Boosted Rule Learning for Multi-Label Classification using Stochastic Gradient Descent

Exploring Potential Improvements using Iterative Modification
Bachelor thesis by Kevin Kampa
Date: Feburary 2021, Date of submission: 02.08.2021

1. Review: Dr. Eneldo Loza Mencía
2. Review: Michael Rapp
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Knowledge Engineering

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Kevin Kampa, die vorliegende Bachelorarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 02.08.2021

K. Kampa

Abstract

Multi-Label classification, the process of assigning multiple different labels to a set of examples is a more complex form of the traditional classification task. It can be achieved using many different approaches. One such approach is using boosted rule models. The idea here is that each rule by itself provides a weak prediction as to the overall labels, but by summing all these weak individual predictions up we can achieve one combined prediction that is actually reasonably strong and is a decent approximation of the learned data. Using an existing and refined gradient boosting framework for the learning of multi-label classification rules called BOOMER [28], we are proposing and developing an extension to the original algorithm in order to facilitate the modification of rules that have already been learned. Prior work in the field focused on other often more simple rule models or tree models (like SGD Trees[14], RIPPER [4] or IREP [12]) and managed to achieve good results when applying concepts such as post-pruning or replacing parts of rules with new ones. While the original algorithm does already provide decent results when using a sufficient number of rules, our goal is to use these insights and apply them to our chosen algorithm. We want to determine whether or not these concepts also work on a stochastic gradient descent boosted rule learner for multi-label classification and can yield better overall models while possibly using less complex rules than were required previously. We will later analyze our finished implementation using some commonly used classification datasets [31] to benchmark and appropriate metrics to evaluate whether any progress was made.

Contents

1 Introduction

To introduce this work I will provide an overview of the topics covered and briefly describe some key aspects. Furthermore, I will outline the overall structure of this thesis.

1.1 Improving Models of a Multi-Label Rule Learner

In the field of knowledge-engineering, or more specifically machine learning, the goal is to extract useful information from data. Ideally as data-scientists we want to develop very general models with a minimal amount of developer input. The less human-input an algorithm has the better its chances of actually discovering useful connections within the data to learn, since in most application of machine learning we humans are not able to comprehend the data in as much detail as an algorithm can, because we can not process these huge amounts of data all at once.
Machine Learning has become a focus of many new research projects due to the ever increasing amounts of data we are producing in our everyday life and which need to be analyzed in order to benefit from said data. While there are many different fields within machine learning, likely the most popular being neural-networks, [1] offer a decent starting point to dive deeper into neural networks. These networks are inspired by an early and incomplete understanding of how our brain works, the area we will be taking a closer look at is rule-learning. [11] Rule learning is one, if not the oldest technique of machine learning and also very closely connected to our way of thinking. We are easily able to understand how a rule is supposed to work and are able for the most part to fully comprehend the operations of simple rules.
Rule-based machine learning can also been done in a number of different ways. For instance we can follow a strategy of ”separate-and-conquer” as presented in several earlier works like [27] or [19]. This probably is the best understood approach to rule learning while also the easiest to understand. The name ”separate-and-conquer” (SeCo) is chosen because the data (or the problem) is divided into several sub-sets. Each sub-set is then covered by a different rule. While this particular approach will not be directly featured in this paper, the idea of ”SeCo” (as presented in [27]) is also implemented as part of the bigger research project which the featured algorithm ”BOOMER” ([28]) is a part of. ”BOOMER” is the multi-label rule learning algorithm this work is trying to extend and add new features to.
This work will focus on a technique known as ”gradient boosting” or ”gradient boosted rules” (as introduced in [28]), the idea is that our rules are all learned on the same chunk of data known as training data. This data is usually also being sampled from and rules are then built using different strategies like top-down or bottom-up rule induction, in the case of ”BOOMER” a greedy top-down search is used to learn rules.
Using the selected strategy and a provided function called a loss function, one can then calculate which new rules (or set of conditions) minimize the training-objective, a calculated value based on the loss-function and current model. The rule’s prediction is also more complex than it would usually be with the separate-and-conquer approach since a rule will provide a real-valued prediction, based on the data it was learned from, instead of a simple whole-numbered 0 or 1 prediction.

By minimizing the loss function we are also performing what is called gradient descent, because we are descending down the calculated gradient with each step, thereby minimizing the estimated error. Gradient boosting as a stage wise learning procedure was initially presented by Friedman et al [8] and was refined by several researchers (for single-label rules in [6] and [9]). A refinement for the application on multi-label rules was developed in the Rapp et al 's "BOOMER" paper [28]. Some real examples for uses of multi-label classification include: image-recognition [3] or automatically associating keywords to news articles among many others. Gradient descent and boosting have become a highly popular topic within machine learning generally in the last years, as evidenced by the popularity of projects like LightGBM [20] or XGBoost [2],[26] which are themselves building on the legacy of early boosting algorithms like AdaBoost [7]. This increased interest is likely linked to the ever-increasing computational power, which allows us to solve more complex gradient equations more quickly and thus mathematically learn or extract information from larger sets of data.

Now that we have briefly outlined the algorithmic landscape of the stochastic gradient descent boosted rule learning approach we will get to the main contribution of this work.
Our goal is to develop and implement an extension to the previously mentioned "BOOMER" algorithm [28]. The original algorithm deals with the learning process as a purely sequential procedure, rules are induced and immediately finalized (no rule-review takes place). This approach appears to offer potential for further improvement: it might be possible that after a number of rules have been learned, that certain parts of, or entire rules have essentially become obsolete or are no longer of a high quality when considering the additional rules learned since its induction. To build on this idea we will develop an algorithmic strategy to re-evaluate an already learned model of rules, and try to modify it within this new context to possibly achieve better performance metrics and improve the model. Similar ideas have successfully been explored for other machine learning approaches like SGDTrees [14] and specifically some simpler rule learning approaches like in [4] and [12]. The idea is to establish an intermediate solution between the simple "SeCo" and the more complex "boosted" approach, which then requires a smaller number of rules than standard boosting while still being more precise than a comparable "SeCo" solution.

1.2 Thesis Structure

We want to briefly outline the scope our the following chapters and highlight their main goals.

- **Chapter 2 "Fundamentals"**:
  In this chapter we want to survey the landscape of machine learning and our selected approach specifically and thereby provide the reader with a general understanding of the current state within this research domain. This will allow the reader to more easily follow the later parts of this thesis. We will break down the individual components of our approach, specifically: Rule Learning, Multi-Label Classification, Stochastic Gradient Descent, Boosting, Loss functions, evaluation metrics while also providing appropriate additional context to related papers on the subjects to enable further reading and research into these topics.
- **Chapter 3 "Approach"**:
  After having established proper context and fundamental knowledge for this thesis, we will proceed to discuss our approach to the problem in more details, while supplying additional illustration, examples and explanations to the reader. We will not extensively revisit parts of the already existing implementation as that is already introduced as part of the "BOOMER" paper, but I will instead focus on the

extensions that were made and how they fit into the existing algorithm. This should allow the reader to fully understand how and why the algorithm was built like it is.

- **Chapter 4 ”Results”**:
  The algorithm which has been presented as part of Chapter 3 ”Approach” will then be tested in this chapter in order to collect data about its real-world performance. In an effort to evaluate whether the selected extensions are beneficial, by also providing a before and after comparison in the form of numerous plots and tables with additional test-results.
- **Chapter 5 ”Conclusion”**:
  Finally, we will conclude the paper by summarizing what we developed as part of this work and also what we learned from the testing of our implementation. We will also present some possible interpretations of these results and provide additional suggestions for future work on the subject.

2 Fundamentals

In this chapter we will go over some fundamental terminology and functions of the algorithm we are working with here. The Boomer algorithm we are working with was introduced during the work of Rapp et al [28] on a Stochastic Gradient Descent Boosting Learning Algorithm, specifically for multi-label rules. Machine learning is a quickly growing field of computer-science that tries to deal with the increasingly vast amounts of data we produce and need to process in a strategic fashion. The process of machine learning can either be done supervised or unsupervised, even a combination of the two is possible. We will focus on supervised learning in this thesis. This refers to the process of extracting knowledge from already correctly labeled ”training-data” (containing a set of labeled attribute-value pairs known as examples). A ”ground-truth” about the data is available for our algorithm to process and it then extracts information to be able to classify other unseen instances (attribute-value pairs without a label) later on. These examples used for testing are known as ”testing-data”.

2.1 Rule Learning

First, we want to narrow down the term rule learning here and explain the idea of how it works. Fürnkranz et al offer a more in-depth dive into rule learning than we will offer here, in the publication ”Foundations of Rule Learning” [11]). According to Fürnkranz et al an instance is usually describing an example without an available ground-truth label, essentially an unclassified/unlabeled example. However, sometimes examples and instances are used interchangeably since both describe attribute-value pairs. Rule Learning describes the task of using information extracted from data to create what can essentially be seen as a number of if-statements. These rules are designed to gradually classify parts or certain attributes of the data by assigning a classification or label prediction to a certain combination of attributes, fitting certain criteria, that was observed in the training-data. A number of these rules will then form a model of rules. This model since it usually consists of several rules combines all these rules into one easily interpreted representation. This is often done in the form of a matrix, which offers the ability to easily and quickly perform computations on the entire model and abstracts individual rules into one accumulated object. As mentioned Fürnkranz et al [11] describe the conceptual challenges of rule learning in a lot more detail, we will stick to only the essential points here.
Every rule consists of both a head and a body. The head refers to the ”then” part of the if-statement if following this earlier analogy. (if THIS then THAT). The prediction contained in the head is a combination of one or more labels and their associated confidence score (often just a binary 1/yes or 0/no). Essentially, this describes how confident our model is that an instance fits the predicted label. These scores can be both negative and positive. This way we can both express positive correlations and negative correlations observed within the data. The body of a rule refers to the conditions the rule is made of, these are used to narrow down our set of instances in order to associate them with the rule’s predicted label and follow a simple structure: attribute, comparison operator and a target value. (e.g. [height $<$ 25]).

Another important and fundamental term as far as rule learning basics go is ”coverage”. A rule’s coverage describes the instances which fit the criteria described by a rule’s conditions. This means an example is considered ”covered” by a rule if and only if this example satisfies every one of the rule’s conditions. Furthermore, the more conditions a rule has the more ”specific” it becomes, because its coverage is decreasing. Similarly, the less conditions a rule has the more general it becomes, because its coverage is increasing due to the less stringent criteria. If rules become ”too specific” during the learning process we will quickly run into what is known among data-scientists as ”overfitting”, this describes the circumstance where a model or rule describes the training data with such a high degree of precision that it no longer provides any useful generalized knowledge, an overfitted model basically looks only for copies of what it already saw during training. Since this is obviously not very useful while trying to extract knowledge and structure from training-data the goal is to always try and avoid overfitting. Ideally a model should be the most general it can be, while still describing a majority of the data correctly. Applying this idea results in a delicate balancing act between over- and underfitting (underfitting being the opposite of overfitting), we need enough specificity to be able to actually classify, but not too much specificity as to just copy our training data. Furthermore, the featured ”BOOMER” algorithm [28] makes use of inductive rule learning, specifically following top-down search as mentioned earlier, its search starts with the most general rule and iteratively refined the rule. During the search the rule continues to become more specific.

2.1.1 Rule Learning Strategies and Ensemble Methods

There are multiple methods of going about building a new rule model, the ones we will feature in this thesis are: separate-and-conquer [27] and especially rule-based gradient boosting [28]. Separate-and-conquer describes the simple idea of learning a rule to describe a subset of the training-data and removing the now covered examples from the pool of available training data for subsequent rules. This way we will end up ”separating” and ”conquering” the available data over time. This approach is also presented in [27] which is part of the same research project as the ”BOOMER” algorithm. It has also been explored in earlier work by Janssen et al who developed a SeCo framework for rule learning in [19]. Gradient Boosting is a modern refinement of the general idea of boosted-learning, which dates back to early algorithms such as AdaBoost [7].
Ensemble strategies describe the concept of applying the same algorithm to several classifiers and combine the results in some way to improve the predictive quality. Boosting as a general ensemble strategy refers to the concept of combining several weak classifiers to form a single combined strong classifier, each weak classifier essentially ”boosts” the next classifier. To be more specific: the learners are learned sequentially and example weights change with each iteration, taking into account the previous results. These ensemble methods are used to achieve better results with otherwise simple algorithms instead of developing huge monolithic algorithms, which quickly become hard to manage. The final boosted model-prediction is determined using the weighted average of all its classifiers. Some more recent and popular implementations of Gradient Boosting specific approaches are XGBoost [2],[26] or LightGBM [20]. Besides Boosting ensembles there is also the idea of bagging an ensemble, which means that a sort of majority voting policy is used to determine the final prediction and all classifier are trained in parallel. Essentially an ensemble of classifiers votes and the most classified result will be selected as the result of the overall ensemble. Examples for and issues of bagged ensembles are also discussed in [13].
Finally, stacking is another ensemble strategy and essentially learns a hierarchy of learners using the output of previous learners for the next level of learners, this approach is similar to how neural networks work. Some of the benefits of stacked classifier ensembles are discussed in [30]

2.2 Multi-Label Classification

Multi-Label classification is a sub-area of traditional classification tasks which only predict single labels. There are many possible use cases like image-recognition or automatic text labelling as featured in [3]. In the case of rule learners multi-label models can be achieved either by learning a set of single-label rules which when accumulated form a combined model that is capable of performing multi-label classification. Or by learning multi-label rule heads, this means that each rule can predict for several different classes at once (e.g. "=> (house = 0.4 ; car = -0.6; dog = 0.2)") Both these ways of constructing a multi-label rule model perform decently well, while the "full-head", the one predicting for several classes, is generally more computationally intensive but can represent more detail. This discussion of Single vs Multi-Label classification is also featured in [27], [28], [22].

2.3 Sampling and Stochastic Gradient Descent

One issue with machine learning can be the incredibly huge amounts of data, sometimes even computers can not process all the data quickly enough. This is especially true while performing the gradient computations required when doing gradient descent, where we need to solve a number of gradient computations during each step. A solution that established itself is dealing with this explosion in compute time by sampling the data, using only smaller subsets of the data. Sampling describes the process of selecting a subset (usually randomly) from a bigger set and learn a model using this subset. If sampling used for a gradient descent operation is essentially chance based we are performing what is known as Stochastic Gradient Descent. Many researchers were able to show over the years that Stochastic Gradient Descent can achieve almost identical performance metrics as a full gradient descent does, while usually generalizing better and thereby underscored its usefulness beyond just speeding up the learning process. [15] However, an important requirement is that we need to ensure the sampling is as unbiased and as representative of the overall data as possible. Sampling-bias can and often will end up ruining a otherwise decent Stochastic Gradient Descent (SGD) solution. Therefore, it is very important to make sure the sampling is as unbiased as possible when employing SGD. Possible solutions like importance sampling for SGD are discussed in [25].

2.4 Loss Functions

A key element required in order to perform gradient descent like "BOOMER" [28] does is a loss-function, a loss-function serves as a heuristic function to guide us towards an optimal solution. In gradient descent we calculate the gradients of that loss-function with respect to our current model state. Using these calculation we can then determine which changes (which rules in our case) are necessary or better and will lead us towards a superior model (descent the gradient). This gradient descent calculation can get overwhelmingly big very fast which is why gradient descent is rarely performed without any sampling today. Sampling as explained in Section 2.3 enables the use of more efficient and generalize-able stochastic gradient descent (SGD). Some examples for loss functions include: F1-Loss, Hinge-Loss, Mean-Squared-Error-Loss, Cross-Entropy-Loss, Hamming-Loss or 0/1-Subset-Loss. Wang et al [32] recently conducted a survey and comparison on 31 popular loss functions in machine learning. The last two mentioned are of particular importance to this thesis and multi-label classification in general. They will be a central loss functions used in this work. I want to briefly explain these particular loss-functions here (analogous to [28]).

Hamming-Loss describes the fraction of incorrect labels among all labels. The mathematical definition is discussed as part of Section 2.5.2 on model evaluation metrics.

Subset-0/1-Loss goes further and describes the fraction of instances for which one or more labels are not predicted correct. In short it describes the amount of "perfect" predictions among all predictions. The mathematical definition is also discussed as part of the evaluation metrics 2.5.3.

2.4.1 Surrogate-Losses

In this thesis we will not directly be using Hamming-Loss and subset-0/1-loss themselves, but instead approximate them using a surrogate loss-function which very closely approximate the original losses (as can be seen in [5]). We do this because our model yields numerical confidence scores, instead of traditional 0 or 1 values in the label vectors. The logistic loss is a continuous loss function, which can be easily minimized for our application. (as explained in [28]). The specific functions used are **label-wise-logistic-loss** and **example-wise-logistic-loss** as depicted in 2.4.1(adopted from [28]). These surrogate losses are commonly used for boosting approaches, especially for single-label classification (as seen in [8]).

$$loss_{label-wise-log}(y_n, \hat{p}_n) := \sum_{k=1}^{K} log(1 + exp(-y_{nk}\hat{p}_n k)) \tag{2.1}$$

$$loss_{example-wise-log}(y_n, \hat{p}_n) := log(1 + \sum_{k=1}^{K} exp(-y_{nk}\hat{p}_n k)) \tag{2.2}$$

2.5 Evaluation Metrics

In this thesis we will mainly use Example-wise F1, Precision and Recall, as well as Hamming Accuracy and Subset-0/1-Loss to evaluate our model's performance. We are also able to evaluate Macro and Micro averaged metrics of these, but those did not provide enough additional value in our experiments to be included in most plots. They might be necessary for further testing, which is why we will briefly cover them as well.

2.5.1 Common Metrics

Some of the most common metrics used to evaluate and test machine learning applications are **Precision**, **Recall** and **F-measure** also known as F1 or F1-Loss. These are most commonly used to evaluate models due to the fact that they allow to easily draw conclusions about the model's overall quality. In some cases these metrics or functions are also referred to as "heuristics" as was done in [17]. They allow us to form statistical conclusions about the model's predictive qualities. These metrics use the absolute amounts of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) to calculate their scores (P # all positives; N # all negatives).
Specifically, the metrics use these values, which have been determined by the model's classifications results on the test data, and calculate a score $\in [0, 1]$ from them. The higher the calculated score for these three, the better the model's performance in regard to that metric.

- Precision is a very well established metric for machine learning in all kinds of classification tasks ranging from simple binary to multi label classification, some related examples featuring its use in multi-label classification include [28],[27], [22]. Its use for binary and multi-label classification is also discussed in [17], [18] and [23]. It describes the amount of true positives among all positive instances (as seen in [11],[27]) and it is defined as:

$$Precision := \frac{TP}{(TP + FP)} = \frac{TP}{P} \tag{2.3}$$

- The Recall as a metric tries to reflect and highlight the amount of relevant labels which were predicted correctly among the total (as described in [11]) and therefore allows us to easily draw conclusions about whether a model is too specific and does not cover enough relevant examples anymore, it is defined as [22]:

$$Recall := \frac{TP}{(TP + FN)} \tag{2.4}$$

- f-measure or f1-Loss essentially reflects a combination of precision and recall and is usually used to determine a healthy balance or trade off between the two. It resembles the harmonic mean of the two values. The specific term f1-measure/loss refers to the regular f-measure with the β parameter set to 1, in order to weigh precision and recall equally and facilitate a balance. The generic and commonly used term for the f-measure is as follows (as discussed in [11] and [29])

$$f_1 = f_{\beta=1} := \frac{(1 + \beta^2) \times Precision \times Recall}{\beta^2 \times Precision + Recall} \tag{2.5}$$

2.5.2 Hamming Accuracy

The hamming loss computes the percentage of incorrectly classified labels. Sometimes Hamming Accuracy is also known simply as "Accuracy" (as per [18]). Therefore, hamming loss is the error associated with the hamming accuracy or:
$Hamming - Accuracy := 1 - Hamming - Loss$
Hamming accuracy thereby computes the percentage of correctly classified relevant (TP) and irrelevant labels (TN) and is defined as follows [27]:

$$Hamming - Accuracy := \frac{(TP + TN)}{(TP + FP + TN + FN)} = \frac{TP + TN}{P + N} \tag{2.6}$$

It is also of note that hamming loss is the gain metric associated with hamming accuracy as per [27] and [21]. Unlike other gain-metrics, loss based metrics however will need to be minimized not maximized to optimize classifier performance. Due to the fact that hamming accuracy and loss are so closely connected and inverted values, hamming accuracy can be calculated in place of the loss to more closely conform with other maximizing gain-metrics. The loss can later be calculated based on the corresponding accuracy. Hamming loss is commonly defined as [27]:

$$Hamming - Loss := \frac{(FP + FN)}{(TP + FP + TN + FN)} = \frac{(FP + FN)}{(P + N)} \tag{2.7}$$

2.5.3 Subset 0/1

A more complex to calculate evaluation metric is the subset 0/1 accuracy, it is based on example-based averaging [27]. By comparing the predicted labels to the true labels of two given vectors V_J and $\hat{V}_J$ for each individual example X_j. As mentioned in [22], [24] and [33] the metric evaluates to 1 if all predicted labels match all true labels, if any one of the labels of the given vectors mismatch the evaluation of the metric is 0. To compute the 0/1 loss on the entire data-set we now need to calculate the mean of all individual examples m, which results in the subset 0/1 accuracy reflecting the amount of perfect predictions among all predictions made as noted by [27], [22], [33]. The subset 0/1 loss can be seen as the inverse accuracy and is calculated as follows:

$$Subset - Accuracy := \frac{1}{m} \times \sum_{i=1}^{m} Y_j = \hat{Y}_j, \begin{cases} 1, & \textit{if all labels are equal;} \\ 0, & \textit{if any one label differs;} \end{cases} \tag{2.8}$$

2.5.4 Evaluating Multi-Label Predictions

Another important area when evaluating a multi label model is the aggregation and averaging of confusion matrices computed using bipartition evaluation metrics, since we cannot simply apply the same methods used for single label data on multi-label classification. (Since we end up with matrices, which need to be transformed, not values) [22], [23]. As presented in [27] the aggregating and averaging of confusion matrices, specifically those calculated from m instances and n labels using bipartition metrics describes the process of breaking down these matrices into a single score which can then be easily compared. Bipartition evaluation metrics are usually considered to be functions on two-dimensional confusion matrices. [21], [24]. These matrices consist of the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) [21] and describe the behavior of the evaluated prediction. As mentioned in [22] equation 2.9 shows the calculation of these values, where as variables y_i and $\hat{y}_i$ refer to true and predicted label vectors, while 1 and 0 describe whether or not the given label is present in the respective one (adapted from [27]).

$$TP_i^j = \begin{cases} 1, & \textit{if } y_i = 1 \textit{ and } \hat{y}_i = 1; \\ 0, & otherwise \end{cases} \tag{2.9}$$

$$FP_i^j = \begin{cases} 1, & \textit{if } y_i = 0 \textit{ and } \hat{y}_i = 1; \\ 0, & otherwise \end{cases} \tag{2.10}$$

$$TN_i^j = \begin{cases} 1, & \textit{if } y_i = 0 \textit{ and } \hat{y}_i = 0; \\ 0, & otherwise \end{cases} \tag{2.11}$$

$$FN_i^j = \begin{cases} 1, & \textit{if } y_i = 1 \textit{ and } \hat{y}_i = 0; \\ 0, & otherwise \end{cases} \tag{2.12}$$

2.6 The BOOMER Algorithm

To conclude this chapter we want to outline the specific BOOMER algorithm [28] as described by its authors in more detail. The BOOMER algorithm [1] is based on their refined gradient boosting framework, which we previously described and which is part of the same publication. The algorithm describes a stagewise process for learning a gradient boosted [28] ensemble of rules, both single and multi-label rules are possible. This ensemble is minimizing a given loss-function. The pseudocode cited in Algorithm 1 depicts their general idea of how the algorithm works. A more detailed explanation can be found in their original paper [28].

[1] Their implementation of it is available at https://www.github.com/mrapp-ke/Boomer

Algorithm 1 Learning an ensemble of boosted classification rules (as depicted in [28] Chapt. 4)

Input : Training examples $D = \{(x_n, y_n)\}_n^N$, first and second derivative l' and l'' of the loss function, number of rules T, shrinkage parameter η

Output : an ensemble of rules F

$G = \{g_n\}_n^N$, $H = \{H_n\}_n^N$ = *calculate gradients and Hessians w.r.t.* l' *and* l''

$f_1 \rightarrow \hat{p}_1$ *with* $b_1(x)$, $\forall$x *and* $\hat{p}_1 = FIND\_HEAD(D, G, H, b_1)$ ▷ *see [28] Section 4.1*

for $t = 2$ **to** T **do**

  G, H = *update gradients and Hessians of examples covered by* f_{t-1}

  $D^{'}$ *randomly draw* N *examples from* D *(with replacement)*

  $f_t : b_t \rightarrow \hat{p}_t$ = *REFINE_RULE(*$D^{'}, G, H$*)* ▷ *see Algorithm 2 adapted from [28]*

  $\hat{p}_t$ = *FIND*$_H EAD$*(D,G,H,*b_t*)*

  $\hat{p}_t = \eta * \hat{p}_t$

return *ensemble of rules* $F = \{f_1, ..., f_T\}$

Initially when creating their rule model, the empty model does not cover anything so predictions about new instances cannot be made. The solution is to start with a default rule, which covers every example and provides a loss-minimizing prediction for the current dataset. (the exact calculations done can be seen in [28] Section 4.1) The next step is to now start inducing increasingly more specific rules, in order to shape our model and start fitting it closer to the data. Each rule does contribute to the final model as discussed earlier, with respect to their calculated confidence scores. During each iteration of rule induction the gradients and Hessians need to be recalculated according to the current model-state (confidence-scores and label predictions) and the true labels. As previously mentioned the algorithm can produce single-label or full-label heads for new rules. The learning of rules after a default rule is induced follows a greedy induction scheme. A first condition is chosen based on the best evaluated quality and iteratively more conditions are added, refining the rule until one of the abort criteria is met. (e.g. no more viable candidate-conditions, condition-limit reached) With each update to the rule body, the rule head also needs to be recalculated because the coverage could have changed. Some additional measures they take with the algorithm include: the sampling (random with replacement) of training-data for each new rule learned in order to produce a more diverse result. As a final measure the algorithm recalculated the rule's prediction, this time taking into account the entire training data, in an effort to avoid overfitting to individual sub-samples. Additionally the shrinkage parameter $\eta \in (0, 1]$, as the name implies, shrinks the predicted scores via multiplication. The shrinkage parameter acts in a similar way as the learning rate does for other machine learning approaches and reduces the effect an individual rule has. [16]

2.6.1 BOOMER and the Refinement of Rules

As the authors [28] state, BOOMER employs a top-down greedy search. This greedy search pattern is common in inductive rule learning. More detailed information about this approach is provided in [11].
The general process of refining a rule is illustrated in Algorithm 2. Without going into too much detail, the process starts with an empty body, successively adding new conditions. The refining conditions either describe the results covered in the nominal case or the averaging of adjacent values in the numeric case. By default the algorithm will also employ random-feature-selection, as introduced for random forests, this ensures a certain diversity of ensemble members. After a new condition is then added, an updated head to determined, which in the case of single-label rules cannot change predicted labels. The selected refining condition will always be the one, which minimized the regularized training objective as seen in 2.6.1, the

equation is introduced in [28] as the main optimization target. It describes the calculation of the element-wise sum of the gradient vectors (g), the sum of the Hessian matrices (H) and a L_2 regularization term which is used to avoid extreme predictions. [28] By default the main stopping criteria is reached if no viable candidate conditions are available.

$$\tilde{R}(f_t) = g\hat{p} + \frac{1}{2}\hat{p}H\hat{p} + \Omega(f_t) \tag{2.13}$$

Algorithm 2 Refine_Rule (as depicted in [28] Sect. 4.2)

Input : Training examples $D = \{(x_n, y_n)\}_n^N$, first and second derivative l' and l'' of the loss function, number of rules T, shrinkage parameter η
Output : an ensemble of rules F

$f^* = f$
$A^{'}$ = *randomly select* $log_2(L-1)+1$ *out of L attributes from D*possible condition c on attributes $A^{'}$ and examples D
$f^{'} : b^{'} \rightarrow \hat{p}^{'}$ = *copy of current rule f*
add condition c to body $b^{'}$
$\hat{p}^{'}$ = *FIND_HEAD*$(D, G, H, b^{'})$ ▷ *see [28], Section 4.1*
if $\tilde{R}(f^{'}) < \tilde{R}(f^*)$ w.r.t G and H **then**
    $f^* = f^{'}$
if $f^* \neq f$ **then**
    $D^{'}$ = subset of D covered by f^*
    return *REFINE_RULE*$(D^{'}, G, H, f^*)$
return *best rule* f^*

3 Approach

In this chapter will talk in-depth about our approach and what changes were made to the algorithm and for which purpose, specifically which algorithmic extensions we introduced. As well as illustrate the thought process behind introducing them.

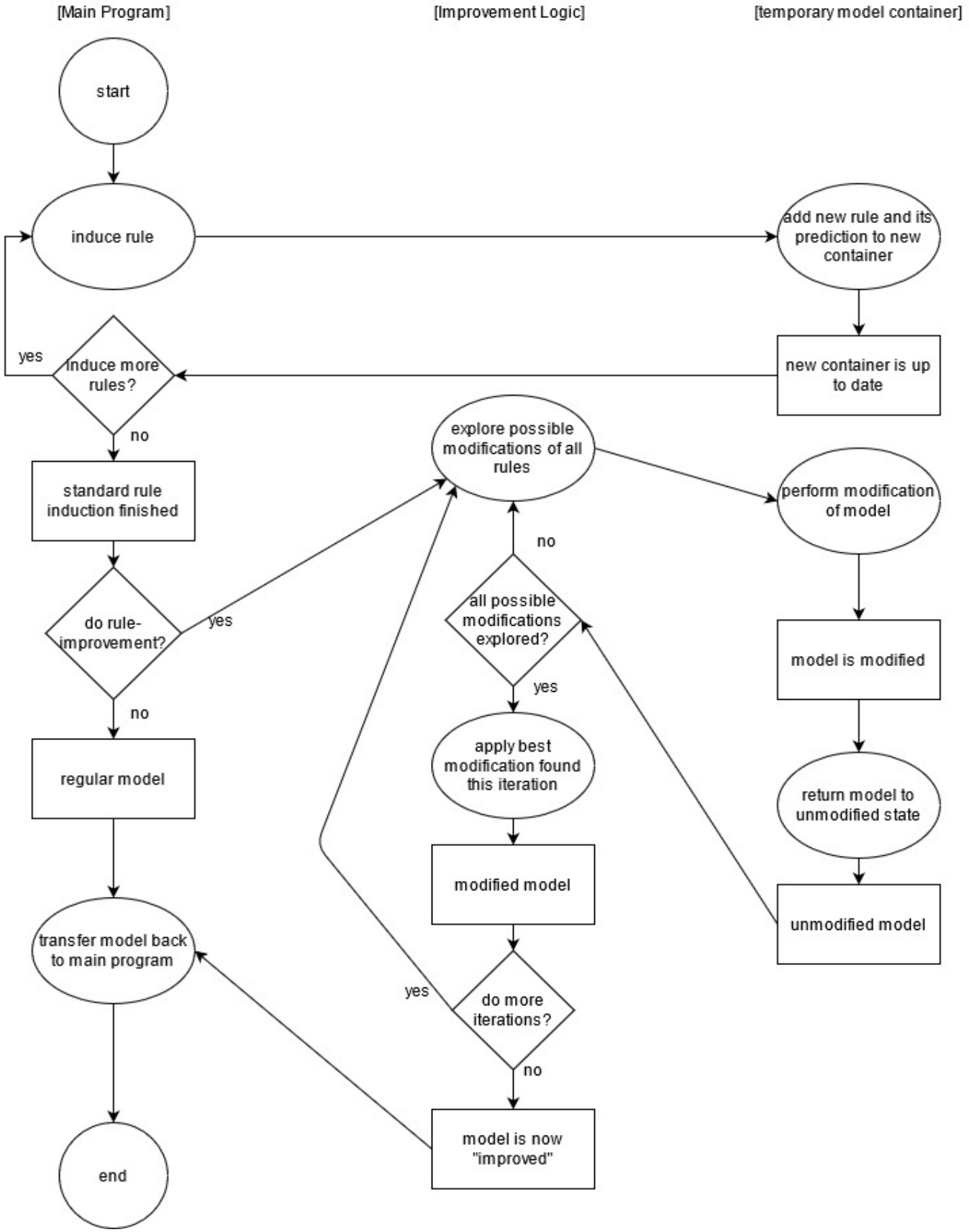
3.1 Accessing Learned Rules

Our first step in enabling the modification of previously induced rules is to make sure we actually have access to all rules that were already learned. We will ignore the default rule here as we do not intend to modify it. For this purpose we introduce a new container, which stores all our rules, the order they were last modified in and their accumulated predictions to enable easy access to all of the rules currently contained in our current model, as well as providing a place to store improvement-specific information. Fig 3.1 depicts the planned program flow used to plan and implement the program extension, following the main-program path strictly from start to end essentially represents the functionality prior to our additions. Once the improvement operations are finished the program resumes with its original implementation. The addition of a new container was necessary because the original representation of the model was not made to be modified after rule induction of a rule was done in the original work [28]. For efficiency purposes the original code uses a storage efficient representation of the model, which does not keep track of enough information to be able to freely edit rules afterwards. The current model's rules are stored ordered inside our container as their individual conditions and their accumulated effect on the model prediction. The so-called "prediction-matrix" stores the sum of the predictions of all rules the model currently contains, per example (y) and per label (x), which is relevant for the processing of our model modifications. This prediction matrix is accessed by the algorithm via a so-called "thresholds" object, which essentially represents this matrix and allows to perform actions on it like filtering, updating and similar operations.

3.2 Introducing Basic Operations for Rule Modification

In this section we will go over all the possible basic operations that could be used to help improve our model and were selected as part of developing our rule-improvement process. These basic operations are essentially tools, which more complex operations can use and be built upon, these themselves are not enough to facilitate meaningful changes. Later, we will go into detail about how the algorithm ends up choosing an improvement and how often it does so.

Figure 3.1: A diagram to visualize the planned program flow



3.2.1 Removing Rules

Now that we can easily access all our current rules in an efficient way, after the initial learning process is completed, we need operations to perform to facilitate our desired improvement. The first and likely most simple operation we could apply to change the model is removing an entire rule, by doing this we not only remove the rule representation with all its conditions (see Algorithm 3 line 7+8), but also remove the rule-specific prediction, meaning the rule-head from our accumulated prediction matrix of the model, which contains all predictions of all rules. To do this we will take the current representation of our model-data called thresholds, which contains the examples we selected from our data for the current iteration and is filterable by conditions. Then we will proceed to iteratively apply all the conditions of our current rule (see Algorithm line 3), to filter our subset of examples and select only the ones actually influencing the prediction of our current rule. Now that we have determined which entries of our model's prediction-matrix need to be modified we can use this information to inverse our current rule's prediction (see Algorithm 3 line 6) and add the inverted prediction back to the previously filtered matrix positions (see Algorithm 3 line 7). This will have the effect of negating the influence the rule had on the prediction of our model, in essence completing the removal process.

Algorithm 3 remove rule

```
1: procedure REMOVERULE(RULE)
2:     iter ← find(rule)
3:     thresholdSubset ← getSubset()
4:     for all conditions of rule do
5:         thresholdSubset.filterThresholds(condition)
6:     localInverse ← rule.head.getInvertedCopy()          ▷ Invert the prediction
7:     thresholdSubset.applyPrediction(localInverse)       ▷ Add inverted prediction
8:     tempRules.erase(iter)
```

3.2.2 Adding Rules

It should also be possible to add new rules to the model in order to replace the ones we previously removed. For this reason we need another new operation to add rules. Similarly to when we remove rules, we will need to add both the rule representation in the form of its conditions (see line 7) as well as its effects on our model's prediction matrix. More specifically, just like when we remove a rule we will once again have to filter our current subset of examples to determine which ones are covered by the current rule within our current model-sampling. (see line 4) This is done in order to determine the relevant positions in our prediction matrix which will need to be updated. Since our new rule is not currently part of our model, its prediction is also not yet part of the prediction matrix. By adding the calculated prediction of our new rule to the prediction matrix at the positions we determined by filtering for its conditions. And adding the conditions representing the rule to our temporary rule container, we now have added the new rule to our model successfully.

Algorithm 4 add rule

```
1: procedure addRule(rule)
2:     thresholdSubset ← getSubset()
3:     for all conditions of rule do
4:         thresholdSubset.filterThresholds(condition)
5:     thresholdSubset.applyPrediction(rule.head)
6:     tempRules.emplaceBack(rule.body)          ▷ Add rule to model container
```

3.3 Introducing more Complex Operations

3.3.1 Re-Learning Rules

Another rather simple operation that could potentially be useful in improving our model quality is to relearn a rule, basically removing an existing rule and learning another new rule to replace it on our current instance-sampling. These operations are particularly inspired by [4] and [12] where they previously used a pruning approach in combination with additional steps to already achieve decent improvements for their model. Since we technically use several basic operations during this step, (namely: 1. remove rule 2. induce new rule 3. add new rule) it should be considered our first ”complex operation”. The induction of the replacement rule is using the regular rule induction process, as already established by the BOOMER algorithm [28], for the rule-induction it does not use a new instance-sampling, but reuses the one that is provided for the current improvement iteration. This is necessary to ensure that all our modification operations can be compared to each other fairly. If we would just sample new instances for the relearning process the result would be that the relearning step could not be fairly evaluated or compared to other steps we would want to take, since it will possibly be biased towards another instance-sampling.

Algorithm 5 relearn rule process during improvement

```
1: bestRule ←The best rule found so far
2: bestQuality ←The evaluated quality of bestRule
3: bestPos ←The best current position
4: removeRule(rule)
5: backupRule ←rule.getCopy()
6: success, rule ←ruleInduction.induceRule()
7: if success then
8:     addRule(rule)
9:     quality ←getRuleQuality(rule)
10:    if quality < bestQuality AND quality < 0 then
11:        bestQuality ←quality
12:        bestRule ←rule
13:        bestPos ←the current Pos
14: removeRule(rule)
15: addRule(backupRule)                          ▷ The original rule is restored
```

3.3.2 Removing conditions

Now that we can add and remove new rules, we can also introduce slightly more complex operations that combine the adding or removing of rules with additional changes. Removing a condition for instance entails removing our original rule, changing its conditions and updating its prediction, then adding it back into our model. In essence we are combining two basic operations with a third modification operation: (1. remove rule 2. modify rule 3. add new rule). The pseudo-code algorithm depicted in 6 shows the functionality of removing a condition in the context of our improvement process. First we remove the original rule from our model like described earlier (see Algorithm 6 line 5). Since the new condition(s) of the modified rule can now possibly cover a broader set of examples we also need to reevaluate and recalculate the rules prediction while we are modifying it to make sure that the rule still reflects the data accurately. This recalculation is done by first filtering our thresholds (see Algorithm 6 line 10), thereby removing all non-relevant entries of our prediction matrix from consideration. Since we are removing a conditions the we will now likely consider more entries of the prediction matrix than we did while removing the original rule, this is because the rule has now become less specific. Using these filtered matrix positions we then recalculate the effect of our remaining conditions on the model to form a new, slightly different prediction for the entire modified rule (see line 11). Both, these new conditions and the new prediction are then added back into the temporary rule container and the model's current prediction matrix respectively and at the matrix positions we determined by filtering for our conditions (see Algorithm 6 line 12). For the sake of simplicity, the current implementation removes conditions from back to front, this approach is common and has been established by prior work on pruning rule models, [12],[10] or [4] come to mind. This enables us to more easily control what the algorithm does, while also dramatically reducing the amount of permutations of conditions which need to be considered. This also helps speed up the process and does not appear to noticeably impact the results. Furthermore, preserving the order in which conditions were learned proves useful for certain rule pruning approaches.

Algorithm 6 remove conditions process during improvement

```
1:  bestRule ←The best rule found so far
2:  bestQuality ←The evaluated quality of bestRule
3:  bestPos ←The best current position
4:  backupRule ←rule.getCopy()
5:  removeRule(rule)
6:  while rule has conditions > MIN_COND do
7:      rule.body.removeLastCondition()
8:      thresholdSubset ← getSubset()
9:      for all conditions of rule do
10:         thresholdSubset.filterThresholds(condition)
11:     rule.head ←recalculatePrediction(thresholdSubset)
12:     addRule(rule)
13:     quality ←getRuleQuality(rule)                    ▷ Evaluate the modification
14:     if quality < bestQuality AND quality < 0 then
15:         bestQuality ←quality
16:         bestRule ←rule
17:         bestPos ←the current Pos
18: removeRule(rule)
19: addRule(backupRule)                                  ▷ original rule is restored
```

3.3.3 Adding Conditions

Our probably most complex operation of adding additional conditions to an existing rule is split into two smaller operations, one operation we call ”induceCondition” responsible to determine how the new condition is supposed to look like and the other ”addConditionToRule” which is solely responsible to actually add the new, modified rule to the model, very similarly to how removing conditions works. Adding a condition also consists of two basic operations and a third operation to modify our rule. (1. remove rule 2. modify rule 3. add new rule) the main difference to the previous operation of removing a condition is that instead of generalizing our rule we are now creating a more specific rule. After we removed the original rule from our model (see Algorithm 7 line 5), we start modifying it. We achieve this by first inducing a new condition (see Algorithm 7 line 7), after we have selected our new condition we can then proceed and once again add it to the current rule conditions (see Algorithm 7 line 8) and begin filtering our thresholds (representation of our instance-sampling) to select only the matrix positions of our prediction matrix that are affected by these new conditions (see Algorithm 7 line 11). Finally, just like we did previously we recalculate the rule’s prediction (see Algorithm 7 line 12), because it is now more specific, its previous prediction is very likely to be slightly inaccurate. After we determined a new prediction based on the relevant examples, we then proceed to add our new prediction to our model’s prediction-matrix at the appropriate positions (see Algorithm 7 line 13) , which we already determined by filtering for the conditions.

During implementation we noticed that it might be useful to be able to add or remove conditions without calculating a whole new prediction, especially if the instance-sampling of the initial learning process is no longer available, but we do still have access to the prediction that was calculated on said sampling. To remedy this it can be useful to provide both ways of adding a condition to a rule: the option to add rules with a pre-calculated prediction as well as just adding another condition with a new recalculated prediction based on the sampling which is currently being used. The process remains the same with the main difference being that we do not recalculate a prediction but just pass it along.

Inducing new Conditions:
As the second part of adding new conditions we first need to determine how the new condition is supposed to look like, for this process we adapted a slightly modified version of how the Boomer algorithm [28] learns rules in general, we essentially reproduce the state of a rule induction that is in progress and execute a single iteration which induces a condition. We achieve this by passing our current rule, which is then used to filter our threshold data down to only the relevant examples and their corresponding prediction-matrix positions, using the rule’s conditions. These are covered by the already existing conditions, those of our ”original rule”. Now we go about finding a single new condition in the same way the initial algorithm would induce a single condition of a newly learned rule, by essentially exploring candidate conditions, evaluating their effects on the model and choosing the one condition that appears to be the most useful as our final refinement condition. This newly induced condition is then added to our model using the procedure previously established during Subsection 3.3.3, where its new prediction based on the current instance-sampling is then calculated.

3.3.4 A Real Example of a Modification

In this subsection we want to present a real example of a rule modification and its effect using the prediction matrix of a very simple model. We are starting with this very simple learned model seen in table 3.1:

Algorithm 7 add conditions process during improvement

```
1: bestRule ←The best rule found so far
2: bestQuality ←The evaluated quality of bestRule
3: bestPos ←The best current position
4: backupRule ←rule
5: removeRule(rule)
6: while backupRule has conditions < MAX_COND do
7:     (success, cond) ←ruleInduction.induceCondition(rule)
8:     rule.body.addCondition(cond)
9:     thresholdSubset ← getSubset()
10:    for all conditions of rule do
11:        thresholdSubset.filterThresholds(condition)
12:    rule.head ←recalculatePrediction(thresholdSubset)   ▷ Recalculate a new prediction, due to coverage
13:    addRule(rule)
14:    quality ←getRuleQuality(rule)                       ▷ Evaluate the modification
15:    if quality < bestQuality AND quality < 0 then
16:        bestQuality ←quality
17:        bestRule ←rule
18:        bestPos ←the current Pos
19: removeRule(rule)
20: addRule(backupRule)                                    ▷ original rule is restored
```

Table 3.1: The initial rules

Rule number	Rule conditions	Rule prediction
0	{}	(play = 0.44, dontplay = -0.44, playmaybe = -0.67)
1	{temperature >70.5 & humidity >77.5 & outlook != 1.0}	(dontplay = 0.68)
2	{windy == 1.0}	(dontplay = -0.58)
3	{temperature >64.5}	(playmaybe = -0.36)
4	{humidity <= 95.5 & humidity >77.5 & outlook != 1.0}	(play = -0.47)

Our first improvement-iteration, after exploring all 4 rules (excluding the default rule) and their modifications, selects ”add condition” as the best step to take according to its evaluated rule quality, meaning the rule quality of this specific modification yielded the biggest model change in quality score. The result is the following set of rules seen in Table 3.2:

Table 3.2: The rules after a modification

Rule number	Rule conditions	Rule prediction
0	{}	(play = 0.44, dontplay = -0.44, playmaybe = -0.67)
1	{temperature >70.5 & humidity >77.5 & outlook != 1.0}	(dontplay = 0.68)
2	{windy == 1.0}	(dontplay = -0.58)
4	{humidity <= 95.5 & humidity >77.5 & outlook != 1.0}	(play = -0.47)
3	{temperature <= 84 & temperature >64.5}	(play = 0.44)

Table 3.3: The initial prediction matrix

Example	play	dontplay	playmaybe
0	-0.0300709	-0.347089	-1.02695
1	-0.0300709	-0.347089	-1.02695
2	-0.0300709	-0.347089	-1.02695
3	0.444444	-1.02901	-1.02695
4	-0.0300709	-1.02901	-1.02695
5	-0.0300709	-1.02901	-1.02695
6	-0.0300709	-1.02901	-1.02695
7	-0.0300709	-0.347089	-1.02695
8	-0.0300709	-1.02901	-1.02695
9	-0.0300709	-0.347089	-1.02695
10	-0.0300709	-0.347089	-1.02695
11	0.444444	-1.02901	-1.02695
12	0.444444	-1.02901	-1.02695
13	-0.0300709	-0.347089	-1.02695

Table 3.4: The modified prediction matrix

Example	play	dontplay	playmaybe
0	0.408844	-0.347089	-0.666667
1	0.408844	-0.347089	-0.666667
2	0.408844	-0.347089	-0.666667
3	0.88336	-1.02901	-0.666667
4	0.408844	-1.02901	-0.666667
5	0.408844	-1.02901	-0.666667
6	-0.0300709	-1.02901	-0.666667
7	0.408844	-0.347089	-0.666667
8	0.408844	-1.02901	-0.666667
9	0.408844	-0.347089	-0.666667
10	0.408844	-0.347089	-0.666667
11	0.88336	-1.02901	-0.666667
12	0.88336	-1.02901	-0.666667
13	0.408844	-0.347089	-0.666667

As a result of that modification our model's prediction matrix changed from its initial state (see Table 3.3.4) to its now refined state (see Table 3.4). As we can see, the prediction of the rule changed from "playmaybe" to "play" after adding the new condition. The algorithm determined that the initial rule covered all examples as we can see from taking a look at what the model is predicting for all examples and label "playmaybe" which was the initial label the rule predicted for (see Table 3.1), cells marked in orange are determined to be covered by rule 3). If we take a look at the modified matrix 3.4 (marked in green: unchanged; marked in orange: changed) we see that the rule's prediction, which changed labels, is removed from all examples it initially covered, and is added back to the new label "play" for the examples covered by the new rule.

Therefore, the new rule coverage is now determined to be covering all examples but example 6. This is why the old prediction (for the old label) was removed from example 6, but no new prediction is added to it (for the new predicted label). Rule 3, which was modified has become more specific and therefore covers less examples. The same logic applies to removing conditions just in reverse order, so a removed condition can cause a rule to cover more examples than before.

As we can see with this really simple example and dataset, the rule modification already causes a large portion of the table to get altered, this will not remain true with larger datasets, because more rules are required to properly describe the detail contained in the data. So any single rule will on average cover less and less of the dataset with increasing size of the dataset.

3.4 Exploring and Evaluating Modifications

Now that we have our fundamental operations available to actually begin and modify our model in the hopes of achieving better performance-metrics, we need to formulate a strategic approach of searching for and selecting the appropriate action to take in order to achieve these desired improvements. Our first step appears to be to iterate over each of the available rules. Then explore all the changes we could apply using our developed operations to said rule and evaluate these modifications in order to judge their usefulness. The existing algorithm already and necessarily provides the tools to evaluate the effect of a rule on the model, expressed essentially as a quality score. This score provides an indicator as to how much the evaluated rule will benefit the model. Previously rules could only be evaluated ”out-of-sample” meaning on examples not within the current sample, we added the possibility to also evaluate rules ”in-sample” meaning on the currently sampled training data. This is done in an effort to test whether the evaluation data has a significant impact on the quality of selected model improvements. After we explored all possible options and chose the modification resulting in the biggest desired model-quality change or the lowest rule-quality-score we now need to apply the improvement we deem to be the most useful and repeat the whole process with our now modified model. To provide a more detailed insight we will explain the two major loops within our improvement process.

3.4.1 Exploration Loop

The first and innermost loop of our rule-improvement process called the ”exploration loop” as already outlined iterates over all existing rules once. During one iteration we apply all of the developed modification options one after the other. Furthermore, after each step is processed a check is done to see whether the modified rule, according to its rule-quality, yields a model improvement over the original rule of the current rule-exploration-iteration. In any case the final action of each individual step is to reset the model back to the state it was in at the beginning of the current rule-exploration-iteration. This guarantees that all options are evaluated on an even playing field and are comparable to each-other. This process is also visualized in Fig 3.1. The three specific steps done are simplified in the diagram as one action.

1. During the first step called ”ADD_COND” we systematically call our ”induceCondition” routine to add new conditions to our rule, up until either:

- the program parameter for maximum amount of conditions is reached,
- no condition can be induced due to the minimum rule coverage dropping too low,
- no useful condition can be found,

- we already induced enough conditions during this iteration to satisfy our ”MAX_MODIFIED_COND” parameter which is responsible to limit the number of conditions we can modify per rule-exploration-iteration.

2. The second step ”REMOVE_COND” systematically removes the last condition of a rule until either:

- only MIN_CONDITIONS remain (1 by default, to avoid essentially changing default rule)
- we already removed enough conditions during this iteration to satisfy ”MAX_MODIFIED_COND

3. The final and last step ”RELEARNING”, will remove the current rule and induce a completely new rule also using the same initial greedy induction algorithm on the current instance-sampling. This replacement rule is then evaluated like before. A last model state reset concludes one such rule-exploration-iteration

3.4.2 Improvement Loop

Algorithm 8 the improvement-logic structure

```
1:  [...]
2:   [... initial model already induced ...]
3:  for NumImprovementIterations do                    ▷ The so-called improvement-loop
4:      [... initialize next iteration ...]
5:      for all rules of induced model do              ▷ The so-called exploration-loop
6:          [... explore rule modifications ...]
7:          [... restoreModelState ...]
8:      if found Improvement then
9:          removeRule(originalRule)
10:         addRule(bestRule)                          ▷ We replace the old with the modified, better rule
```

Once we found possible improvements and evaluated them we then need to apply the correct improvement to our current model using the improvement-loop.
The improvement-loop, our outermost loop (a very simplified version of which is depicted in algorithm 8), repeats as many times as was indicated with the corresponding parameter (rule-improvement-iterations, depicted as ”NumImprovementIterations” in Algorithm 8) and is responsible for actually applying the individual improvements once, every time we finish an improvement iteration. The actual content of the loop is essentially what has been presented in the earlier three ”complex operations” and is therefore not repeated in this pseudo-code. A visual representation (fig 3.1) of the loop can also be gathered from the initial program diagram on which the code structure is based on. Previously, during our exploration of all potential modifications in the exploration-loop we also collected data on which iteration yielded the current-best modification. After having verified that an improvement was actually found we then remove the rule corresponding to the exploration iteration that yielded this best improvement and replace it with the best modified rule we found. This means that during each improvement-iteration (one iteration of this loop) the model ”improves” or ”modifies” only one of its rules using one of the steps above according to the evaluation of the given loss function on the current model. This completes one full iteration of the rule-improvement process and should theoretically have improved our overall model-quality slightly. We would expect that if we were to limit our taken steps to only adding and removing conditions that a certain convergence of our metrics should arise.

4 Results

In this chapter we will present the results of our experiments in order to gain some insight into whether the modifications proved to be useful. Our evaluation of the changes that were made to the algorithm will perform some comparisons between different algorithmic parameters, as well as different datasets. Plotting these results should give us a general idea of what the algorithm is doing, and how the changes affect the model's predictions and therefore its predictive qualities. We will also try to figure out which parameters yield the most beneficial results. The evaluated metrics are: example-based F1, Precision and Recall, Hamming Accuracy/Hamming Loss as well as Subset 0/1 loss/accuracy. These appeared to be the most useful metrics to evaluate our experiments. Macro Averaged F1, Precision and Recall, Micro-averaged F1, Precision and Recall are also available, but did not carry enough meaningful, additional information for our experiments. However, we will rarely report all these metrics for every single tested case, since they tend to behave similarly and to illustrate certain effects its more useful to restrict the amount of data presented at once. Should we notice divergent behavior we will note this.

4.1 Testing Methodology

In this section we will outline our testing approach as well as present our results. Based on these results we will later try to develop conclusions in order to outline potential for further research. Our general expectation going into the testing and evaluation is that our algorithm should achieve a slight improvement of the target metric (with possible deterioration of others) or remain about level if no real improvements can be found. We will use the results gathered as a basis for additional experiments and to help guide the further direction of these experiments. We will also examine in more detail unexpected behavior in order to try and figure out possible causes.

4.1.1 Datasets

To evaluate the results we will use some of the most common and publicly available datasets (provided via the Mulan project [31]) used to test multi-label classification algorithms. Specifically, we use the following datasets during our testing. (see Table 4.1.1)

4.1.2 Parameters

For the evaluation we use the algorithm running in its original configuration as a baseline to compare against. Then we run our modified algorithm which applies a number of improvement iterations after the initial model is built, as discussed earlier. The control-run will appear at iteration 0 on the plots, indicating 0

Table 4.1: The datasets used, provided via MulanProject [31]

dataset	instances	attributes	labels	cardinality	density
birds	645	260	19	1.014	0.053
emotions	593	72	6	1.868	0.311
flags	194	19	7	3.3926	0.485
yeast	2417	103	14	4.237	0.303
image	2000	294	5	1.236	0.247
scene	2407	294	6	1.074	0.179

improvement-iterations were done. The main parameters we will take a closer look at are: instance-sub-sampling, feature-sub-sampling, the loss-function, max-rules and rule-improvement-iterations. We are also able to vary or disable certain improvement operations, namely: ADD_COND, REMOVE_COND, RELEARN, as well as the number of modified conditions per iteration. The default configuration will focus on ADD_COND and REMOVE_COND. The mode of evaluation can be set to either in-sample or out-of-sample. This second set of parameters is currently set at compile time and therefore we will change these in cases where we identify them to be able to yield additional information. The shrinkage parameter will by default remain at 1, to avoid that the default rule overpowers any changes made to the model. This is useful because our tested models will mostly consist of 20-40 rules, which is much less than BOOMER would usually induce. Any data presented will also feature the settings used.

4.2 Evaluation the Collected Data

To present the collected data in a simple and easily interpretable way we implemented a simple python script, to plot a series of runs done by the algorithm in a graph. The additional data provided in the tables associated to a certain configuration will feature their scores as percent values, the purpose here is that these are easier to read in a table format. Some abbreviations we will use in the following section are: instance-sub-sampling - ISS, feature-sub-sampling - FSS and cross-validation - CV

4.2.1 Evaluating Label-Wise-Logistic-Loss

The first configuration we tested was using the label-wise-logistic-loss, the BOOMER algorithm [cite boomer] uses this loss as a surrogate for the hamming accuracy. This means that this loss will try to optimize our Hamming Accuracy metric. (the inverse of the Hamming Loss)

1. 20rules 40 iterations, ISS=bagging, No FSS, No CV
In this first configuration the algorithm will be learning 20 rules on the data, use simple bagging for the instance-sub-sampling, not do feature sub-sampling only do ”ADD_COND” and ”REMOVE_COND” (with a limit of 1 per iteration) and evaluate the modifications out-of-sample, meaning on all the currently un-sampled examples.

We can see in Fig 4.1 and 4.2 that in the case of the birds and emotions dataset we see no real upwards trend for our target metric (Hamming Acc.) this remains true for the other datasets as can be gathered from 4.2. These ”jumpy” variations in metrics, which we can observe around the initial starting value could

Figure 4.1: Birds 20 It 40, NoFSS, ISS=Bagging, No CV

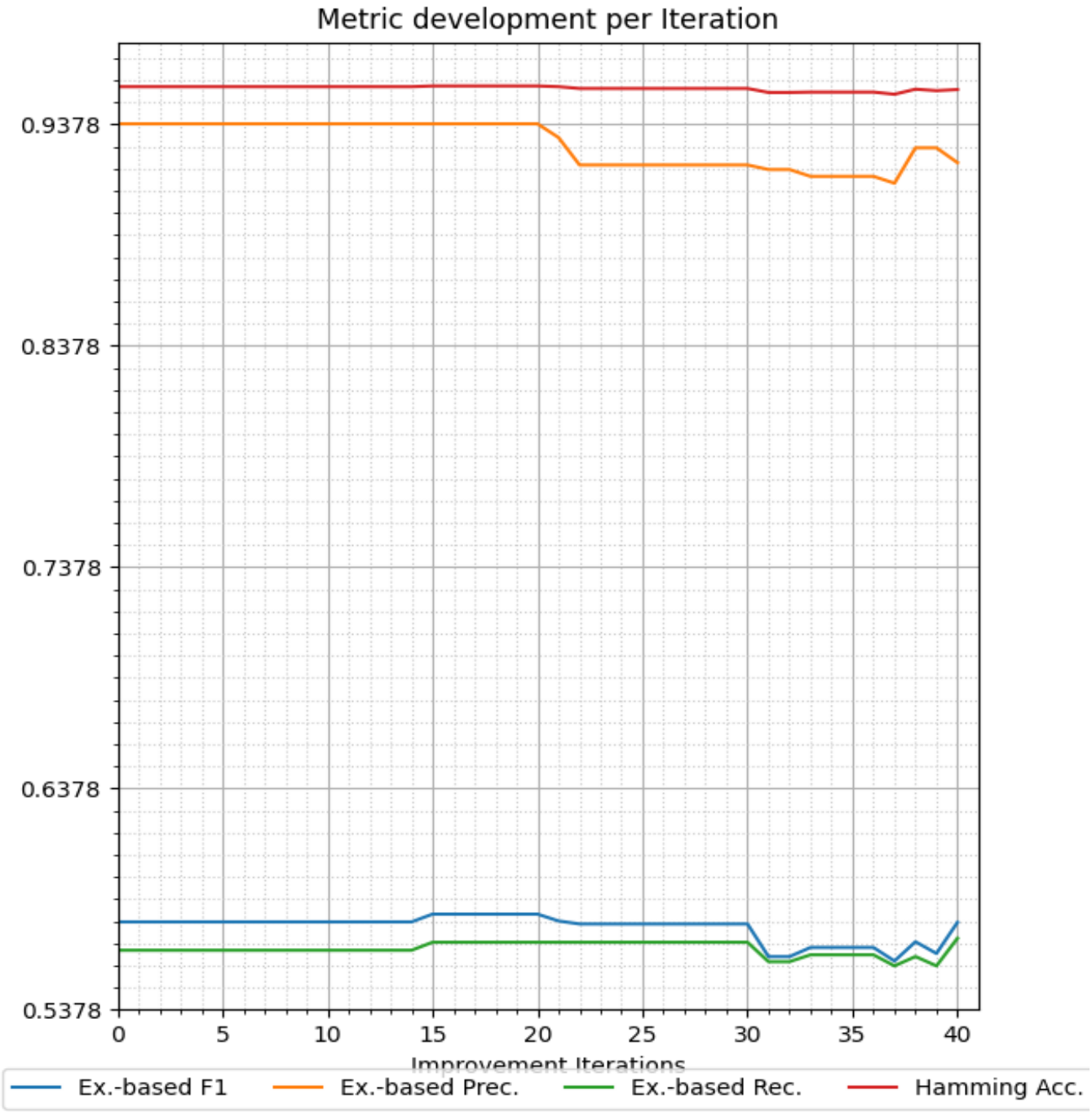


Figure 4.2: Emotions 20 It 40, NoFSS, ISS=Bagging, No CV

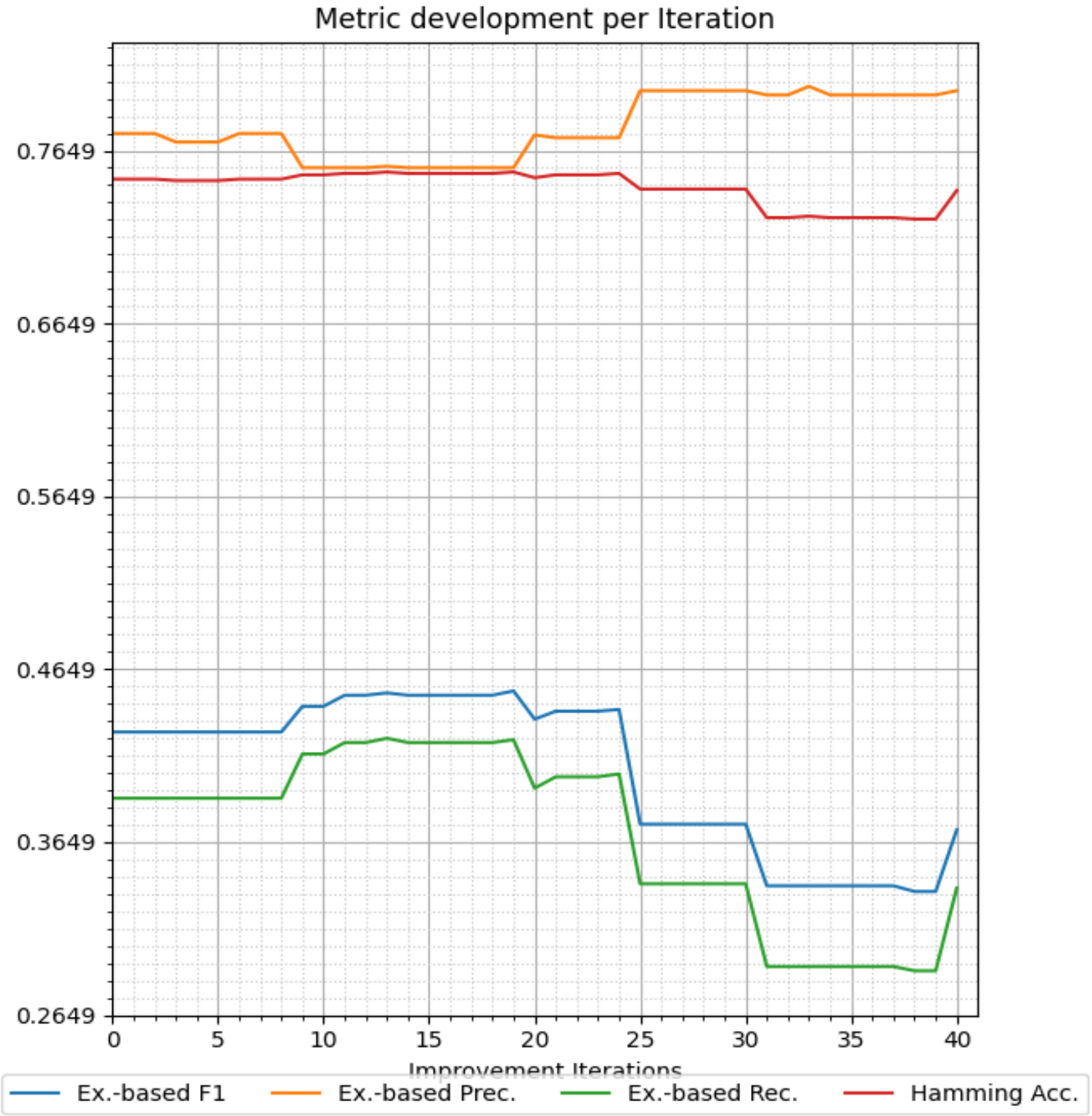


Table 4.2: NoFSS, ISS=bagging, No CV, test,outofsample, additional data, score-format in percent

dataset	#iterations	Hamming Acc.	Ex.-basedF1	Ex.-based Precision	Ex.-based Recall	Evaluation on
birds	0	95.4863	57.7517	93.8080	56.4705	test-data
birds	10	95.4863	57.7517	93.8080	56.4705	test-data
birds	20	95.5189	58.1026	93.8080	56.8317	test-data
birds	30	95.4049	57.6588	91.9504	56.8317	test-data
birds	40	95.3560	57.7310	92.0536	57.0123	test-data
emotions	0	74.8349	42.8547	77.4752	39.0264	test-data
emotions	10	75.0000	41.8151	79.2904	38.2838	test-data
emotions	20	75.1650	42.2937	79.3729	38.7788	test-data
emotions	30	75.2475	42.4752	79.7854	38.7788	test-data
emotions	40	75.1650	41.6501	80.0330	37.7887	test-data
flags	0	70.9890	68.1159	70.6923	70.1282	test-data
flags	10	71.4285	69.4322	70.4358	72.1282	test-data
flags	20	70.5494	68.5518	68.7948	71.6666	test-data
flags	30	70.7692	68.6617	69.0512	71.6666	test-data
flags	40	68.3516	66.7322	67.8717	69.1538	test-data
yeast	0	78.0806	51.7101	74.0603	42.4102	test-data
yeast	10	78.0806	51.7101	74.0603	42.4102	test-data
yeast	20	78.0806	51.7101	74.0603	42.4102	test-data
yeast	30	78.0806	51.7101	74.0603	42.4102	test-data
yeast	40	78.0806	51.7101	74.0603	42.4102	test-data
image	0	78.0806	51.7101	74.0603	42.4102	test-data
image	10	78.7174	33.8677	81.3293	33.1663	test-data
image	20	78.7174	33.8677	81.3293	33.1663	test-data
image	30	78.6773	35.0701	80.4275	34.3687	test-data
image	40	78.6372	35.7047	79.6259	35.0701	test-data
scene	0	85.9392	35.5908	88.2246	36.3712	test-data
scene	10	85.8416	35.4793	87.8065	36.3712	test-data
scene	20	85.8416	35.4793	87.8065	36.3712	test-data
scene	30	85.8416	35.4793	87.8065	36.3712	test-data
scene	40	84.8244	36.2318	82.0791	38.9214	test-data

be caused by several things, for instance: instance sampling at each iteration could be causing ”unstable” changes, which are not actually better on the entire data, but are fitting just the sampled data. It could also be possible that the initial model is already ”too good” to find valid improvements and the algorithm is unable to find really useful changes due to the limited changes we allow it to make.
One odd thing of note is that the yeast dataset seems to be somehow unaffected by whatever is causing this behavior. Finally, we also can not yet rule out that some sort of error within the process of selecting the correct improvement.
It could be that something is causing the algorithm to select an ”improvement” by accident which was not compared to the correct value, thereby replacing a rule with one that is actually worse.
Due to the nature of machine-learning algorithms testing for these sorts of errors quickly and efficiently proves rather difficult since usually a ”desired” or ”perfect” solution to the problem is not yet known. We could verify, using the program logs and testing outputs, that the algorithm consistently finds better evaluations (better quality scores) for the modifications applied. An earlier version of the implementation, which used a slightly different improvement strategy, also did provide actual metric-improvements although these were not consistent enough. We were also able to verify that each implemented operation has the desired effect on the model, more specifically: ”REMOVE_COND” removes a specified amount of conditions, ”ADD_COND” adds a specified number of new conditions to the selected rules and ”RELEARN” learns a new rule in its place, so it appears most likely that if the cause for the observed behavior is found inside the code, that its located within the modification selection part of the implementation, given what we were able to observe. Table 4.2 shows the start and endpoints after 40 iterations with several mid-points, evaluated on the testing-data, which all appear to follow this same trend. It is evident that the expected slight upwards trend does not arise, even after performing a number of improvement iterations. The target metric still remains at about the same level it started at, while most other metrics tend to drop a bit. Furthermore, due to each iteration using a different sample of the training data the algorithm almost always finds an improvement to apply. The fact that this configuration does not provide a consistent trend, this does not meet our expectations. It is also a first hint that something about the process itself might not be working as intended.

2. 40rules 40 iterations No FSS, ISS=bagging, No CV

Next we looked at whether increasing the number of rules has a beneficial effect on the behavior we previously observed. As we can see in Fig 4.3 increasing the number of rules only really reduces the effects of each individual modification (dampens fluctuations) but the behavior remains largely unchanged, the metrics still rise and fall in apparently arbitrary intervals. And the overall trend remains towards a worse model. The additional data in Table 4.3 also confirms that observation. However, yeast once again remains particularly stable.

Figure 4.3: Birds 40 It 40, NoFSS, ISS=bagging, No CV

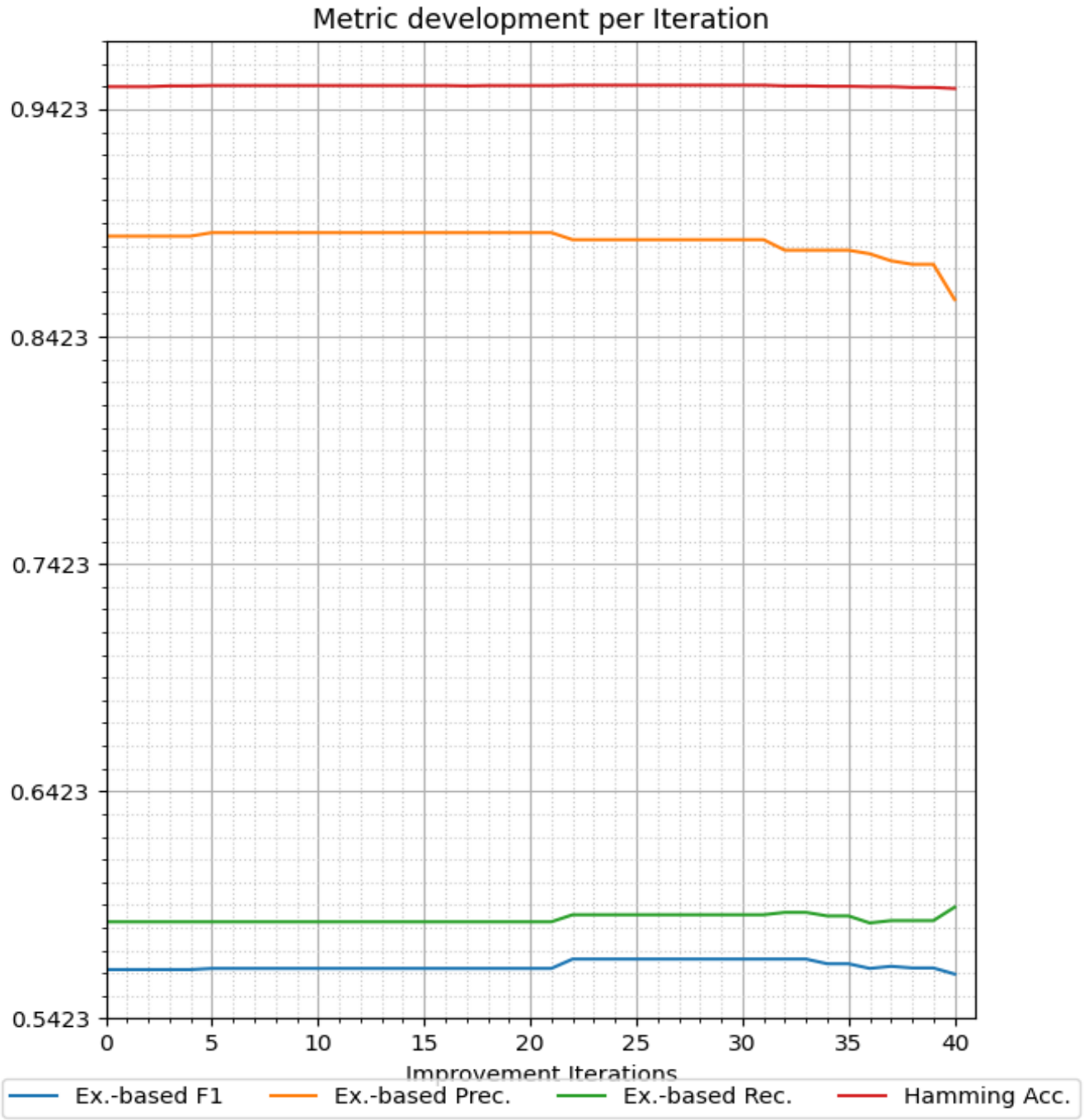


Table 4.3: 40 It 40 NoFSS, ISS=bagging, No CV, test,outofsample, additional data, score-format in percent]

dataset	#iterations	Hamming Acc.	Ex.-based F1	Ex.-based Precision	Ex.-based Recall	Evaluation on
birds	0	95.2256	56.3791	88.6481	58.4829	test-data
birds	10	95.2745	56.4382	88.8889	58.4829	test-data
birds	20	95.2745	56.4382	88.8889	58.4829	test-data
birds	30	95.2958	56.8435	88.4932	58.7925	test-data
birds	40	95.1442	56.1831	85.8617	59.1279	test-data
emotions	0	75.4959	49.9457	68.9356	58.2525	test-data
emotions	10	75.6666	54.2899	69.8151	59.5959	test-data
emotions	20	75.5775	55.9311	68.6881	59.5959	test-data
emotions	30	74.6699	52.3338	65.3282	52.2727	test-data
emotions	40	74.6699	52.3338	65.8333	52.2727	test-data
flags	0	76.7296	74.5455	73.2821	78.2821	test-data
flags	10	75.1648	72.2685	71.6415	75.1538	test-data
flags	20	75.1648	72.2721	71.9487	75.1538	test-data
flags	30	74.5934	71.9559	76.6666	75.5384	test-data
flags	40	73.8461	71.8874	74.3223	75.8461	test-data
yeast	0	78.2676	54.65824	72.1988	47.5581	test-data
yeast	10	78.2754	54.67947	72.1988	47.5854	test-data
yeast	20	78.2754	54.67947	72.1988	47.5854	test-data
yeast	30	78.2754	54.67947	72.1988	47.5854	test-data
yeast	40	78.2754	54.67947	72.1988	47.5854	test-data

3. 20rules 40 iterations no feature subsampling, no instance subsampling, evaluated on training data

Our next experiment is trying to specifically test whether or not our instance-sampling is possibly hindering our algorithm by creating samples that are not representative of the entire training set. This would could cause our model to become worse while training on sampled data. This is done by removing instance-sub-sampling (ISS), feature-sub-sampling(FSS), while evaluating the model on the training-data. We would expect, now that we are evaluating on the training-data and using no ISS that we should be seeing either a constant unchanging development or a very slight upwards trend. We also use 10folds for cross validation to further reduce the influence of poor data splits which could occur if doing no CV.

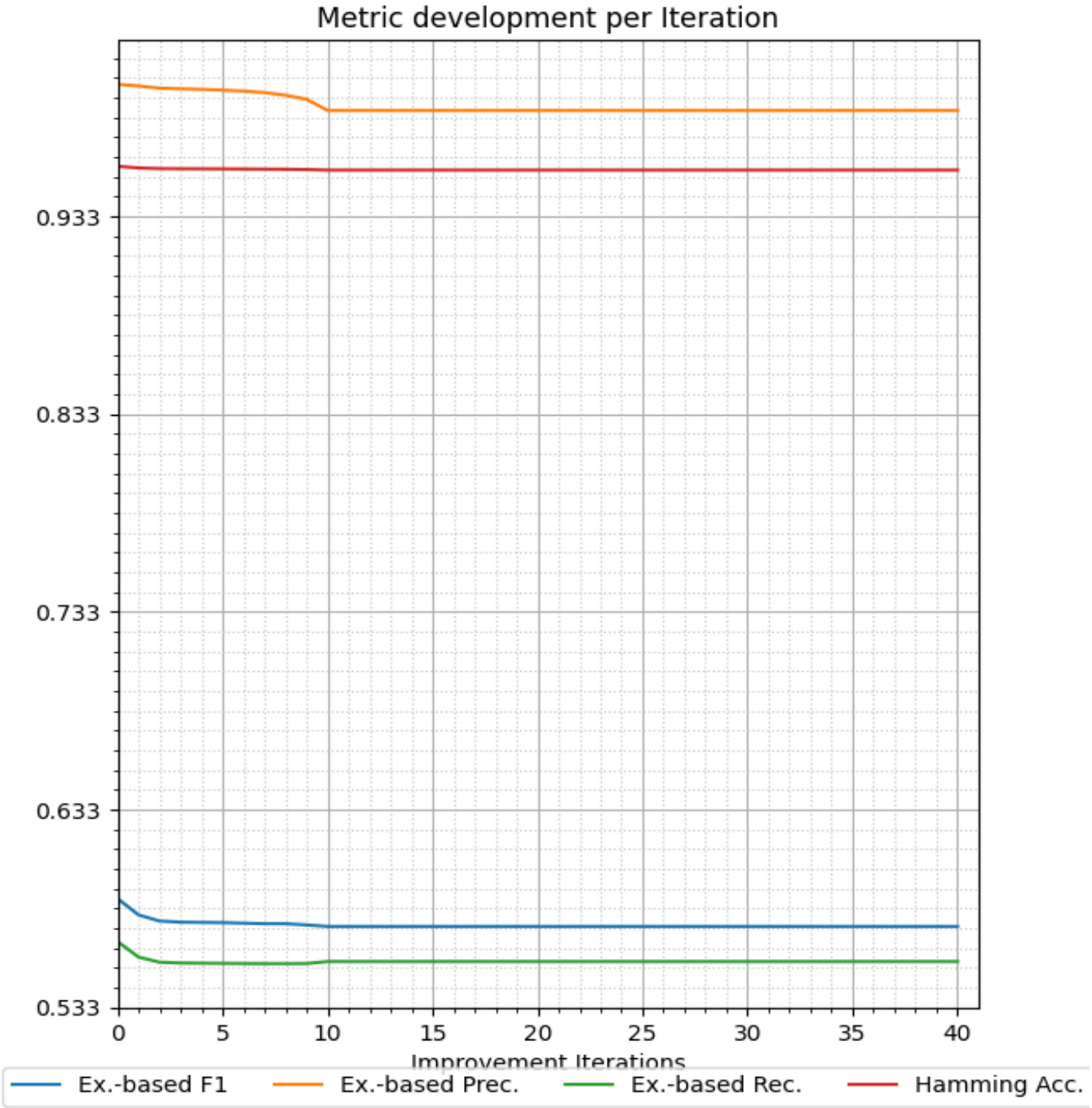
As Figure 4.4 and Table 4.4 show our initial expectation for the test are once again not full-filled, it appears as though the development has become a lot more stable but it is somehow still possible to achieve reductions in the target metric. It appears as if poor ISS might be part of the reason our model is not improving, but there has to be more to it, since we still experience a decline of metrics initially. This remains true even without performing CV.
However, we can not yet conclude that this is the only cause for this unexpected behavior, because we still observed some losses. An interesting note is that the algorithm appears to suddenly stop decreasing its model quality after several iterations, indicating a stable state. The reason for this might be that the lack of instance-sampling is causing our algorithm to at some point no longer be able to select viable improvements. An additional observation confirming this is that all folds of the CV stopped finding any improvements after a number of changes were applied, that explains why the graphs all flatten out after some iterations. A reason

for the lack of further improvements is likely the lack of instance-sampling, causing us to stop at a presumably ”ideal” model. We also need to start considering other explanations for this odd behavior.
One possible option might be that one of our modification operations is causing problems and on occasion is responsible for the selection of bad improvements for some reason, to take a closer look at this we will run a short evaluation without cross validation (CV) but with ISS activated and color the plot according to the currently selected operation. This way we can correlate operations to the change in model quality and possibly draw conclusions about the effect this way.

Table 4.4: 20 It 40, NoFSS, NoISS, 10F CV, EvalOn Training, in-sample, additional data, score-format in percent

dataset	#iterations	Hamming Acc.	Ex.-based F1	Ex.-based Precision	Ex.-based Recall	Evaluation on
birds	0	95.8357	58.8122	99.9827	56.6364	train-data
birds	10	95.6443	57.4038	98.6581	55.6288	train-data
birds	20	95.6443	57.4038	98.6581	55.6288	train-data
birds	30	95.6443	57.4038	98.6581	55.6288	train-data
birds	40	95.6443	57.4038	98.6581	55.6288	train-data
emotions	0	84.9040	58.3363	98.3104	52.7009	train-data
emotions	10	84.4857	56.8128	98.6851	51.0744	train-data
emotions	20	84.3201	55.9313	99.0787	50.1529	train-data
emotions	30	84.2983	55.8876	99.0880	50.0905	train-data
emotions	40	84.2983	55.8876	99.0880	50.0905	train-data
flags	0	80.8215	76.0123	82.2949	73.4122	train-data
flags	10	79.7735	74.9686	81.3809	72.3527	train-data
flags	20	79.0702	74.1728	80.7475	71.7739	train-data
flags	30	78.9314	74.0115	80.7990	71.4775	train-data
flags	40	78.9314	74.0115	80.7990	71.4775	train-data
yeast	0	79.8619	54.0061	77.2188	44.3370	train-data
yeast	10	79.8619	54.0061	77.2188	44.3370	train-data
yeast	20	79.8619	54.0061	77.2188	44.3370	train-data
yeast	30	79.8523	53.9979	77.1800	44.3370	train-data
yeast	40	79.8523	53.9979	77.1800	44.3370	train-data

Figure 4.4: Birds 20 It 40, NoFSS, NoISS, 10F CV, EvalOn Training, in-sample



4. 20rules 40 iterations No FSS, ISS=bagging, evaluated on test data, insample ; color-coded operations

We can conclude from the coloring (see fig 4.5; Green: ADD_COND; Red: REMOVE_COND) that while a majority of operations are ”REMOVE_COND” the noticeable dips appear not to be correlated to any specific operation (on neither evaluation: test vs training-data).
Additional available program logs appear to confirm this assumption, since they show no clear correlation between model changes and the selected modification either. Therefore, it appears unlikely that the act of adding and removing conditions is causing the issue in itself, since we can see both rising and falling metrics following both these operations. This detailed analysis of our testing results has unfortunately not yet yielded an obvious explanation for the observed behavior. It is possible that even after extensive verification of the individual parts of the algorithm that we missed something. Both, the individual functions behavior's and the model's prediction matrix remained ”logically consistent” during their evaluation process.
This means no big changes or sudden big swings in values and no ever increasing error-terms were observed. These could cause the model to become worse over time. Our experiments were unable to help us identify a clear cause for the unexpected decline of the metrics. It could be possible after reviewing our test results, that we missed a flaw in the ”improvement-logic” implementation. Specifically, a flaw related to the selection process appears to be a likely explanation at this point.
However, since we were also unable to observe any meaningful improvement even when evaluating on the training-data while also not performing ISS it could also be the case that the approach of iterative rule improvement does not work particularly well on boosted rule models. If the algorithm, by its nature, is not able to correctly identify ”better” modifications reliably, that would explain what we are able to observe. It appears as though the algorithm is currently able to select modifications resulting in a worse model, even though in terms of rule quality an improvement was calculated. This particular observation appears to further support the assumption that our approach needs to be refined to work on boosted rule models. Due to the evaluated quality being based on the chosen loss-function, we can be reasonably certain that the quality score itself should not be causing these issues. Although, it might be that in order to correctly identify valuable modifications for boosted rule models, additional parameters to regulate the evaluation of the rule qualities are needed. The assumption that we need to process the quality score of a rule before using it appears to be supported by the observation that a decline was possible while logs reported an increasing quality score and not performing ISS on the training data.
Our collected data thus far has on average been relatively stable with some strong hints of some odd behavior. While the target metric especially does not appear to stray very far from its starting value, this could just be related to us restricting the size of the changes and only allowing small ones. It also does not appear plausible that the operations themselves are faulty since this should be unlikely to produce such ”stable” behavior. Given that metrics flattened somewhat quickly when testing without ISS (see 4.4), it appears as though the type of instance sampling also plays a role in the quality of selected rule-improvements (and the lack of these), which does make sense since the learning process also benefits from unbiased sampling.

Figure 4.5: Birds 20 It 40, NoFSS, ISS=Bagging, No CV, Eval On Testing-data, outofsample, colorcoded operations

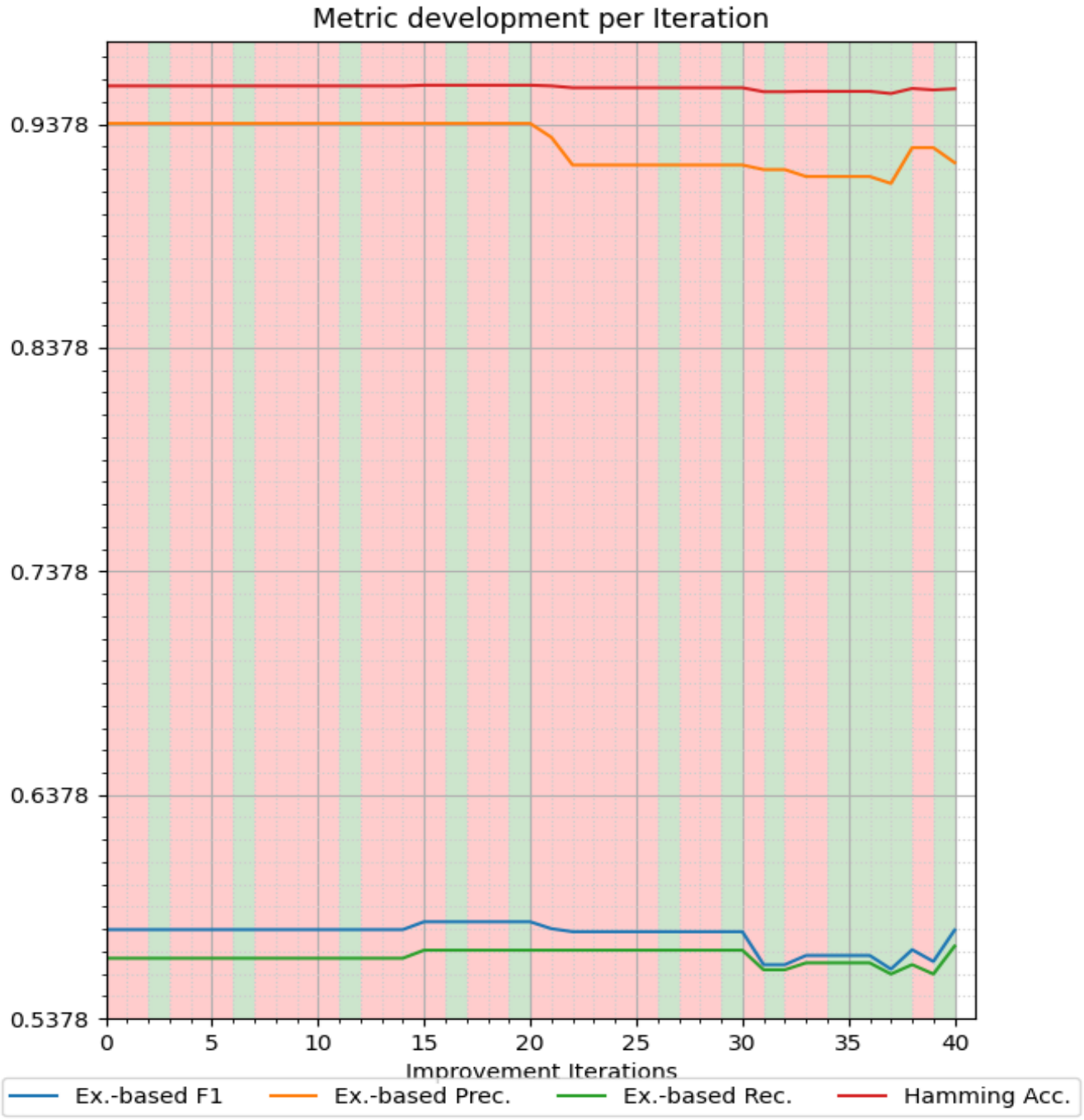


Table 4.5: Some observed model-size decreases

Dataset	#rules	#iterations	model-size (in kb)	# total conditions
birds	20	0	3.19	63
birds	20	20	2.84	53
birds	20	40	2.68	49
emotions	20	0	6.19	109
emotions	20	20	5.58	97
emotions	20	40	5.15	89
flags	20	0	1.93	89
flags	20	20	1.83	83
flags	20	40	1.79	81

4.2.2 Additional observations

One thing we could observe is that our algorithm currently tends to reduce the model size. Given that most operations performed remove conditions, therefore produce less specific rules with less conditions, this is to be expected. A result is that we are compressing the model while retaining metrics, usually close to the original ones. This could potentially be a useful feature for very large models if weighting cost/benefit of this compression ahead of time. And the user is aware of the potential costs in terms of a slightly reduced model quality. Table 4.5 table depicts some general trends we were able to observe with respect to model sizes. These trends obviously apply to both the file-size and the length of individual rules. In these few examples we were able to in the best case (birds20) reduce the number of conditions down to 78% of their starting amount, while retaining almost 99% of its target metric performance. The worst case (flags20) still reduced the number of conditions to 91% while suffering a reduction in the target metric to 96%, so it appears that if going strictly by the numbers the compression could actually be considered useful in certain cases.

4.2.3 Evaluating Example-Wise-Logistic-Loss

The example-wise-logistic-loss is used as a surrogate loss for the subset 0/1 loss which is why we did replace the Hamming Acc with the 0/1 subset loss in our data presentation. Even though the example-wise loss behaves largely the same we do still want to include one of these plots in order to provide a more complete picture of the observations. The main difference we can observe in Figure 4.6 and Table 4.6 is that we observed more frequent and noticeable fluctuations. This is likely caused by the full head refinement (multi-label rule heads) we are doing for the example-wise logistic loss. Subset 0/1 Loss increases overall, so the model as previously said appears to behave largely equivalent. Figure 4.6 shows the emotions dataset performed on example-wise-log-loss. The behavior remains unsteady with a tendency towards a higher loss, thus a worse target-metric. Table 4.6 depicts the associated results of the remaining datasets.

Figure 4.6: Ex.-w. Emotions 20 It 40, NoFSS, ISS=Bagging, No CV, Eval On Testing-data, outofsample

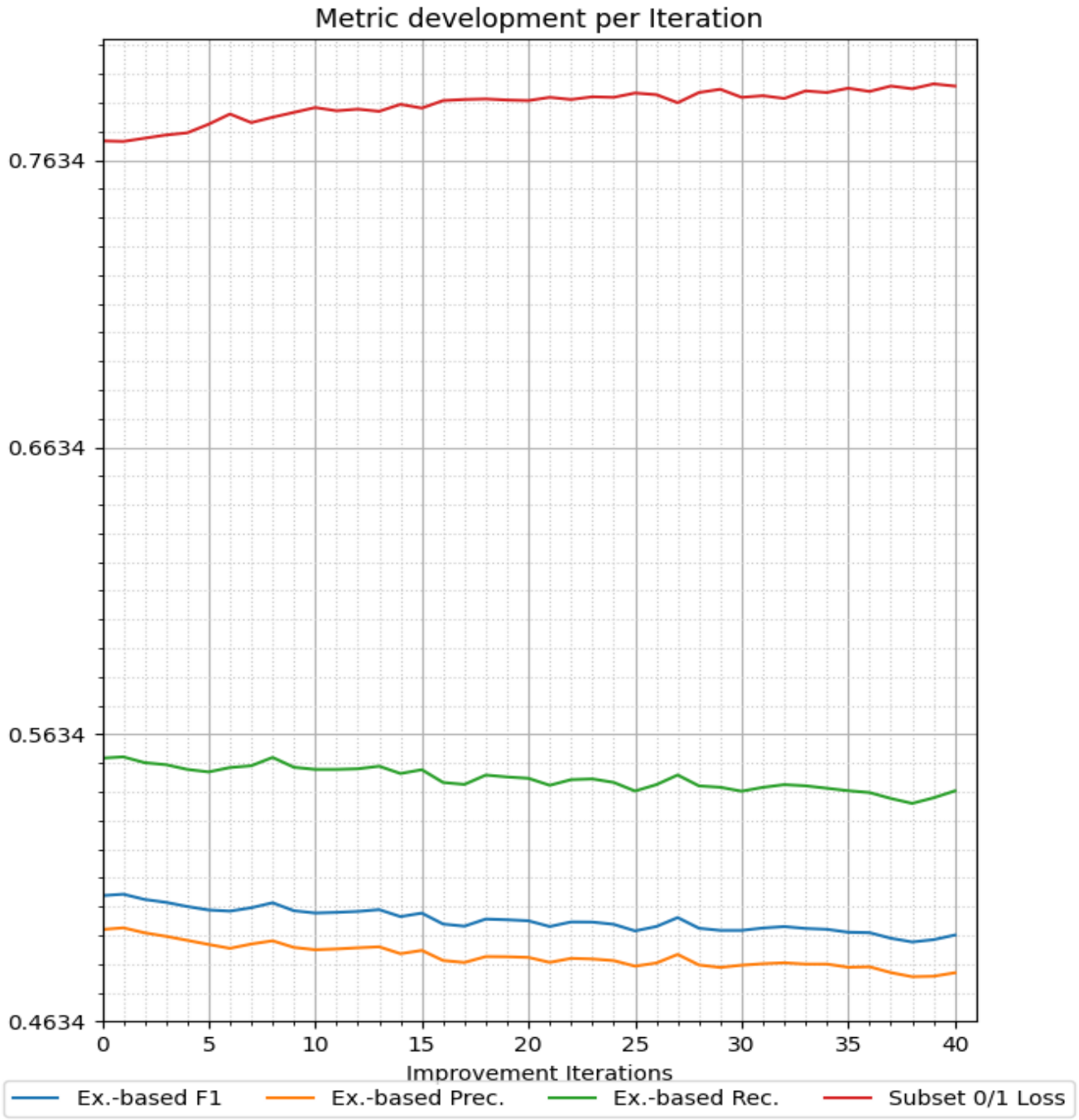


Table 4.6: Ex.-w. 20 It 40, NoFSS, ISS=bagging, 10F CV, EvalOn Testing-data, out-of-sample, additional data, score-format in percent

dataset	#iterations	Subset 0/1 Loss.	Ex.-based F1	Ex.-based Precision	Ex.-based Recall	Evaluation on
emotions	0	77.0095	50.7161	49.5360	55.5063	test-data
emotions	10	78.1705	50.1106	48.8306	55.1129	test-data
emotions	20	78.4142	49.8394	48.5650	54.8036	test-data
emotions	30	78.5266	49.5052	48.2933	54.3538	test-data
emotions	40	78.9196	49.3431	48.0288	54.3642	test-data
flags	0	85.1694	67.6840	64.4989	75.4362	test-data
flags	10	88.7655	65.6302	61.2211	80.3611	test-data
flags	20	87.7927	66.4642	61.5774	80.9171	test-data
flags	30	87.5609	65.8312	62.1780	78.7925	test-data
flags	40	87.6765	66.2068	62.3451	79.1503	test-data
yeast	0	86.6178	59.9422	61.9260	62.2603	test-data
yeast	10	86.3558	59.6929	61.8779	61.6779	test-data
yeast	20	86.4662	59.7263	61.8701	61.8145	test-data
yeast	30	86.6179	59.5067	61.6241	61.5614	test-data
yeast	40	86.7328	59.6285	61.6516	61.8215	test-data
image	0	70.6333	36.9790	39.9583	36.0333	test-data
image	10	70.7722	36.9901	39.9416	36.1083	test-data
image	20	71.4444	36.9207	39.6916	36.3416	test-data
image	30	71.5833	36.5738	39.3944	35.8898	test-data
image	40	71.9055	36.6151	39.2694	36.1509	test-data
scene	0	73.2310	28.9746	30.0832	28.4222	test-data
scene	10	74.0065	28.2422	29.3723	27.6790	test-data
scene	20	75.3315	26.9726	28.1304	26.3956	test-data
scene	30	76.0194	26.3031	27.4702	25.7215	test-data
scene	40	76.0517	26.1785	27.2994	25.6199	test-data

4.2.4 Additional Notes

While performing the ”RELEARN” operation the current implementation tended to gravitate towards doing mostly that and especially if not performing ISS it produced near identical ”relearned” rules. It appears as though the relearn operation does not provide much value for boosted rule models. Experiments allowing more conditions to be changed at once, caused slightly bigger ”swings” while not affecting the overall behavior. Enabling FSS caused a very small stabilizing effect in some experiments, but this did not appear to be significant and is likely just connected to the fact that we sample during each improvement iteration and thereby achieve a similar effect as not performing FSS would have. The dataset yeast appears to be especially resistant to the decline of model metrics regularly caused by our algorithm in other datasets. Other research on post-pruning (like [4], [12],[10]) has sometimes been using a separate sample of the data to perform improvements, one which has not yet been used during training, since we have not been doing this, mainly for simplicity purposes it might be one reason contributing towards the poor results. Another thing of note is, that earlier research especially on post-pruning was able to consistently achieve model-improvements, which strengthens the suspicion that either our approach/implementation might be flawed or that the con-

cept of iterative model improvement does not work on gradient boosted rule models like we expected it to and possibly needs further refinement. Moreover, the new algorithm was only tested on the Boosting implementation ”BOOMER”[28] of the associated rule learning research project so far, a separate-and-conquer implementation as presented in [27] is also available to conduct further tests on.

5 Conclusion

We will end the thesis with some concluding remarks and outline what did work and what did not. Finally, we will derive implications for future work in the field of rule learners.

5.1 Concluding the Results

To conclude this paper, we developed an extension to the BOOMER [28] algorithm on the basis of several prior works like [14], [4], [12], [10] and in the hopes of achieving an improvement of the model quality and target metrics by performing these modifications. As can be seen in Chapter 4 our testing showed that the general idea of improving our model was occasionally successful but was not able to consistently improve our models in each step. A majority of changes either did not influence our results or even worsened them. We were unable to identify the specific reason for this behavior as of now. Our assumption is, that either the entire idea of iteratively improving boosted rule models using quality-scores does not work or our specific implementation is flawed in some way. The evidence does not appear to be sufficient to confirm either assumption as true yet. The algorithm manages to select ”worse” modifications even though those should technically produce a worse quality-score than the unmodified rule. This might be related to the model behaving in a way we did not anticipate, perhaps a new parameter is required to further regulate the calculated quality-score. Relearning did not appear to have a major impact on the quality of modifications achieved. Furthermore, relearning rules also tended to be favoured as the best operation while activated. This is likely because the change of adding an entire rule outweighs smaller changes. Our algorithm also did perform slightly better when using feature-sub-sampling, but the overall observations in terms of trends as displayed in 4 still held true. While our approach appears quite plausible in theory, our testing was unable to verify a consistent improvement does indeed occur. The ”improvements” appear to be pretty inconsistent or even worsened the model if compared to the starting model. Overall, it appears as though our results were not conclusive enough to be able to disprove that the approach definitely does not work or to prove that a malfunction must be the reason. We found indicators supporting both these main assumptions. Nevertheless, we were able to identify that the algorithm still manages to compress our model’s with minimal losses in the metrics optimized by the loss function. In some cases the difference between compression and target-metric-loss was as big as 21% as can be seen in 4.2.2

5.2 Future Work

Future research on this specific topic should be trying to develop a better understanding of how changes to a boosted rule model after induction do affect the model quality and why/how they do that. As noted earlier the observations appear to show that rule-quality alone is not enough to identify good modifications.

Prior work like [4], [12],[10] and [14] which appeared to show promising results for modifications to (rule) models in general by using pruning (and modification) approaches were not able to be reproduced during the evaluation of our Stochastic Gradient Descent Boosted Rule Learning approach. It might be possible that rule improvement does not work particularly well for Boosted Rule Learners. This prior research [4], [12],[10] and [14] used slightly different types of modeling after all. However, I think more research needs to be done to come to a final conclusion on this. Furthermore, applying the developed algorithm to randomly generated rule model's (which will usually be terrible) could be a useful step in verifying that the implementation does actually and consistently achieve its goal with sufficiently bad starting models. This would enable us to conclude whether the models generated by BOOMER are already "too good" for our algorithm to improve upon or whether the implementation still needs more work. In the same line of thinking it could prove interesting to apply the improvement algorithm on the Separate and Conquer rule learner [27], that is also available as part of the overall research project [28]. Additionally, it might be necessary to calculate a Δ-quality, to more accurately evaluate the change in quality when comparing the original rule and the modified one with vastly different qualities. The algorithm was built under the assumption that any modified rule that is improving will by default also get selected due to a higher quality-score. It might currently be the case that this assumption can be violated. It is possible that a re-implementation of the ideas presented in this thesis could remove doubts about possible implementation errors and put the assumption that boosted Rules are poorly modify-able on firmer ground.
Extending the currently existing improvement algorithm by adding even more complex operations (e.g. "replace condition", add and remove in one iteration via a candidate-search) or even conducting a version of Beam-Search on improvement candidates could yield interesting results. Also other interesting approaches would be to use the implementation with more and different loss-functions (like F1 loss or one feature in [32]). Using several different sampling methods (apart from bagging,Random-Feature-Selection) for instance and feature sampling, to validate the effects of sampling quality on the improved models, as was noted earlier during our test-observations. It might also prove prudent try and perform the modifications of the algorithm with previously unseen training-data. Finally, as was noted in Section 4.2.2 unexpected compression behavior was visible as a result of running the improvement-algorithm, likely due to the tendency to remove more conditions than we add, this might also be an interesting direction to dive deeper into to facilitate slightly easier storage of huge models.

Bibliography

[1] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11 (2018), e00938.

[2] Tianqi Chen and Carlos Guestrin. "XGBoost: A scalable tree boosting system". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data mining*. 2016, pp. 785–794.

[3] Zhao-Min Chen, Xiu-Shen Wei, Peng Wang, and Yanwen Guo. "Multi-label image recognition with graph convolutional networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 5177–5186.

[4] William W Cohen. "Fast effective rule induction". In: *Machine learning proceedings 1995*. Elsevier, 1995, pp. 115–123.

[5] Krzysztof Dembczynski, Wojciech Kotlowski, and Eyke Hüllermeier. "Consistent multilabel ranking through univariate losses". In: *arXiv preprint arXiv:1206.6401* (2012).

[6] Krzysztof Dembczyński, Wojciech Kotłowski, and Roman Słowiński. "Ender: a statistical framework for boosting decision rules". In: *Data Mining and Knowledge Discovery* 21.1 (2010), pp. 52–90.

[7] Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139.

[8] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)". In: *The Annals of Statistics* 28.2 (2000), pp. 337–407.

[9] Jerome H Friedman and Bogdan E Popescu. "Predictive learning via rule ensembles". In: *The Annals of Applied Statistics* 2.3 (2008), pp. 916–954.

[10] Johannes Fürnkranz. "Pruning algorithms for rule learning". In: *Machine learning* 27.2 (1997), pp. 139–172.

[11] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač. *Foundations of rule learning*. Springer Science & Business Media, 2012.

[12] Johannes Fürnkranz and Gerhard Widmer. "Incremental reduced error pruning". In: *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 70–77.

[13] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. "A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.4 (2011), pp. 463–484.

[14] Henry Gouk, Bernhard Pfahringer, and Eibe Frank. "Stochastic gradient trees". In: *Asian Conference on Machine Learning*. PMLR. 2019, pp. 1094–1109.

[15] Moritz Hardt, Ben Recht, and Yoram Singer. “Train faster, generalize better: Stability of stochastic gradient descent”. In: *International Conference on Machine Learning*. PMLR. 2016, pp. 1225–1234.

[16] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: DataMining, Inference, and Prediction*. 2009.

[17] Frederik Janssen. “Heuristic rule learning”. [Online; accessed 01-August-2021]. PhD thesis. 2012.

[18] Frederik Janssen and Johannes Fürnkranz. “On the quest for optimal rule learning heuristics”. In: *Machine Learning* 78.3 (2010), pp. 343–379.

[19] Frederik Janssen and Markus Zopf. “The SeCo-framework for rule learning”. In: *Proceedings of the German Workshop on Lernen, Wissen, Adaptivität-LWA2012*. 2012.

[20] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. “LightGBM: A highly efficient gradient boosting decision tree”. In: *Advances in Neural Information Processing Systems* 30 (2017), pp. 3146–3154.

[21] Oluwasanmi Koyejo, Nagarajan Natarajan, Pradeep Ravikumar, and Inderjit S Dhillon. “Consistent Multilabel Classification.” In: *NIPS*. Vol. 29. 2015, pp. 3321–3329.

[22] Eneldo Loza Mencía. “Efficient Pairwise Multilabel Classification”. [Online; accessed 01-August-2021]. PhD thesis. 2013.

[23] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook,volume2*. 2005.

[24] Eneldo Loza Mencía and Frederik Janssen. “Learning rules for multi-label classification: a stacking and a separate-and-conquer approach”. In: *Machine Learning* 105.1 (2016), pp. 77–126.

[25] Deanna Needell, Nathan Srebro, and Rachel Ward. “Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm”. In: *Mathematical Programming* 155.1-2 (2016), pp. 549–573.

[26] Didrik Nielsen. “Tree boosting with XGBoost-why does XGBoost win ”every” machine learning competition?” MA thesis. NTNU, 2016.

[27] Michael Rapp. “A Separate-and-Conquer Algorithm for Learning Multi-Label Head Rules”. [Online; accessed 24-July-2021]. MA thesis. 2016.

[28] Michael Rapp, Eneldo Loza Mencía, Johannes Fürnkranz, Vu-Linh Nguyen, and Eyke Hüllermeier. “Learning Gradient Boosted Multi-label Classification Rules”. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Frank Hutter, Kristian Kersting, Jefrey Lijffijt, and Isabel Valera. Cham: Springer International Publishing, 2021, pp. 124–140. ISBN: 978-3-030-67664-3.

[29] Yutaka Sasaki and R Fellow. “The truth of the F-measure, Manchester: MIB-School of Computer Science”. In: *University of Manchester* (2007), p. 25.

[30] M Paz Sesmero, Agapito I Ledezma, and Araceli Sanchis. “Generating ensembles of heterogeneous classifiers using stacked generalization”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 5.1 (2015), pp. 21–34.

[31] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. “Mulan: A java library for multi-label learning”. In: *The Journal of Machine Learning Research* 12 (2011), pp. 2411–2414.

[32] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. “A comprehensive survey of loss functions in machine learning”. In: *Annals of Data Science* (2020), pp. 1–26.

[33] Shenghuo Zhu, Xiang Ji, Wei Xu, and Yihong Gong. “Multi-labelled classification using maximum entropy method”. In: *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. 2005, pp. 274–281.