Universiteit Leiden

Master Computer Science

A multiple testing correction based approach for encoding human generated features in MDL-based rule list learning

Name:	Tim Opdam
Student ID:	s2332515
Date:	01/06/2025
Specialisation:	Data Science
1st supervisor:	Lincen Yang
2nd supervisor:	Matthijs van Leeuwen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Rule learning models are widely used interpretable models that, using ordered sets of rules (boolean expressions), describe subsets of the data. Among the state-of-the-art rule list models are models — such as the SSD++ algorithm — that make use of the minimum description length (MDL) principle, a framework that selects the model for which the code length of the model and the data combined is the smallest. Rules learned as part of a rule list can be used to create new binary features that describe the same data. Domain experts may select good rules from previously learned rule lists to add to the dataset as features, however, this human-in-the-loop approach is underexplored for models using the MDL principle. MDL-based model selection can be seen as a form of hypothesis testing in which the code length of the model functions as the multiple testing correction term. When new features based on rules are added, this correction term no longer holds, and the type-I error will thus not be properly controlled in these hypothesis tests, which may lead to learning overly complex models. With the connection between MDL-based rule list learning, hypothesis testing, and multiple testing correction we propose a new correction method that restores the connection between multiple testing correction and model code length when the new human-selected features are added to the dataset. To achieve this correction method we also introduce a new model encoding scheme for the SSD++ algorithm based on the principles of multiple testing correction, on which we base our correction. We empirically explore the effects of added features on MDL-based rule list learning with three different encoding schemes. 1) the SSD++ encoding scheme, 2) our new encoding scheme without correction, and 3) our new encoding scheme with correction applied to the new features. We have tested encoding schemes on synthetic and several real-world datasets, and from these results we have found that treating the added features as normal features increases the risk of learning overly complex models for all tested encoding schemes. The models trained with our correction suffer from the same problems, and leads to very similar results as the uncorrected model. This may suggest that future work should look outside of a purely MDL context to resolve this issue.

Contents

1 Introduction

Machine learning is a process in which an algorithm learns from data for various purposes, such as prediction tasks or data exploration. It is extensively used in many tasks, such as optical character recognition when scanning written texts [20], fraud detection by banks [2], recommending systems in shops [31], and many more.

In machine learning tasks, domain experts can play multiple roles because we cannot rely on the data alone to create good models. These roles include feature selection, where the expert selects the most informative features of the dataset, and feature engineering, where the expert creates new informative features to add to the dataset. In feature selection, the expert selects the relevant features to be used by the machine learning model, simplifying the dataset for the machine learning task. Domain experts can also utilise their prior knowledge of the domain to introduce additional information that is not present in the dataset to allow the machine learning models to better do their task. Feature selection is done for several reasons, such as removing known irrelevant features, simply reducing the dimensionality of the dataset for faster model training, or reducing chances of overfitting occurring [16].

In feature engineering, the engineered features are most often calculated from the other pre-existing features, with the intent to improve the performance of the models trained on this dataset [15]. Several methods are used for feature engineering; one such method is manual work by a human domain expert, another method is an automated process. It can be beneficial to combine human knowledge and intuition with machine learning to achieve models with a higher predictive accuracy. Thus, by having a domain expert in the loop that selects which learned features to add to the dataset to train future models, it is possible to train better models by making use of human intuitions.

One type of machine learning model is a rule list model, such as the SSD++ model [26]. Rule lists are a type of interpretable machine learning model that, through an ordered set of Boolean expressions, describes different sections of the data in a human-understandable way. A simple example of a rule list can be seen in Figure 1.

IF	has_spikes = false		THEN	$P(\text{Healthy}) = 0.05$	usage = 100
ELIF	weight < 600 AND	age > 1	THEN	$P(\text{Healthy}) = 0.3$	usage = 100
ELSE				$P(\text{Healthy}) = 0.8$	usage = 100

Fig. 1: An example of a simple rule list for a mock dataset of hedgehog health, where the target variable has two values: *Healthy*, denoting if the hedgehog is healthy, and *Unhealthy*, if it is not. Note that the probability that the animal is unhealthy is not shown in this rule list as this probability is simply $1 - P(\text{Healthy})$. Usage refers to the size of the subgroup (i.e, the number of samples in this subgroup). In this mock dataset, *weight* is in grams and *age* is in years.

In this example, describes a different subset of the data in a humanly interpretable way. Each rule in this example describes the probability that the hedgehog is healthy based on some Boolean expression describing characteristics of the animal. These expressions consist of conditions such as 'if the animal weighs less than 600 grams',

describing the status of the hedgehog. The default rule (the last rule, denoted by the ELSE statement) describes the remaining samples that were not covered by the previous rules.

An earlier mentioned example of a rule list learning model is the SSD++ model. The SSD++ model is a heuristic model based on the minimal description length principle [14] (MDL principle) that learns new rules using greedy beam searches.

The MDL principle is a model selection principle that states that the model that allows you to describe the data and the model with the shortest code length (i.e., using the fewest bits) is the best model. The addition of the code length of the model factors in the model selection is to reduce overfitting [14]. The description length, i.e., code length, of the model is calculated using a model encoding scheme, and the data encoding scheme is used to calculate the code length of the data using this model [14]. The MDL principle has been successfully applied to many machine learning tasks [11].

When a rule list is learned, a domain expert may like a rule and decide to add this rule as a feature to the dataset. A rule list model trained on this dataset is now able to describe the same subset of the data using a rule consisting only of the newly created feature, instead of the longer rule that this feature was based on. For MDL-based rule lists, i.e., methods that make use of the MDL principle to select what rule to add to the list, this means that the new model is simpler, as only a single feature is now used to describe the same data. This means a lower code length is required to encode the model, as a shorter rule requires a shorter code length to encode compared to longer rules. Because describing the same subset of the data is cheaper (requiring fewer bits), the model is more likely to describe subsets of this subset, leading to more, more complex rules. In addition to the increase in dimensionality of the dataset, this could lead to some potential problems such as the algorithm learning overly complex models.

In this thesis, we investigate the impact of adding features chosen by humans in MDL-based models, as this interaction is underexplored for MDL-based models. We want to deal with this potential problem when human domain experts introduce new features to the dataset for training MDL-based models, whilst still preserving the potential benefits of introducing these newly engineered features. We intend to explore this problem by connecting multiple testing correction, more specifically, Bonferroni correction [4], a method used to control the false positive rate when doing a series of statistical test, and MDL-based rule lists, to motivate why adding features based on rules can be a problem for learning a rule list. This connection, as we will show in Chapter 4, is that the model code functions as a multiple testing correction term when we transform the code length comparison for learning new rules into a statistical test. This code length comparison is to decide, according to the MDL principle, which rule is the best to add to the rule list, where the shortest code length is the best.

When features that are based on rules are added to a dataset we see that this connection breaks. This is because the model code length no longer properly functions as the correction term when the newly introduced features are used (we will handle this in more detail in Section 4.1.2). Since the multiple testing correction term no longer functions as it should, this may result in errors during rule learning.

In this thesis we will thus explore how we can encode the added features for an MDL-based rule list model, such that the connection between MDL-based rule learning and multiple testing remains intact. Our proposed model encoding scheme is a modification of the encoding scheme of the MDL-based SSD++ rule list model. With our proposed model encoding scheme we intend to correct the code length required for encoding the features introduced by the feature engineering using the principles behind multiple testing correction. Thus, by correcting the code length of the rules using the newly added features, we preserve the connection between MDL-based rule learning and multiple testing correction.

Our contributions are: connecting multiple testing correction to MDL-based rule list learning, as well as introducing a new model encoding scheme for rule list learning based on this connection. We also introduce a method that uses multiple testing correction to correct the code length required for encoding the newly created features to preserve this connection.

The process of adding features we are specifically looking at is the process in which first a rule list is learned on the data, after which a human domain expert uses some of the learned rules as new features. For example, in the rule list from Figure 1, a human could transform the second rule into a new feature called *underweight* to more easily describe that section of the data. We will refer to such features as constructed features.

To test the effectiveness of our proposed encoding scheme and correction method, we conduct a series of experiments on various datasets where we compare models trained with our proposed correction to models without applied correction. These experiments show that our correction does not show major differences in model performance compared to an uncorrected model, showing near identical ROC-AUC scores and only minor differences in total code length.

This may suggest that for MDL-based models, an MDL approach based on multiple testing correction may not resolve the potential issue of learning overly complex models that could arise from introducing new engineered features to a dataset. This may be because the difference in code length of the corrected and uncorrected models is relatively small compared to the code length of the data, lowering the potential impact that modifying the model code length may have.

First, we will look at related work in Chapter 2, after which we will go into more detail into how the SSD++ algorithm that our encoding scheme is based on functions in Chapter 3, with the main focus on the encoding scheme the algorithm uses. In this section we will also review the basics of likelihood ratio tests and multiple testing correction. After that, in Chapter 4, we will first go over the connections between MDL and multiple testing correction, after which we will explain our proposed model encoding scheme and how our correction method works. Next, in Chapter 5, we go over the experiments to empirically demonstrate the effects of our method and the results of those experiments. And finally, in Chapter 6 we go over our conclusions.

2 Related Work

In this section we start by looking at MDL-based models in machine learning and their wide range of applicability. We next look at previous work in hypothesis testing for rule learning and pattern mining, as well as using MDL for hypothesis testing. Next we look at previous work in rule learning algorithms that make use of previously learned rules, and finally, previous work in human-in-the-loop feature engineering approaches.

In this thesis we are studying MDL-based methods. The MDL principle has been widely used in many machine learning contexts such as pattern mining [32], association rule learning [9], and discretisation [35]. The MDL principle has been successfully utilised in many machine learning tasks, in a wide variety of contexts. However, the encoding schemes used in MDL models have not yet been studied in a human-in-the-loop context. We will be exploring this by specifically looking at an MDL-based rule list model. We will connect this model to hypothesis testing to support why this may result in problems and how we might solve such a problem.

Previous works have already made use of hypothesis testing to learn interesting subsets in data [18] [17]. In these works, statistical tests are used to evaluate the candidate subsets to add to the model. We will also use hypothesis testing as the method to learn subsets of the data using a rule list model. However, for this we will be focusing on an MDL-based model where we propose a new model encoding scheme, making use of the connection between MDL and hypothesis testing.

Previous work has also discussed MDL as a hypothesis testing method [13], showing that the comparison of code lengths of two MDL models can be seen as a hypothesis test. We expand on this by also including the multiple testing correction term in the context of rule learning.

In this thesis we will be making use of previously learned rules in order to help in learning a rule list model. Using such previously learned rules in order to construct new rule lists is a common practice [33] [34] [6]. By only considering previously learned rules as candidates to add to the model the search space is much more limited, allowing for a less computationally demanding, and thus faster, algorithm. However, previous works do not consider MDL-based models for constructing rule lists using previously learned rules. In our work, we are investigating how to use previously learned rules in an MDL-based rule list model. We will be considering these previous rules that were picked by experts to be added to the training dataset as features, more like a feature engineering context.

There are already many feature engineering methods, both automatic and interactive. The interactive methods allow for a human domain expert to interactively apply their domain knowledge to create new features. These feature engineering methods often work for any model as they only modify the data. One example of an interactive feature engineering method is the dataset evolver[22]. Dataset evolver is a tool for classification tasks that supports the user in feature engineering tasks. This tool gives the user suggestions for new constructed features that they may accept and add to the dataset, or reject. This allows for human intuition to interactively play a role in machine learning tasks. We too will consider features added by experts chosen from a suggested pool (a previous rule list), however, this work does not take into account the potential problems that may arise when more features are added, focusing only

on creating new features. An example of such a problem that may arise is the risk of learning overly complex models when features are added to a dataset.

We will be looking at an interactive feature engineering method that does take such potential problems into account, specifically for MDL-based models. For this, we will be creating a model encoding scheme for a rule list algorithm that allows for a correction method that aims to increase the code length of the features, while still trying to preserve their positive effects.

3 Preliminaries

In this section, we will first go over the SSD++ algorithm [26] that our work is based on, starting with the encoding schemes used, followed by the process of rule learning. After that, we review the basics of hypothesis testing, with the main focus on likelihood ratio tests and multiple testing correction.

3.1 SSD++

The SSD++ algorithm is a greedy heuristic rule list learning algorithm based on the MDL principle that iteratively adds one rule at a time until no better rule is found using a beam search. A rule is a combination of Boolean expressions that describe a subsection of the data, and a rule list is a series of rules describing the entire dataset.

In this section, we will first elaborate on how the code length is calculated for the data encoding and the model encoding, which, when added together, form the total code length used to evaluate the rules. Then, the search process for discovering new rules will be covered.

3.1.1 Data Encoding

We will first go over how the code length for the data, i.e., how many bits are required to describe the data, D, is calculated.

The total code length needed to encode all target values that take a single class value is the number of occurrences of the class multiplied by the negative log probability of that class. This is the code length needed to encode that particular class when using a Shannon-Fano encoding [29]. This is repeated for every class c in the set of unique classes, $\mathcal{Y}$, after which the parametric complexity [14] is added to obtain the total code length of the data.

The code length calculation for the data encoding of a single subgroup a_i is given by Equation 1 [26], which uses a Normalized Maximum Likelihood (NML) code for its calculation.

$$L_{NML}(D|a_i) = -\sum_{c \in \mathcal{Y}} \left[n_c \log(\frac{n_c}{n}) \right] + \mathcal{C}(k, n) \tag{1}$$

For predictive rule lists, the data of all subgroups, as well as the data of the default rule and the code length required to encode them, are calculated using this method.

In this equation $\mathcal{Y}$ represents the set of unique values (classes) that appear in the target variable y (such as *Healthy* and *Unhealthy* from the earlier example in Figure 1), and c is an individual class of $\mathcal{Y}$. n_c represents the number of occurrences of class

c in the subgroup, and n is the total number of samples in the subgroup. $\frac{n_c}{n}$ thus represents the empricial probability of class c in this subgroup. Finally, $\mathcal{C}(k, n)$ is the parametric complexity, which serves as a penalty term to normalise the maximum likelihood estimate to reduce overfitting. This complexity can be calculated with only the cardinality k of the target variable, i.e., the number of unique values in $\mathcal{Y}$, and the number of samples n in the subgroup [21].

Thus, to calculate the NML code length of the data given a subgroup, first a Shannon-Fano encoding is used, as seen in the first term. In this encoding, the code length required to describe the data is equal to the negative log-probability of the data. This probability is derived empirically from the data in the subgroup.

For example, if we were to encode the data of the first rule of the example in Figure 1, we would first encode the first class of the target variable: *Healthy*. This class has a probability of 0.05 in this subgroup and would thus have $0.05 * 100 = 5$ samples with this value. This class would then need a code length of $5 * -\log(0.05))$ to describe all occurrences in this rule. We then repeat this for the *Unhealthy* class, to obtain a code length of $95 * -\log(0.95)$, and calculate the parametric complexity $\mathcal{C}(2, 100)$, which, when everything is added together is the code length for the data of this first subgroup.

To calculate the code length of the data for the entire rule list, for every rule in the list, as well as the default rule, we calculate the data code length of that subgroup with Equation 1, with the statistics n, n_c of that subgroup. When the code lengths of all these subgroups are added together, the code length of the data is obtained.

3.1.2 Model Encoding

Next, we will be covering the method to calculate the code length required to describe the model.

To calculate the code length required to encode the rule list, i.e., the model, first, the number of rules $|S|$ is encoded using the universal code for integers [27] $L_{\mathbb{N}}$. After that, the code length of each rule is added.

The code length of a single rule is calculated by first encoding the length of the rule using the universal code for integers, then the features used are encoded. This is done by encoding the combinations of features of this length that are calculated using a combinatorial. Finally, the value of each variable, v, used in the rule is encoded. If v is categorical, the code length can be calculated by taking the log of the cardinality. For numeric variables, first, the number of operators (either 1 or 2) is encoded using the universal code of integers that is restricted to values of only 1 and 2 ($L_{\mathbb{N}|2}$). Next, given the number of operators and cut points (how many sections the numeric features should be split with equal frequency binning), the total number of all possible value combinations can be calculated. The log of this total number of combinations is taken to then obtain the code length of the numeric variable.

The encoding scheme for encoding the model is seen in Equation 2.

$$L(M) = L_{\mathbb{N}}(|S|) + \sum_{a_i \in S} \left[L_{\mathbb{N}}(|a_i|) + \log \binom{f}{|a_i|} + \sum_{v \in a_i} L(v) \right] \tag{2}$$

In this equation, S is the set of subgroups (i.e., rules) in the rule list, and thus $|S|$ is the number of rules. a_i refers to the i-th subgroup of S and $|a_i|$ the length of rule i. f is the total number of features in the dataset, and v is a variable that is used in the current rule a_i.

For calculating the code length of the variables $L(v)$ (i.e., the features of the dataset present in this rule), Equation 3 is used. In the equation are separate cases for categorical variables and numeric variables.

$$L(v) = \begin{cases} \log(|\mathcal{X}_v|), & \text{if } v \text{ is categorical} \\ L_{\mathbb{N}|2}(n_{op}) + \log(N(n_{op}, n_{cut})), & \text{if } v \text{ is numeric} \end{cases} \tag{3}$$

In Equation 3, $|\mathcal{X}_v|$ is the cardinality of variable v, n_{op} is the number of operands the condition uses, which can be either 1 (e.g $age \geq 1$ where age is one of the features of the dataset) or 2 (e.g., $1 < age \leq 4$). n_{cut} is the number of cut points used to split the numeric features so there are not infinite possible options for using numeric features. n_{cut} is a parameter of the algorithm that can be set beforehand. $N(n_{op}, n_{cut})$ calculates the number of possible combinations of conditions for numerical variables based on the number of operands and cut points. To discretise the numeric features, equal frequency binning is used.

For example, if we were to again encode the first rule of the example in Figure 1, we first encode the length of the rule using the universal code for integers. Since the rule only has a single condition ($has\_spikes = false$), the length is 1, and we get a code length of $L_{\mathbb{N}}(1)$. Next, assuming the data has a total of 3 features, we encode the combination of features of length 1 used in the rule. There are $\binom{3}{1}$ combinations of length 1, so we need a code length of $\log \binom{3}{1}$ to encode this feature. Finally, we encode the value of the used feature *has_ spikes*. *has_ spikes* is a binary categorical feature and following Equation 3 thus needs $\log(2)$ bits to encode the value. When we add this together, we obtain the code length required to encode this first rule.

3.1.3 Learning rules

Next, we will be covering the process of how the SSD++ algorithm learns new rules. First, we will cover the process of searching for new rules using a beam search, after which we will cover how the candidate rules (found rules that are not yet added to the rule list) are evaluated. The optimisation target for this search is the two-part code length, i.e., the code length of the model encoding plus the code length of the data encoding.

3.1.3.1 Beam search

In this algorithm, the rules are searched using a beam search. A beam search is a greedy algorithm that is used to limit search spaces by only expanding the w best candidates every iteration, where w is the beam width.

For rule learning, first, all candidate rules of length 1 are generated and evaluated. The metric used to evaluate the candidate rules is the normalised compression gain [26] (See Section 3.1.3.2). The overall best, w candidates, i.e., the candidates with the highest normalised gain, are then stored for the next iteration. In the next iteration of the beam search, new candidate rules are generated by expanding the w best rules to rules of length 2. Then, the best w candidates are stored again to be expanded in the next step. The search stops when no more improvements are made, and the w best candidates remain the same. The candidate rule that was evaluated the best is then chosen to be added to the rule list.

Rules are added one by one this way until the best rule no longer improves on the total code length of the rule list.

3.1.3.2 Normalised gain

The normalised compression gain used to evaluate candidate rules is calculated by normalising the difference in code lengths between the rule list with and without the candidate rule.

The absolute gain, i.e the total difference in code length between the rule lists with and without the candidate rule, is normalized by dividing it by $|n_{a_i}|^\beta$, where $|n_{a_i}|$ the usage of group a_i (the number of samples in the group), and β, which has a value in $[0, 1]$, is the normalisation level, a parameter of the SSD++ algorithm [26]. We thus get

$$\Delta_\beta L(D, M \oplus a_i) = \frac{L(D, M) - L(D, M \oplus a_i)}{|n_{a_i}|^\beta}, \tag{4}$$

where $\Delta_\beta L(D, M \oplus a_i)$ is the normalised compresion gain for adding the new rule a_i, $L(D, M)$ is the total code length of the data and rule list without the new rule, and $L(D, M \oplus a_i)$ is the total code length of the data and rule list with the rule a_i added.

3.2 Hypothesis testing

We will now briefly review hypothesis testing. In a hypothesis test, usually two hypotheses are compared, the null hypothesis and the alternative hypothesis, in order to either accept or reject the null hypothesis. A type-I error, also referred to as a false positive, is rejecting the null hypothesis when it is actually true. The null hypothesis is rejected if the probability that the observed data originates from the probability distribution of the null hypothesis is below some predetermined probability α. In practice, this α, i.e., the significance level, is often set to 0.05. The threshold value to beat (i.e., the minimum value required to achieve the required probability to reject the null hypothesis) in order to achieve this significance level is often referred to as the critical value.

3.2.1 Likelihood ratio tests

A type of statistical test comparing two models is the likelihood ratio test (LRT), which compares the maximum likelihood of the data D for the null hypothesis M_0 to the maximum likelihood of the same data for the alternative hypothesis M_1. A requirement for an LRT is that the models M_0 and M_0 must be nested, that is, the

more complex model M_1 must be able to be transformed into the simpler M_0 by imposing constraints to M_1. The ratio r can then be defined as

$$r = \log \frac{\max_{\theta_1} P(D|M_1, \theta_1)}{\max_{\theta_0} P(D|M_0, \theta_0)}, \tag{5}$$

where θ_0 and θ_1 are the parameters for the models M_0 and M_1.

If the ratio r is above a predetermined threshold t, then the alternative model M_1 is picked and the null hypothesis is rejected. This threshold value is picked to control the type-I error rate in order to achieve the predetermined significance level.

3.3 Multiple testing correction

The family-wise error rate is the probability that at least one of the tests in a series of tests results in a type-I error. This is formally defined by

$$FWER = 1 - (1 - \alpha)^m, \tag{6}$$

where m is the total number of performed tests.

When doing a single test, the $FWER$ is equal to the significance level α. However, when more than one tests are performed, the $FWER$ will always be above α. This means that when multiple tests are done in succession, it is more likely to receive a false positive result in at least one of those tests.

To reduce the chances of false positives when doing multiple tests, a multiple testing correction is needed. One method of doing this is by using the Bonferroni correction [4]. The correction by a Bonferroni correction is simply to compensate for the total number of tests m by reducing the significance level α for each test to $\frac{\alpha}{m}$. Leading to a corrected $FWER'$ of

$$FWER' = 1 - (1 - \frac{\alpha}{m})^m. \tag{7}$$

This corrected $FWER'$ approximates, but never exceeds, α when a series of m tests are performed [12]. Thus, with a Bonferroni corrected significance level, the family-wise error rate can be controlled to retain the predetermined significance level over all tests.

For a Bonferroni correction, it is also possible to have tests with different significance levels as long as the combined total of these test specific significance levels adds up to α.

4 Method

In this section, to motivate our proposed encoding scheme and correction method, we will first connect MDL, more specifically the SSD++ algorithm, to likelihood ratio tests, as well as multiple testing correction. Then, we will go over our own encoding scheme and how the proposed correction would function.

4.1 Code length and multiple testing correction

To explain our method of correction, we must first show how the model encoding and multiple testing correction are related by transforming the code length comparison of candidate rules into an LRT. In this LRT, the model encodings function as a multiple testing correction term, as we will show in this section.

In the case of rule learning, where we want to find the best rule to add to the rule list, we first compare an empty rule M_0 (i.e., no rule) with a rule with a single condition M_1. For simplicity, we will be ignoring the difference in the encoding of the length of the rule as this is, in practice, only a small part of the code length. This also holds for a non-empty rule M_0 and an M_1 with one extra condition, as the only difference between these rules is the extra condition. The rule M_1 will be chosen if the combined code length of M_1 is smaller than the combined code length M_0, so if

$$L(D|M_1) + L(M_1) < L(D|M_0) + L(M_0), \tag{8}$$

M_1 is a better rule compared to M_0.

We will first show how this code length comparison of two candidate rules also functions as a statistical test. To show that this indeed functions as a hypothesis test, we must first rewrite the code length of the data encoding from Equation 1 as

$$L(D|M) = -\log(P(D|M)) + \mathcal{C}.$$

Here, we interpret the data code length as the negative log likelihood of the data calculated using the probability estimates of the compared rules plus the NML-complexity term.

If we insert this back into Equation 8 we obtain,

$$-\log(P(D|M_1)) + \mathcal{C}_{M_1} + L(M_1) < -\log(P(D|M_0)) + \mathcal{C}_{M_0} + L(M_0), \tag{9}$$

which can then be rewritten as

$$\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M0} + L(M_1) - L(M_0), \tag{10}$$

where $\mathcal{C}_{M_0}$ is the NML complexity term of the null hypothesis, and $\mathcal{C}_{M_1}$ is the NML complexity of the candidate rule.

This has turned the code length comparison of Equation 9 into a likelihood ratio test. In this LRT, the difference in the NML-complexity terms, $\mathcal{C}_{M0} - \mathcal{C}_{M1}$, as we will show next, functions as the critical value, and the difference in model code length as the multiple testing correction term.

Next, we show how the difference in model code length functions as a multiple testing correction term and how the NML-complexity terms function as the critical value. We will do this by first looking at the LRT and significance level without the model code lengths, and the effect introducing this term has on the significance level of the test.

If we were to ignore the model code length, and thus the correction term, the LRT would look like

$$\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0}, \tag{11}$$

where the critical value is $\mathcal{C}_{M_1} - \mathcal{C}_{M_0}$.

The no-hypercompression-inequality (Equation 12) [14] states that for a dataset D^* generated by a distribution P^*, the probability that a code different from P^* compresses the data more by at least K bits is 2^{-k}, and thus exponentially small.

$$P(-\log P^*(D^*) \geq -\log Q(D^*) + K) \leq 2^{-K} \tag{12}$$

The no-hypercompression-inequality can then be rearranged for an LRT context to

$$P\left(\log \frac{Q(D^*)}{P^*(D^*)} \geq K\right) \leq 2^{-K}, \tag{13}$$

in which K is the critical value.

Now, as per the no-hypercompression-inequality, the type-I error rate of the LRT without correction (the difference in model code length) of Equation 11 is

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0}\right) \leq \alpha. \tag{14}$$

For this, we will be assuming that the probability distribution of H_0 follows a Bernoulli distribution with parameter θ_0 that is estimated from the data[1]. According to this equation, the significance level α of this test is equal to $2^{-(\mathcal{C}_{M_1} - \mathcal{C}_{M_0})}$, and thus $\mathcal{C}_{M_1} - \mathcal{C}_{M_0} = -\log(\alpha)$. The difference in NML-complexity terms thus functions as the critical value in this LRT.

Now, when the difference in model code length is added to the test, we obtain a type-I error rate of

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0} + L(M_1) - L(M_0)\right) \leq 2^{-(-\log(\alpha) + L(M_1) - L(M_0))}. \tag{15}$$

We will be denoting $log(m) = L(M_1) - L(M_0)$, where m is an estimate of the number of models searched to find the current rule. m can thus also be seen as the number of hypothesis tests performed. This is to show that adding $log(m)$ to the LRT test changes the significance level in such a way that it functions as a multiple testing correction.

4.1.1 Multiple testing correction and model code length

We will now show how the model encoding functions as the multiple correction term by first assuming a simplified model encoding scheme. We will use this simplified

[1] In this hypothesis test, H_0 is that adding the new rule will not improve the rule list, and the alternative hypothesis H_1 is that adding this rule will improve the rule list.

encoding scheme to show how to obtain the number of rules searched (and thus the correction term) from the model encoding scheme. To show that the difference in model code lengths is indeed related to the number of rules searched, we will, as mentioned, assume a simplified model encoding scheme. In this simplified encoding scheme, instead of encoding the combination of features using $\log\binom{f}{|a_i|}$, we encode the used features individually. Encoding features one by one is also common in practice. One reason for this is that this ensures that longer rules have larger model code lengths. With this method, as there are f possible options to choose from when adding a feature to a rule, each used feature would require a code length of $\log(f)$ to encode. This leads to a code length of $|a_i| * \log(f)$ to encode all of the used features in a rule of length $|a_i|$. We will also leave out the encoding of the length of the rule $L_{\mathbb{N}}(|a_i|)$ and the encoding of the number of rules $L_{\mathbb{N}}(|S|)$. Since these code lengths are, in practice, much smaller than the code length required for encoding a rule, we will ignore them for the sake of simplicity.

We thus get the following as the simplified model encoding scheme,

$$L_{simplified}(M) = \sum_{a_i \in S} \left[|a_i| * log(f) + \sum_{v \in a_i} L(v) \right] \tag{16}$$

With this simplified encoding scheme, the difference in code lengths between M_1 and M_0 will then be $\log(f) + \log(|v_i|)$, where v_i is the newest variable introduced in M_1. This is because M_0 is an empty rule and M_1 only has a single condition in this scenario. Note that this also holds when comparing a non-empty M_0 and an M_1 that is M_0 with one extra condition.

The difference in code length between M_0 and M_1 is $\log(f * |v_i|)$. Assuming that all f features have a cardinality of $|v_i|$ for simplicity, the number of rules searched to find M_1 then is $m = f * |v_i|$ as there are f possible features, each having $|v_i|$ values.

With this estimated number of rules searched m, when we insert this back into Equation 15, we get

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0} + \log(m)\right) \leq 2^{-(-\log(\alpha)+\log(m))},$$

which then becomes

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0} + \log(m)\right) \leq \frac{\alpha}{m}. \tag{17}$$

We thus see that when the model encoding is added, the critical value α gets divided by the number of additional rules searched. This number of rules searched can also be regarded as the number of hypothesis tests performed, thus functioning as a Bonferroni correction, showing that the difference in model code length does indeed function as a multiple testing correction.

4.1.2 How constructed features break the connection

Now, when a constructed feature is added, the correction term no longer reflects the number of rules searched when this feature is used. This is because this new binary feature does not reflect the number of rules searched for the rule that was used to create this feature. For encoding a rule just consisting of this newly constructed binary feature, following Equation 16, only $\log(f+1)+\log(2)$ bits are required ($f+1$ because there is one more feature introduced), and thus $(f+1)*2$ rules searched. Logically, the rule that was used to construct the new feature likely has either more conditions and/or a higher cardinality, thus resulting in a much higher number of estimated rules searched than $2*(f+1)$. This thus breaks the connection between multiple testing correction and $L(M_1)-L(M_0)$ because the actual number of rules searched no longer matches the difference in code length.

4.2 Correction

In order to resolve this discrepancy in the number of rules searched when features are added to a dataset, we propose a new model encoding and correction scheme such that the connection between $L(M_1)-L(M_0)$ and the number of rules searched, m, still holds. We will do this by correcting the code length required for encoding these newly constructed features in such a way that they reflect the complexity of the rules they are based on.

In order to achieve this correction, we modify the code length of the rules containing constructed features. In this modification, instead of encoding the feature as if it were a normal feature, we first calculate the number of rules searched to find the rule this feature was based on and encode that to obtain the code length.

Since we are only modifying the code length of the conditions of a rule that uses constructed features, we must make some changes to the model encoding scheme. These changes are required to allow us to replace the code lengths of individual conditions with the corrected code length.

4.2.1 Our model encoding scheme

We will now go over the basis of our new model encoding scheme. We will first explain the concept by considering only categorical features before expanding to numeric features. We will be using the term condition to describe the individual Boolean expressions separated by the AND statements of a rule (e.g., the rule: $weight < 600$ AND $age > 1$ has two conditions: weight < 600, and age > 1).

In our new model encoding scheme, we will only encode the rules that are searchable by the greedy beam search. Since a greedy beam search can only cover a subset of the entire search space, only the rules in that searchable subset are encoded.

For example, a beam search with a single beam containing the rule $weight < 600$ will never search the rule $has\_spikes = false$ AND $age > 1$ because this rule does not expand on the rule currently in the beam. This rule will never be considered, and it is thus not necessary to encode this rule.

This would give a closer representation of how many rules were actually searched to find and add an additional condition, leading to a more accurate correction term.

For the simplest scenario, we will first assume that we are working with a beam width of 1. We will also be assuming that features can not be used twice in the same rule, i.e., a feature can only appear in a single condition of the rule. This is to prevent rules from containing redundant conditions or an impossible combination of conditions. Features that appear in a condition with two operators (e.g $1 < age \leq 4$) are treated as a single occurrence of that feature.

With a beam width of 1, the number of rules searched, m, can be calculated using Equation 18.

$$m = \sum_{j=0}^{|a_i|} (f - j) * |v_{a_{ij}}| \tag{18}$$

In this equation, a_i is the i-th rule, and $|a_i|$ represents the number of conditions in said rule (i.e., the length of the rule). f is the total number of features in the data set, and $|v_{a_{ij}}|$ is the number of candidate values of the j-th feature (the feature that is part of the j-th condition) of rule a_i.

To calculate the total number of searched rules, we calculate the number of searched rules to find each individual condition and add it together. For example, in a dataset with three binary features, you would need to search $3 * 2 = 6$ (three features, each having two possible values) rules to find the first condition of the rule. So, we first calculate how many rules were searched to find the first condition, and then we do the same for all other conditions of the rule.

To obtain the number of rules searched to add a singular condition, we multiply the number of unused features $(f - j)$ by the number of unique values of the chosen feature. For example, for the first condition, there are f possible features, and the feature we are encoding has $|v_{a_{ij}}|$ possible values. Thus, an estimated $f * |v_{a_{ij}}|$ rules are searched to find this first condition. To find the second condition, only $f - 1$ features are possible.

We will now cover how the numeric features are handled in our encoding scheme as these do not have a cardinality that we can use to calculate the number of rules searched in Equation 18. Therefore, we make use of the fact that numeric features are divided by a number of cut points (n_{cut}), which is a parameter of the algorithm.

To find the cardinality of numeric features $k_{\#}$, we first find the total number of possible value combinations given a single operator and the number of cut points ($N(1, n_{cut})$). We add this to the number of possible value combinations for two operators with the same number of cut points ($N(2, n_{cut})$). These numbers of combinations are found in the same way as seen in Equation 19.

$$k_{\#} = N(1, n_{cut}) + N(2, n_{cut}) \tag{19}$$

The total code length of a model using this encoding scheme would then be

$$L(M) = L_{\mathbb{N}}(|S|) + L_{\mathbb{N}}(w) + \sum_{a_i \in S} \left[L_{\mathbb{N}}(|a_i|) + \log \left(w * \sum_{j=0}^{|a_i|} (f - j) * |v_{a_{ij}}| \right) \right] \tag{20}$$

where we first encode the constants, the number of rules in the rule list $|S|$, and the beam width w, using the universal code for integers. Then, we encode each rule in the rule list as follows: for each rule, we first encode the length of the rule using the universal code for integers. We then calculate the number of rules searched and multiply it by the beam width. We multiply the number of searched rules on a single beam as calculated in Equation 18 by the beam width to generalise to a wider beam. This results in the maximum number of rules that can be searched in w beams. This method does not account for the possible overlap of the beams and, thus, only gives the upper bound of the number of possible rules instead of the actual amount. Finally, we calculate the code length required to encode these rules by taking the log of this number of rules. Because there are only a fixed number of searchable rules, we can use the log instead of using the universal code for integers. When we again assume that all features have the same cardinality, the number of rules searched that we calculated is exactly the number of possible rules, meaning we can thus encode each possible rule this way.

Now that the number of rules searched for each condition is calculated individually, it is possible to modify specifically the conditions using constructed features. Additionally, the code length now represents the actual number of rules searched using a beam search and, thus, how many rules were actually tested.

Using this encoding scheme, the difference in model code lengths of no rule (M_0) and a rule (M_1) still functions as a correction term when we insert this into Equation 15. For simplicity, we are still ignoring the difference in the encoding of the length of the rule, and the difference of the encoding of the number of rules. We then get:

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0} + \log\left(w * \sum_{j=0}^{|a_i|}(f-j) * |v_{a_{ij}}|\right)\right) \leq \frac{\alpha}{w * \sum_{j=0}^{|a_i|}(f-j) * |v_{a_{ij}}|},$$

which, because inside the log is the number of searched rules, then becomes

$$P_{H_0}\left(\log(\frac{P(D|M_1)}{P(D|M_0)}) > \mathcal{C}_{M_1} - \mathcal{C}_{M_0} + \log(m)\right) \leq \frac{\alpha}{m},$$

thus showing that our model encoding scheme can still function as the multiple testing correction term.

4.2.2 Our correction method

Now that the encoding scheme has been sufficiently changed to allow the correction to be applied, we will now discuss the process of the correction in more detail.

In order for the correction to be applied, we will be assuming that the following information will be provided for each constructed feature: the length of the rule they represent, and the features used in each condition of the rule.

For each constructed feature, the number of rules searched for the rule it represents can then be calculated with Equation 18 using the provided information. The corrected

number of rules searched (the number of rules searched to find the rule the constructed feature was based on) for a constructed feature v will be denoted as $n_{corrected}(v)$.

With this new information, the calculation for the number of rules searched must now also take the constructed features that require the corrected number into account, therefore, the calculation in Equation 18 will be replaced by Equation 22 when correction needs to be applied.

$$n_{rules}(j) = \begin{cases} n_{corrected}(v_j), & \text{if } v_j \text{ is a constructed feature} \\ (f-j) * |v_{a_{ij}}|, & \text{otherwise} \end{cases} \tag{21}$$

$$m = w * \sum_{j=0}^{|a_i|} n_{rules}(j) \tag{22}$$

In these equations, $n_{rules}(j)$ is the number of rules searched to find the j-th condition. This is split into two cases. The first is that the condition uses a regular feature, and thus, no corrective actions need to be taken. The second case is that the feature used is a constructed feature. In this case, instead of calculating the number of rules searched as normal, we find the number of rules searched to find the rule that this constructed feature was based on $n_{corrected}(v_j)$. This is calculated as per Equation 18, which is then multiplied by the beam width. This forms the basis of our correction.

When this is further applied, the complete calculation for the corrected model code length would be:

$$L(M) = L_{\mathbb{N}}(|S|) + L_{\mathbb{N}}(w) + \sum_{a_i \in S} \left[L_{\mathbb{N}}(|a_i|) + \log \left(w * \sum_{j=0}^{|a_i|} n_{rules}(j) \right) \right] \tag{23}$$

Using this corrected encoding scheme, a rule using a constructed feature will now have an increased code length compared to code length of the same rule without correction. This correction thus replaces the number of rules searched to find this condition using a constructed feature with the number of rules searched to find the rule the constructed feature was based on. This corrected code length now also reflects the complexity of the rules the constructed features are based on whilst still having a lower code length than them. Now, the only difference between their code lengths is the encoding of the length of the rule ($L_{\mathbb{N}}(|a_i|)$).

This allows the inherent overfitting limiting of MDL to better judge the complexity of constructed features, whilst still encouraging the use of these features as they have a smaller code length compared to their original rule. And since the code length of rules using the corrected features do not have the exact same code length as the same rules using the rules these features represent, different models may be able to be found when the constructed features are used.

5 Experiments

In our experiments we will demonstrate on both simulated datasets and real-world datasets the effects of treating constructed features as normal features. More specifically, we will look at the model complexity, predictive power and robustness against learning spurious rules of models trained with increasing numbers of constructed features. First, we start with the SSD++ models, where we begin by looking at a synthetic dataset where the true rules are known, after which we look at the real-world datasets. Next, in Section 5.8, we will repeat the experiments with our proposed encoding and correction scheme.

5.1 Experiment setup: simulated dataset

First we look at a simulated dataset where the true rule set used to generate the data is known. For the simulated data, we generated a binary dataset with 5000 samples from a fixed set of five rules. These five rules all have lengths of two or three conditions, and the total number of features in the dataset is six $(x_1, ..., x_6)$, with target variable Y. The exact rules can be seen in Figure 2. The dataset was created by generating a 5000×6 matrix with random binary values. The value of the target variable Y was then assigned as per the rules in Figure 2.

R1:	IF	$x_2 = true$	AND	$x_4 = true$			THEN	$P(Y = true) = 0.95$
R2:	ELIF	$x_1 = true$	AND	$x_3 = true$	AND	$x_5 = true$	THEN	$P(Y = true) = 0.8$
R3:	ELIF	$x_1 = false$	AND	$x_2 = false$	AND	$x_5 = true$	THEN	$P(Y = true) = 0.65$
R4:	ELIF	$x_2 = true$	AND	$x_4 = false$			THEN	$P(Y = true) = 0.5$
R5:	ELIF	$x_3 = false$	AND	$x_4 = true$			THEN	$P(Y = true) = 0.35$
D:	ELSE							$P(Y = true) = 0.2$

Fig. 2: The rules used to generate the data for the simulated experiments

When adding the constructed features to the dataset for the experiment we have added each of the five true rules as a feature to the dataset.

For the models trained using the simulated data we only have to look at the learned rules to determine if something has gone wrong because the true rules are already known. Thus, if we see that the trained models have rules that do not exist in the true rule set, we know that the model has fitted to the random noise in the data. The exact metrics used are discussed in Section 5.2.

In order to see the potential issues caused by adding features based on rules to a dataset, we start training SSD++ models with no features added as the baseline. Then we train with increasing numbers of constructed features added to the dataset. This allows us to see if adding constructed features to a dataset leads to a different set of learned rules compared to the true rules.

To reduce the randomness of the results, the models were trained and tested on 10 stratified folds for all datasets. This keeps the percentage of samples roughly the same for each class in each of the 10 training sets. This was done using the `StratifiedKFold`

function from scikit-learn[25] with the `randomstate` set to 0. Of these 10 results, the mean and the standard deviation of these folds were reported for all measured metrics.

5.2 Metrics: simulated data

For the experiments using a synthetic dataset we will use two methods to evaluate the models. Since the true rule set of this dataset is already known, we first compare the number of learned rules to the true number of rules to see if these differ. Next we will compare the coverage of these rules to see if the learned rules cover the same samples as the true rule set. To compare these rules we will use the Jaccard index, which can be calculated by dividing the number of samples that appear in both compared rules by the total number of unique samples of both rules combined. The Jaccard index is calculated for all combinations of learned rules and true rules as the n-th learned rule may not always correspond to the n-th true rule.

5.3 Results: Simulation

In Figure 3 we can see the average number of rules of the 10 folds for a model trained without added features, as well as a model trained with features. The true number of rules for this dataset is also displayed.

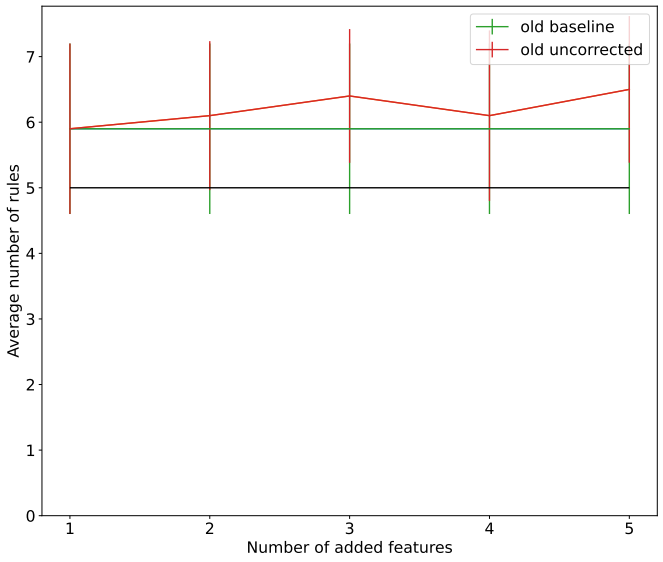


Fig. 3: The average number of rules in a rule list trained with increasing numbers of added features (old uncorrected) compared to a rule list trained without added features (old baseline) with the true number of rules (black) as reference.

In the figure, the old baseline is the baseline SSD++ model trained without added features. The old uncorrected model is the SSD++ model trained with additional features. We can see that the baseline model learns on average one more rule than there are rules in the true model. This shows that there already is some fitting on the random noise in the baseline model. We also see that when additional features are introduced into the dataset, the average number of learned rules increases compared to the baseline with no added features. This means more spurious rules are learned when more features are added, and that the added features increase the risk of learning spurious rules compared to not adding them.

In Figure 4 we can see the Jaccard similarity of these learned rules to the true rules for a model with no added features, a model with a single added feature, and a model with five added features. The results in these figures are from the same randomly chosen fold. The true rules (including the default rule) are on the x-axis, and the learned rules are on the y-axis.

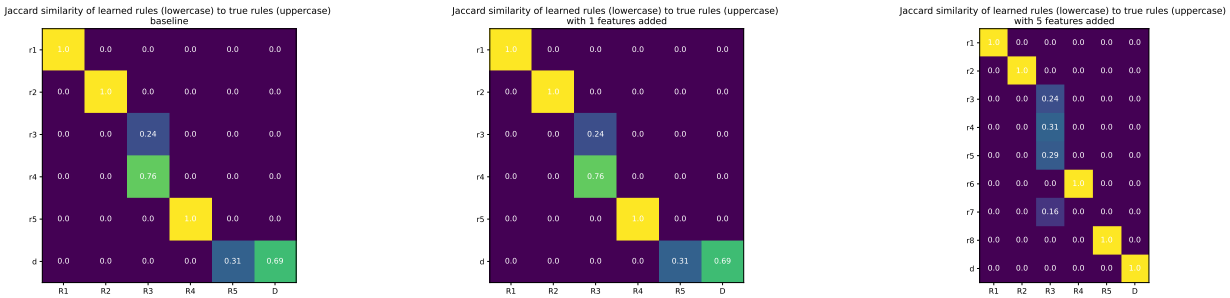


Fig. 4: The Jaccard index between the true rules (x-axis) and the learned rules (y-axis) for a model trained without added features, with 1 added feature, and 5 added features

In the figures with no added features and 1 added feature, we see that they are the same. When we compare these to the true rules, we see that three of the rules (r1, r2, and r5) directly match true rules. We also see that the samples from the third true (R3) rule are split between two rules (r3 and r4), this shows that, indeed, the model has fitted to random noise because any difference between r3 and r4 would be due to randomness in the data. Finally, we see that the samples that should be part of the fifth true rule are found in the default rule of the learned model. This may be because the noise in the data causes the merged default rule to be slightly more optimal as the default rule and fifth true rule are quite close in value.

In the figure with 5 added features, we see that the third true rule is now split between four rules. Showing that the fitting on random noise has gotten worse. We also see that the fifth true rule is now no longer in the default rule. This is likely because the fifth true rule has now been added as a feature, meaning that adding a rule using this feature requires a smaller code length. This likely caused the addition of the rule to be better than incorporating it in the default rule.

We have seen that adding more features has worsened the learning of spurious rules. With no features added, a true rule was split over two learned rules, but with 5 added features, it has split even further to four rules. We can thus conclude that there is indeed some risk of problems occurring when multiple features are added to a dataset. This increase in rules learned when more features are added may sometimes be beneficial as it can help in discovering true rules, but it also makes the model less robust against fitting to random noise.

5.4 Experiment setup: real world datasets

We repeat this experiment on several real-world datasets. The datasets used can be seen in Table 1, which describes the sizes of the datasets and the number of classes of the target variable. These datasets are of various sizes to better see the effects in different scenarios. We will again be using 10 folds to reduce the randomness of the result.

Name	Samples	Features	Target Labels
Ipums [28]	8844	56	2
Muskv2 [5]	6598	166	2
Student success [19]	4424	36	3
Apple quality [23]	4000	8	2
Optdigits [3]	3823	64	10
Diabetes [1]	2460	8	2
Auction verification [24]	2043	7	2
Mobile price [30]	2000	20	4
Wine quality (red) [7]	1599	11	6
Heart failure [8]	918	11	2
Mine [36]	338	3	5
Cirrhosis [10]	276	17	3

Table 1: The datasets used for the experiments

5.4.1 Adding features to datasets

To select the rules to add as features to the real-world datasets, we first train an SSD++ model on these datasets. We then choose the first 10 rules of these models to add as constructed features to the datasets. For each rule to be added as a feature, all samples in the dataset that fulfil the conditions specified in the rule get marked as *true* in the new feature, the rest are marked *false*. Note that this may include samples not covered by the rule, as we are not excluding the samples covered by earlier rules that may also fulfill the conditions of the added rule. We are thus treating the rules to add as features as standalone rules, because this is simpler to interpret. This way, the described feature only needs a single rule to describe it instead of requiring the entire rule list.

For datasets where fewer than 10 rules were learned, only those $n < 10$ rules were added as the constructed features for those datasets. The decision to use the first 10

rules as the constructed features was made due to insufficient domain knowledge on all datasets, and thus we would not necessarily recognize the interesting rules.

5.5 Metrics: real-world datasets

For the real-world datasets, different metrics are required as the true rules are unknown. Instead, we will be measuring the code lengths of the models and their predictive power.

5.5.1 Code length

The first metric used is the two-part code length of the model, i.e., the code length of the data (Eq. 1) plus the code length of the model (Eq. 2). This measures the compression achieved by the model, and thus how well the model is able to describe the data (a smaller code length thus means a better compression). Since models were trained with different numbers of features, using the model code length without any changes leads to models having different model code lengths for the same rules. In order to be able to compare these models without such inequalities, we will be encoding the models as if they used the original datasets without added features. For this to work, we must first replace all constructed features used in the rule list with the rules these features represent since the original dataset does not have these features.

So for a dataset with features $x_1, x_2, ..., x_n$ and constructed features $z_1, z_2, ..., z_m$, all constructed features are substituted by the rules they represent for all rules in the rule list such that no rule contains any constructed feature. For example, if z_1 represented the rule "$x_1 > 1$ **and** $x_2 > 1$", then substituting z_1 in the rule "$z_1 == True$ **and** $x_3 > 1$" would result in "$x_1 > 1$ **and** $x_2 > 1$ **and** $x_3 > 1$". We then encode the rules using the encoding scheme from the method section (Eq. 20).

This metric was chosen because we can easily compare models and see which models are better from an MDL perspective, which is the model with the lowest two-part code length. A lower two-part code length means that the model has found a better balance between model complexity and data compression. If we see the two-part code lengths increase as more features are added, we can conclude a worsening performance due to these features.

5.5.2 ROC-AUC

The second metric used is the area under the receiver operating characteristic curve (the ROC-AUC score). This is a commonly used metric to score model performances with a score in the range $[0, 1]$, where 1 is the score for the perfect model. The receiver operator curve plots the true positive rate against the false positive rate, where a larger area under the curve indicates a better model. For the datasets with non-binary targets, the AUC for each class is computed against all other classes (one vs rest) and the macro (unweighted) average was taken in order to obtain the ROC-AUC score for the model.

The ROC-AUC score is calculated on the data of the training set, as well as the data of the test set. The ROC-AUC score was used to also see the predictive

performance of these models in addition to the achieved compression seen in the code length.

5.6 Results: Code length

Next, we will start by looking into the effect of these added features on the performance of the models. The first performance-based metric we look at is the code length of the model and the combined two-part code length of the model and data encoding.

When a single constructed feature is added to a dataset, we see in Figure 5 that for half of the datasets, the model code length, and thus model complexity, increases. For the other half, it decreases to a lesser degree. When more and more features are added, these differences tend to become more pronounced in most cases, as can be seen in the graph with 10 features added. However, in Figure 6 we can see that for most datasets with an increase in model complexity, the total code length remains very similar to the baseline model. This suggests that although the model becomes more complex, there is no increase in performance, and thus a lower total code length, showing the potential problem of increased complexity without a gain in performance (such as learning spurious rules). Although we can see an exception in the auction verification dataset, all other cases with an increased model complexity do not show an improvement in their total code length, suggesting that adding features indeed risks some problems. The presence of datasets that do show a better model from an MDL perspective when features are added (and thus have a lower total code length) may suggest that adding the features can also show a positive impact of human-added features. Suggesting that adding human insight into a dataset can improve models trained on that data.

5.7 Results: ROC-AUC

The next metric we are looking at is the average ROC-AUC scores of the models over the 10 folds.

In Table 2 the average ROC-AUC score for the models trained on each dataset for 1 to 10 added features. Column 0, shows the score of the baseline model that was trained without added features.

In these tables, for most datasets we see almost no change in ROC-AUC score for the models trained with added features on the test set. The three datasets that do show the largest difference are the cirrhosis dataset which shows a noticeable decrease in ROC-AUC score when features are added, and the auction verification and mobile_price set which show an increase.

From only looking at the ROC-AUC scores, we can see that for most datasets the ROC-AUC does not change when constructed features are added. It is possible that these models have learned spurious rules like we have seen with the simulated data, in such a case there will be no improvement in the ROC-AUC scores of these models since these spurious rules describe the same data.

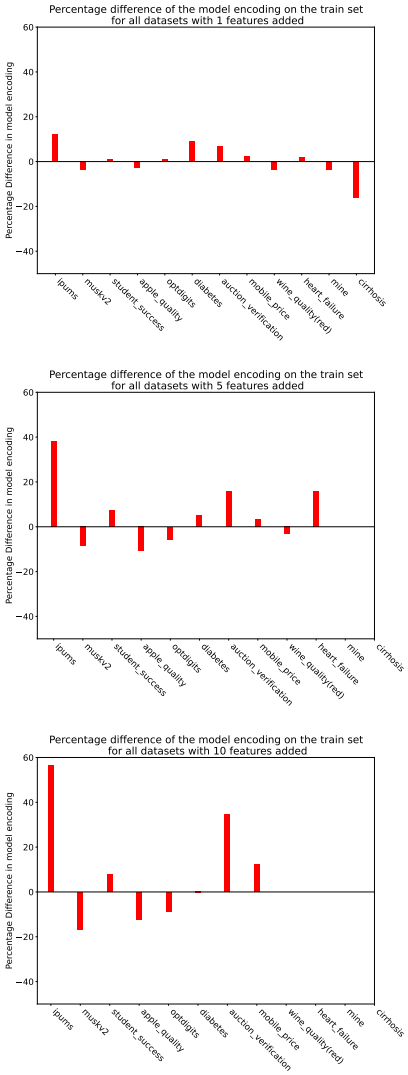


Fig. 5: The difference (in percentage) of the code lengths between the model encodings of the models trained without additional constructed features (the baseline), and models with 1, 5, and 10 added features.

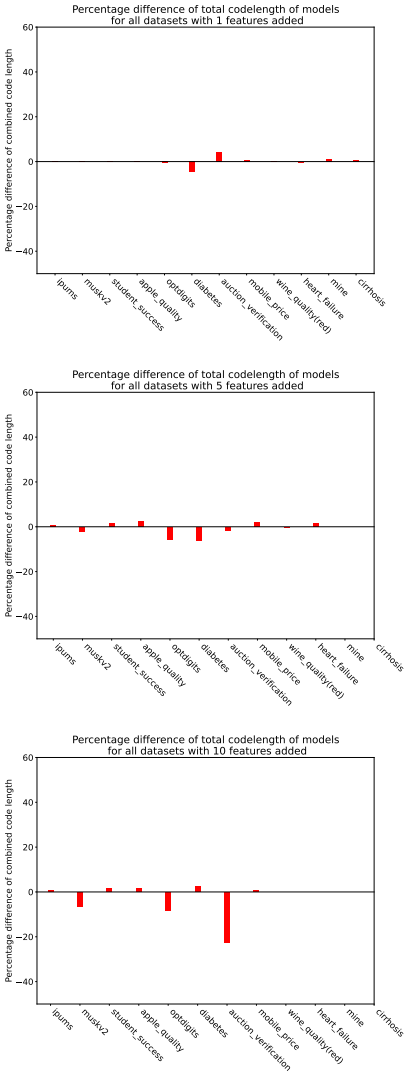


Fig. 6: The difference (in percentage) of the code lengths between the total code lengths of the models trained without additional constructed features (the baseline), and models with 1, 5, and 10 added features.

Dataset	0	1	2	3	4	5	6	7	8	9	10
muskv2	0.934	0.916	0.935	0.926	0.926	0.926	0.938	0.938	0.939	0.936	**0.942**
ipums	0.545	0.553	0.554	0.564	0.568	0.576	**0.577**	0.574	0.574	0.577	0.574
apple_quality	**0.798**	0.792	0.779	0.796	0.787	0.781	0.797	0.796	0.8	0.8	0.796
mine	0.762	0.753	0.775	**0.813**							
auction_verification	0.886	0.901	0.905	0.903	0.913	0.905	0.905	0.95	0.947	0.96	**0.964**
diabetes	0.912	0.918	0.906	0.884	0.86	**0.92**	0.899	0.91	0.894	0.901	0.913
heart_failure	0.836	0.844	0.828	0.829	0.857	0.86	**0.861**				
optdigits	0.807	0.804	0.817	0.831	0.819	0.819	0.82	0.834	0.82	**0.835**	0.823
wine_quality(red)	0.716	0.72	0.709	**0.726**	0.716	0.72	0.688	0.708			
student_success	0.792	0.8	0.803	0.795	0.8	0.798	0.797	0.802	0.804	**0.806**	0.803
cirrhosis	**0.552**	0.439	0.439								
mobile_price	0.393	0.382	0.398	0.403	0.404	0.401	0.402	0.41	0.409	0.421	**0.423**

Table 2: The average ROC-AUC on the test set over 10 folds for each dataset with 0 (baseline) to 10 added features trained using the SSD++ algorithm.

5.8 Correction experiment

Next we will be repeating the experiments using our proposed model encoding scheme instead of the encoding scheme from SSD++ using the same metrics. In this experiment we will be comparing the performances of the models with correction applied, and without correction (and thus treating the added features as normal features). As with the previous experiment, these models are trained with an increasing number of constructed features, and as a baseline to compare to, a model trained without the added features is used. All of these models are trained using our proposed model encoding scheme.

We will be using the same metrics as before, to see if our proposed multiple testing correction has a positive effect on models trained using it. For training the models, the same settings were applied as in the previous experiment.

5.8.1 Results: Simulation

We will now again start by looking at the simulated dataset. This time the models are trained with our proposed model encoding scheme, both with, and without our correction from Equation 23. The results of this can be seen in Figure 7, where we again look at the average number of rules compared to the true number of rules. In this figure, the new baseline refers to the baseline model trained with our modified encoding scheme. The new uncorrected model refers to the models trained with added features, but without applying the correction. The new corrected model refers to the models trained with the modified encoding scheme with correction applied.

Again, we see that all models, including the corrected model, have a greater number of rules than the true model. This may be because each individual rule requires a smaller code length in our encoding scheme than their equivalent in the SSD++ encoding scheme. We are thus still encountering the same problem we have seen in the SSD++ model. In this case however, we see the opposite happening, both the uncorrected and the corrected model with features added have learned fewer rules than the baseline using our modified encoding scheme. At five added features, we see a similar number of rules as the original encoding scheme. Perhaps the increase in the

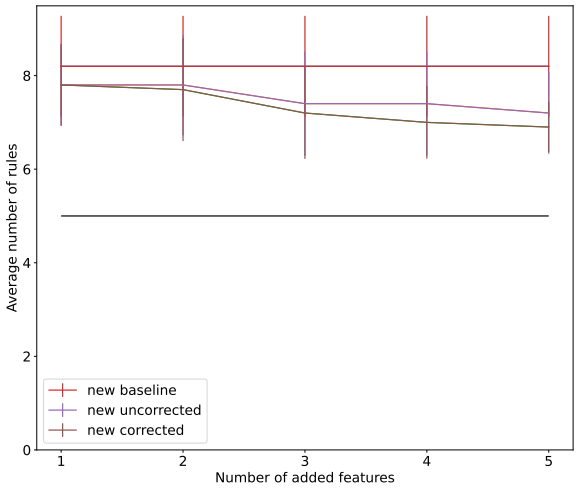


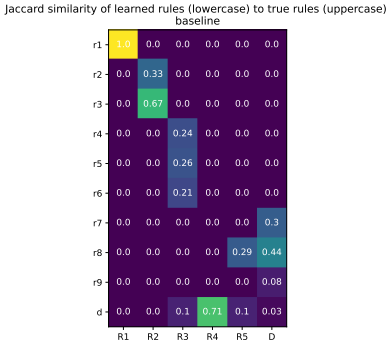
Fig. 7: The average number of rules in a rule list trained with increasing numbers of added features without correction (new uncorrected), and with correction (new corrected), compared to a rule list trained without added features (new baseline) with the true number of rules (black) as reference.

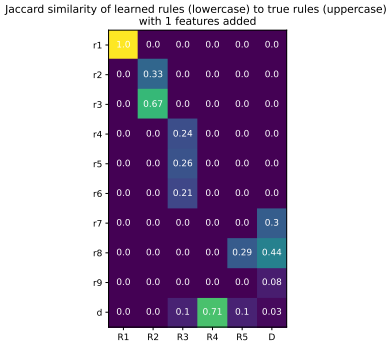
number of features has a higher impact in our encoding scheme than in the encoding scheme used in SSD++, thus leading to fewer rules learned when more features are added, as each rule has a higher required code length.

Since the results from the corrected models are so close to the results of the uncorrected models, it is likely that he correction has little to no impact on the number of learned rules. This may be due to the code length difference for encoding the same rule with correction and without correction being relatively small compared to the data encoding, resulting in similar numbers of rules.

In Figures 8 and 9 we can see the Jaccard similarity of these learned rules to the true rules for a model with no added features, a model with a single added feature, and a model with five added features. Figure 8 shows the uncorrected models and 9 shows the corrected models. The results in these figures are from the same randomly chosen fold. The true rules (including the default rule) are on the x-axis, and the learned rules are on the y-axis.

In these figures, we see that both the uncorrected and the corrected results are the same. We see the same similarity scores for the learned rules for all numbers of added features. With zero and one added features, we see that several of the true rules are split up between two or more rules, which shows that the model is again fitting to random noise in the data. We also see that more rules were learned compared to the SSD++ model, like we have seen in Figure 7. With five added features, we see a





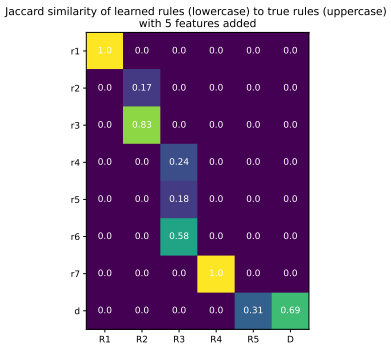
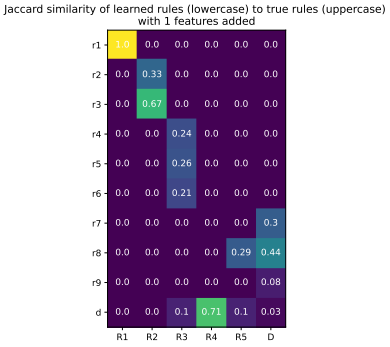


Fig. 8: The Jaccard index between the true rules (x-axis) and the learned rules (y-axis) for a model trained without added features, with 1 added feature, and 5 added features. All models are trained using our proposed encoding scheme without correction applied



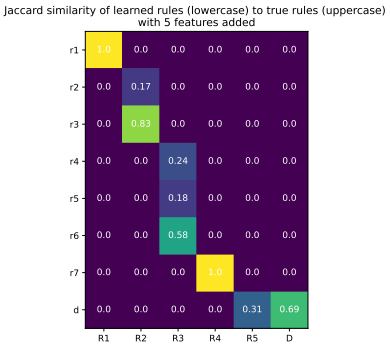


Fig. 9: The Jaccard index between the true rules (x-axis) and the learned rules (y-axis) for a model trained without added features, with 1 added feature, and 5 added features. All models are trained using our proposed encoding scheme with correction applied (except for the baseline)

somewhat better result, as the true rules are now split between fewer rules, but the same problem is still present.

5.8.2 Results: Code length

Next, we will be looking at the code lengths of the model encoding in Figure 10 in order to see if applying correction influenced the learned model complexity. For both the corrected and the uncorrected models, the code lengths were calculated as using the same encoding scheme as described in Section 5.5.1 for a fair comparison. In these figures, the new uncorrected model refers to the models using our encoding scheme without correction, treating the constructed features as normal features. The corrected model refers to the model using our encoding scheme with correction applied.

We see that the difference in the encoding scheme does make a difference in terms of model code length of the learned models, however, we still see that for models trained with more features, the difference between the baseline (the model trained with no features) tends to increase. When correction is applied we still see the same behaviour as the uncorrected model, but often to a slightly lesser degree.

In Figure 11 we look at the combined code length for the corrected and uncorrected models. In this figure, we see that the corrected and uncorrected models behave very similarly to each other. Like the models trained with the SSD++ model, we still see that most of the models that had an increase in their model complexity have a total code length near identical to the baseline, suggesting that the same problems still occur with our proposed encoding scheme, even when correction is applied. It is likely that in those models, the rules are split into multiple rules like we saw in the simulated datasets and are thus not actually improving.

5.8.3 Results: ROC-AUC

Next, we look at the ROC-AUC scores for the corrected and uncorrected models trained using our modified encoding scheme.

In Table 3a the ROC-AUC scores for the uncorrected models trained with the modified encoding scheme are shown, and in Table 3b the results from the corrected models are shown. In the tables the average ROC-AUC score for the models trained on each dataset for 1 to 10 added features. Column 0, shows the score of the baseline model that was trained without added features.

We again do not see any noticeable difference in the ROC-AUC scores between the baseline and the models trained with added features (again the auction verification dataset shows the greatest increase in performance). Additionally both the corrected and the uncorrected models show near identical ROC-AUC scores for all datasets, suggesting that our correction does not have a large impact in improving model performance.

5.9 Discussion

We have performed experiments on datasets with increasing numbers of constructed features added. We performed these experiments on a simulated dataset with known

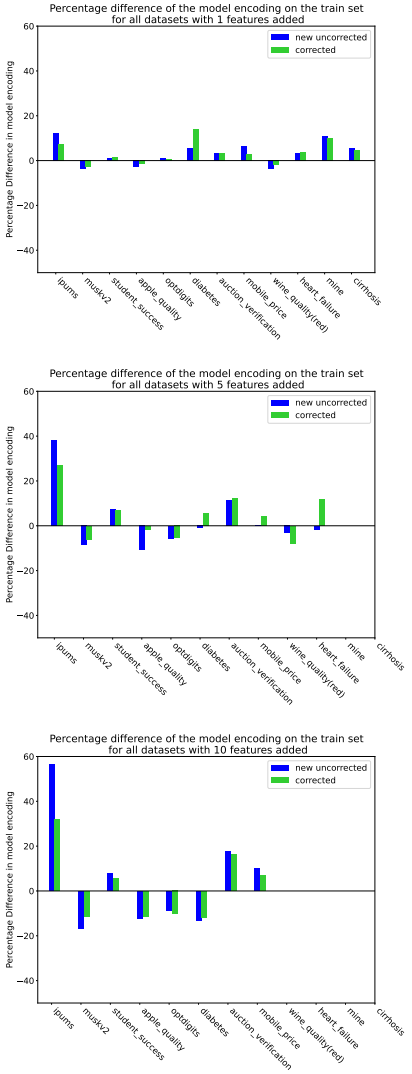


Fig. 10: The difference (in percentage) in the model encoding code length for the models trained without additional constructed features (the baseline), and models with 1, 5, and 10 added features. Where dark blue is our modified encoding scheme without correction applied, and green our modified encoding scheme with correction applied.

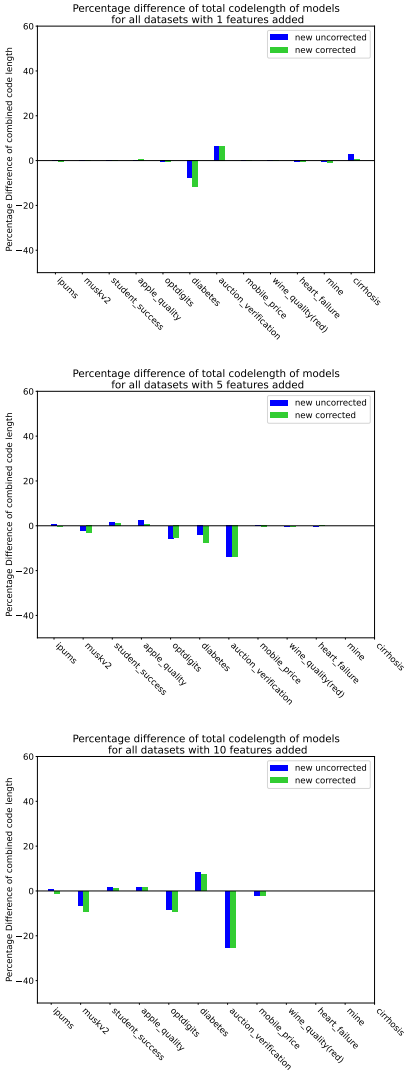


Fig. 11: The difference (in percentage) in the combined two-part code length for the models trained without additional constructed features (the baseline), and models with 1, 5, and 10 added features. Where dark blue is our modified encoding scheme without correction applied, and green our modified encoding scheme with correction applied.

Dataset	0	1	2	3	4	5	6	7	8	9	10
muskv2	0.932	0.933	0.933	0.928	0.93	0.93	0.932	0.929	0.932	**0.944**	0.941
ipums	0.579	0.585	0.578	0.577	0.586	0.589	0.586	0.59	**0.6**	**0.6**	0.599
apple_quality	0.793	0.787	**0.795**	0.78	0.78	0.784	0.781	0.791	0.782	0.775	0.785
mine	**0.773**	0.766	0.771	0.77							
auction_verification	0.87	0.882	0.874	0.875	0.931	0.932	0.932	0.971	0.959	**0.965**	**0.965**
diabetes	0.935	0.962	**0.976**	0.952	0.915	0.94	0.94	0.909	0.905	0.922	0.899
heart_failure	0.819	0.823	0.832	0.829	0.823	**0.839**	0.836				
optdigits	0.791	0.803	0.816	0.822	0.819	0.819	0.812	0.831	0.82	0.831	**0.836**
wine_quality(red)	0.697	**0.722**	0.671	0.714	0.689	0.698	0.699	0.685			
student_success	0.794	0.795	0.792	**0.797**	0.795	0.789	0.788	0.793	0.788	0.793	0.788
cirrhosis	0.527	**0.536**	0.522								
mobile_price	0.412	0.417	0.407	0.416	0.404	0.418	0.406	0.416	0.417	**0.424**	0.416

(a) The average ROC-AUC scores on the test set over 10 folds for each dataset with 0 (baseline) to 10 added features trained using our proposed encoding scheme without correction applied.

Dataset	0	1	2	3	4	5	6	7	8	9	10
muskv2	0.932	0.932	0.935	0.929	0.932	0.933	0.935	0.934	0.928	**0.951**	0.942
ipums	0.579	0.59	0.583	0.588	0.578	0.591	0.593	0.6	0.588	0.593	**0.595**
apple_quality	0.793	0.788	0.787	0.789	0.788	0.797	0.787	0.789	0.794	**0.798**	0.783
mine	**0.773**	0.763	0.762	0.771							
auction_verification	0.87	0.882	0.874	0.875	0.932	0.933	0.933	0.971	0.953	0.961	**0.968**
diabetes	0.935	**0.986**	0.938	0.969	0.971	0.962	0.958	0.953	0.953	0.929	0.909
heart_failure	0.819	0.825	0.832	0.826	0.816	0.821	**0.841**				
optdigits	0.791	0.803	0.813	0.824	0.817	0.82	0.811	**0.834**	0.823	0.828	0.829
wine_quality(red)	0.697	**0.718**	0.674	0.701	0.701	0.702	0.696	0.716			
student_success	0.794	**0.796**	0.79	0.795	0.786	0.791	**0.796**	0.794	0.786	0.787	0.788
cirrhosis	0.527	0.526	**0.536**								
mobile_price	0.412	0.41	0.418	0.408	0.424	0.413	0.42	0.411	0.418	0.415	**0.428**

(b) The average ROC-AUC scores on the test set over 10 folds for each dataset with 0 (baseline) to 10 added features trained using our proposed encoding scheme with correction applied.

Table 3

true rules, as well as on real-world datasets. We started with performing these experiments with the SSD++ model. From these experiments we have seen that when more constructed features are introduced, the models are more likely to learn spurious rules. On the simulated data, we saw that when more constructed features were added, the model learned more spurious rules than without any added features. Also, on the real-world datasets, we saw that for half of the datasets, the model complexity increased when constructed features were introduced, this increase tended to be more pronounced when more features were added.

Next, we performed the same experiments using our encoding scheme, both with and without correction applied to rules using constructed features. From the results of the models with our correction applied, we can see that the correction had little impact in changing the learned models across all metrics used. Even with correction, we see a similar increase in model complexity as for models without correction. In these cases, the combined code length is roughly the same as the baseline (the model trained on a dataset without added features). This means that the models with correction, just like the uncorrected models, become more complex without improving

the achieved compression, making the model unnecessarily complex. The ROC-AUC scores on the test sets were similarly unaffected by the correction, showing near identical results to the uncorrected model. This correction thus has not reduced the risk of problems occurring when constructed features are introduced. Perhaps the lack of impact of correcting the code length of added features may be due to the relatively small difference in code length between the corrected feature and the uncorrected feature compared to the much larger code length for encoding the data. Such a relatively small difference may not be large enough to make a noticeable difference in rule learning.

6 Conclusion

Domain experts adding rules as features to a dataset when feature engineering can lead to improved model performance when models are trained on these modified datasets. However, doing this potentially risks some problems, such as the model learning overly complex rules. This is because these constructed features allow the model to describe a subsection of the data in a shorter code length compared to the code length of the rule used to create the feature. This shorter code length allows the algorithm to learn additional rules, thus increasing the model complexity.

In this thesis we have looked at how to encode such constructed features for an MDL-based rule list model. To achieve this, we have connected MDL-based rule list learning to hypothesis testing and multiple testing correction as the basis of our model encoding scheme. In this connection, we have seen that the code length comparison for learning rules can be transformed into a likelihood ratio test where the difference in model code lengths of the candidate rule functions as the multiple testing correction term. For our new model encoding scheme we handle each condition of the rule individually, calculating the number of rules that has been searched to add each additional condition. The total number of rules searched is then used to encode the rule.

Using this new model encoding scheme, we have also proposed a correction method that corrects the code length required for encoding the constructed features. This correction method is based on multiple testing correction, where the corrected code length required to encode the constructed features is based on the number of rules searched to find the rule that this feature was based on. In our experiments, we have first compared SSD++ models trained on datasets without constructed features, to SSD++ models trained with increasing numbers of features. This was done to empirically demonstrate the risk of introducing these constructed features to a dataset. In these experiments we have demonstrated that with the SSD++ algorithm, treating these constructed features as normal features can indeed lead to some problems, such as rules being split on random noise when more features are introduced.

We have also repeated these experiments using our model encoding scheme. In these experiments, we have again trained models on a dataset without added constructed features. These models were then compared to models trained on datasets with increasing numbers of constructed features. This was done for models both with, and without, our correction applied to the constructed features. In the experiments using our proposed encoding scheme, both with and without correction applied to the

constructed features, we have seen that the models trained with correction do not show a major difference in predictive performance compared to the uncorrected models. Additionally, models with correction still suffer from the same problem of fitting to noise in the data when more constructed features are added.

From these experiments, we have concluded that our multiple testing correction-based encoding scheme had little effect in the rule learning process, leading to very similar results to an uncorrected model. This suggests that the potential problems caused by treating constructed features as regular features may require a different approach from code length-based methods to resolve.

6.1 Future work

Since the issue of fitting to noise still persists when the code length required to encode the constructed feature is corrected following the principles of multiple testing correction, it might be beneficial to look outside of a purely MDL context to solve this problem. One such example could be using a validation set to test if adding a candidate rule to the rule list actually improves the rule list.

Alternatively, it is possible to improve the limitations in this work, although it is likely that this will not significantly influence the results. These limitations include improving the calculation for the number of searched rules, as in this implementation, it is only an estimate, only providing the true number of searched rules if all features have the same cardinality. Another point to improve is taking into account that the beams in the beam search may have some overlap in their saved rules when estimating the number of rules searched with more than one beam.

References

[1] Ehab Aboelnaga. *Diabetes*. Version 1. Retrieved May 17, 2024 from https://www.kaggle.com/datasets/ehababoelnaga/diabetes-dataset.

[2] Musaab Mohammad Alhaddad. "Artificial intelligence in banking industry: a review on fraud detection, credit management, and document processing". In: *ResearchBerg Review of Science and Technology* 2.3 (2018), pp. 25–46.

[3] E. Alpaydin and C. Kaynak. *Optical Recognition of Handwritten Digits*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50P49. 1998.

[4] C.E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilità*. Pubblicazioni del R. Istituto superiore di scienze economiche e commerciali di Firenze. Seeber, 1936.

[5] David Chapman and Ajay Jain. *Musk (Version 2)*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C51608. 1994.

[6] Guoqing Chen et al. "A new approach to classification based on association rule mining". In: *Decision Support Systems* 42.2 (2006), pp. 674–689. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2005.03.005.

[7] P. Cortez et al. "Modeling wine preferences by data mining from physicochemical properties". In: *Decis. Support Syst.* 47 (2009), pp. 547–553. URL: https://api.semanticscholar.org/CorpusID:2996254.

[8] fedesoriano. *Heart Failure Prediction Dataset*. Version 1. Retrieved May 17, 2024 from https://www.kaggle.com/fedesoriano/heart-failure-prediction.

[9] Jonas Fischer and Jilles Vreeken. "Sets of Robust Rules, and How to Find Them". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Ulf Brefeld et al. Cham: Springer International Publishing, 2020, pp. 38–54. ISBN: 978-3-030-46150-8.

[10] Thomas R. Fleming and David P. Harrington. *Counting processes and survival analysis*. John Wiley & Sons, 2013.

[11] Esther Galbrun. "The minimum description length principle for pattern mining: a survey". In: *Data Mining and Knowledge Discovery* 36.5 (2022), pp. 1679–1727. DOI: 10.1007/s10618-022-00846-z.

[12] Jelle J. Goeman and Aldo Solari. "Multiple hypothesis testing in genomics". In: *Statistics in Medicine* 33.11 (2014), pp. 1946–1978.

[13] Peter Grünwald and Teemu Roos. "Minimum description length revisited". In: *International Journal of Mathematics for Industry* 11.01 (2019), p. 1930001. DOI: 10.1142/S2661335219300018.

[14] Peter D Grünwald. *The minimum description length principle*. MIT press, 2007.

[15] Jeff Heaton. "An empirical analysis of feature engineering for predictive modeling". In: *SoutheastCon 2016*. 2016, pp. 1–6. DOI: 10.1109/SECON.2016.7506650.

[16] A. Jović, K. Brkić, and N. Bogunović. "A review of feature selection methods with applications". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2015, pp. 1200–1205. DOI: 10.1109/MIPRO.2015.7160458.

[17] Junpei Komiyama et al. "Statistical Emerging Pattern Mining with Multiple Testing Correction". In: *Proceedings of the 23rd ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining. KDD '17. Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 897–906. ISBN: 9781450348874. DOI: 10.1145/3097983.3098137.

[18] Stephane Lallich, Olivier Teytaud, and Elie Prudhomme. "Association Rule Interestingness: Measure and Statistical Validation". In: *Quality Measures in Data Mining*. Ed. by Fabrice J. Guillet and Howard J. Hamilton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 251–275. ISBN: 978-3-540-44918-8. DOI: 10.1007/978-3-540-44918-8_11.

[19] Mónica V. Martins et al. "Early Prediction of student's Performance in Higher Education: A Case Study". In: *Trends and Applications in Information Systems and Technologies*. Ed. by Álvaro Rocha et al. Cham: Springer International Publishing, 2021, pp. 166–175. ISBN: 978-3-030-72657-7.

[20] Jamshed Memon et al. "Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR)". In: *IEEE Access* 8 (2020), pp. 142642–142668.

[21] Tommi Mononen and Petri Myllymäki. "Computing the multinomial stochastic complexity in sub-linear time". In: *Proceedings of the 4th European Workshop on Probabilistic Graphical Models*. 2008, pp. 209–216.

[22] Fatemeh Nargesian et al. "Dataset Evolver: An Interactive Feature Engineering Notebook". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). DOI: 10.1609/aaai.v32i1.11369. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11369.

[23] Elgiriyewithana Nidula. *Apple Quality*. Version 1. Retrieved May 17, 2024 from https://www.kaggle.com/datasets/nelgiriyewithana/apple-quality.

[24] Elaheh Ordoni, Jakob Bach, and Ann-Katrin Fleck. "Analyzing and Predicting Verification of Data-Aware Process Models–A Case Study With Spectrum Auctions". In: *IEEE Access* 10 (2022), pp. 31699–31713. DOI: 10.1109/ACCESS.2022.3154445.

[25] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[26] Hugo M. Proença et al. "Robust subgroup discovery: Discovering subgroup lists using MDL". In: *Data Mining and Knowledge Discovery* 36.5 (Aug. 2022), pp. 1885–1970. ISSN: 1573-756X.

[27] Jorma Rissanen. "A Universal Prior for Integers and Estimation by Minimum Description Length". In: *The Annals of Statistics* 11.2 (1983), pp. 416–431.

[28] Steven Ruggles and Matthew Sobek. *IPUMS Census Database*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5BG63. 1999.

[29] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[30] Abhishek Sharma. *Mobile Price Classification*. Version 1. Retrieved May 17, 2024 from https://www.kaggle.com/datasets/iabhishekofficial/mobile-price-classification.

[31] George Stalidis et al. "Recommendation Systems for e-Shopping: Review of Techniques for Retail and Sustainable Marketing". In: *Sustainability* 15.23 (2023). ISSN: 2071-1050.

[32] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. “Krimp: mining itemsets that compress”. In: *Data Mining and Knowledge Discovery* 23.1 (2011), pp. 169–214. DOI: 10.1007/s10618-010-0202-x.

[33] Tong Wang et al. “A Bayesian framework for learning rule sets for interpretable classification”. In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 2357–2393. ISSN: 1532-4435.

[34] Hongyu Yang, Cynthia Rudin, and Margo Seltzer. “Scalable Bayesian Rule Lists”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 3921–3930. URL: https://proceedings.mlr.press/v70/yang17h.html.

[35] Lincen Yang, Mitra Baratchi, and Matthijs Leeuwen. *Unsupervised discretization by two-dimensional MDL-based histogram*. Feb. 2023. DOI: 10.1007/s10994-022-06294-6.

[36] Cemal Tugrul Yilmaz, Hamdi Tolga Kahraman, and Salih Söyler. “Passive Mine Detection and Classification Method Based on Hybrid Model”. In: *IEEE Access* 6 (2018), pp. 47870–47888. URL: https://api.semanticscholar.org/CorpusID:52304953.