. The rationale behind this choice would be that it leads to an efficient exploration of the values in the domain of the thresholds.

However, this initialisation does not take the data themselves into account at all. Therefore, we prefer to use an approach based on *percentiles* in the training set. For n thresholds per feature, the initial threshold values will go from the $\left(100\frac{1}{n+1}\right)^{\text{th}}$ percentile to the $\left(100\frac{n}{n+1}\right)^{\text{th}}$ percentile. Note that we use a quantile-based approach as in DRNet’s initial, fixed implementation, but only for the initialisation of the thresholds. During training, these will be free to evolve towards other values.

4.1.2 Implementation

In practice, the thresholds layer is represented by a *matrix*, denoted T , containing the learnable thresholds associated to each feature. This is reminiscent of the way in which linear layers are implemented, but we will later note some differences. The entries of T are initialised as described in Section 4.1.1.

As the way in which categorical or mixed input features are handled will be the topic of Section 4.1.3, we focus here on *numerical* input features. The dimensions of this thresholds matrix T are $(n_f \times n)$ with n_f denoting the number of input features and n , as usual, the maximum number of thresholds per feature.

Forward Pass. A *forward pass* through the thresholds layer works in the following way. After normalisation, the input of the thresholds layer is $x \in [0, 1]^{n_f}$. Let $x_i \in [0, 1]$ denote the i^{th} feature of x . During the forward pass, for each $i \in \{1, \dots, n_f\}$, x_i ’s value is compared to those of the n thresholds in $T_{i,:}$, yielding an n -dimensional vector whose k^{th} entry is equal to 1 if $x_i \leq T_{i,k}$ and 0 otherwise. All of those vectors are then gathered into a single one of dimension $n_f \cdot n$, which constitutes the output of the thresholds layer.

We note here some dissimilarities with the way in which a classical linear layer would operate. We do not multiply the input by T , which would yield an output vector of dimension n , but instead we produce one output feature for every single entry of T , by comparing it to the relevant input feature x_i .

As the output of the thresholds layer is a binary vector, this allows this new part to seamlessly integrate with the rest of the network: the AND layer directly takes as input, the output of the thresholds layer. Therefore, one can visualise this new layer as pre-processing the data before injecting it into (an adapted version of) the original architecture of DRNet. The key interest being, of course, that this

pre-processing is *dynamic*: it evolves during the training of the model, in line with the other layers of DRNet. An illustration of this process is presented on Figure 4.1, in the case where $n = 3$.

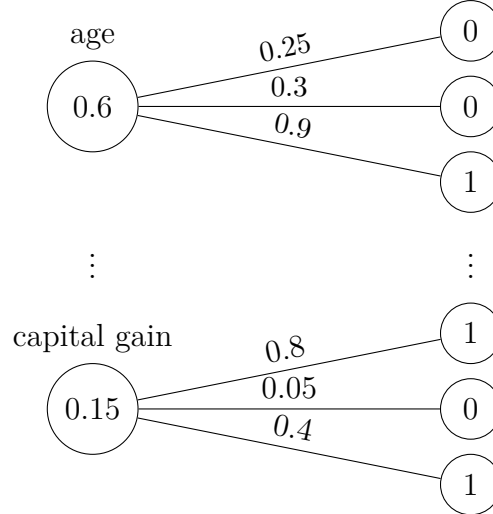


Figure 4.1: Forward pass for *numerical*, normalised input values. The features originate from the *house* dataset (see Section 5.1), but the values are not necessarily representative of those encountered in the dataset.

As observed on Figure 4.1, the thresholds for a given feature are not necessarily ordered. Indeed, order has no importance in our model, as the predicates within each rule are interchangeable.

In this example, we have $T_{1,:} = [0.25 \ 0.3 \ 0.9]$, and $T_{nf,:} = [0.8 \ 0.05 \ 0.4]$. Other features are not represented but behave in the same manner. Finally, as all thresholds on Figure 4.1 are in $[0, 1]$, they are all “active”, in the sense that they can binarise some inputs to the value 1, and others to the value 0. This is not always the case, as will be discussed later regarding *clipping*.

In the Python implementation of our model, the forward pass is performed efficiently thanks to vectorised operations. It performs the comparison operation described above. One may then notice that this operation is not differentiable, as it corresponds to a *step function*, as represented on Figure 4.2.

In this sort of situation, differentiable “approximations” of a step function, such as the *sigmoid* function, naturally come to mind, as illustrated on Figure 4.3. This means that instead of computing $\text{bin}(T_{i,k} - x_i)$, we would compute $\sigma(T_{i,k} - x_i)$,

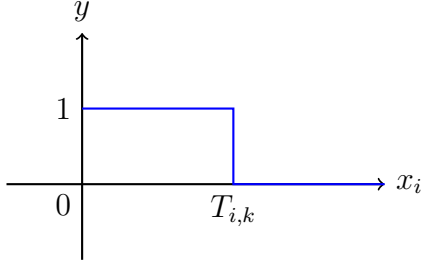


Figure 4.2: Step function representing the application of threshold $T_{i,k}$ to feature x_i .

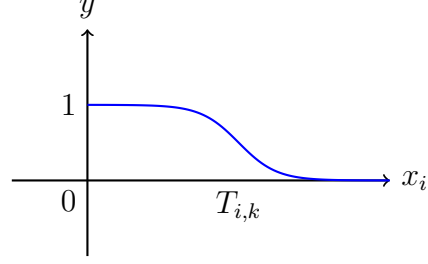


Figure 4.3: “Softer” version of the step function, using a sigmoid.

where these functions are defined respectively as

$$\text{bin}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}, \quad \text{and} \quad \sigma(x) = \frac{1}{1 + e^{-x}}.$$

We observe indeed that the bin function does perform the required comparison operation, as $\text{bin}(T_{i,k} - x_i) = 1$ iff $x_i \leq T_{i,k}$. But on the other hand, $\sigma(T_{i,k} - x_i)$ may take any value in $[0, 1]$. This does not fit our model’s purpose as, to allow it to remain interpretable, the AND layer must strictly take values in $\{0, 1\}$. The output of a sigmoid function would thus need to be further binarised before entering the AND layer, which renders the sigmoid function useless. Therefore, it seems unavoidable to use a non-differentiable step function as part of our thresholds layer. This problem thus requires some consideration when performing a *backward pass*, as described hereafter.

Backward Pass. A *backward pass* through the thresholds layer involves computing the gradient of the network’s output – or rather of the loss function $L(\hat{y}, y)$ computed from the predicted output $\hat{y}$ and the expected one y – with respect to both the thresholds and the layer’s input. As observed previously, the operation performed by the thresholds layer is not differentiable. Therefore, we choose to apply the same approach which was used in [6] regarding the AND layer, that is, using a *straight-through estimator* [28], the main idea being to *copy* the gradient of the output of the layer.

It is then necessary to consider the *dimensions* of the different objects we are handling, in order to define the gradients correctly. These special considerations arise from the fact that as discussed previously, the thresholds layer does not act as a classical linear layer regarding the dimensions of its inputs, parameters matrix and outputs. Let us thus consider an input of dimensions $(B \times n_f)$ with B the *batch size* used in the training process. Indeed, in practice, inputs will not be passed through the model individually, but rather in batches. After passing this

input through our thresholds layer whose parameters are contained in the matrix T of dimensions $(n_f \times n)$, the dimensions of the output are¹ $(B \times (n_f \cdot n))$. This means that when using a straight-through estimator and copying the gradient of the output layer, we are left with an object of dimensions $(B \times (n_f \cdot n))$.

The gradient with respect to the thresholds is then obtained by computing the mean over the B inputs in our batch, leaving us with a vector of size $n_f \cdot n$, which can be easily resized as a matrix of dimensions $(n_f \times n)$, thus fitting the dimensions of T . Similarly, the gradient with respect to the network's input is obtained by resizing that of the output as a tensor of dimensions $(B \times n_f \times n)$, and then taking the mean over the last dimension. The idea in both cases is that several elements contribute to the same gradient. Therefore, we compute it by aggregating over all of them.

Clipping. The initial values of the thresholds lie in the $[0, 1]$ interval, and so do the values of the data we are dealing with after normalisation. It would thus be natural for the thresholds, throughout the whole learning process and for the final rule set, to remain in this interval as well. However, this is of course not guaranteed in general. A positive (resp. negative) gradient for a threshold could lead this threshold, through the learning process, to become smaller than 0 (resp. larger than 1). This is why we prefer to limit the range of values that each threshold can take. In practice, this is done after each forward pass, by *clipping* the values of all thresholds to the interval $[-0.1, 1.1]$. This prevents the threshold values from straying too far away from the interval of interest.

After learning, a threshold outside $[0, 1]$ – i.e., since we performed clipping throughout – a threshold in $[-0.1, 0[\cup]1, 1.1]$ can be interpreted as a threshold which is not *active*. Indeed, as described in Section 4.2, such thresholds lead to predicates that are either always satisfied, or never satisfied. These do not allow discriminating between different inputs, rendering them effectively useless. This is especially prevalent in the case where the maximum number n of thresholds is set to high values. If there are not as many as n groups into which a given feature could be divided using thresholds, then some of those n thresholds will be useless.

4.1.3 Categorical Data

The thresholds layer was first added to DRNet with the purpose of integrating the binarisation of *numerical* features more closely with the rest of the learning

¹The $\times$ and $\cdot$ notations that we use are not to be mistaken: $\times$ is used to separate the different dimensions of a matrix or tensor, while $\cdot$ is the product of two scalars within one of these dimensions.

process. But how can this layer handle *categorical* data? For those, there is usually no notion of order between the different categories for which we could use thresholds.

As was the case in the original implementation of DRNet, categorical features can simply be binarised through *one-hot encoding* during pre-processing, yielding one new binary feature for each value that the initial feature could take. As such, they could already be passed through to the AND layer of the network. But we would prefer for numerical and categorical features to be handled in the same manner, i.e., also applying our new thresholds layer to the binary features obtained from one-hot encoding.

However, any threshold in the $[0, 1[$ interval will *flip* our one-hot encoding: every 0 will become a 1, and vice versa. Note that the value 1 is not included, but 0 is. Indeed, as the $x_i \leq T_{i,k}$ inequalities computed during the forward pass are not strict, a threshold of 1 would leave any 1 unchanged, while a threshold of 0 would still flip any 0. Moreover, any threshold $T_{i,k} < 0$ or $T_{i,k} \geq 1$ will respectively lead to all binary values to be cast to 0 or to 1.

A simple trick allowing us to pass these features through the thresholds layer is then to *already flip* the one-hot encoding during pre-processing: each categorical feature with m categories will be encoded through $(m - 1)$ 1's and one 0. Applying any threshold in the $[0, 1[$ interval to such features – e.g., 0.5 which is the value we opted for – will then directly flip them back to their initial one-hot encoding. This means that the categorical features passed through the rest of the network will preserve their initial interpretation, and the rules learned from those will make sense with regard to the initial data. Finally, since these thresholds need not be learned, they are fixed back to their value of 0.5 after each pass through the network.

This entire process is illustrated on Figure 4.4 for $n = 2$ thresholds. In this example, we consider one categorical feature named “colour” which can take three values: *red*, *green* or *blue*. For the input example in Figure 4.4, that value is *green*. Applying the *flipped* one-hot encoding then yields three new binary features: one for each potential value for “colour”. As the initial value was *green*, the *col_green* feature takes the value 0, while the other two values are 1. These can now be passed through the thresholds layer of our network. As we consider binary features, all thresholds are fixed at 0.5. Comparing the previous values to these thresholds finally allows falling back on the usual one-hot encoded values, which constitute the input of the subsequent AND layer.

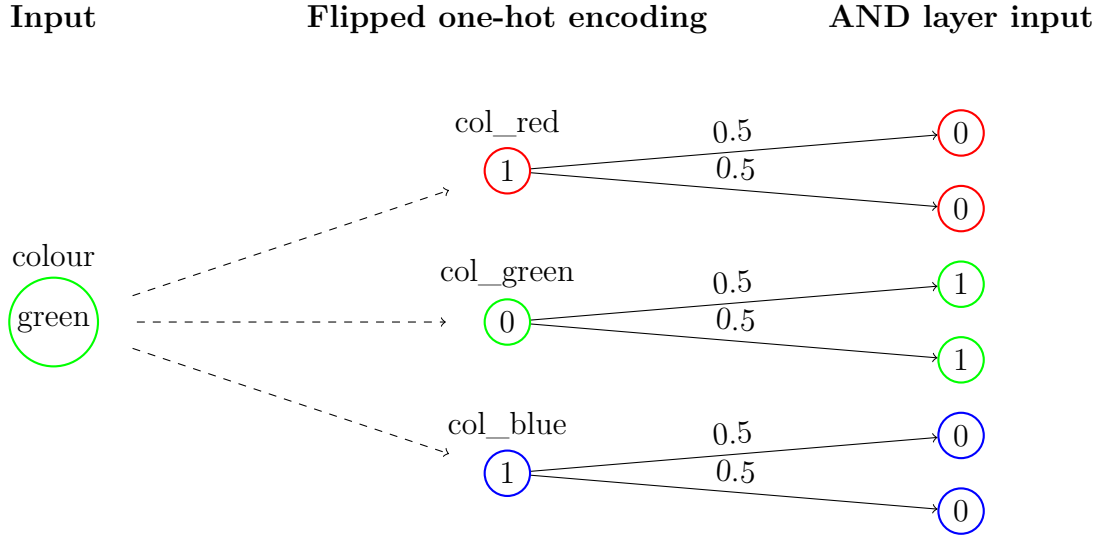


Figure 4.4: Representation of one categorical feature, from its initial form to the way it is passed through the AND layer. The first set of arrows is represented as dashed, as those do not represent actual links (or *synapses*) in the network.

We observe that each one-hot encoded feature is converted into n different features when passed through the thresholds layer, which has the disadvantage of increasing the number of parameters of the network (but those added parameters remain fixed to 0.5), as well as risking complexifying the learned rules, or to make them unsatisfiable. An obvious alternative would have been to only create one new feature, instead of n , for each of the one-hot encoded ones. However, we did not opt for this approach, as the benefits of ours outweigh its drawbacks. Indeed, as we have observed up to now, representing the thresholds layer as a single matrix of dimensions $(n_f \times n)$ is extremely convenient². In particular:

- This allows to vectorise the operations of the forward pass.
- We have based our gradient backpropagation through the thresholds layer on the dimensions of this matrix and the resulting output.
- This approach lets us reason about numerical and categorical features – and the way in which they interact with the thresholds layer – in the exact same way.
- Post-processing techniques can be more generic, having the same effect on both numerical and categorical features. This is explained in Section 4.2.2.

Finally, in the event of redundancy or contradictions in the learned rules, the prun-

²With the “one feature” approach, thresholds could not be gathered into such a matrix anymore. Indeed, while numerical features would each have n associated thresholds, categorical features would only have one. Other entries of a thresholds matrix could not even be filled with zeros for these features, as zeros are actual threshold values with a meaning.

ing techniques described in Section 4.2 will be applied, leading to an equivalent but simpler rule set.

4.1.4 Training Process

The training process of DRNet is split into different phases, as was noted in Section 3.3. When the parameters of a layer are updated, those of the other one remain fixed. Given that we have prepended a new thresholds layer before the AND layer, we must adapt the training, which now follows a *three-phase learning strategy*. The principle remains the same: updating the parameters of each of the three layers one layer at a time. One advantage is that we can pick the learning rate used for our new layer separately from the one used for the other two. In practice, we have observed that setting it to lower values than those provided better results; as otherwise, the threshold values tended to vary too quickly.

As there are now three different types of phases, we have more choice regarding their *order*. The parameters of the AND and OR layers are initialised at random. Therefore, it would not make sense to start training the thresholds layer parameters based on those values. We thus start with two phases that train these two layers one at a time, before moving on to the thresholds layer. As usual, this process is then repeated until we reach the maximum total number of epochs that we have chosen. It remains to choose which layer to start with among the AND and the OR layer, and which one to terminate with. In practice, the setup that has worked best is to start with the training of the AND layer, and finish with that of the OR layer. Training the OR layer last has sometimes been observed to lead to rule sets that were too sparse, at the cost of accuracy. But with a sufficient number of training epochs, this effect was largely reduced, and training the OR layer last has allowed to obtain sparser rule sets without much impact on accuracy.

4.2 Post-processing

Although our network provides a final decision rule set after having been trained, it is not always the case that all the rules, and all the predicates within them, are actually useful to make the final decisions. Therefore, in our quest for sparse but accurate rule sets, a crucial step is to apply some *post-processing* to the rules.

4.2.1 Uninformative Rules

First off, it can happen in practice that not all the rules selected by the OR layer contribute to the decisions. In particular, some rules are not even observed in the

training set, i.e., there is no example in the training set for which all predicates that compose the rule are satisfied. During the training process, such rules have thus never been applied to perform a prediction, and can be pruned. Similarly, rules that are always verified – e.g., due to redundancy among certain features – may also be pruned.

In practice, both of these types of rules are removed by keeping track of which rules apply to which training examples. Let us consider a dataset with B examples, such that for each $i \in \{0, \dots, B-1\}$, R_i is the set of rules that apply to the i^{th} example. Then, the sets

$$S_1 = \bigcup_{i=0}^{B-1} R_i, \quad S_2 = \bigcap_{i=0}^{B-1} R_i$$

respectively contain the rules that apply to at least one training example, and those that apply to all of them. We can thus limit ourselves to the $S_1 \setminus S_2$ rule set.

4.2.2 Interactions between Thresholds

After removing entire rules, we can now consider post-processing techniques that allow to remove some predicates from *within* a given rule. As several thresholds are applied to the same feature, it can easily happen that some of them are *redundant*. Indeed, we can consider the case where, in a given rule, we both have the predicates $x_i \leq t_1$, and $x_i \leq t_2$. Then the *largest* of the two thresholds is redundant:

$$x_i \leq \min(t_1, t_2) \Rightarrow x_i \leq \max(t_1, t_2).$$

In a similar manner, for negative conditions, if we both have, in a given rule, NOT $(x_i \leq t_1)$ and NOT $(x_i \leq t_2)$, then the *smallest* of the two thresholds is redundant:

$$\text{NOT } (x_i \leq \max(t_1, t_2)) \Rightarrow \text{NOT } (x_i \leq \min(t_1, t_2)).$$

This post-processing approach thus has the potential to make each rule of the rule set much shorter. Each feature will appear at most once in its positive form and its negative form.

Conversely, different thresholds which apply to the same feature might be *contradictory*, which requires the entire rule to be pruned. This only happens in the following case: both predicates $x_i \leq t_1$ and NOT $(x_i \leq t_2)$ are present in the rule, with $t_1 \leq t_2$. Note that the edge case where $t_1 = t_2$ also requires pruning. This will especially be important when considering categorical features, as explained hereafter.

Due to the way in which our network handles categorical features, described in Section 4.1.3, this method of pruning allows – without any adaptation required – to prune redundant or contradictory rules regarding such categorical features. Indeed, let us consider for example a categorical feature named “Sunny”, which can only take the values 0 and 1. A single rule containing the “Sunny” predicate several times corresponds to the $Sunny \leq 0.5$ feature being present more than once within the same rule. Then, applying our pruning method, all but one of these occurrences will be removed. Similarly, a rule that would both contain “Sunny” and “NOT Sunny” would be removed from the rule set, due to the fact that it displays both the $Sunny \leq 0.5$ and NOT ($Sunny \leq 0.5$) features, which are contradictory.

We conclude this part by observing that *redundant* predicates are expected to arise in practice. Indeed, the model can easily consider several predicates as useful to perform accurate predictions, since they actually are, but one of them would be sufficient. On the other hand, a well-trained model should seldom contain *contradictory* rules. Those obviously do not apply to any training example, and their pruning should thus already be performed by the OR layer of our network, which is indeed what is observed a vast majority of the time in practice, as displayed in Section 5.4.3.

4.2.3 Clipping

As mentioned in the part of Section 4.1.2 that describes the *clipping* process, we do not strictly restrict the values of our thresholds to the $[0, 1]$ interval. Rather, we allow them to grow as high as 1.1, or to shrink as low as -0.1 . Such values outside $[0, 1]$ can lead to four different situations, which can be gathered into two groups, and all allow for some pruning:

- When a threshold $T_{i,k}$ is < 0 , then the condition $x_i \leq T_{i,k}$ is never satisfied. Similarly, when $T_{i,k} > 1$, the condition NOT ($x_i \leq T_{i,k}$) is impossible to satisfy. Such false predicates lead to the entire rule in which they appear evaluating to false as well. Therefore, in these cases, the entire rule can be pruned.
- On the other hand, if $T_{i,k} > 1$, then $x_i \leq T_{i,k}$ is always satisfied. The same can be said about NOT ($x_i \leq T_{i,k}$) when $T_{i,k} < 0$. If such predicates occur, they can thus be pruned from the rule that contains them.

4.3 Parameters Initialisation

A final element that we endeavoured to improve relates to the *initialisation* of the weights that constitute the AND layer and OR layer of the original DRNet

(i.e., without a thresholds layer). Indeed, in the implementation from [6], those are initialised *randomly*, either according to a uniform distribution, as mentioned in the paper, or to a normal distribution, as found in the code. It is thus natural to wonder how to initialise these weights in a smarter way, and whether this can improve the model’s performance.

The idea we opted for was to learn a first rule set using the RIPPER model [27], and to initialise the weights of DRNet based on this rule set. Several approaches have been considered:

- directly representing the RIPPER rule set into DRNet, without modification,
- the same principle, but where we also pick random values for the weights corresponding to predicates or rules that are not part of the RIPPER rule set,
- suggesting the RIPPER rules for a part of the DRNet rule set, but initialising the rest randomly, to allow for more opportunities of identifying new ones.

Playing around with these approaches, we found that the last one tended to give results that were slightly better than the others. But in general, we did not observe any real gain in using any of those approaches compared to a random initialisation. For this reason, and the fact that we found it preferable for our model’s results to not depend on those provided by RIPPER, we have decided to abandon this idea in order to focus on the points presented in the previous sections.

4.4 Conclusion

The thresholds layer is the central element of our extension of DRNet, and allows the binarisation of the features to be carried out directly within the network. We have carefully defined the different aspects of this new layer, from its initialisation to the effect it has on the data to its training process. Each step has involved many choices, for which we have made sure to favour the interpretation, ease of use and adaptability of this component. This way, we already make a great step forward in meeting part of the objectives of this work.

Furthermore, by pondering about the quality of the rule set returned by our model, we have found several improvement opportunities through post-processing. Thereby, we have identified ways in which our model could return redundant or contradictory elements, which are now removed from the final rule set.

Chapter 5

Performance

The enhancements that we have performed on DRNet have already allowed us to fulfil one of our objectives: that of versatility, as the network can now handle numerical and categorical features in an analogous manner. It is now time to test how well the model performs with regard to our other objectives, namely the accuracy, the simplicity of the rules, and a potential efficiency gain.

To this end, we have tested it on several datasets, described in Section 5.1. After presenting the other models to which we have compared ours in Section 5.2 and our experimental setup in Section 5.3, we finally display and interpret our results in Section 5.4.

5.1 Datasets

Our model has been tested on a variety of datasets. These include the four that were used in the original DRNet paper [6] – respectively named *adult*, *magic*, *house* and *heloc* – as well as a few additional datasets in order to perform more extensive tests and tackle specific issues: *drugs*, *occupancy* and *rockburst*. While references to the first four datasets can be found in [6], all the remaining ones are from the UCI Machine Learning Repository [30].

For each dataset, we dedicate a subsection in which we describe the data that it contains. We also mention specificities about these data, such as their type, as well as potential pre-processing steps that were performed.

5.1.1 Adult

In this dataset, the aim is to correctly predict whether a person makes more than \$50 000 per year or not, based on census data. Both numerical (e.g., age, number of hours of work per week) and categorical (e.g., marital status, native country) features are present. This sort of mixed dataset is relevant to test our model, as it can easily handle both types of data.

5.1.2 Magic

The topic of interest for these data is astronomy. In particular, the task consists in discriminating between two types of *particle showers*, based on features that were extracted from an image of said shower. All of those features are numerical, which is fitting in order to evaluate the way in which our thresholds layer produces new binary features from them.

5.1.3 House

This constitutes a classical machine learning problem, in which the price of a house is to be predicted, using regression. It has been turned into a binary classification problem by the creators of this version of the dataset: the goal is now to determine whether the house price is above or below the average, based on a set of all numerical features.

5.1.4 Heloc

In this fourth and final dataset used in [6], the task is to predict whether homeowners who apply for a *Home Equity Line of Credit (HELOC)* will reimburse the required amount within a delay of two years. The prediction is based on features that mostly revolve around the applicants' previous credits and finances, and are all considered as numerical. That is, even features that were initially categorical, have been treated as numerical in [6], certainly due to the notion of order among them.

5.1.5 Drugs

The drug consumption dataset gathers features about different subjects and their consumption of a variety of drugs. Those range from food products such as chocolate to illegal substances such as cocaine, while even including an imaginary drug in order to identify individuals who would overclaim their use. For each of these drugs, a classification problem then consists in identifying the person's use level, among seven possible classes. As our work only considers binary classification, we follow the approach recommended by the dataset creators, by gathering five of those classes into a wider "User" class, while the remaining two constitute the

“Non-user” class. Among all nineteen available drugs, we then choose to focus exclusively on cannabis, as quite a lot of people use it (unlike harder drugs) but not an overwhelming part of the population (unlike chocolate or coffee), thus striking a good balance between both classes. Imbalanced classes, on the other hand, will be further discussed in Section 5.1.7.

Finally, another particularity of this dataset is the fact that all features have been *quantified*, and are thus represented by real values. But for our model, it is not necessary to exclusively work with real values: considering mixed features is quite interesting, as mentioned in Section 5.1.1. Therefore, we choose to map some of the numerical features back to categorical ones when appropriate. Namely, this was done for the *age*, *gender*, *education*, *country* and *ethnicity* features.

5.1.6 Occupancy

Based on a few features such as the temperature and humidity measured in an office room, the aim is to determine whether someone is present in this room or not. This dataset was chosen because it has an additional interesting aspect: *temporality*. Indeed, these measurements are collected approximately every minute, and are presented successively. A single train-test split is thus already incorporated in the dataset. It would not be wise to randomly split the data in another manner to perform cross-validation, as we would lose this temporal aspect.

As the precise date and time at which each measurement was taken are provided as a feature, they can be used in our rules. For example, given that the room that we consider is located in an office building, we would not expect it to be occupied outside office hours, and especially not during the night. Therefore, we decided to create two new features. First, from the provided date, we create a categorical feature whose value is the day of the week to which this date corresponds. As an example, “2015-02-04” becomes “Wednesday”. The point being that again, in an office building, we can reasonably expect a room’s occupancy to vary based on the day of the week. Secondly, we have converted the time into a numerical feature, which our network is directly able to handle. As such, the time “17:15” is mapped to the numerical value 17.25.

Furthermore, in order to account for the fact that data at a given minute are highly related to those collected during the previous minutes, we add *rolling window* averages to our dataset. For instance, we add a “CO2_avg_10” feature which contains the average CO2 level in the room from the 10 previous measurements, starting from the current one. The idea here is to identify trends that carry on over time: if the average CO2 level during 10 minutes is above a certain threshold, it might be a more reliable way to detect the presence of someone in the room,

rather than a one-off variation. The ability of our model to handle this prediction problem is studied in Section 5.4.4.

5.1.7 Rockburst

In this final dataset, we find data about the seismic activity measured during 8-hour shifts of mining activity. We use these in order to predict whether a *seismic bump* of high intensity will be observed during the next shift. This acts as a surrogate for the event of a *rock burst*, which is both rare and dangerous. These two facts are the key factors for which we have chosen this dataset. Indeed, those imply that few positive examples are present in the dataset, and that correctly predicting those is critical. This is thus an opportunity to evaluate how our model performs in *imbalanced* classification contexts – even though it was not originally developed with this precise goal in mind – and is explored in Section 5.4.5.

5.2 Other Models

When considering the accuracy and the simplicity of rule sets, it is natural to compare our model to other existing ones, in order to evaluate its relative performance. Namely, we use as reference the original, unmodified version of DRNet, as well as the RIPPER rule learning algorithm, which uses a more classical approach to extract rules than our own model, and was also used for comparison in [6].

5.2.1 Original DRNet

This is the Python implementation of DRNet from which we have started to build our extensions. As such, this network does not contain a thresholds layer. Rather, numerical features are binarised through quantile discretisation with a fixed number of (at most) 9 thresholds – as described in Section 3.4 – and categorical features are transformed into binary ones using a regular (non-flipped) one-hot encoding. We also note that unlike in our new model, numerical features are not normalised before the quantile thresholds are applied to them.

Regarding training, we use all the same parameters as for our new model. These are described in Section 5.3. Furthermore, we turn back to a two-phase process, in which the AND layer and the OR layer are alternatively trained for a given number of iterations. For the sake of comparison, the total number of epochs, the number of successive training epochs per layer (named *num_alter* hereafter) and the batch size are set to the same values as for our new model, whereas they were respectively set to 10 000, 1000 and 2000 in [6]. Finally, we of course do not apply any of the post-processing methods described in Section 4.2 to the rules obtained with the original DRNet, as this step was not a part of this model either.

5.2.2 RIPPER

The RIPPER algorithm [27] implementation that we use is named *JRip*, and is provided in the *Weka* library for Java, in its stable 3.8.6 version, developed by the University of Waikato [31]. In order to easily use the methods provided by Weka in our Python programs, we make use of the `py4j` Python package, which allows handling Java objects from a Java Virtual Machine directly inside Python.

When using JRip, we perform the same pre-processing steps as with the original DRNet. In particular, we apply quantile discretisation to numerical features in order to binarise them, as required by the fact that this particular implementation of RIPPER does not handle numerical features. For each fold during cross-validation, the training and test sets are then separately converted to *ARFF* (*Attribute-Relation File Format*) [31]. This type of file is separated into two parts, first listing each attribute of the dataset and the types of values it is associated with in the *header*, and then listing the data. For each categorical feature, the set of values that it can take must be provided in the header. This is the format used to represent data in the JRip implementation. Afterwards, the RIPPER model is trained, before being evaluated on the test set.

5.3 Procedure

We now present the different parameters and approaches that were chosen in order to run our experiments. For all of our experiments, 5-fold cross-validation was used, unless otherwise indicated. The folds were defined randomly, but with a fixed seed for reproducibility. When training both versions (old and new) of DRNet:

- The learning rate of the Adam optimiser [14] is fixed to 10^{-2} and no weight decay is used.
- The regularisation parameters of the AND layer and the OR layer are respectively fixed to $\lambda_1 = 10^{-2}$ and $\lambda_2 = 10^{-5}$.
- The maximum number of rules has remained set to 50, as this has never been a limiting factor in practice, in the sense that the rule sets consistently contained less than 50 rules. On the other hand, it could be interesting to try setting this parameter to lower values in order to force the rule sets to be sparser. But this did not seem necessary in our case, as sparse rule sets were obtained anyway, and a value of 50 provides greater room for manoeuvre during training. We will however come back to this issue later.
- The three-phase learning approach described in Section 4.1.4 is used for our new model, with the *num_alter* parameter, representing the consecutive

number of epochs per phase, being set to 400. For the original DRNet, the two-phase learning approach from Section 3.3 is used, with the same value for *num_alter*.

- The total number of training epochs is set to 3200. Combined with the previous point, this means that, for our new model, six phases are first performed, allowing to train each layer twice. This way, each layer has the opportunity to learn from the updated values in the other layers. This is followed by one final training phase of the AND layer – which allows improving the predicate choices based on the latest thresholds values – and one training phase of the OR layer in order to update the choice of rules to create the final rule set which constitutes our model. This number of epochs is quite low per se, when compared to the 10 000 epochs used in [6], which allows for a faster training process in our case.
- The batch size is set to 400 for most datasets. However, for the *drugs* dataset, this parameter is fixed to 100, as this dataset is smaller, containing only 1885 examples in total. This is also the case for *rockburst*, which only contains 2584 examples.

Finally, for our new model, the maximum number n of thresholds is a parameter whose impact is studied in the first experiment of Section 5.4. Afterwards, we set its value to the one that was observed to mostly perform best during this experiment.

5.4 Results

Now that our setup has been specified, we are able to carry out the experiments that allow us to quantify the performance of our model. Each of the following subsections considers one specific question regarding the impact of a parameter, the comparison with other models, or the way our model is able to adapt to new situations. Displaying, analysing and interpreting our results then allows answering those questions.

5.4.1 Maximum Number of Thresholds

The parameter n that we have been using since the beginning of Section 4.1 is related to the number of thresholds that are available for a given feature in our model. As we have noted when discussing *clipping*, all of those thresholds might not be used for every feature in practice, hence why this quantity represents the *maximum* number of thresholds for a feature. Therefore, this can be considered as a quantification of the expressiveness of the thresholds layer: larger amounts of thresholds lead to more precision in the expression of our rules, but also risk increasing the complexity of the rule set.

In this first experiment, we thus wonder: *what is the impact of this parameter on the following three metrics: accuracy, number of rules, and average number of predicates per rule?* The last two quantities are used to measure the rule set’s complexity. Naturally, we strive to maximise the accuracy, while minimising the complexity. We thus study the way in which those vary when considering different values for n , ranging from $n = 1$ – an extreme case in which only one threshold is available per feature – up to $n = 40$, which we consider to provide plenty of freedom regarding the choice of thresholds for each feature.

For all of those metrics, the values were *averaged* over 5-fold cross-validation. When plotted, around each value, we find a visual cue proportional to the *standard deviation* across those folds. Finally, for conciseness, even though the average number of predicates per rule has itself been averaged over the different folds, we still call the resulting quantity “average number of predicates per rule”.

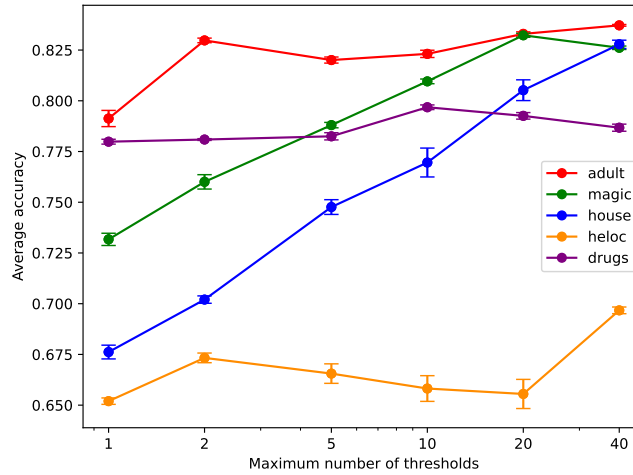


Figure 5.1: Accuracy of the model with respect to the maximum number of thresholds per feature n . The abscissa axis uses a logarithmic scale.

Figure 5.1 displays the way in which our model’s accuracy evolves with n , for our five datasets of interest. We observe as a general trend that the accuracy tends to increase with n , which was expected, as the model gains in expressiveness. For *heloc* however, the accuracy slightly decreases from $n = 2$ to $n = 20$. The differences are however quite small, especially when considering the standard deviation, which is larger in those cases. Even for that dataset, we observe a 5% accuracy gain when going from $n = 1$ to $n = 40$.

Despite these observations, we cannot simply conclude that a larger value of n is always better. We suspect that it will also tend to increase the complexity of our

rule set. As we aim for a good accuracy-simplicity trade-off, we must thus take both factors into account.

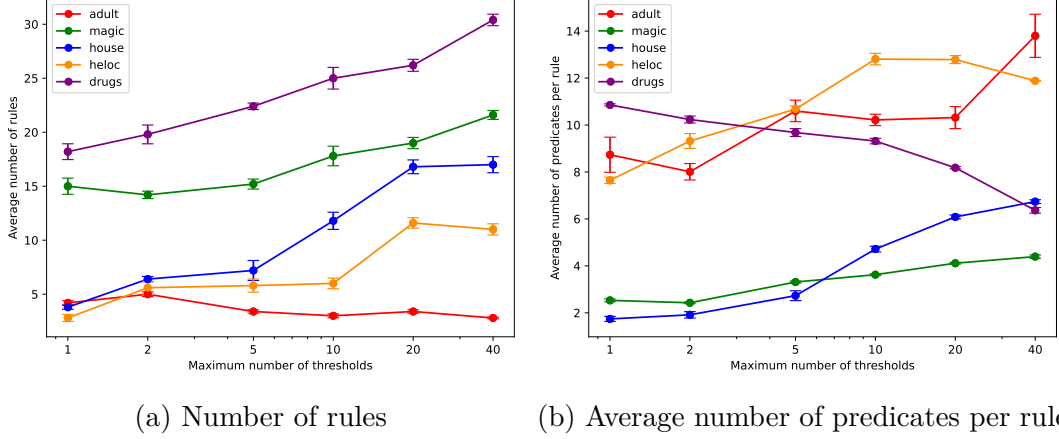


Figure 5.2: Complexity of the rule set with respect to the maximum number of thresholds per feature n , *after* post-processing. The abscissa axis uses a logarithmic scale.

And indeed, we notice on Figure 5.2 that the complexity of the rule set does tend to increase with n . For the *magic*, *house* and *heloc* datasets, this is both noticeable when considering the number of rules and the average number of predicates per rule. For *drugs* however, the latter quantity tends to decrease when n increases. But we directly notice on 5.2a that this effect is compensated by the increase in number of rules. Finally, for *adult*, we observe a similar but opposite phenomenon.

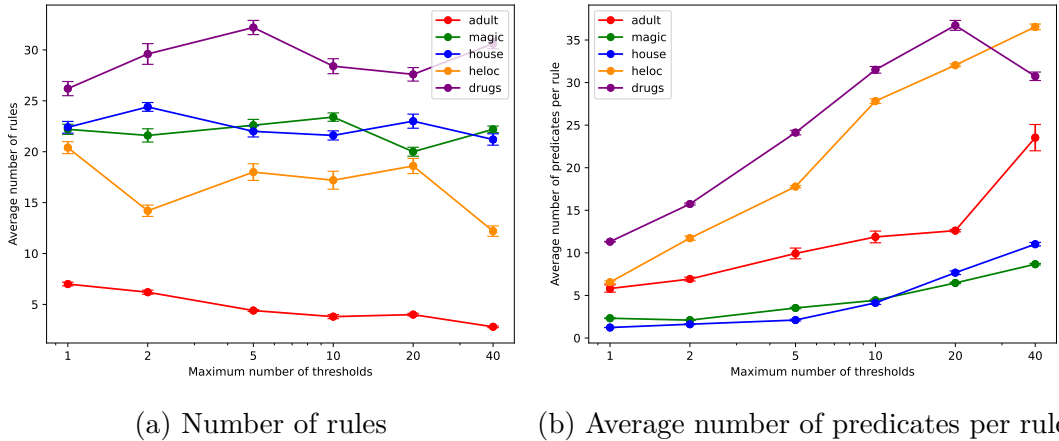


Figure 5.3: Complexity of the rule set with respect to the maximum number of thresholds per feature n , *before* post-processing. The abscissa axis uses a logarithmic scale.

It is also worthwhile to study the impact of n on the rule set's complexity *be-*

fore performing our post-processing steps, which is what we represent on Figure 5.3. Here, we notice that increasing n does not generally lead to modifying the number of rules that constitute our model before post-processing. Comparing Figure 5.3a to 5.2a reveals however that fewer of those rules are pruned when n increases. As for the average number of predicates per rule, it generally follows a significant upwards trend, depicted on Figure 5.3b, which is much more noticeable than on Figure 5.2b. This makes sense, as increasing the maximum number of thresholds increases the potential for redundant thresholds within a rule, as well as for thresholds outside the $[0, 1]$ interval leading to predicates that are always satisfied. Those are all pruned by our post-processing methods. Further analysis of the post-processing’s effectiveness will take place in Section 5.4.3.

Given our observations, we can now settle on a value of n for each dataset, to which we will fix this parameter for the next experiments. When considering complexity, we base our decisions on the results from Figure 5.2, i.e., after post-processing, as those are the rules that make up our final rule set. For *adult* and *magic*, we choose a value of 20. Indeed, further increasing n leads to an increased complexity that is not justified by the marginal increase – or even decrease, in the case of *magic* – in accuracy. Regarding *house* and *heloc* however, picking a value of 40 is justified, as it leads to quite substantial accuracy gains, while (almost) not impeding on the model’s complexity. Finally, for *drugs*, we settle on a value of 10, at which the accuracy is the highest, and where we strike a decent compromise between the number of rules and the average number of predicates per rule. We notice however that choosing a smaller value of n would have led to almost equivalent accuracy, making this decision less straightforward than for the other datasets. These final values are gathered in Table 5.1 for convenience.

Dataset	<i>adult</i>	<i>magic</i>	<i>house</i>	<i>heloc</i>	<i>drugs</i>
n	20	20	40	40	10

Table 5.1: Values to which the parameter n is fixed in the following experiments.

5.4.2 Models Comparison

We can now tackle the question: *how does our model compare to the other two presented in Section 5.2?* Again, we study this question based on the same three metrics, to check whether our model can compete with those, or even outperform them.

The three models perform quite similarly in terms of accuracy, as depicted on Figure 5.4. Which model performs best is quite dataset-dependent, and the differences are generally quite small. On *house* and *heloc* however, we observe that our

model performs noticeably worse than RIPPER in terms of accuracy. One of the contributing factors is the fact that our model displayed more variability across the folds during cross-validation. Precisely, for those two datasets, it performed much worse on one of the folds than on the remaining four, thus leading to a lower average accuracy. We also observe, interestingly enough, that the RIPPER model used here consistently achieves higher accuracies than those reported in [6], which is probably due to implementation differences between JRip and the version used in [6].

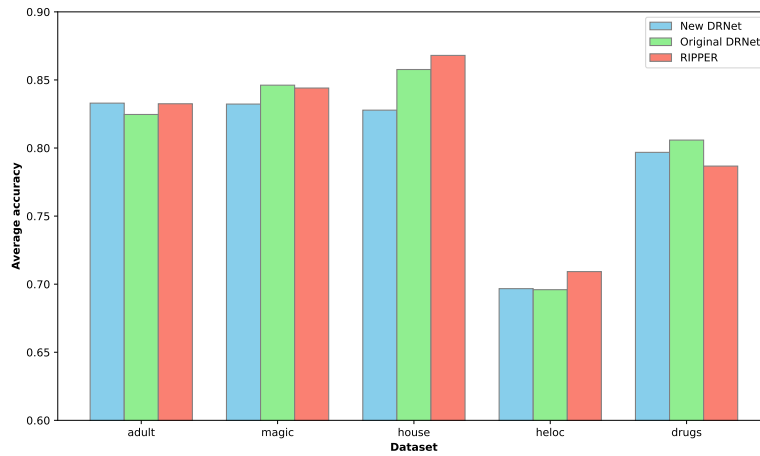
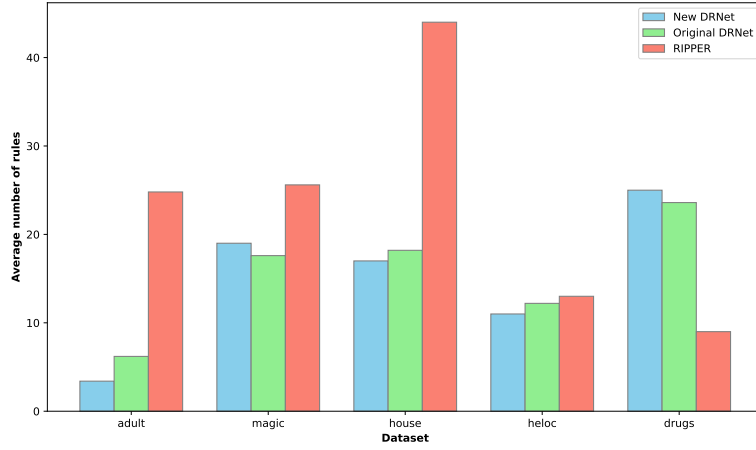


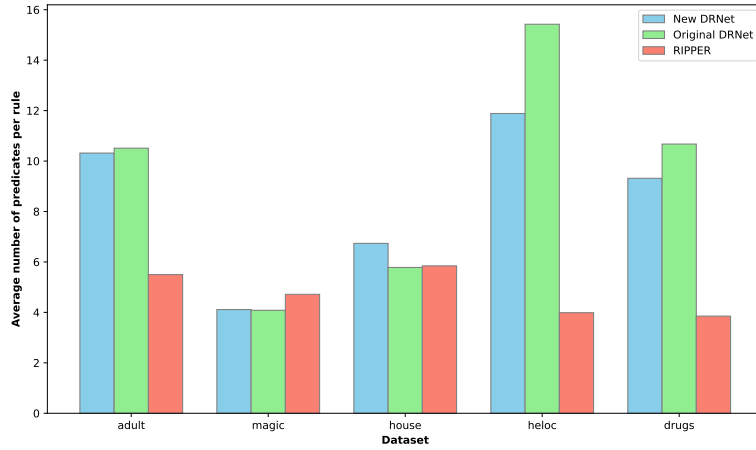
Figure 5.4: Accuracy comparison of the models.

The differences are far more pronounced when considering the rule set complexities, as seen on Figure 5.5. When comparing the rule set complexities of the new DRNet to the old one, results are quite similar when considering the contributions from both Figures 5.5a and 5.5b.

Comparing those to RIPPER, however, leads to very different observations. On the *adult* dataset, both versions of DRNet display much fewer rules, but each one of those contains more predicates on average. On *magic* and *house*, our model clearly outperforms RIPPER, by containing substantially fewer rules. On the other hand, RIPPER unequivocally outperforms our model on the last two remaining datasets: *heloc* and *drugs*. Turning back to Figure 5.2, we observe that even when considering other values of n , our model does not obtain rule sets that contain, at the same time, as few rules and as few predicates per rule as RIPPER's for the *drugs* dataset. For *heloc* however, our model could get closer to RIPPER's performances in terms of rule set sparsity by considering lower values of n , albeit at the cost of decreased accuracy as observed on Figure 5.1. Therefore, for those two datasets, our model (and DRNet in general) struggles to strike the same accuracy-simplicity trade-off as RIPPER does. As we have noted, this is



(a) Number of rules



(b) Average number of predicates per rule

Figure 5.5: Comparison of the rule set complexities for the different models.

mostly due to the sometimes large amount of predicates per rule in our model. Therefore, we might wonder about the effectiveness of our post-processing, which is what we focus on next.

5.4.3 Post-processing Effectiveness

In this part, we want to understand: *how effective are our post-processing methods in reducing the rule set's complexity?*

Observing Figure 5.6, we find the type of trend that we would expect. For all datasets, post-processing the rules allows to both decrease the number of rules, and the average number of predicates inside of them. While the former is guaranteed (no post-processing will add any rule), the latter is not. Indeed, when using other experimental parameters, we observed in practice that the average number

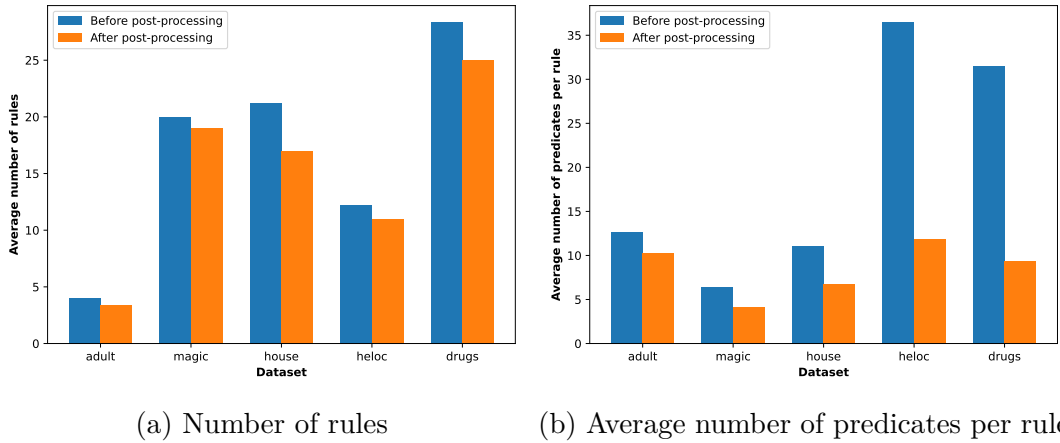


Figure 5.6: Rule set complexity before and after post-processing.

of predicates per rule could be higher after post-processing. This happens in cases where many rules are entirely removed, while few predicates inside of the remaining ones are pruned, which naturally leads to an increase in the ratio that defines the average number of predicates per rule.

The fact that the differences on Figure 5.6a are quite small indicates however that our model does not output many rules that can be discarded: those are already not chosen by the OR layer. The larger differences observed on Figure 5.6b are both expected and reassuring, as redundant predicates are pruned from their respective rules, leading to a sparser rule set.

But the most interesting observation comes from Figure 5.6b: the datasets for which the number of pruned predicates is the largest, are *heloc* and *drugs*. These are precisely the ones for which we observed in Section 5.4.2 that our model struggled to be competitive with RIPPER in terms of accuracy-simplicity trade-off. This suggests that the initial rules obtained by our model before post-processing for both of those datasets are far too complex, and that other steps could be taken to enforce finding sparser rule sets from the beginning. Some potential ideas, which we leave for future work, could involve coming up with a modified loss function that already penalises redundant predicates during the training process, as well as possibly developing more complex post-processing techniques. One example would be to verify whether, inside a given rule, some predicates are always satisfied when the other ones are, making those superfluous.

5.4.4 Temporal Data

We now switch our focus to the way our model is able to perform when presented with the *occupancy* dataset presented in Section 5.1.6. As a short reminder, we have considered creating two types of additional features to handle the temporal aspect of these data:

- a categorical representation of the day of the week at which the measurements were collected, and a numerical representation of the time;
- aggregates of the previous measurements, computed as rolling window averages.

The question we try to answer is then: *how well is our model able to handle data with a temporal aspect, when provided with different combinations of these new features?* This is thus a different type of experiment than the preceding ones: here, we vary the set of input features given to our model.

For this dataset again, we need to choose a value for the parameter n . When using large values such as 20, we observed that our model was able to reach very high levels of accuracy on this dataset, albeit with complex rules. Thus, in order to make the task more challenging, and based on what we have learned from the previous experiments, we settle for a lower value with $n = 5$. Furthermore, as mentioned in Section 5.1.6, we work with a single train-test split. But we have decided to still run the experiment five times, and present the mean results over these five different runs. The randomness is due to the random initialisation of the weights of the AND layer and the OR layer. Other parameters are unchanged from the previous experiments. Those results are gathered into Table 5.2, based on our three usual metrics of interest.

Setting	Accuracy	Nb. rules	Avg. nb. predicates per rule
No added feature	0.82	19.4	3.56
Day & time features only	0.89	21.2	6.03
Aggregate features only	0.91	21.0	5.78
Both types of features	0.96	27.8	6.48

Table 5.2: Model performance on the *occupancy* dataset for different types of input features. The results are an average over five separate runs.

Without taking the temporal aspect into account, as seen in the first row of Table 5.2, the accuracy is quite low, especially when considering the fact that almost 80% of instances in the test set belong to the same class. As we observe in the next two rows however, either taking the day and time into account, or tracking aggregated versions of the other features, are each able to individually increase the accuracy. In the former case, as expected, we often observe predicates such

as “NOT Saturday” or “NOT Sunday” in the rule set. Thursday also appeared in several rules, potentially meaning that the habits of the people in this office building differ on that day as well. As for the aggregate features, they are often found together with their non-aggregate counterparts. As an example, rules that incorporate “Humidity_avg_10” usually also incorporate “Humidity”.

We observe that the separate effects of using the day and time features or the aggregate features are quite similar, both increasing the accuracy and the complexity of the rule set, with the second approach having a slight advantage. The last row of Table 5.2 shows that combining them allows to effectively use both of those approaches’ predictive power, as we observe the highest accuracy level by far. However, as often, this is at the cost of a more complex rule set.

5.4.5 Imbalanced Data

In this last experiment, we are interested in our model’s behaviour when presented with highly imbalanced data, as is the case in the *rockburst* dataset. Indeed, out of its 2584 entries, only 170 are part of the positive class, which represents 6.58% of the dataset. Our question is thus: *how should our model and evaluation metrics be adapted when considering such imbalanced data?*

First off, we observe that when left unchanged, our model essentially never predicts the positive class. In some cases, the rule set is even *empty*. Indeed, through this approach, it is able to correctly classify a vast majority of the examples (all the negative ones), while being maximally sparse. However, this is pretty much the opposite of what we are aiming for. Indeed, as explained in Section 5.1.7, the critical aspect is to correctly classify positive examples. In particular, we want to correctly classify as many positive examples as possible. Indeed, in the context of the *rockburst* dataset, a false negative would translate into a serious safety hazard in real life.

Therefore, we need to make our model “understand” the crucial aspect of correctly predicting the positive class. As we know, the loss function that it attempts to minimise during training comprises a binary cross-entropy (BCE) loss term, as well as extra regularisation terms to promote sparsity. But the BCE is symmetric with respect to both classes.

The solution is thus to multiply the contributions of the positive class to the loss by a *weight* $w > 1$. This way, each positive example will have as much influence as w negative examples on the loss. Furthermore, it is evident that in this context, our model’s performance should not be evaluated by computing its accuracy, but

rather by considering the *recall*. Other metrics such as *precision* could turn out to be more relevant in other contexts. As explained in the documentation of the `torch` Python module that has been used to implement our model, picking a value of w larger than 1 tends to increase the recall, while values smaller than 1 favour precision. Therefore, the solution we have opted for makes sense with regard to our new metric of interest.

Setting the parameter n to the value of 20 for flexibility and the weight w to 15, we find ourselves in a setting where the sets of positive and negative examples have a similar impact on the objective. Indeed, there are roughly 15 times more negative examples than positive ones in *rockburst*. In this context, through 5-fold cross-validation, we observe recall values of around 50%, although with a large variance across folds, and decreased accuracy of around 75%. When using much larger weights such as $w = 50$ to place even more emphasis on the correct classification of the positive class, we consistently obtain recall values above 90%, while the accuracy is further decreased to the low value of 40%. Again, finding a fitting balance between the correct classification of positive examples and that of negative examples is application-dependent.

5.5 Conclusion

Through a careful analysis of our model’s behaviour, we have been able to examine its sensitivity with respect to the maximum number of thresholds. In particular, while the model’s accuracy generally tends to increase with this parameter, so does the complexity of the resulting rule set. Comparisons with the original DR-Net model and RIPPER have allowed to further highlight this phenomenon, as for similar accuracy, the complexity of our rule set ranges from very satisfying on some datasets to quite disappointing on others. Turning to the efficiency of our post-processing techniques has revealed that on those problematic datasets, the complexity is already significantly higher before post-processing than on the other ones. This suggests that modifying the way in which complexity is fundamentally handled in DRNet could help alleviate this issue.

Finally, in our search for versatility, we have studied the way in which our model is able to react to datasets with different properties. When presented with the right features in the context of temporal data, it is able to achieve significantly higher performance than by ignoring this temporal aspect, although at the cost of increased complexity. Further, in contexts where the prediction of one class is more challenging and/or important than that of the other, we have shown the necessity of modifying the loss function used during the model’s training process.

Chapter 6

Conclusion

The main aim of this thesis was to develop an interpretable and performant binary classification model. The need for interpretability in machine learning models being justified by its use in critical contexts, as well as higher user adoption. Exploring the existing approaches, we have found a separation between two types of classification models. On one hand, rule learning models are highly interpretable. On the other, neural networks reach high levels of accuracy and support efficient learning techniques. Recently, those two approaches have been united into a new paradigm, where decision rule sets are learned from neural networks.

We have thus started our journey by reviewing concepts related to the worlds of deep learning and rule learning. Analysing the complex architecture and training process of neural networks has made us understand why those are considered to be black-box models. This was followed by an overview of the rich landscape of rule learning techniques, in which we identified recurring steps involved in the process of building a set of rules for classification.

It is then through the scope of the DRNet model that we have ventured into the world of interpretable neural networks. Whereas the weights that constitute a neural network are generally considered a means to an end, DRNet uses them directly to form its classifier. To achieve this, the model is split into two parts, respectively responsible for creating rules, and for gathering some of them into a rule set. Through this approach, DRNet is able to be highly accurate in classification tasks, while also being interpretable as a rule set.

Despite these positive results, DRNet is not a perfect model. Firstly, we have considered that the way in which data are injected into the model presents room for improvement. Adding a thresholds layer has allowed for the binarisation process

to not be fixed, but rather to evolve in agreement with the other parts of the network. Through this, we also handle numerical and categorical data in analogous ways, allowing for a unified vision of the binarisation process.

The second main improvement opportunity has risen from the underlying implementation of our model, and the impact it may have on the complexity of the rule set that constitutes our classifier. To address this issue, we have developed post-processing techniques which simplify the set of rules while keeping it equivalent to the one obtained by the network. Our thresholds layer has turned out useful in this step as well, allowing for our post-processing to behave equivalently, regardless of the underlying data types.

Experimenting with our model has further highlighted its versatility, as it was able to effectively use new features to increase its accuracy, and was also easily adaptable to settings involving imbalanced data. Furthermore, when compared to other classification models in different experiments, ours remained competitive in terms of accuracy, but the complexity of the rule sets that it created was not always satisfactory. Further exploration of the reasons for this phenomenon has revealed that our network was sometimes producing rules that were needlessly long.

Penalising the redundancy of the rule set within the network could then be a potential direction for future research. In parallel, more complex post-processing solutions could be developed, in order to produce rule sets that are maximally sparse. For this, itemset mining principles could be used on the set of rules output by the network. If necessary, one may even prune rules to the point where the model's accuracy is decreased, if this allows for the rule set's complexity to sufficiently decrease in return. Finally, in our unrelenting quest for versatility, we may be interested in finding ways to adapt our model to more complex data types, such as sequential, spatio-temporal, or even image-based data.

Bibliography

- [1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell et al. *Language Models Are Few-Shot Learners*. 2020. (URL)
- [2] Shahab S Band, Atefeh Yarahmadi, Chung-Chian Hsu, Meghdad Biyari, Mehdi Sookhak, Rasoul Ameri, Iman Dehzangi, Anthony Theodore Chronopoulos, and Huey-Wen Liang. *Application of Explainable Artificial Intelligence in Medical Health: A Systematic Review of Interpretability Methods*. In: Informatics in Medicine Unlocked, Volume 40. 2023. (URL)
- [3] Mrinal R. Bachute, Javed M. Subhedar. *Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms*. Machine Learning with Applications. Volume 6. 2021. (URL)
- [4] E. Yurtsever, J. Lambert, A. Carballo and K. Takeda. *A Survey of Autonomous Driving: Common Practices and Emerging Technologies*. In: IEEE Access, vol. 8, pp. 58443-58469. 2020. (URL)
- [5] C3.ai - *Model Interpretability Is Critical to Driving Adoption*. (URL)
- [6] Litao Qiao, Weijia Wang and Bill Lin. *Learning Accurate and Interpretable Decision Rule Sets from Neural Networks*. 2021. (URL)
- [7] Alexander Jung. *Machine Learning: The Basics*. 2022. (URL)
- [8] Warren S. McCulloch and Walter Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. 1943. In: Bulletin of Mathematical Biophysics, Volume 5, pp. 115-133. (URL)
- [9] Cornell University course blog for INFO 2040/CS 2850/Econ 2040/SOC 2090. *Neural Networks and Machine Learning*. 2015. (URL)
- [10] Kunihiko Fukushima. *Cognitron: A Self-Organizing Multilayered Neural Network*. 1975. In: Biological Cybernetics, Volume 20, pp. 121–136. (URL)

- [11] John S. Bridle. *Training Stochastic Model Recognition Algorithms as Networks Can Lead to Maximum Mutual Information Estimation of Parameters*. 1989. In: NIPS'89: Proceedings of the 2nd International Conference on Neural Information Processing Systems, pp. 211-217. (URL)
- [12] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. 2017. (URL)
- [13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Internal Representations by Error Propagation*. 1986. In: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations, pp. 318-362. (URL)
- [14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. (URL)
- [15] Richard E. Bellman. *Adaptive Control Processes: A Guided Tour*. 1961. Princeton University Press. (URL)
- [16] Lei Chen. *Curse of Dimensionality*. 2009. In: Encyclopedia of Database Systems, pp. 545-546. (URL)
- [17] Xia Hu, Lingyang Chu, Jian Pei, Weiqing Liu and Jiang Bian. *Model Complexity of Deep Learning: A Survey*. 2021. (URL)
- [18] Sebastian Thrun and Lorien Pratt. *Learning to Learn*. 1998. (URL)
- [19] Kyoung-Su Oh and Keechul Jung. *GPU Implementation of Neural Networks*. 2004. In: Pattern Recognition, Volume 37, Issue 6, pp. 1311-1314. (URL)
- [20] Razvan Pascanu, Tomas Mikolov and Yoshua Bengio. *On the Difficulty of Training Recurrent Neural Networks*. 2013. (URL)
- [21] Sepp Hochreiter and Jürgen Schmidhuber. *Long Short-Term Memory*. 1997. In: Neural Computation, Volume 9, Issue 8, pp. 1735–1780. (URL)
- [22] Johannes Fürnkranz and Tomáš Kliegr. *A Brief Overview of Rule Learning*. 2015. In: Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke and Dumitru Roman (Eds.). Rule Technologies: Foundations, Tools, and Applications (RuleML 2015), pp. 54-69. (URL)
- [23] Rakesh Agrawal, Tomasz Imieliński and Arun Swami. *Mining Association Rules between Sets of Items in Large Databases*. 1993. (URL)
- [24] Bing Liu, Wynne Hsu and Yiming Ma. *Integrating Classification and Association Rule Mining*. 1998. (URL)
- [25] Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules*. 1994. (URL)

- [26] Johannes Fürnkranz. *Pruning Algorithms for Rule Learning*. 1997. In: Machine Learning, Volume 27, pp. 139-172. (URL)
- [27] William W. Cohen. *Fast Effective Rule Induction*. 1995. (URL)
- [28] Yoshua Bengio, Nicholas Léonard and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. 2013. (URL)
- [29] S. Gopal Krishna Patro and Kishore Kumar Sahu. *Normalization: A Preprocessing Stage*. 2015. (URL)
- [30] Markelle Kelly, Rachel Longjohn and Kolby Nottingham. *The UCI Machine Learning Repository*. (URL)
- [31] Eibe Frank, Mark A. Hall and Ian H. Witten. *The WEKA Workbench*. 2016. Online Appendix for “Data Mining: Practical Machine Learning Tools and Techniques”, Morgan Kaufmann, Fourth Edition, 2016. (URL)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl