Scalable Histogram-based Induction of Gradient Boosted Multi-label Rules

Bachelor thesis by Lukas Johannes Eberle
Date of submission: April 6, 2021

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

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
FB20
Knowledge Engineering
Group

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Lukas Johannes Eberle, die vorliegende Bachelorarbeit 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 Fall 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 der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

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

Darmstadt, 6. April 2021

L. Eberle

Contents

List of Figures

Liste der Algorithmen

Zusammenfassung

Klassifikation ist ein Gebiet von Machinellem Lernen, was zum Gebiet der künstlichen Intelligenz gehört. In dieser Arbeit schauen wir uns das spezifische Gebiet der Multi-Label Klassifikation über Regellerner an. Dabei werden historische Daten genutzt, um Modelle zu lernen, die Instanzen ähnlicher Daten mehrere Labels korrekt zuordnen können.
Dabei gibt es folgendes Problem: Mit vielen Daten nimmt die Zeit, die benötigt wird, um das Modell zu lernen, drastisch zu. Wir untersuchen hier einen Ansatz, der versucht, unter der Verwendung von „Unsupervised Binning“-Methoden, die Anzahl der vom Algorithmus untersuchten Bedingungen zu reduzieren und ihn damit geeigneter für große Datensätze zu machen. Damit wollen wir die Trainingszeit reduzieren, aber geben dafür auch etwas Genauigkeit auf. Dazu gruppieren wir mehrere Beispiele aus einem Datensatz in einem Behälter, genannt „Bin“. Den gesamten Datensatz teilen wir damit auf mehrere solcher Bins auf.
Je nach Datensatz und verwendeten Methoden unterscheiden sich die Ergebnisse hierbei stark. Dabei zeichnet sich ein Trend ab, auf dessen Basis explizite Empfehlungen möglich sind.

Abstract

Classification is an area of machine learning, which belongs to the area of artificial intelligence. In this thesis we will look at the specific area of multi-label classification via rule learners. Historical data is used to train models that can correctly assign multiple labels to instances of similar data correctly.
The problem with this family of algorithms is that with a lot of data, the time it takes to train the model increases dramatically. We will look at an approach that tries to reduce the number of conditions examined by the algorithm using “Unsupervised Binning” methods and thus make it more suitable for large data sets. With this approach we want to reduce the training time by giving up accuracy. To do this, we group several examples in a bin. The entire data set is therefore split into multiple bins.
The results differ greatly depending on the data set and the methods used. A trend is emerging on the basis of which explicit recommendations can be made.

1 Introduction

Artificial Intelligence (AI) and Knowledge Engineering have become more important in recent years and continue to do so, as computation power increases. They are versatile and can be used in many circumstances. AI includes, among other methods, Machine Learning (ML), which itself is divided in supervised and unsupervised ML. [Bro20] While AI is a broad umbrella term for any program, which mimics the cognitive functions of the human brain[Rus10], ML has the more refined task, to learn and use models from data[Gru15] with which it can then predict various outcomes.
ML is useful in modern computational problems, like Computer Vision[Seb05], which is for example used to implement self-driving cars, and Natural Language Processing[Pow89], which is used to compute or produce human speech or human-like speech respectively. These processes often use supervised learning. This means the models are trained with data, which was recorded beforehand. [Rus10] This data is called historic data. The fight against the Corona-Virus is fought with ML, as well[AAM20], which should stand as a testament for how much perceived potential ML has right now.
AI and Machine Learning are enormous business opportunities, too. [Bro20] So it comes as no surprise, that tech-giants like Google and Amazon produce a lot of AI services, like Google Cloud Services[1] and AWS [2].
Supervised Multi-Label Classification Algorithms take a special place in this development. As historic data collection increases constantly, since we collect more data every day, they become more powerful. The second reason is that these methods can assign multiple labels at once, like the name suggests, which can handle a lot of problems. An example would be recognizing multiple street signs in one picture for a self driving car. This class of algorithms can be used to label unseen data, which is similar to the training data, with multiple labels. It can also predict labels for data, which were previously unknown.
We look at the two ensemble methods, which can be used to implement multi-label classification; decision tree boosting, like XGBoost[Vis19] or LightGBM[Sha18], and rule learning, like BOOMER[Rap+20], where our focus lies in this thesis.

1.1 Motivation

As we gather more structured information our data sets grow larger, as well. First and foremost, this is a great benefit, as it makes ML algorithms more precise, but they also consume time and computational resources to a greater extent as smaller ones. So we need to accelerate the process to compensate for slower hardware improvement and increasing data points. [Lit20]
We want to find a way to benefit from the bigger data collections, while using less resources or reduce the training time respectively. The initial question was; if we only consider a specific subset, can we still profit from our larger data collection. In other words we want to make the induction of gradient boosted multi-label rules more scalable by reducing the number of possible conditions induced.

[1] `https://cloud.google.com/docs`
[2] `https://aws.amazon.com/`

In this approach we sacrifice some accuracy to reduce computation time or the required computational resources respectively. We achieve this trade-off by summarizing similar values into a bin and the whole data set in a set of these bins. We concentrate our efforts on non sparse feature values with unsupervised methods. There are some approaches, which handle sparse data, and supervised methods, which consider different scores that are calculated at run-time. We will look at those in the Section 2.2.

1.2 Structure of this Thesis

This thesis is structured as follows:

- **Fundamentals**: In Chapters 2 and 3 we look at all necessary details to understand what we are doing. In Chapter 2 we look at the prerequisites for this work, including the BOOMER algorithm, which is our baseline, and relevant related work. In Chapter 3 we look at Example Binning, which will be the focus of this thesis.
- **Implementation**: In the Chapter by the same name, Chapter 4, we look at our implementation and highlight some design decisions and unique pitfalls, which we will encounter by expanding on the fundamentals.
- **Discussion**: In the third and final part of this thesis, we evaluate our solution by looking at some experiments and their results in Chapter 5. We conclude in Chapter 6, if the approach works and sum up what we learned by working with it, as well as a short outlook on possible future work.

2 Fundamentals

This chapter introduces all prerequisites, necessary for the approach discussed in this paper. This mainly includes the underlying gradient boosted multi-label rule learning algorithm, called BOOMER as well as, related work, which follows a similar approach of generalizing data.

2.1 Learning Gradient Boosted Multi-label Rules

The goal of rule learning is to generate a set of formal rules, which can be used to label a given example. This form of labeling allows a human comprehension of what the algorithm does. [FGL12]
For this paper we have to develop a basic understanding of how boosted multi-label rule learning works. It is a special case of rule learning, where we want to predict multiple labels at once. In addition gradient boosting is used to improve our predictions. We look at the BOOMER algorithm described in source [Rap+20], as it is used as a baseline for this project. Most of the information in this section is covered in the Source [Rap+20].

2.1.1 Multi-label Classification

When we speak of classification, we usually talk about assigning *labels* to given unseen *examples*. These examples are also sometimes called *instances*. Let us look at a mathematical description of an example, before describing it more informally:
Any given example can be represented as a vector $x = (x_1, ..., x_K) \in \mathcal{X}$ with $\mathcal{X} = A_1, ..., A_K$, $1 < K \in \mathbb{N}$ and x_i is a value describing A_i.
This mathematical concept is roughly described in natural language as follows: An example consists of a list of values which represent an *attribute*. This could be for example the top speed, if we want to classify vehicles, or temperature, if we want to classify weather. An attribute or *feature* is a data point we measured beforehand. An example has multiple attributes and each attribute is resembled by a value. The last part is not necessarily always the case, if our data is sparse. Sparse data means there is a default value, which is not explicitly stored. An example for this is data, collected by a sensor, which only records data, if its state changes. [Sri18] This way there is one value for each change but for each time there was no change the default value is set. But since the binning would be a lot more complicated and would probably produce worse results, when accounting for sparsity, it was not looked into further.
Now that we know what an example is, let us look at the classes or labels, we want to assign to them. We want to assign several labels from a predefined set of labels $\mathcal{L} = \{\lambda_1, ..., \lambda_L\}$. So we associate each example with a label vector $y = (y_1, ..., y_L) \in \mathcal{Y}$ with $\mathcal{Y} = \{0, 1\}^L$ and the interpretation: y_i indicates, if label λ_i is assigned (1) or not (0). An example for labels would be genre classifications for movies. A movie could be labeled “Action”, “Drama” or “Thriller” or any combination of them. We could also go back and look at the example we introduced for attributes to differentiate between the multi-label and the single label approach:

When labeling vehicles we could have labels like “Truck”, “Sportscar” or “ATV”, but since we would only give one label to one example, this is called a single-label approach, we would not need a vector in this case.
The task is to find a model which can assign the correct label vector to a previously unknown example vector. To achieve this we train the model with a data set, where we know the correct labels for each example. This is called a supervised process, since we use the labeled data as a “teacher” and do not let the algorithm figure out patterns on its own.

2.1.2 Rules and Boosting

A *rule* consists of a *head* and a *body*. The body consists of a set of *conditions*, which are tested against attribute values[LJ] and they evaluate to true, if all conditions are true, otherwise they evaluate to false. The head, in our specific case, assigns a numerical score to each label. [Rap+20]
To produce a good model we obviously need to use a lot more rules than just one, since a single rule is rarely a good classifier. If a condition is true, we add the heads to our current prediction. This method of combining the results of a lot of weak classifiers, like our single rule, is an ensemble approach called boosting. [Zho12]
For binary or discrete labels the scores would be discrete, but since we use boosting, in this specific instance of a rule learning algorithm, we use real numbers for our scores.
The conditions are of the form $C_i = A_j R c$ with A_j like in the last section, is an attribute, $R \in \{\leq, >\}$ and $c \in \mathbb{N}$. The body as a set of multiple conditions is written as $\{C_1, ..., C_k\}$. The head is a vector with one entry per label $h = (l_1, ..., l_L)$.
Let us look at an example. A typical rule looks like this:

$$\{A_{97} \leq 0.2821 \wedge A_{22} > 0.0402 \wedge A_{17} \leq 0.8902\} \rightarrow (0.79, -0.35).$$

Examples which fulfill a rule are called *covered* by this rule.
Assuming there are only two labels, we would check the attributes A_{97}, A_{22} and A_{17} in their corresponding condition. If all the conditions evaluate to true, we would add the head $(0.79, -0.35)$ to our previous prediction.
This algorithm has an addition to deal with nominal values. We will look at it briefly, since our approach is not fit for these. Nominal values are values which have no specific order. An example of nominal values in our domain, would be values which represent weather, for example $0 = sunny$, $1 = rainy$, $2 = foggy$ and so on. The values represent bigger concepts, which can not be sorted and with which we can not do any kind of math. BOOMER uses these values to learn conditions, but only builds rules with the operators “$=$” or “$\neq$”, since checks for equality still work on numerical feature values.
There is still one detail missing: How can we generate an initial prediction to add to? The first rule we learn is always the default rule, which has an empty set for its body. This means the body is true for every example. It also includes a prediction for each label as its head. So we have at least a simple prediction for each label. There are unavoidably some labels, which the default rule predicts correctly, but we want to approve upon it to make predictions as accurate as possible for each example and label.

2.1.3 Thresholds

For constructing the conditions for a rule, several thresholds could be used. In order to learn a condition for a rule we need to sort the features. In a sorted feature list the borders between two distinct values are called *thresholds*. So the threshold t_n between x_n and x_{n+1} with $x_n, x_{n+1} \in x$ and x defined as in 2.1.1, would be $t_n = \frac{x_n + x_{n+1}}{2}$. Each threshold t_n is always as far away from the two original values as possible, while still

fulfilling the condition $x_n < t_n < x_{n+1}$, since we only consider thresholds between distinct values, which by definition cannot be equal.
We can continue our weather example. We used a sensor to record temperatures. It produced the following array of temperatures:

-1	4	8	9	12	19	21	26

We can calculate the threshold between the first two values with the formula we introduced, by inserting the corresponding values: $t_0 = \frac{-1+4}{2} = 1.5$. We could calculate all other thresholds the same way.
To learn a rule we look at those thresholds. We check if it is more beneficial for our model to predict something, when a given feature value is "less or equal" or "greater than" the threshold. This is done by determining the optimal scores to be predicted by the resulting rule and assessing its quality according to the loss function.
In our example we would look at our threshold $t_0 = 1.5$, so we check the parameter temperature against it. We will probably see that a temperature ≤ 1.5 increases the chances of the label "snowy" significantly, so we would chose it as a condition.
There are $n - 1$ thresholds, if n is the number of distinct examples. In our temperature list there would be 7 thresholds in total, since there are 8 examples. To confirm this intuition, we first look at one value. There is no other value so we can not build a threshold, so there are 0 thresholds for $n = 1$. This was expected since $1 - 1 = 0$. If we add another distinct value so that $n = 2$, then we can build a threshold between them. We would have $1 = 2 - 1$ thresholds. This is true, whenever we add a distinct example, so there are $n - 1$ thresholds for n as the number of distinct feature values.
For each threshold there are two possible conditions and each new example introduces a new threshold to each feature, as long as these values are not already in the feature list. If they were, the value in question would not be distinct and would therefore not produce a new threshold. While scaling linearly with all those factors, this can still introduce a problem, if we use a lot of examples with a lot of features, so our goal is to reduce the number of possible conditions.

2.1.4 Filtering

After we learn a condition, the feature values, which fulfill the condition, are called *covered* by this condition. We remove the examples, which belong to the corresponding feature values. After removing those values, the environment has changed and we guarantee that the next condition, we learn, is a new one.
This sounds fairly simple, but if we look into the structures we described, this gets a lot more complex. In 2.1.3 we noted, that in order to function properly the feature vectors have to be sorted. So the feature values from our original example x_i are not in the position with index i of each feature vector. So to filter out a covered example we have to iterate over each feature vector, to search the corresponding value form our example.
Even though this operation has a time complexity of $O(instances * features)$ the first time around and becomes faster with each already filtered example, it can use a lot of computation time, as this also becomes slow with big data sets. So we have to be especially careful with the implementation of the filtering, since this method has to be used for every condition in the resulting model.
Let us look at the weather example again. We recorded the humidity (feature1) and the wind speed (feature2), as well and got this result:

	ex0	ex1	ex2	ex3	ex4	ex5	ex6	ex7
feature0	-1	4	8	9	12	19	21	26
feature1	60	10	90	45	25	20	10	5
feature2	10	0	5	15	20	5	10	20

The first step would be to sort the values, which means we lose the assignment to the examples. It would look like this:

feature0	-1	4	8	9	12	19	21	26
feature1	5	10	10	20	25	40	60	90
feature2	0	5	5	10	10	15	20	20

We learn the same condition as before, so we learned a condition on the basis of feature0 of example0. Now we need to filter example0, so we need to find the rest of its feature values in the table. We need to find the value 60 in feature1. We have to check six values until we find it in the seventh spot. In feature2 we need to find the value 10, we find it on our fourth step and filter it. We do not care which 10 we filter as long as we filter the value corresponding to the example. So we finished our filtering process.

feature0	**X**	4	8	9	12	19	21	26
feature1	5	10	10	20	25	40	**X**	90
feature2	0	5	5	**X**	10	15	20	20

2.2 Related Work

The reduction of possible conditions proved to be an effective method. It is not an uncommon field of study, since the number of possible conditions is one of the worst scaling parts in algorithms like BOOMER, which build large ensembles. Furthermore this problem is not limited to rule learning algorithms, but also effects tree boosting algorithms, like the popular XGBoost[Vis19] and LightGBM[Sha18], which makes research in this area relevant. We will look at Split Finding, which inspired the idea of example binning, Weighted Quantile Sketch and other supervised methods, which also take the quality of our prediction into account.

2.2.1 Split Finding

Split Finding is a part of decision tree learning. It is relevant here, because tree learning is comparable to rule learning. To put it casually, rule learning is tree learning, if we only consider one branch of the tree. Each node along this branch is one condition and we reach the leaf, if all notes in the branch evaluate to true.
Split Finding describes, how to find the cutoff between two branches. So we could compare the split point of a tree to the threshold of a rule. This is one of the most time consuming tasks in tree boosting algorithms. [Sha18]
We look at three methods for Split Finding[Yil19]:

- **Pre-sorted**: Every possible split point in a sorted feature list is evaluated
- **Histogram-based**: Continuous feature values are sorted into bins
- **Entropy-based binning**: Finds the most pure bins, which means that the majority of values in the bin correspond to the same label.

The Pre-sorted algorithm is comparable to what we already have. The possible split points are the thresholds of our implementation and we check each one. So this method is not especially interesting for our goal.
The Histogram-based Split-Finding is a little more complex as the Pre-sorted Split-Finding. We define a maximum number of discrete values per feature we want to look at. Then the values are grouped together into a number of bins, which is equal to the given maximum number. [Bah21] We will look at two possible implementations of how we can handle this binning in more detail in Chapter 3.
These two approaches are called *unsupervised* methods. We call every method which has no clear target values and uses no reward system unsupervised. The algorithm should find patterns by itself. [HS99]
A *supervised* approach would be Entropy-based binning, since we calculate a so called entropy based score on each class label, which is information we have to reintroduce to the algorithm. This means we interfere, which makes the algorithm supervised.[Say21a]

2.2.2 Gradient One-Side Sampling

The gradient of a given feature value gives us important information about how well we can already predict it. A small gradient means we can make a good prediction for the associated value, on the other hand a large gradient means we can not make a good prediction. This is where GOSS comes in, it uses four steps to find better splits with this information[Yil19]:

1. Sort the data example-wise by the absolute value of their summed gradients in descending order
2. Select the top $a * n$ examples
3. Select $b * n'$ random examples from the not selected examples
4. Multiply the sample from the examples with small gradients by $\frac{1-a}{b}$

The following conditions apply: $0 \leq x \leq 1 : x \in a, b : a, b \in \mathbb{N}$, n is the total number of examples and n' is the total number of remaining examples. The relation $n' = n - (n * a)$ applies, as well.
This is a lot more complicated so we look into what happens here. In the first step we generate a list of our examples. By sorting we guarantee that the first example in the list is the one where our prediction was the worst, the last example is the one we can predict best and all other values are distrusted between them accordingly.
In the second step we take a subset of all of our examples. By varying the a value, we can vary how much of this set is considered bad enough to focus on.
In the third step we reintroduce some of the cut examples. We can already make better predictions for them, but we should not just ignore them. To reduce focus on them we only put a random subset based on b of them back in. By varying b we can change how much we want to put back in.
In the last step we introduce an additional weight to soften the impact of the previously cut examples. The equation $\frac{1-a}{b}$ evaluates to a bigger value, if both values are small, since we do not have a lot of examples in this case, we should not dilute them further. If both values are bigger the equation evaluates to a smaller value, since the examples we cut previously do not matter as much. The equation is more sensitive to changes of b than a.
So by manipulating the two values a and b we can vary the focus between well known examples and lesser known examples. [Bah21]

2.2.3 Exclusive Feature Bundling

We looked at how we can summarize examples, but the number of features is also a deciding factor. [Sha18] The basic idea of EFB is to reduce features by combining mutually exclusive features, which means they do not have non-zero values at the same time. [Yil19]
Let us look at the following example, while explaining what EFB does:

feature1	feature2	feature3	feature4	feature5
0	1	0	3	0
0	2	2	0	0
3	0	0	0	5
1	0	6	0	0

We see intuitively by the definition feature1 and feature2 are mutually exclusive, as well as feature3 and feature4.
EFB uses three steps to find these relations. First it calculates a conflict measure for each combination of features or bundles. This is done be dividing features with overlapping non-zero values by the total number of mutually exclusive features in the bundle. This measure gets bigger with the number of conflicts. Secondly it sorts the features by the number of non-zero instances, beginning with the feature with the least zero instances and going to the feature with the most zero instances. Lastly EFB loops over the generated list and assigns features to existing bundles, if the conflict measure is small enough and it is put in a newly created one, if the measure is too large. [Sha18]
The merging process is more intuitive. We calculate an offset, as the span of values in a bundle and add it to the values, which we want to add. This is necessary since we want to bundle the features, but still differentiate between them. [Sha18]
So let's look at our example step by step. There is no need to sort them since they already are in our desired order.

- **Feature1** creates a new bundle "bundle1", since there is no bundle, yet. At this point bundle1 looks like this $0, 0, 0, 0$, so the span is 0, since we don't have two distinct values. We don't need to add anything to the values of feature1 and we merge it into the bundle. After this step the bundle is as follows: $0, 0, 1, 3$.
- We look at **feature2** which has the conflict measure 0 to bundle1. This is small enough to add it. So we calculate the span of the bundle $3 - 0 = 3$ so our offset is 3. After adding feature2 with its modified non-zero values, we get $4, 5, 3, 1$ for our bundle1. Note that feature1 and feature2 contained a 1, but this feature value is now distinguishable through the addition of the offset.
- Now we check for **feature3**. It has a conflict measure of 0.5. This is not good enough and we create a new bundle, "bundle2", the same way we did for bundle1 and get $0, 2, 0, 6$ as our new bundle.
- **Feature4** has a conflict measure of 0.25 with our first bundle and 0 with our second bundle, so we add it to bundle2. Our bundle2 has a span of $6 - 0 = 6$ so we add 6 to all non-zero values and add those to our bundle. This results in $9, 2, 0, 6$ for bundle2.
- **Feature5** has the same conflict measures as feature4, so we want to add it to bundle2, as well. This time the offset is 9 since our span has increased. So the result for bundle2 is $9, 2, 14, 6$.

This way we summarized five features into two bundles. The following table summarizes the results.

feature1	feature2	feature3	feature4	feature5	bundle1	bundle2
0	1	0	3	0	4	9
0	2	2	0	0	5	2
3	0	0	0	5	3	14
1	0	6	0	0	1	6

2.2.4 Weighted Quantile Sketch

Weighted Quantile Sketch (WQS) is a supervised method to speed up split-finding, while minimizing accuracy loss. It was first introduced in the original XGBoost paper[CG16], the following information originates from this paper.
WQS basically divides bins by combined weights instead of number of entries or value ranges. This section gives a quick overview of this algorithm and focuses on the description of the core functionality.
In the following examples we will use the data set:

$$\mathcal{D} = \{(x_0, w_0), (x_1, w_1), (x_2, w_2), ...(x_n, w_n)\} \text{ with entries } x \text{ and weights } w.$$

Weights and Ranks

As a first step we need to give each entry in the data set a numerical value, called weight, which corresponds to how good our prediction of this given value is. The weight has to be large for an entry, which has no good prediction, and has to get smaller as the prediction gets better. A good example for a function which fulfills these conditions is the loss function $\mathcal{L}$.
Now that we have weights, we define ranks, which represent the combination of the weights of all previous entries. WQS uses these ranks to divide the data set into quantiles.

$$r_{\mathcal{D}}^{-}(y) = \sum_{(x,w)\in\mathcal{D}, x<y} w$$
$$r_{\mathcal{D}}^{+}(y) = \sum_{(x,w)\in\mathcal{D}, x\leq y} w$$

$r_{\mathcal{D}}^{-}(y)$ is the rank of y without its own weight, $r_{\mathcal{D}}^{+}(y)$ is the rank of y with its own weight added. We also define $w(\mathcal{D})$ as the sum of all weights of $\mathcal{D}$, as such

$$w(\mathcal{D}) = \sum_{(x,w)\in\mathcal{D}} w.$$

A useful mental image for ranks is a stack of all previous weights with or without respectively the current weight as seen in 2.1.
It also helps to look at an example. Let's define a really simple example:

$$\mathcal{D}_{ex} = ((x_0, 1), (x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1), (x_5, 1)) \tag{2.1}$$

with data points x_n and weights $w_n = 1$. Note that you would not use WQS for such a small data set, because Pre-sorted split finding would be fast enough and more precise. This example would produce the following ranks as seen in 2.2:

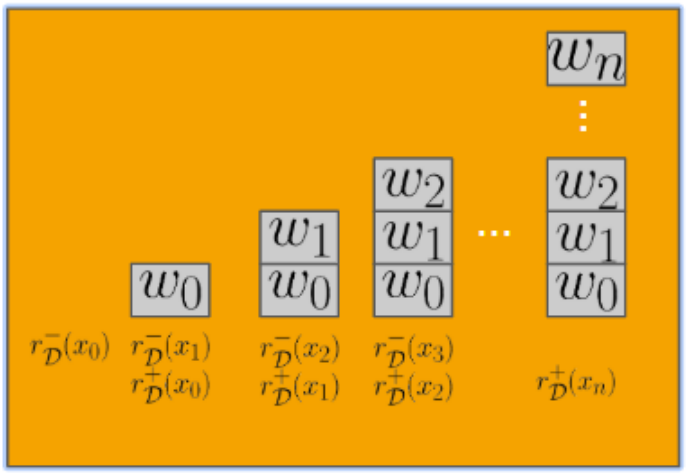


Figure 2.1: Image: Graphic Representation of Ranks

$$
\begin{aligned}
r^-_{\mathcal{D}_{ex}}(x_0) &= 0,\\
r^+_{\mathcal{D}_{ex}}(x_0) = r^-_{\mathcal{D}_{ex}}(x_1) &= 1,\\
r^+_{\mathcal{D}_{ex}}(x_1) = r^-_{\mathcal{D}_{ex}}(x_2) &= 2,\\
r^+_{\mathcal{D}_{ex}}(x_2) = r^-_{\mathcal{D}_{ex}}(x_3) &= 3,\\
r^+_{\mathcal{D}_{ex}}(x_3) = r^-_{\mathcal{D}_{ex}}(x_4) &= 4,\\
r^+_{\mathcal{D}_{ex}}(x_4) = r^-_{\mathcal{D}_{ex}}(x_5) &= 5,\\
r^+_{\mathcal{D}_{ex}}(x_5) &= 6
\end{aligned}
$$

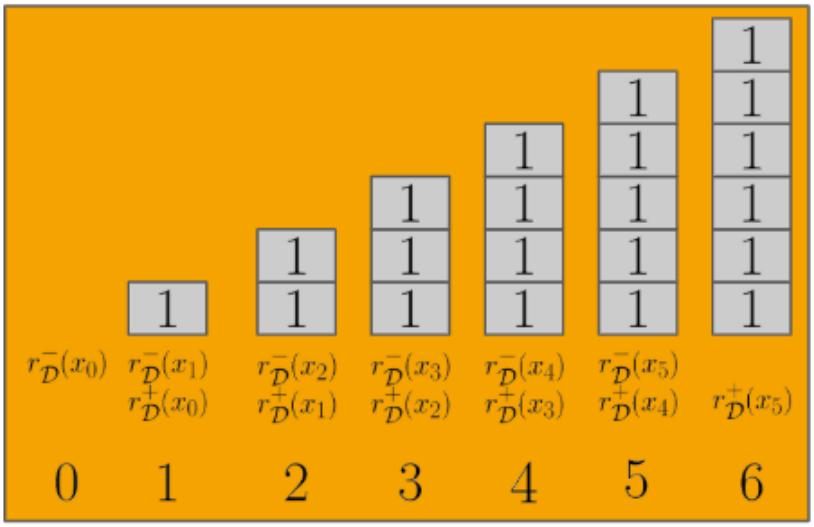


Figure 2.2: Image Graphic Representation of the Example

Query Function

The Query Function $g(\mathcal{D}, d)$ (1) takes in the data set $\mathcal{D}$ and numerical parameter d. d is the weight increment, we are searching for. The function returns the data point with the rank closest to d. To understand this function it is useful to imagine a number line from 0 to $w(\mathcal{D})$ with the weights as increments.

Algorithmus 1 : Query Function $g(\mathcal{D}, d)$

Input: $d : 0 \leq d \leq w(\mathcal{D})$
Input: $\mathcal{D} = \{(x_0, w_0), (x_1, w_1), (x_2, w_2), ..., (x_n, w_n)\}$
if $d < \frac{1}{2}(r^-_{\mathcal{D}}(x_0) + r^+_{\mathcal{D}}(x_0))$ **then**
  return x_0;
end
if $d \geq \frac{1}{2}(r^-_{\mathcal{D}}(x_n) + r^+_{\mathcal{D}}(x_n))$ **then**
  return x_n;
end
Find i such that:
$\frac{1}{2}(r^-_{\mathcal{D}}(x_i) + r^+_{\mathcal{D}}(x_i)) \leq d < \frac{1}{2}(r^-_{\mathcal{D}}(x_{i+1}) + r^+_{\mathcal{D}}(x_{i+1}))$;
if $2d < r^-_{\mathcal{D}}(x_i) + w_{\mathcal{D}}(x_i) + r^+_{\mathcal{D}}(x_{i+1}) - w_{\mathcal{D}}(x_{i+1})$ **then**
  retutn x_i
else
  retutn x_{i+1}
end

We look at Algorithm 1. First the algorithm checks, if d is smaller than the first rank or larger than the largest rank and returns one of them accordingly. These edge cases have to be dealt with first, because the following code has no way of handling these due to a lack of predecessor or successor respectively. Then we search i such that the rank of x_i is less or equal to d and such that the rank of its successor x_{i+1} is greater than d. In the final step, we have to decide which of our candidates, x_i and x_{i+1}, is closer to d to return it.
For example if we want to find a data point in our simple example set $\mathcal{D}_{ex}$ from 2.1 with a rank close to 3.2, we would call $g(\mathcal{D}_{ex}, 3.2)$. We walk through this example step by step:

1. We check if 3.2 is in the weight area of x_0: $3.2 < \frac{1}{2}(0 + 1)$ which is $false$, so we go to the next step.
   - If it were $true$, we would return x_0 and terminate the algorithm
2. Now we check if 3.2 is in the weight area of x_5: $3.2 \geq \frac{1}{2}(5 + 6)$ which is also $false$ and like before we go to the next step.
   - Also like in the last step, we would return the corresponding data point x_5, if the check returns $true$.
3. The algorithm searches for a data point, such that d is between its rank and the rank of its successor. In our case this would be x_2
   - Proof: $\frac{1}{2}(r^-_{\mathcal{D}_{ex}}(x_2) + r^+_{\mathcal{D}_{ex}}(x_2)) = \frac{1}{2}(2 + 3) = 2.5$ and $\frac{1}{2}(r^-_{\mathcal{D}_{ex}}(x_3) + r^+_{\mathcal{D}_{ex}}(x_3)) = \frac{1}{2}(3 + 4) = 3.5$, with $2.5 \leq 3.2 < 3.5$
   - This step is a black box were we could use any search algorithm
4. Now we check if 3.2 is in the weight range of x_2: $2 \times 3.2 < (2 + 1) + (4 - 1) \rightarrow 6.4 < 6$ so this is also $false$, so we return x_3 as a result and terminate the algorithm.
   - As before, if it were $true$, we would return x_2

3 Unsupervised Example Binning

To improve the scalability of the algorithm, we want to group some examples into one bin. This way we expect to reduce the number of possible conditions. So we try to find subsets of similar examples to group together. If we group just any examples, we would not only lose a lot of precision, but we would have to change the process with which the thresholds are calculated. The following two solutions are called unsupervised binning methods [Say21b], the simplest approach to find similar values for each bin. Since they are only binning one time and do not change the assignments afterwards, they should be more run-time efficient than the more complicated algorithms introduced in the previous chapter.
In this chapter we look at everything we need to know for our implementation.

3.1 Equal-Frequency Binning

When we look at a sorted list of distinct numbers and pick any of them, the one which came before is always the biggest value which is less than the picked one. In the same way the next value will be the smallest value bigger than the current one. The Equal-Frequency approach uses this fact and looks at the number of occurrences, to determine which values belong together. [Say21b; Pod20]
We group a specific amount of values x in each bin and create n bins with $n, x \in \mathbb{N}$. The idea is, when we know x, we can just iterate over the sorted list and put x values together in one bin, x values in the next one and so on. In theory we would get n bins with this approach. In reality there are less than n bins most of the time, since we need natural numbers for x and n, and they can not be divided accurately enough.
With $total$ for the total numbers of values in the feature list, we can apply the following equation to determine the number of examples per bin x:

$$x = \left\lceil \frac{total}{n} \right\rceil \tag{3.1}$$

Note that we are rounding the solution up. The first obvious reason is, we can only add a natural number of examples to a bin. The second reason will get obvious, when we look at the implementation later on in Chapter 4.
If we look at a feature value list: $23, -2, 6, 1, 1, 3, 5, 4, 12, 6, 9, -1, 0$ and want to generate five bins we will get:

$$\left\lceil \frac{13}{5} \right\rceil = 2,6 = 3 \tag{3.2}$$

So we get three examples per bin. Also we would get a result like shown in image 3.1.
Note that we sorted the list before grouping the values in bins. We also put equal values in the same bin, even if we have to put more examples in one bin. Since we want to group similar values, we have to avoid separating equal values, because they are so similar that they are indistinct.

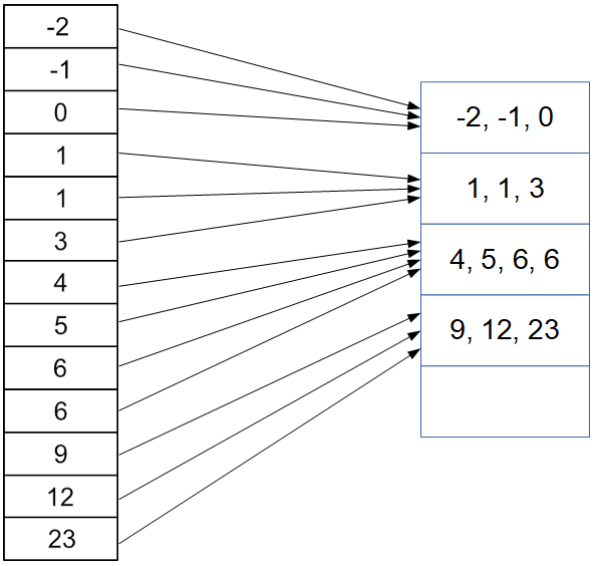


Figure 3.1: Equal-Frequency Binning: Bin placement example

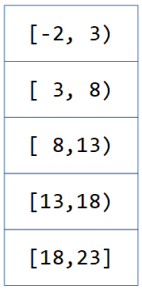
As you can see we stop if there are no values left. In the example the last bin stays empty, because of that. In practice this is no problem and it would be more complex to circumvent this phenomenon. Since we picked this algorithm for its simplicity, we want it to stay as simple as possible, while guaranteeing its functionality. It is also possible, that the last filled bin is not completely filled. If for example the 23 was missing here, the last bin would only contain $9, 12$, so two values instead of the expected three. This is also no problem in terms of functionality, so we keep it as is, too.

3.2 Equal-Width Binning

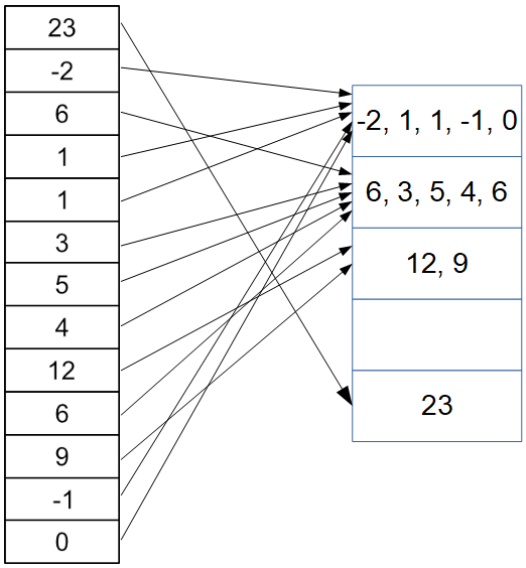
We can get a lot more direct with the grouping of values with a common attribute. For the Equal-Width Algorithm we define a span or range of values for each bin. Every example, which is covered by the range, will be put in this bin. [Say21b; Pod20]
We do not need a sorted list of examples for this method, but we need to know the lowest and highest example value. This can be found out easily by going through the list, while memorizing the smallest and biggest value. This is a lot less complex than sorting the list. Finding the minimum and maximum has a time complexity of $O(n)$, since we have to look at each value exactly once, while the most efficient sorting algorithms have a time complexity of $O(n * \log n)$. [Knu98]
First we determine the range of values over all examples, using the searched two values, then we can divide that by the total number of bins we want n, with n defined as previously. The result is the range per bin. Note that we do not even need to know, how much examples there are. This relation is represented by this equation:

$$range = \frac{|maximum - minimum|}{n} \tag{3.3}$$

We need the absolute in the denominator, because, if all our values are negative, we would produce a negative range, which would lead to false results. We would start to calculate the range for the first bin with the smallest value and subtract from that. So there would be no value in this range expect the smallest one. All



(a) Equal Width Binning: Span of the bins



(b) Equal Width Binning: Bin placement example

Figure 3.2: Equal-Width Examples

other bins would be completely empty, since there are no values less than the smallest one, but we would only consider such values.
If we look at the same list as before: $23, -2, 6, 1, 1, 3, 5, 4, 12, 6, 9, -1, 0$ and want to generate the same amount of bins. Our range would be:

$$\frac{|23 - (-2)|}{5} = \frac{25}{5} = 5 \tag{3.4}$$

We would generate bins with the spans shown in example 3.2a. Please note that we made the last border inclusive to include the highest value of the example list. Of course we have to make this exception in our implementation, as well.
We group the values like shown in example 3.2b. The number of examples per bin can vary a lot. For example a feature list with 18 values between 0 and 1 exclusively and a single 0 and a single 10, would generate one bin with 19 values, one with one value and 8 empty ones for $n = 10$. This example is a bit extreme, it is not expected to occur often, but it is not impossible, so we need to consider it. And even in less extreme cases there can be empty bins between filled bins, this is something we have to keep in mind for the implementation. But as far as the binning itself is concerned, this is no problem.
In the implementation we obviously can not intuitively decide in which of the calculated ranges a given value belongs. The following function returns the index of the bin i for a given feature value x with the span s and the minimal value m:

$$i = \left\lfloor \frac{x - m}{s} \right\rfloor \tag{3.5}$$

4 Implementation

In this chapter we go over the changes and additions, we made to the base algorithm, we described in Section 2.1. We start with the data structure, which will contain the bins, and the binning observer, which fills the data structure while binning.
Then we will look into the practical implementation of the concepts, we depicted in Chapter 3. In other words the actual binning.
This chapter concludes by showing the implementation of the filtering process for our new data structure. We will focus on the two types of filtering, dynamic and static filtering, and the two actual filter functions.

4.1 Data Structures

To add the histogram-based approach to the existing algorithm without influencing other options, we need a new data structure, which contains the bins and supplies the information, that existing functions need, as if we had a normal data set.
Also there has to be an algorithm, which handles the creation of the bins, since we do not want to handle this in the binning functions. When we learn in which bin a feature value belongs, we notify the binning observer. The binning observer puts the value in the bin and updates the information in the bin to reflect the new value.

4.1.1 Datatype Bin

To store a bin, its structure does not actually need to access the values of all examples, which are assigned to it, for most of the run-time. We only need to know all feature values when we filter the examples. Usually we only have to access the smallest and the largest value, as well as the numbers of examples in each bin. The reason for this is that we calculate the thresholds between two bins as the mean between the maximum value of a bin and the minimum value of its successor. It can be noted as

$$t_n = \frac{max(x_n) + min(x_{n+1})}{2}.$$

The two bins, we consider for a threshold, are always adjacent to each other. A bin x_n is therefore considered for t_{n-1} and t_n, as long as it is not the first or last bin for a given feature vector. In these cases we would only consider them for t_n or t_{n-1} respectively. So we store the highest and lowest value separately in our data structure to access them easily and quickly without having to look through the entire list of feature values.
The data structure `Bin` contains the `maxValue`, the `minValue`, which are stored as floating point numbers, the `numExamples`, which is stored as an integer, and, for the sole purpose of filtering, a linked list containing the examples, called `Examples`.

The Image 4.1 provides a graphical representation of two adjacent bins. If the number of examples is zero after the binning, we ignore and remove the bin.

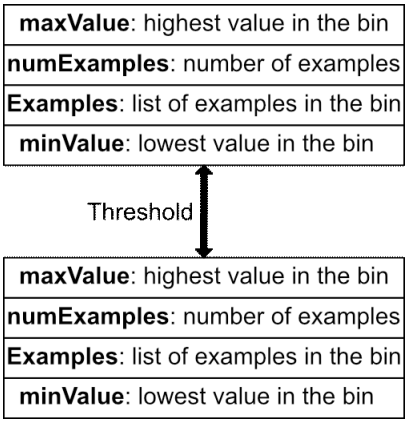


Figure 4.1: Bin Structure

4.1.2 Binning Observer

To fill the bins, we use an observer, which is notified every time we determine in which bin an example belongs. The observer fills a vector, which holds all possible bins, with the corresponding examples. It also simplifies our binning, since it handles the data structure by increasing the number of examples and setting the current minimal value, as well as the current maximal value.
Algorithm 2 describes the functionality of the observer. It starts by taking in the index of the bin, where the

Algorithmus 2 : Bin Update $f_{bin-update}(index, exampleIndex, exampleValue)$

Input: $index, exampleIndex, exapmleValue$
bin = $getBin(index)$
bin.numExamles++
if *bin.maxValue* $<$ *exampleValue* **then**
  bin.maxValue = exampleValue
end
if *exampleValue* $<$ *bin.minValue* **then**
  bin.minValue = exampleValue
end
append $Example(index)$ to $Examples$

feature value should be put in (`index`), the index of the example (`exampleIndex`), to which the feature belongs, and the actual feature value to store (`exampleValue`). The two if-conditions check the incoming value against the stored minimal and maximal value respectively. We initialized the maximum value as the smallest possible value and the minimum value as the largest possible value of their data type, this way we do not need to check anything else, since the first value will override both of them naturally.

4.2 Binning Algorithms

In this section we will look at the practical implementation of the two unsupervised binning methods, Equal-Frequency Binning and Equal-Width Binning, which where introduced in Chapter 3. We will also list some design decisions and evaluate them by giving some alternatives.

4.2.1 Equal-Frequency Binning

Algorithmus 3 : Equal-Frequency Binning $f_{eq-fr}(n, \mathcal{D})$

Input: $n : 0 < n \leq m$: Number of Bins
Input: $\mathcal{D} = \{x_0, x_1, x_2, ..., x_m\}$: Feature Vector
$sort$ Feature Vector
numElementsPerBin = $\lceil m/n \rceil$
binIndex = 0
previousValue = 0
for $i \leftarrow 0$ **to** m **by** 1 **do**
  currentValue = $value\_of(x_i)$
  if *previousValue* $\neq$ *currentValue* **then**
    binIndex = $\lfloor i /$ numElementsPerBin$\rfloor$
  end
  previousValue = currentvalue
  notifyObserver(binIndex, $index\_of(x_i)$, currentValue)
end

Algorithm 3 describes the Equal-Frequency Binning method. As described previously our feature list has to be sorted, to use this method. So we start by sorting the feature vector. The variable `numElementsPerBin` is calculated with Equation 3.1. We talked about rounding up as a method to ensure that we generate a whole number. It is also used to ensure that the indexes, we generate here, are valid. To show this we can prove that rounding down to the next whole number would be insufficient by a contradicting example. If we have a feature vector containing $0, 1, 2, 3, 4, 5, 6$ and we want 3 bins, the equation would look like this $num = \frac{7}{3} = 2,3333$. To demonstrate the problem we round the number down. So we want to put 7 elements into 3 bins by putting 2 elements in each bin. This does not work, we have six spots for seven values. If we round up we get 3 bins with 3 elements each and can fit all elements with free spaces after the last value.
There are other solutions, like adding an extra bin for overflowing values. But this introduces the problem, that we would generate more bins, than we want. Another solution would have been to put all values, which would overflow, in the last bin, like we do in Equal-Width binning. In the worst case the last bin would contain $2n - 1$ elements for n being the number of elements in any one of the other bins. This would be a worse solution as well, since our last bin, together with the first one, are the only bins which contribute to only one threshold. Every value, which comes after the first one in the last bin, is only relevant, if all smaller values are filtered out, so we want to avoid making this bin bigger than the others. There is one other design which could be used. We could calculate the number of overflowing elements $n_{oe} = total \mod n$ and add a single example to the first n_{oe} bins. This solution would produce good results, as well, but was cut in favor of the simpler solution.
We also have to set the variable for the previous value to 0. Obviously the value of x_{-1} does not exist, but 0 is

a work around, which works like we intend it to. In the for-loop we check if the previous value is the same as the current value, if that is the case we would like to put it in the same bin. We want to have all equal values in the same bin, otherwise we would have two adjacent bins, where the maximum value of the bin with the lower index is the same as the minimum value of the bin with the higher index. This would generate this exact value as a threshold. A value within the set is a worse threshold than one between two values, because we can not differentiate between the two values, if they are equal.
Now we can look at the reason, why we choose 0 for our first previous value. We have to look at the following two cases:

1. *The first value* $x_0 = 0$: The if-condition evaluates to `false`, so we skip the index calculation and put x_0 in the bin with index 0, since we did not change `binIndex` from its initial value 0, yet. The first value belongs in this bin. The bin must have 1 or more spaces, since we didn't fill anything in and the number of examples per bin has to be greater than 0.
2. *The first value* $x_0 \neq 0$: The if-condition evaluates to `true`, so we calculate the index of the bin, we want to put our example in. Because we are looking at x_0, i has to be 0. 0 divided by any value is still 0, so we put the first example in the bin with index 0 as expected.

After that we set our previous value to our current value and the next iteration works as intended. We could have used any value as the previous value in the first iteration, but 0 seems like the most natural solution.
We glossed over the calculation for the bin index, but this is an important step, as well. In Chapter 3 we said we want to iterate over the list and fill the first bin with elements corresponding to the calculated number of examples per bin. After that we want to do the same for the next bin and for each one coming after, until we have no examples left, while keeping in mind that equal values always have to end up in the same bin. For the mathematical explanation we will call the bin index $x(i)$, since we want to calculate it in relation to i. The number of elements per bin will be called n. i goes from 0 to the total number of elements, so this iterates through the index of each value in the feature vector. We use an example with a feature vector with 7 elements, which do not overlap, the feature values do not matter in the calculation, except if two or more values are equal. In this example we want 4 bins, this means we want 2 examples per bin. The indexes are calculated as such:

$$
\begin{aligned}
x(0) &= \left\lfloor \tfrac{0}{2} \right\rfloor = 0 \\
x(1) &= \left\lfloor \tfrac{1}{2} \right\rfloor = 0 \\
x(2) &= \left\lfloor \tfrac{2}{2} \right\rfloor = 1 \\
x(3) &= \left\lfloor \tfrac{3}{2} \right\rfloor = 1 \\
x(4) &= \left\lfloor \tfrac{4}{2} \right\rfloor = 2 \\
x(5) &= \left\lfloor \tfrac{5}{2} \right\rfloor = 2 \\
x(6) &= \left\lfloor \tfrac{6}{2} \right\rfloor = 3
\end{aligned}
$$

By looking at these indexes, we see there are two values for the first, second and third bin, and one for the last one. We use the property of the integer division that there are exactly n natural numbers x which can be divided by n to get an arbitrary and fixed result y.
This way each value is put in the right bin and we stop, if there is no example left. Note that we will increase i, even if the value is put in the same bin as the previous value, this could lead to empty bins between filled

ones, if there are a lot of equal values in a given feature vector. This is no concern, since we already handle empty bins by removing them.

4.2.2 Equal Width Binning

Algorithmus 4 : Equal-Width Binning $f_{eq-w}(n, \mathcal{D})$

Input: $n : 0 \leq n \leq m$: Number of Bins
Input: $\mathcal{D} = \{x_0, x_1, x_2, ..., x_m\}$: Feature Vector
max = $max\_value\_of(\mathcal{D})$
min = $min\_value\_of(\mathcal{D})$
spanPerBin = (max−min)/n
for $i \leftarrow 0$ **to** m **by** 1 **do**
    currentValue = $value\_of(x_i)$
    binIndex = $\lfloor$((currentValue−min)/spanPerBin)$\rfloor$
    if *binIndex* $\geq n$ **then**
        binIndex = $n - 1$
    end
end

In this method the same value always ends up in the same bin, because we calculate a span, or range, of values for each bin and equal values belong to the same span. We are not settled on a specific amount of examples per bin. Our span per bin is calculated by the span of all values divided by the amount of bins we want. This is represented through the initial three lines of pseudo code in Algorithm 4. Now we iterate over all feature value indexes starting with 0 and going up to the total number of examples. We fetch the value of the feature on the index we look at. Now we can calculate the corresponding indexes. We look at the simple example $4, 5, 5, 1, 3, 2, 8$ and we want 4 bins. The span s is calculated as such:

$$s = \frac{8-1}{4} = 1.75$$

Note that we do not need to round this value, since we are not operating on indexes but values, these can be real numbers and are not limited to integers. To show how the for-loop determines the bin index, we use $f(i)$ as the function that returns the bin index for the value at feature vector index i with the span 1.75 and the minimum 1, which we have already calculated.

$$\begin{aligned}
f(0) &= \left\lfloor \frac{4-1}{1.75} \right\rfloor = 1 \\
f(1) &= \left\lfloor \frac{5-1}{1.75} \right\rfloor = 2 \\
f(2) &= \left\lfloor \frac{5-1}{1.75} \right\rfloor = 2 \\
f(3) &= \left\lfloor \frac{1-1}{1.75} \right\rfloor = 0 \\
f(4) &= \left\lfloor \frac{3-1}{1.75} \right\rfloor = 1 \\
f(5) &= \left\lfloor \frac{2-1}{1.75} \right\rfloor = 0 \\
f(6) &= \left\lfloor \frac{8-1}{1.75} \right\rfloor = 4 \Rightarrow 3
\end{aligned}$$

We reduced the calculated index of the biggest example by 1 to avoid addressing a fifth bin. We talked about this in Chapter 3, the last range has to be inclusive, while all other ranges have to include the lower border and exclude the upper border, else we would have to include a value on the border in two bins. This exception is made by the if-condition, following the index calculation.
We will always produce this overflowing index for exactly the highest value in a feature matrix. Through changing the equation, it becomes clear why. We replace the current value with the largest value max, since this is the case we want to examine. We use s again as symbol for span per bin, n is the number of bins we want, $f(i)$ represents the function, which calculates the bin index from the example index again, and i is the index of the highest value.

$$
\begin{aligned}
f(i) &= \frac{max - min}{s} && |s = \tfrac{max-min}{n} \\
f(i) &= \frac{max - min}{\frac{max-min}{n}} \\
f(i) &= \frac{(max - min) * n}{(max - min)} && = n
\end{aligned}
\tag{4.1}
$$

Since our index starts at 0 the n-th bin's index is $n - 1$, so n is never a valid index, even though it will always be assigned to the highest value. Like mentioned before, we just set it to the biggest index $n - 1$.

4.3 Filtering

We have to ensure that our environment, the data we use, changes to learn a new condition each time, else we would have the same data in every iteration and we would produce the same condition every time. We do ensure this change by filtering examples, which are covered by a rule. But, through binning, the feature values for one example could end up in multiple bins with entirely different indexes. So if we learn a rule, which covers the first two bins of one feature value, we can simply delete those two bins from this feature list. But we also have to remove all the examples they contain. We have to find all other feature values of these examples in all other bins. Image 4.2 shows schematically how this distribution looks in the data structure, if we just learned a rule based on the first two bins of the first feature.
We expect the filtering of the bins to take, as long as, if we had not binned at all. Since we have to go through each bin to look for each feature value of a given example, which is covered, the time complexity should be $O(instaces * features)$ for the first iteration and should get faster with each iteration, since we filter out examples. This is equivalent to filtering without binning, see Subsection 2.1.4.

4.3.1 Dynamic and Static Filtering

By removing feature values from a bin, we could change the minimum or the maximum value of a bin. Two different approaches to handle this event were implemented, referred to as the dynamic and the static filtering respectively.

Dynamic Filtering This method is called dynamic, because we *dynamically* adjust the minimum and maximum value of any given bin, once the feature value, corresponding to one of these values, is removed. For example, if we have a bin containing $1, 2, 3, 4$ and remove the example, which contains the feature value 4 we would

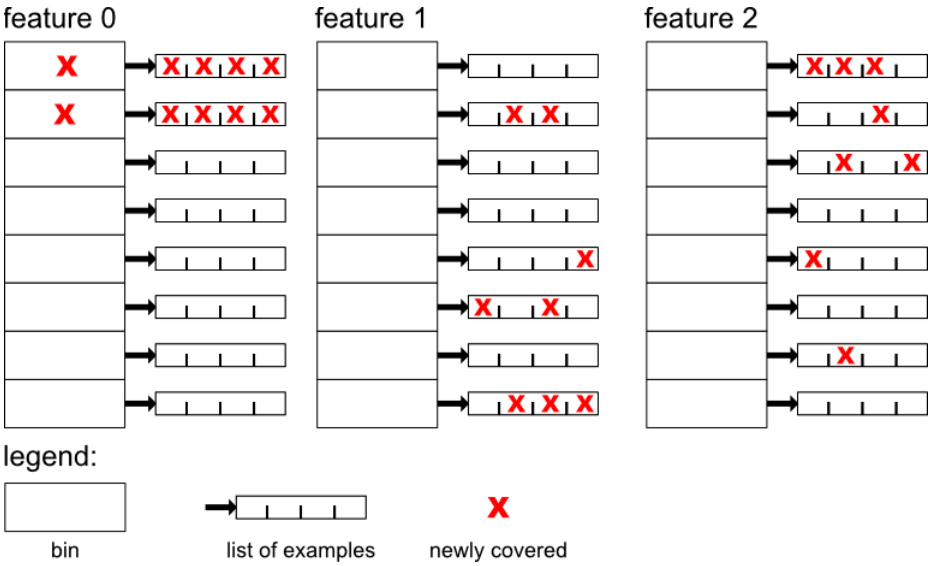


Figure 4.2: Filter Example

set the maximal value from 4 to 3. This way all bins always have the correct boundaries. Because of this it is expected, that this method is more accurate than the static method.

Static Filtering Static filtering describes the opposite. We do not adjust the minimum or maximum value, after we first created the bin, even if we remove the corresponding value. If we look at our last example, the maximum value would not change after the removal of the value 4. This is expected to be less accurate than dynamic filtering, but it will help us to evaluate how effective the dynamic solution is.

4.3.2 Filter Functions

Two filter functions were implemented; `filterAnyVector`, which is called before, we look at possible refinements, if the rule has changed since we last filtered, `filterCurrentVector`, which is called on the feature vector, which belongs to the feature, which is used for the refinement. By going into more detail we will encounter the term cache entry. In reality there are some more details here, but this term represents a crucial concept and can not be omitted. The cache entry is a data structure, which holds our bins and supplies some fast methods to access information about our bins. We start by looking at `filterAnyVector`, since we can build on it in the explanation of `filterCurrentVector`.

filterAnyVector

We want to filter multiple vectors such that only elements remain in the vectors, which are also covered by the condition, we just learned. We refer to Algorithm 5.

Algorithmus 5 : `filterAnyVector`

Input: $BinVector, cacheEntry$
maxElements = $BinVector.length$
filteredVector = $cacheEntry.getVector$
wasEmpty = $false$
if *filteredVector is* $null$ **then**
    Make a new empty cacheEntry with size max
    Set filtereVector to the new cacheEntry
    wasEmpty = $true$
end
i = 0
for $r \leftarrow 0$ **to** $maxElements$ **by** 1 **do**
    numExamples = 0, maxValue = $-\infty$, minValue = ∞
    for *each Example* **do**
        if *Example is covered* **then**
            The following two condition is only for dynamic binning
            if *exampleValue* < *minValue* **then**
                minValue = exampleValue
            end
            if *maxValue* < *exampleValue* **then**
                maxValue = exampleValue
            end
            if *wasEmpty* **then**
                addExample
            end
            numExamples++
        end
        else
            if *not wasEmpty* **then**
                remove Example
            end
        end
    end
    if *not wasEmpty* **then**
        swap(i, r)
    end
    if *numExamples > 0* **then**
        BinVectorEntry i = Bin r; i++
    end
end
update filteredVector; update cacheEntry

In the beginning `filteredVector` can be null, which means it is empty, if we do not have a `cacheEntry` already. In this case we have to make a new `cacheEntry` with the size of `maxElements`, which represents the number of elements in the bin vector at this time. Filtering reduces the number of elements and we will never add elements back in. This is why, we do not need more spaces than `maxElements` in the `cacheEntry`. So we initialize it with this size. We have to remember that we made a new `cacheEntry`, so we set `wasEmpty` to $true$.

i is the index of the bin we are currently looking at in the filtered list. If a bin is filtered out i is not increased and the next unfiltered bin will take over its index in the filtered list.
The outer for-loop iterates over each entry of the bin vector, which means it iterates over single bins. In dynamic binning we start by updating the borders of the bin if necessary. This part resembles what Algorithm 2 does to set these values. After this we continue with both variations by adding an example, if the cache was empty. If the cache was not empty, the value is already in the collection and has not to be added, we just do not remove it later. If the example was not covered and the cache was not empty, we have to remove the example. If the example is not covered, we just do not add it in the case that the cache was empty.
If we removed a bin, we swap the current entry with the entry on the spot i. This way we close a possible gap in the data. In case $i < r$, we removed a bin before the current one so we swap the current entry onto the next open spot in the vector, which is i. If $i = r$ we did not remove an entry and the swap function does nothing. The case $i > r$ is not possible, since we increment r in every iteration, while we only increase i, if a entry was selected.
The cache entry is updated by setting the number of conditions to the new number of conditions. This is important to know, in order to know, when to filter again. If the number of conditions in the cache is different to the number of conditions overall, we need to filter.

filterCurrentVector

As mentioned before the filtering of the vector, we used to learn a condition is somewhat easier, since we can look at entire bins without checking each example, see feature0 in 4.2. We refer to Algorithm 6.

Algorithmus 6 : `filterCurrentVector`

Input: $BinVector, cacheEntry, conditionEnd, covered$
numTotalElements = $BinVector.length$
wasEmpty = $false$
filteredVector = $cacheEntry.getVector$
if *covered* **then**
  numElements = conditionEnd
end
else
  if *numTotalElements* > *conditionEnd* **then**
    numElements = numTotalElements - conditionEnd
  end
  else
    numElements = 0
  end
end
if *filteredVector is* $null$ **then**
  Make a new empty cacheEntry with size $numElements$
  Set filtereVector to the new cacheEntry
  wasEmpty = $true$
end
i = 0
if *covered* **then**
  start = 0; end = conditionEnd
end
else
  start = conditionEnd; end = numTotalElements
end
for $r \leftarrow start$ **to** end **by** 1 **do**
  for *each Example* **do**
    omitted updates
    if *wasEmpty* **then**
      addExample
    end
  end
  if *not wasEmpty* **then**
    swapExamples(i, r)
  end
  update Bin; i++
end
update filteredVector; update cacheEntry

First we have to look at the two new input parameters `conditionEnd` and `covered`. `conditionEnd` specifies the cut-off, where the learned condition ends, it will also be called *split point* in the following explanation. The parameter `covered` tells us, if the feature values until this point are covered ($true$) or if the feature values after this point are covered($false$).
The following block of nested conditions uses this information to determine, how much elements should be in the resulting vector. When the split point is larger than or equal to the total number of elements, we set

`numElements` to 0, in this case there are no elements covered by the condition and the outer for-loop, which is the main part of this function, will be skipped entirely.
Note that we do not have to make an equivalent check, if the condition is covered. If the split point is larger than the total number of elements, everything is covered and there is no problem. On the other hand, if it is 0 and no example is covered, we would again skip the loop.
As briefly mentioned the outer for-loop is the center piece of this function. To determine over which part of the bin vector it should iterate we set the parameters `start` and `end`. Now we know which bin vector elements are covered and where they are. The for-loop iterates over them, the inner for-loop iterates over the examples of the bin and updates other parts of the BOOMER algorithm, which are not relevant to this paper, about which examples are covered. Then, if our cache was empty, it adds the example. If it was not empty we leave the cache as is. After the loop we swap the covered examples to the beginning of the list. We update the filtered vector and the cache entry like in Algorithm 5.

5 Evaluation

In this chapter we will look at experiments. These were run to evaluate our solution. The data sets used, will be introduced, before looking at how the experiments where set up, followed by some metrics, which tell us how the new solution performed in comparison to the fundamental algorithm. Lastly we analyze the results.

5.1 Data Sets

We will use the following data sets to evaluate our solution:

Name	Domain	Instances	Attributes	Labels	Source
mediamill	video	43907	120	101	[Sno+06]
scene	image	2407	294	6	[Bou+04]
yeast	biology	2417	103	14	[EW02]

- **mediamill**: Mediamill contains not only the most examples of those data sets, but also the most labels. This data set can provide us with general trends corresponding to these facts.
- **scene**: Scene is fairly comparable with yeast but we would expect a better acceleration since scene has three times the attributes that yeast has, which means each example less will have a bigger impact.
- **yeast**: As mentioned under scene, we expect less acceleration from yeast.

We will focus on data sets from three different domains. All of these data sets are dense, since we expect the best results, if there is only minimal sparsity. This means there are only a small amount or even no values which are set to a default value, typically 0. If the opposite was to be true, there would be a lot of feature values equal to the default value, which would almost certainly take a bin for themselves. This effect is more likely to occur, if we increase the number of bins. Bins, which only contain default values, can typically only generate bad conditions, since the default value often tells us nothing about the example. Also more feature values lead to more thresholds, which means there is an increased amount of possible conditions, so we can reduce this amount further. We expect better results for mediamill, since it is the biggest data set and our solution should scale with the number of instances, since we can group them in less bins relative to the total number of examples.

5.2 Experimental Setup

The following hardware was used to conduct the experiments:

- CPU: AMD Ryzen 7 3800X - 8 cores @ 3.90 GHz - 128 GB RAM

All data sets, previously described, were ran without binning to establish a baseline. Then we ran our two methods Equal-Frequency Binning and Equal-Width Binning with static and dynamic filtering and the maximal amount of bins set to $2, 4, 8, 16, 32$ and 64 respectively. In preliminary experiments it was found out that a bin ratio, a fraction of bins relative to the amount of distinct feature values, could be set to low values, as low as 0.01, without losing a lot of accuracy. The minimal amount of bins possible, which is 2, was used to find out how low the number of bins could be set. We want to find general trends, since tuning the parameter would take a lot of time. This is the reason for using 2^n for the number of bins.
We also used 8 parallel CPU threads. Each time the algorithm learns exactly 1000 rules, one default rule and 999 rules with non-empty bodies. For every experiment we ran, we used 10-Fold-Cross-Validation to ensure that the results are as reliable as possible.

5.3 Metrics

We use the following two metrics, to compare the training time and accuracy of our new methods to our original algorithm.
These metrics should fulfill the following three conditions:

- The values should be largely between 0 and 1
- A bigger value should always be better than a smaller one
- The value has to be relative to our baseline

These three conditions ensure a comparability between all values.

5.3.1 Relative Speed Up (RSU)

The speed up in relation to the original training time is calculated like this:

$$RSU = \frac{t - t'}{t} \tag{5.1}$$

With t being the time the original algorithm took and t' the time that the new variant took.
Let us look at our conditions:

- The values should be largely between 0 and 1. We look at the following three cases:
  - The RSU equals 1, if we need 0 seconds to train our model: $\frac{t-0}{t} = \frac{t}{t} = 1$
    We can not accomplish a training time of 0 seconds, so we can not get a score of 1 or above.
  - The RSU equals 0, if the new method takes exactly as long as the baseline: $\frac{t-t}{t} = \frac{0}{t}$
    This is an undesirable result, since we want to get faster.

  - This metric can become negative, if the new method takes longer than the old one. In this case the speed up is below zero and the algorithm does not improve the training time with this parameters. So we do not need to compare it, since we already know that this run did not achieve the goal, we set.

- A bigger value should always be better than a smaller one: The less time the new algorithm takes, the better. So if t' gets smaller the RSU should increase and if t' gets bigger the RSU should get smaller. If we look at the formula, we see this is true, as well.
  - $RSU \uparrow = \frac{t - t'\downarrow}{t}$
  - $RSU \downarrow = \frac{t - t'\uparrow}{t}$
- The value has to be relative to our baseline: To ensure this we divide by the time the original algorithm took.

5.3.2 Relative Accuracy Improvement (RAI)

We look at the improvement of predictive performance of the original algorithm and the binning algorithm in relation to the default rule and to each other. For the accuracy we used the example-wise F1 Accuracy. The F1 score is calculated by this equation:

$$F1 = \frac{2 * p * r}{p + r} \tag{5.2}$$

With p being the precision score and r being the recall score. They are calculated as such:

$$p = \frac{tp}{tp + fp} \qquad r = \frac{tp}{tp + fn} \tag{5.3}$$

tp is the number of true positives, fp and fn are the number of false positives and negatives respectively. This information is taken from Chapter 11 of [Gru15].
We calculate the metric with the following formula:

$$RAI = \frac{a' - a_d}{a - a_d} \tag{5.4}$$

With a being the F1 accuracy of the original algorithm; a' the F1 accuracy of the new variant and a_d the F1 accuracy of the default rule. It was also considered to use the Hamming Accuracy in addition to the example-wise F1 Accuracy, but this provided no additional information in the interpretation of the results, so this information is not shown.

- The values should be largely between 0 and 1. Here we have to consider three cases, as well:
  - This value falls below zero if the default rule is more accurate than the accuracy of our new algorithm. This is the worst case, since the default rule is the fastest execution possible and should be the least accurate. In this case the algorithm did not achieve its goal and we do not need to compare the run with successful ones.
  - This value can go above 1, if the new variant is more accurate than the original. We do not expect to see this.

  - We expect the relation $a_d \leq a' \leq a$. In this case this value is between 0 and 1, which would fulfill our condition by definition.

- A bigger value should always be better than a smaller one, so a bigger value should represent a higher accuracy of the new variant. This is true, since a', our new accuracy and only interesting value, stands in the numerator.
  - $RAI \downarrow = \frac{a' \downarrow - a_d}{a - a_d}$
  - $RAI \uparrow = \frac{a' \uparrow - a_d}{a - a_d}$
- The value has to be relative to our baseline. Since this value is a percentage of the accuracy improvement of the baseline in relation to the accuracy of the default rule, this is true by definition.

5.4 Analysis of the Results

5.4.1 Reduction of Possible Conditions

Let us first look at the initial goal, we set. Did we reduce the number of possible conditions? Table 5.1 lists all possible conditions the algorithm evaluated, sorted by filtering method, binning method and number of bins for each data set.

Filter	Binning	#Bins	scene	yeast	mediamill
	none		1450768146	231211774	41932149631
dynamic	eq-freq	2	1750107	548611	251721
dynamic	eq-freq	4	6944302	2320729	919638
dynamic	eq-freq	8	17391623	6390232	2714850
dynamic	eq-freq	16	37155631	14217348	7489660
dynamic	eq-freq	32	72990901	27919739	19794368
dynamic	eq-freq	64	138392054	50716692	50285234
dynamic	eq-width	2	2797645	902658	296645
dynamic	eq-width	4	7533213	2675458	1099830
dynamic	eq-width	8	17068047	5161349	3524338
dynamic	eq-width	16	32579712	9383721	10506240
dynamic	eq-width	32	59331650	16762195	28282732
dynamic	eq-width	64	104842206	28972828	66328852
static	eq-freq	2	293267	102306	119532
static	eq-freq	4	879135	307181	360634
static	eq-freq	8	2050299	717747	841563
static	eq-freq	16	4385543	1536986	1808266
static	eq-freq	32	9017264	3187645	3731542
static	eq-freq	64	18217466	6466111	7596822
static	eq-width	2	293614	102658	119557
static	eq-width	4	881263	307040	360066
static	eq-width	8	2046628	706643	843959
static	eq-width	16	4332737	1467458	1806289
static	eq-width	32	8756175	2864691	3711693
static	eq-width	64	17108261	5401588	7461155

Figure 5.1: Table: Number of possible conditions

We can see, that all runs with binning produce less possible conditions than the original algorithm with no binning, which is marked by none in column binning. We can also see that within the same binning and filter method more bins mean more possible conditions and less bins mean less possible conditions, as was to be expected. If we look at the differences between the binning methods, we see that equal-frequency produces less possible conditions with a small amount of bins, while equal-width produces less possible conditions with more bins. If we look at the differences between both filtering methods, we see that both reduce the number of

possible conditions by up to factor of hundred thousand (100000), but the static filtering consistently reduces them more. In the most extreme cases static filtering reduces the number of possible conditions by up to factor 10 more than dynamic binning.
The explanation for these observations, can be as follows:

1. **Reduction of conditions**: As said earlier, this was our goal. We only look at the thresholds between the bins, not between each value, so we have to check less possible conditions.
2. **Scaling of possible conditions with number of bins**: With less bins there are less thresholds between bins and less possible conditions.
3. **Difference between binning methods**: Equal-Width produces a lot more empty bins, when the number of bins is high. A larger number of bins leads to a shrinking range for each individual bin, which leads to less filled bins, because the likelihood of values belonging in a certain range decreases. Bins which contain no value can not be considered for thresholds, therefore there are less thresholds meaning less possible conditions.
4. **Difference between filtering methods**: Through the adjustments to the minimum and maximum values of the bins, dynamic binning introduces new thresholds every time, we filter. This allows a good refinement for a rule. Static binning keeps the same thresholds through the entire process, so the refinement can not improve the quality of the rule as much. In the following analysis there are more instances, which indicate this.

5.4.2 Comparing binning time and filtering time

Since binning and its corresponding filtering are the new methods, we should look at those. We use two filter functions in our implementation, `filterAnyVector` and `filterCurrentVector`, so their times were tracked separately. The vector reading time of the baseline runs are also tracked to compare it to the time the binning method took, because in both cases these are the times we need to prepare the data for the rest of the implantation. Since the tables became large, they are divided by data sets. Within a given data set the values are the most comparable. Keep in mind that the filter time can be longer than the total training time, since the training time is evaluated over the whole process not considering the parallel threads one by one and the filtering time is the added filtering time of each single thread. This is no problem since we only look at the relation between the filtering times and not in relation to the total training time. The results for the data set yeast are shown in Table 5.2, the results for the data set scene are shown in Table 5.3 and the results for the data set mediamill are shown in Table 5.4.

Filter	Binning	#Bins	bin/read time	filterAny	filterCurrent
	none		0.0128	1.5928	0.0300
dynamic	eq-freq	2	0.0493	21.4579	0.1681
dynamic	eq-freq	4	0.0531	18.4005	0.1611
dynamic	eq-freq	8	0.0491	17.4991	0.1680
dynamic	eq-freq	16	0.0426	12.7241	0.1419
dynamic	eq-freq	32	0.0527	12.6704	0.1475
dynamic	eq-freq	64	0.0492	12.8009	0.1496
dynamic	eq-width	2	0.0444	18.7269	0.1944
dynamic	eq-width	4	0.0460	15.8273	0.1963
dynamic	eq-width	8	0.0550	14.8321	0.1738
dynamic	eq-width	16	0.0535	11.2105	0.1356
dynamic	eq-width	32	0.0475	11.5641	0.1340
dynamic	eq-width	64	0.0461	11.8573	0.1322
static	eq-freq	2	0.0608	4.2029	0.0556
static	eq-freq	4	0.0603	4.9233	0.0353
static	eq-freq	8	0.0608	5.3916	0.0275
static	eq-freq	16	0.0608	5.3050	0.0244
static	eq-freq	32	0.0524	5.4348	0.0249
static	eq-freq	64	0.0553	5.5505	0.0254
static	eq-width	2	0.0442	5.0016	0.0312
static	eq-width	4	0.0442	5.4108	0.0231
static	eq-width	8	0.0462	5.5826	0.0259
static	eq-width	16	0.0440	5.5533	0.0303
static	eq-width	32	0.0526	5.7064	0.0311
static	eq-width	64	0.0476	5.9018	0.0314

Figure 5.2: Table: Yeast: Filter and Binning Time in seconds

Filter	Binning	#Bins	bin/read time	filterAny	filterCurrent
	none		0.0322	5.0003	0.0749
dynamic	eq-freq	2	0.0630	61.5551	0.1643
dynamic	eq-freq	4	0.0613	52.3145	0.1472
dynamic	eq-freq	8	0.0629	50.6810	0.1509
dynamic	eq-freq	16	0.0628	50.0758	0.1552
dynamic	eq-freq	32	0.0755	31.9343	0.1153
dynamic	eq-freq	64	0.0613	33.7805	0.1178
dynamic	eq-width	2	0.0555	52.4810	0.1746
dynamic	eq-width	4	0.0633	50.8859	0.1588
dynamic	eq-width	8	0.0563	54.4200	0.1730
dynamic	eq-width	16	0.0535	54.7226	0.1722
dynamic	eq-width	32	0.0580	36.7527	0.1297
dynamic	eq-width	64	0.0611	39.4525	0.1338
static	eq-freq	2	0.0595	9.4030	0.0449
static	eq-freq	4	0.1139	10.6361	0.0275
static	eq-freq	8	0.0840	11.3099	0.0255
static	eq-freq	16	0.0939	11.5614	0.0268
static	eq-freq	32	0.0928	11.9503	0.0285
static	eq-freq	64	0.0851	13.2867	0.0309
static	eq-width	2	0.0685	12.6272	0.0209
static	eq-width	4	0.0624	13.2460	0.0235
static	eq-width	8	0.0397	13.4723	0.0317
static	eq-width	16	0.0561	14.1846	0.0334
static	eq-width	32	0.0550	14.9697	0.0384
static	eq-width	64	0.0530	16.3386	0.0437

Figure 5.3: Table: Scene: Filter and Binning Time in seconds

The four main observation, we can make are:

1. Binning is not necessarily slower than just reading the data, especially for the data set mediamill, but in most cases binning does take longer. Even in those cases the overhead is in the micro second range and is negligible in relation to the filtering time.
2. Equal-Width binning tends to take less time. This is what we expected, since the Equal-Width binning does not need to sort the data.
3. The filtering with binning takes a lot more time than without, even though we expected it to be roughly the same.
4. Static filtering is faster then the dynamic method, but still slower than the original.

The last two observation require a bit more attention. The time complexity of filtering with binning is the same as the time complexity of filtering without binning, namely $O(instances * features)$. The difference is the new data structure bin, through which we have to navigate. For more information on this structure refer to the first section of Chapter 4. By examining this structure, we can see that we have to look at a bin in

Filter	Binning	#Bins	bin/read time	filterAny	filterCurrent
	none		0.2695	89.8053	11.4798
dynamic	eq-freq	2	0.1947	1332.5035	6.8262
dynamic	eq-freq	4	0.1892	836.0623	11.1205
dynamic	eq-freq	8	0.1814	643.2856	14.7502
dynamic	eq-freq	16	0.1792	610.9052	19.2853
dynamic	eq-freq	32	0.3630	630.3168	26.4567
dynamic	eq-freq	64	0.2159	679.7267	36.1455
dynamic	eq-width	2	0.1975	463.8279	14.7757
dynamic	eq-width	4	0.1723	612.4378	17.8907
dynamic	eq-width	8	0.1931	764.7619	25.3364
dynamic	eq-width	16	0.2217	923.6635	39.8141
dynamic	eq-width	32	0.2374	1091.2539	55.8251
dynamic	eq-width	64	0.1937	1170.2363	67.4630
static	eq-freq	2	0.2215	1060.0089	6.4623
static	eq-freq	4	0.3138	536.6678	9.1521
static	eq-freq	8	0.2667	401.6495	10.1953
static	eq-freq	16	0.2459	400.5129	10.4783
static	eq-freq	32	0.2509	403.5872	10.6057
static	eq-freq	64	0.2394	416.3299	10.7104
static	eq-width	2	0.1744	266.0313	12.8286
static	eq-width	4	0.2239	352.9561	12.3975
static	eq-width	8	0.1705	435.8694	12.2357
static	eq-width	16	0.2198	495.4262	12.4519
static	eq-width	32	0.2354	553.6663	12.5622
static	eq-width	64	0.1826	577.2671	12.6227

Figure 5.4: Table: Mediamill: Filter and Binning Time in seconds

the feature vector, then iterate through the linked list of examples in this bin `Examples`. Here is the crucial difference to the baseline implantation, which uses only a feature vector, which is implemented as an array, to search through. The new implementation has to go through an array of bins and additionally through a linked list for each entry in the bin. The initial array is smaller as it only contains bins instead of all feature values, but the linked lists are slow to iterate through, so it seems like the smaller array size cannot mitigate this additional expenditure.
Let us look at the last observation. The implementation of dynamic filtering is the same as the one of static filtering with the only exception being two if-conditions, which check if the minimum or maximum value has changed. For further information refer to Section 4.3.
These additional instructions can not cause such a great difference in training time. Our assumption from earlier, dynamic filtering introduces more thresholds and makes more refinements, which leads to more filter calls, would also explain this phenomenon. To confirm or reject the assumption, that the filtering is called less often, we tracked how often the filtering methods were actually called, which is listed in Table 5.5.

Data Set	Binning	#Bins	dyn Any	dyn Current	static Any	static Current
yeast	eq-freq	2	454105	4495	99073	999
yeast	eq-freq	4	684954	6861	101034	1017
yeast	eq-freq	8	847151	8578	103362	1043
yeast	eq-freq	16	923110	9410	109598	1086
yeast	eq-freq	32	924909	9474	118115	1169
yeast	eq-freq	64	899345	9275	128605	1273
yeast	eq-width	2	847164	8528	99511	999
yeast	eq-width	4	997246	10389	110495	1113
yeast	eq-width	8	925361	9691	129708	1311
yeast	eq-width	16	869022	9100	145387	1464
yeast	eq-width	32	837526	8740	153389	1552
yeast	eq-width	64	801659	8368	159601	1613
scene	eq-freq	2	1472743	5085	287691	999
scene	eq-freq	4	2070752	7262	297396	1026
scene	eq-freq	8	2286415	8056	308770	1070
scene	eq-freq	16	2352545	8318	331258	1149
scene	eq-freq	32	2361713	8384	360317	1241
scene	eq-freq	64	2338368	8313	391766	1352
scene	eq-width	2	2797233	9688	289579	999
scene	eq-width	4	2587006	9087	328479	1131
scene	eq-width	8	2676832	9431	354962	1227
scene	eq-width	16	2560263	9029	372001	1287
scene	eq-width	32	2498146	8823	390700	1345
scene	eq-width	64	2430160	8584	402267	1388
mediamill	eq-freq	2	132157	1119	118553	999
mediamill	eq-freq	4	189800	1609	118533	999
mediamill	eq-freq	8	277445	2360	118878	1002
mediamill	eq-freq	16	398334	3395	119456	1006
mediamill	eq-freq	32	549235	4672	120003	1010
mediamill	eq-freq	64	721425	6129	120777	1017
mediamill	eq-width	2	188499	1625	118011	999
mediamill	eq-width	4	294267	2520	120421	1017
mediamill	eq-width	8	468263	3989	123295	1040
mediamill	eq-width	16	724927	6214	119735	1065
mediamill	eq-width	32	994358	8476	124931	1093
mediamill	eq-width	64	1200702	10206	129260	1120

Figure 5.5: Table: Number Filter Calls

We can see, the static filtering methods get called less often. Sometimes the `filterCurrent` method is only called 999 times with static filtering, which means it is called once per learned rule. This is the least amount possible. Now that we confirmed that the filter methods are called less often by using static filtering, we look at our assumption and analyze its plausibility:

1. **Dynamic Filtering** adjusts the thresholds of the bins.
   This introduces more thresholds, which allow for refinements, which are evaluated better.
2. **Static Filtering** keeps the same thresholds through the entire run time.
   This leads to limited improvement opportunities for additional refinements.

We know that the first line of 1. and 2. are true, since this is how we defined the methods in Section 4.3. That dynamic binning produces more thresholds follows from the fact that adjusting minimal and maximal values also changes the thresholds between them. Since filtering is called every time we evaluate a refinement, we know that dynamic binning evaluates more refinements from the data in Table 5.5. So this assumption is at least plausible.

5.4.3 Tendencies from Scatter-Plots

At this point we look at some scatter plots and establish general trends, which are looked into in more detail later. They depict the relationship between RSU (Speed Up) and RAI (Accuracy Improvement). We abbreviated dynamic filtering as *Dyn*, static filtering as *Stat*, Equal-Frequency Binning as *Eq-Fr* and Equal-Width Binning as *Eq-W*. Since labeling the data points with the number of bins used would make the scatter-plots confusing, tables with the data, which was used to generate the plots will be included below them.

Yeast

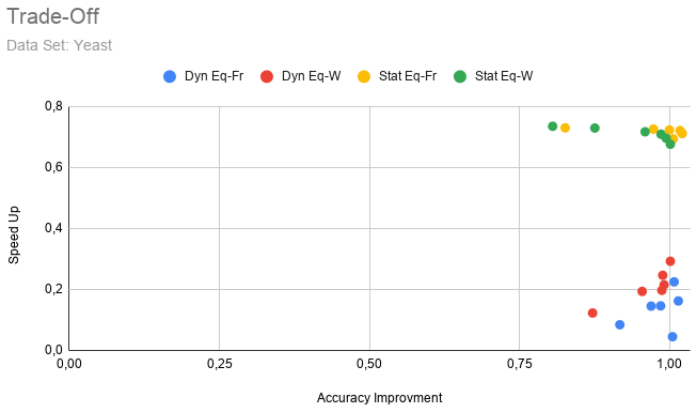


Figure 5.6: Scatter-Plot: Trade-Off Yeast; Data: Table 5.7

Filter	Binning	#Bins	RAI	RSU
	no binning		1	0
dynamic	eq-freq	2	0.9170	0.0833
dynamic	eq-freq	4	0.9692	0.1444
dynamic	eq-freq	8	0.9849	0.1452
dynamic	eq-freq	16	1.0075	0.2240
dynamic	eq-freq	32	1.0144	0.1614
dynamic	eq-freq	64	1.0047	0.0440
dynamic	eq-width	2	0.8716	0.1218
dynamic	eq-width	4	0.9542	0.1928
dynamic	eq-width	8	0.9908	0.2144
dynamic	eq-width	16	1.0011	0.2917
dynamic	eq-width	32	0.9884	0.2457
dynamic	eq-width	64	0.9870	0.1963
static	eq-freq	2	0.8260	0.7297
static	eq-freq	4	0.9730	0.7256
static	eq-freq	8	0.9996	0.7228
static	eq-freq	16	1.0173	0.7208
static	eq-freq	32	1.0209	0.7113
static	eq-freq	64	1.0060	0.6937
static	eq-width	2	0.8051	0.7348
static	eq-width	4	0.8754	0.7292
static	eq-width	8	0.9591	0.7169
static	eq-width	16	0.9856	0.7089
static	eq-width	32	0.9941	0.6956
static	eq-width	64	1.0011	0.6758

Figure 5.7: Table: Trade-Off Yeast

First we look at the scatter-plot produced by our experiments on the data set yeast in Scatter Plot 5.6. The divide between static and dynamic binning is directly visible, since data points generated with static binning are grouped in the top right hand corner, while the data points created by methods with dynamic binning are grouped in the bottom right hand corner. This means static filtering leads to faster learning on this data set. This was not expected, but we can explain this with what we already know. Static binning leads to less possible conditions and needs less time to filter, see Table 5.1 and Table 5.2 respectively.
On this data set, we do not seem to lose a lot of accuracy with a small amount of bins and we even see an increase of accuracy in relation to the original algorithm, if we use 16 or more bins with Equal-Frequency Binning. Equal-Width Binning produces results better than the original algorithm with 16 bins and dynamic filtering or 64 bins and static filtering. In the case of the yeast data set binning seems to work like Random Instance-Sub Sampling [PS10], since it breaks down a set into a smaller sub-sets. Even though binning does not introduce randomness, it still functions in a similar way and increases accuracy.
If we look at the table 5.6 we can identify the points, which are grouped together. With this information we can deduce that runs with $8, 16, 32,$ and 64 generally produce similarly good results for each filtering-binning setup.

Scene

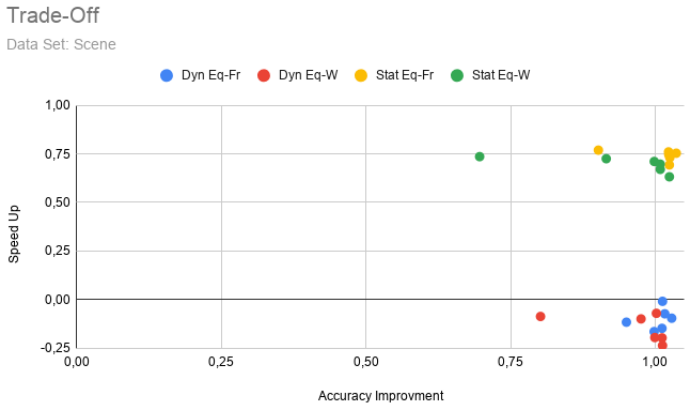


Figure 5.8: Scatter-Plot: Trade-Off Scene; Data: Table 5.9

Filter	Binning	#Bins	RAI	RSU
	no binning		1	0
dynamic	eq-freq	2	0.9504	-0.1164
dynamic	eq-freq	4	1.0167	-0.0732
dynamic	eq-freq	8	1.0287	-0.0959
dynamic	eq-freq	16	1.0119	-0.1480
dynamic	eq-freq	32	1.0128	-0.0089
dynamic	eq-freq	64	0.9983	-0.1647
dynamic	eq-width	2	0.8019	-0.0868
dynamic	eq-width	4	0.9757	-0.0995
dynamic	eq-width	8	0.9995	-0.1953
dynamic	eq-width	16	1.0128	-0.2365
dynamic	eq-width	32	1.0022	-0.0706
dynamic	eq-width	64	1.0120	-0.1972
static	eq-freq	2	0.9019	0.7694
static	eq-freq	4	1.0233	0.7604
static	eq-freq	8	1.0366	0.7537
static	eq-freq	16	1.0237	0.7465
static	eq-freq	32	1.0255	0.7308
static	eq-freq	64	1.0246	0.6932
static	eq-width	2	0.6966	0.7360
static	eq-width	4	0.9156	0.7253
static	eq-width	8	0.9985	0.7111
static	eq-width	16	1.0088	0.6974
static	eq-width	32	1.0089	0.6704
static	eq-width	64	1.0247	0.6322

Figure 5.9: Table: Trade-Off Scene

We can see similar tendencies in Scatter Plot 5.8 like in the runs with the data set yeast, but the divide between static and dynamic filtering is even worse for the dynamic method, since the RSU in every case is less then zero. This means this method provides no benefit over the original algorithm. Quite the opposite is true. Dynamic filtering leads to longer training times and slows the algorithm down. It rarely increases the accuracy, but not by enough to consider running it over the original algorithm and taking more time.
On the flip side, static filtering produces good results. With Equal-Frequency Binning it increases the accuracy for each number of bins greater than 2, with Equal-Width for each number greater than 8. The RSU is also notable since it is comparable to the results of static filtering on the yeast data set, which already saved a lot of time.
We can assume that the amount of attributes play a significant role in these changes, since it is the biggest difference between those two data sets. By checking Table 5.3 we can see that dynamic filtering took much more time in relation to the static filtering, compared to the other two data sets. This is also backed by the data from Table 5.5, since we see the same pattern with the filter calls. The cause could be what we discussed before: Dynamic filtering introduces more thresholds, which allows for more refinements and consequently leads to more filter calls. Since this happens for each feature, we can assume that dynamic filtering will take even longer with an increasing amount of features.

Mediamill

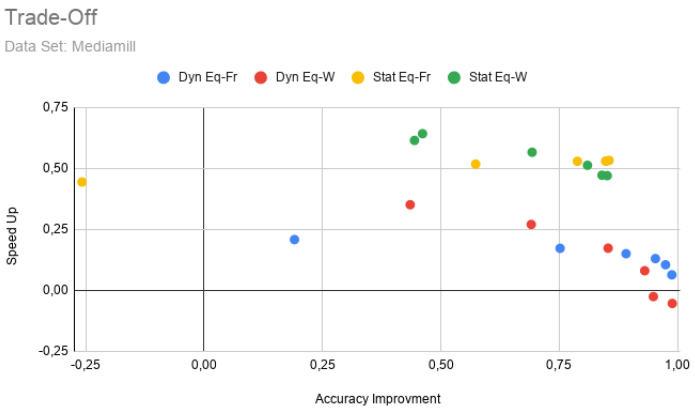


Figure 5.10: Scatter-Plot: Trade-Off Mediamill; Data: Table 5.11

Filter	Binning	#Bins	RAI	RSU
	no binning		1	0
dynamic	eq-freq	2	0.1909	0.2078
dynamic	eq-freq	4	0.7517	0.1718
dynamic	eq-freq	8	0.8909	0.1498
dynamic	eq-freq	16	0.9529	0.1297
dynamic	eq-freq	32	0.9743	0.1044
dynamic	eq-freq	64	0.9876	0.0631
dynamic	eq-width	2	0.4352	0.3507
dynamic	eq-width	4	0.6908	0.2699
dynamic	eq-width	8	0.8530	0.1724
dynamic	eq-width	16	0.9303	0.0798
dynamic	eq-width	32	0.9486	-0.0263
dynamic	eq-width	64	0.9886	-0.0544
static	eq-freq	2	-0.2575	0.4441
static	eq-freq	4	0.5734	0.5173
static	eq-freq	8	0.7882	0.5289
static	eq-freq	16	0.8482	0.5295
static	eq-freq	32	0.8550	0.5327
static	eq-freq	64	0.8484	0.5298
static	eq-width	2	0.4613	0.6423
static	eq-width	4	0.4443	0.6147
static	eq-width	8	0.6926	0.5661
static	eq-width	16	0.8095	0.5127
static	eq-width	32	0.8512	0.4702
static	eq-width	64	0.8401	0.4719

Figure 5.11: Table: Trade-Off Mediamill

In Scatter Plot 5.10 we see a noticeable data point on the left, we did not see before. The Table 5.11 determines that this value belongs to the run with the parameters static filtering, Equal-Frequency binning and 2 bins. We already indirectly noted, that the minimum number of bins (2) leads to the worst results, as expected. Static binning leads to worse accuracy on this data set in general. These factors with the worse performance of our methods in terms of accuracy on this data set led to a worse accuracy than the default rule with the given parameters, we did not expect to sacrifice so much accuracy. This is the worst case, since the default rule should be improved upon and not diminished. One possible explanation is, that the split point is chosen so poorly that most of the features are not represented correctly anymore and the algorithm learns false rules. This theory is supported by the fact that other approaches still achieve positive results. If we look at the two Equal-Width Binning results, we see they perform a lot better. They would find a better split point which is exactly in the middle of the total span of values. This tells us a lot more about the feature values and we can learn better conditions. Continuing this assumption, the dynamic filter approach can mitigate a bad split point in case of Equal-Frequency Binning with 2 bins, since it updates the bin borders and therefore moves the split point.
In this experiments the accuracy actually decreased with all parameters, instead of increasing it with some. Static binning only reaches about 85% of the accuracy improvement the original algorithm achieved. With dynamic binning we still reach about 99%. This was to be expected and fulfills the assumption about this approach we formulated in the beginning: “We want to sacrifice some accuracy to reduce the training time.” The increase we saw until now was more surprising.

Dynamic filtering is slower than the original, when used with 32 or 64 bins and the Equal-Width method. This filter method is faster with other parameters, but still not as fast as the static filter method, which reduces the training time to about half of the original.
So here we have to actually decide; do we want to sacrifice some accuracy to gain more speed, or do we want to invest more time to benefit the accuracy. We could tune the parameters accordingly.

5.4.4 Comparison of Binning Methods

To refine our understanding of the binning methods in terms of accuracy and training time, we look at both in direct comparison. We will take a look at the corresponding Table 5.12, which reformats the data, we previously looked at. This should help to compare it. We could simply count the number of times each algorithm was faster or more accurate to find a general pattern. Considering that negative values in one of both categories makes the used method useless under the given parameters, we do not consider the scene-dynamic runs since both methods produce a negative RAI. We also consider Equal-Frequency better in both categories for the mediamill-dynamic runs with 32 and 64 bins, because Equal-Width is slower in each case. Equal-Width is considered better in both categories for mediamill-static with 2 bins, since Equal-Frequency reduces the accuracy below the accuracy of the default rule.

	Equal-Frequency	Equal-Width	Total
RAI	26	4	30
RSU	16	14	30
Total	42	18	60

We can conclude, that Equal-Frequency produces more accurate results. For the speed up we can not conclude a clear advantage of one or the other, since the differences are not that clear.

			RAI		RSU	
Data Set	Filtering	#Bins	Equal-Freq	Equal-Width	Equal-Freq	Equal Width
yeast	dynamic	2	0.9170	0.8716	0.0833	0.1219
yeast	dynamic	4	0.9692	0.9542	0.1444	0.1928
yeast	dynamic	8	0.9849	0.9908	0.1452	0.2144
yeast	dynamic	16	1.0074	1.0011	0.2240	0.2917
yeast	dynamic	32	1.0144	0.9884	0.1614	0.2457
yeast	dynamic	64	1.0047	0.9870	0.0440	0.1963
yeast	static	2	0.8260	0.8051	0.7297	0.7348
yeast	static	4	0.9730	0.8754	0.7256	0.7292
yeast	static	8	0.9996	0.9591	0.7228	0.7169
yeast	static	16	1.0173	0.9856	0.7208	0.7089
yeast	static	32	1.0209	0.9941	0.7113	0.6956
yeast	static	64	1.0060	1.0011	0.6937	0.6758
scene	dynamic	2	0.9504	0.8019	-0.1164	-0.0869
scene	dynamic	4	1.0167	0.9757	-0.0732	-0.0995
scene	dynamic	8	1.0287	0.9995	-0.0959	-0.1953
scene	dynamic	16	1.0119	1.0128	-0.1480	-0.2365
scene	dynamic	32	1.0128	1.0022	-0.0089	-0.0706
scene	dynamic	64	0.9983	1.0120	-0.1647	-0.1972
scene	static	2	0.9019	0.6966	0.7694	0.7360
scene	static	4	1.0233	0.9156	0.7604	0.7253
scene	static	8	1.0366	0.9985	0.7537	0.7111
scene	static	16	1.0237	1.0088	0.7465	0.6975
scene	static	32	1.0255	1.0089	0.7308	0.6704
scene	static	64	1.0246	1.0247	0.6932	0.6322
mediamill	dynamic	2	0.1909	0.4352	0.2078	0.3507
mediamill	dynamic	4	0.7517	0.6908	0.1718	0.2699
mediamill	dynamic	8	0.8910	0.8530	0.1498	0.1724
mediamill	dynamic	16	0.9529	0.9303	0.1297	0.07980
mediamill	dynamic	32	0.9743	0.9486	0.1044	-0.0263
mediamill	dynamic	64	0.9876	0.9886	0.0631	-0.0544
mediamill	static	2	-0.2575	0.4613	0.4441	0.6424
mediamill	static	4	0.5734	0.4443	0.5173	0.6148
mediamill	static	8	0.7883	0.6926	0.5289	0.5661
mediamill	static	16	0.8482	0.8095	0.5296	0.5127
mediamill	static	32	0.8550	0.8512	0.5327	0.4702
mediamill	static	64	0.8484	0.8401	0.5298	0.4719

Figure 5.12: Table: RAI and RSU comparison of binning methods

5.4.5 Comparison of Filtering Methods

If we use the same method like in the last section to compare filtering methods, we can make some general statements about them as well. Since we do not discard the values of dynamic binning on the scene data set, we get a few values more. In this case we also reordered the tables to get a better view of the values. RAI and RSU are shown in Table 5.13.

	dynamic	static	Total
RAI	14	22	36
RSU	1	35	36
Total	15	57	72

This distribution shows that static filtering is always faster than the dynamic approach. The one data point, where we evaluate the dynamic filtering as the better one, is the one instance, where static binning caused a negative accuracy value. We can conclude that static binning is faster. In almost two thirds of experiments static filtering returned a more accurate model. But since the worse values are limited to mediamill, the biggest data set, it is possible that static filtering performs even worse, if the data sets get bigger.

			RAI		RSU	
Data Set	Binning	#Bins	dynamic	static	dynamic	static
yeast	eq-freq	2	0.9170	0.8260	0.0833	0.7297
yeast	eq-freq	4	0.9692	0.9730	0.1444	0.7256
yeast	eq-freq	8	0.9849	0.9996	0.1452	0.7228
yeast	eq-freq	16	1.0075	1.0173	0.2240	0.7208
yeast	eq-freq	32	1.0144	1.0209	0.1614	0.7113
yeast	eq-freq	64	1.0047	1.0060	0.0440	0.6937
yeast	eq-width	2	0.8716	0.8051	0.1219	0.7348
yeast	eq-width	4	0.9542	0.8754	0.1928	0.7292
yeast	eq-width	8	0.9908	0.9591	0.2144	0.7169
yeast	eq-width	16	1.0011	0.9856	0.2917	0.7089
yeast	eq-width	32	0.9884	0.9941	0.2457	0.6956
yeast	eq-width	64	0.9870	1.0011	0.1963	0.6758
scene	eq-freq	2	0.9504	0.9019	-0.1164	0.7694
scene	eq-freq	4	1.0167	1.0233	-0.0732	0.7604
scene	eq-freq	8	1.0287	1.0366	-0.0959	0.7537
scene	eq-freq	16	1.0119	1.0237	-0.1480	0.7465
scene	eq-freq	32	1.0128	1.0255	-0.0089	0.7308
scene	eq-freq	64	0.9983	1.0246	-0.1647	0.6932
scene	eq-width	2	0.8019	0.6966	-0.0869	0.7360
scene	eq-width	4	0.9757	0.9156	-0.0995	0.7253
scene	eq-width	8	0.9995	0.9985	-0.1953	0.7111
scene	eq-width	16	1.0128	1.0088	-0.2365	0.6975
scene	eq-width	32	1.0022	1.0089	-0.0706	0.6704
scene	eq-width	64	1.0120	1.0247	-0.1972	0.6322
mediamill	eq-freq	2	0.1909	-0.2575	0.2078	0.4441
mediamill	eq-freq	4	0.7517	0.5734	0.1718	0.5173
mediamill	eq-freq	8	0.8909	0.7883	0.1498	0.5289
mediamill	eq-freq	16	0.9529	0.8481	0.1297	0.5296
mediamill	eq-freq	32	0.9749	0.8550	0.1044	0.5327
mediamill	eq-freq	64	0.9876	0.8484	0.0631	0.5298
mediamill	eq-width	2	0.4352	0.4616	0.3507	0.6424
mediamill	eq-width	4	0.6908	0.4443	0.2699	0.6148
mediamill	eq-width	8	0.8530	0.6926	0.1724	0.5661
mediamill	eq-width	16	0.9303	0.8095	0.0798	0.5127
mediamill	eq-width	32	0.9486	0.8512	-0.0263	0.4702
mediamill	eq-width	64	0.9886	0.8401	-0.0544	0.4719

Figure 5.13: Table: RAI and RSU comparison of filtering methods

6 Conclusion

This chapter summarizes what we learned and gives an outlook on future work. We will also look briefly at an improvement, which was already made.

6.1 Deduction

In this section we look at what we found out and give some recommendations based on the experience gained, while working with the solution, we examined in this paper.

6.1.1 Summarizing the observations

First things first: We succeeded in reducing the number of possible conditions. In fact we could reduced them by up to factor 100000, as shown by our experiments.
The overhead of the binning is marginal, but the filtering is far more difficult to optimize. Even though we implemented a static filtering method, which leads to faster results, than the dynamic method, which was thought to be more accuracy conserving, while being just as fast.
We also saw that the trade-off between accuracy and training time does not need to sacrifice a lot of accuracy to reach a decent time reduction. On the data sets with less instances we could even improve the accuracy of the predictions. On the data set with the most instances, we lost the most accuracy.
The Equal-Frequency Binning produces more accurate results than our Equal-Width method. We can not make a well-founded statement about which of them is faster.
By comparing filter methods, we could determine, that static filtering generally produces better results. But we do not know, if this result is scalable, yet.
From our scatter plots we can guess that in general a number of bins between 8 and 32 is advisable. There probably is an optimal amount of bins for each combination of data set, binning method and filter method, but to find it, you would have to try out each integer greater or equal to 2 and less than the original amount of instances. This would take a lot of time which would defeat the purpose of binning, which is, after all, finding a good enough model in less time.

6.1.2 Recommendations

It is recommended to start the algorithm with Equal-Frequency Binning and static filtering, as well as a number of bins between 8 and 32. This should also make a bad result, like we saw on the data set mediamill with this binning-filtering combination and only 2 bins unlikely, since there should be enough thresholds to learn good conditions. Is the data set a lot bigger and a high accuracy is crucial, dynamic binning should be used, which could still achieve a decent speed up.

6.2 Future Work

To wrap everything up, we look at what can be done in the future. This is a special outlook since one of the suggestions was already implemented, while finishing this paper, so we will take a look at it first.

6.2.1 Improving Implementation

We implemented the example list of the data structure bin as linked lists, this was mainly done to enable dynamic filtering. This could be one of the main reasons, why filtering is so slow. Since we saw that dynamic filtering is not necessarily better than static filtering, we could implement an array which saves the indexes of contained examples for each bin.
This is exactly what Michael Rapp did using the results from this paper as a baseline. Our experiments were run on this implementation, as well. The results are compared to the current static filtering in tables 6.1 and 6.2. The column “difference” is built by subtracting the old results from the new ones, which means positive results signal an improvement by the new implementation and negative results are a deterioration in relation to the current algorithm. Please note that the difference was calculated before rounding the results, so it can vary a bit.
When we look at the RAI comparison, we see what was to be expected. Since we do basically the same as before, but with another underlying data structure, the accuracy should not change and it does not really change. Most of the differences stay below one percent point and we see positive and negative changes seemingly at random. So we can conclude that the accuracy stays the same.
When we look at the RSU we see a clear increase. On the two smaller data sets the speed increases by up to 21 percent points. On the bigger one we see an increase by up to 41 percent points.
So we can reasonably assume by focusing on static binning, we can change to a more suitable data structure to increase the speed up beyond the point of the algorithm introduced in this thesis.

Data Set	Binning	#Bins	linked list	array	difference
yeast	eq-freq	2	0.8260	0.8260	0
yeast	eq-freq	4	0.9730	0.9696	-0.0033
yeast	eq-freq	8	0.9996	0.9930	-0.0066
yeast	eq-freq	16	1.0173	1.0256	0.0083
yeast	eq-freq	32	1.0209	1.0193	-0.0017
yeast	eq-freq	64	1.0060	1.0116	0.0056
yeast	eq-width	2	0.8051	0.8056	0.0005
yeast	eq-width	4	0.8754	0.8719	-0.0035
yeast	eq-width	8	0.9591	0.9599	0.0008
yeast	eq-width	16	0.9856	0.9884	0.0028
yeast	eq-width	32	0.9941	1.0038	0.0096
yeast	eq-width	64	1.0011	1.0016	0.0005
scene	eq-freq	2	0.9019	0.9019	0
scene	eq-freq	4	1.0233	1.0228	-0.0004
scene	eq-freq	8	1.0366	1.0402	0.0037
scene	eq-freq	16	1.0237	1.0312	0.0076
scene	eq-freq	32	1.0255	1.0347	0.0093
scene	eq-freq	64	1.0246	1.0350	0.0104
scene	eq-width	2	0.6966	0.6954	-0.0013
scene	eq-width	4	0.9156	0.9258	0.0103
scene	eq-width	8	0.9985	1.0023	0.0037
scene	eq-width	16	1.0088	1.0161	0.0073
scene	eq-width	32	1.0089	1.0170	0.0081
scene	eq-width	64	1.0247	1.0166	-0.0080
mediamill	eq-freq	2	-0.2575	-0.2575	0
mediamill	eq-freq	4	0.5734	0.5737	0.0003
mediamill	eq-freq	8	0.7883	0.7826	-0.0056
mediamill	eq-freq	16	0.8482	0.8397	-0.0085
mediamill	eq-freq	32	0.8550	0.8624	0.0074
mediamill	eq-freq	64	0.8484	0.8451	-0.0033
mediamill	eq-width	2	0.4613	0.4613	0.0001
mediamill	eq-width	4	0.4443	0.4439	-0.0004
mediamill	eq-width	8	0.6926	0.7035	0.0109
mediamill	eq-width	16	0.8095	0.7997	-0.0099
mediamill	eq-width	32	0.8512	0.8364	-0.0148
mediamill	eq-width	64	0.8401	0.8581	0.0180

Figure 6.1: Table: RAI comparison with new filter method

Data Set	Binning	#Bins	linked list	array	difference
yeast	eq-freq	2	0.7297	0.8265	0.0968
yeast	eq-freq	4	0.7256	0.8379	0.1123
yeast	eq-freq	8	0.7228	0.8394	0.1166
yeast	eq-freq	16	0.7208	0.8297	0.1089
yeast	eq-freq	32	0.7113	0.8371	0.1258
yeast	eq-freq	64	0.6937	0.8198	0.1261
yeast	eq-width	2	0.7348	0.8412	0.1064
yeast	eq-width	4	0.7292	0.8489	0.1197
yeast	eq-width	8	0.7169	0.8513	0.1344
yeast	eq-width	16	0.7089	0.8449	0.1360
yeast	eq-width	32	0.6956	0.8415	0.1458
yeast	eq-width	64	0.6758	0.8288	0.1530
scene	eq-freq	2	0.7694	0.8566	0.0872
scene	eq-freq	4	0.7604	0.8630	0.1025
scene	eq-freq	8	0.7537	0.8542	0.1006
scene	eq-freq	16	0.7465	0.8551	0.1086
scene	eq-freq	32	0.7308	0.8539	0.1231
scene	eq-freq	64	0.6932	0.8405	0.1473
scene	eq-width	2	0.7360	0.8630	0.1270
scene	eq-width	4	0.7253	0.8648	0.1395
scene	eq-width	8	0.7111	0.8581	0.1469
scene	eq-width	16	0.6975	0.8585	0.1610
scene	eq-width	32	0.6704	0.8528	0.1824
scene	eq-width	64	0.6322	0.8482	0.2160
mediamill	eq-freq	2	0.4441	0.8590	0.4148
mediamill	eq-freq	4	0.5173	0.8619	0.3446
mediamill	eq-freq	8	0.5289	0.8555	0.3265
mediamill	eq-freq	16	0.5296	0.8527	0.3231
mediamill	eq-freq	32	0.5327	0.8571	0.3245
mediamill	eq-freq	64	0.5298	0.8632	0.3333
mediamill	eq-width	2	0.6424	0.8615	0.2191
mediamill	eq-width	4	0.6148	0.8566	0.2418
mediamill	eq-width	8	0.5661	0.8588	0.2928
mediamill	eq-width	16	0.5127	0.8580	0.3454
mediamill	eq-width	32	0.4702	0.8618	0.3916
mediamill	eq-width	64	0.4719	0.8619	0.3900

Figure 6.2: Table: RSU comparison with new filter method

6.2.2 Implementing more complex binning methods

As discussed in section 2.2, there are more binning methods. We looked at Gradient One-Side Sampling, Weighted Quantile Sketch and Exclusive Feature Binning.
The first two can be used as a basis to implement supervised example binning methods. To do so we need additional information, namely the gradients. To expand our current algorithm to handle this, we would need to gain this additional information from other parts of BOOMER. We implemented a method `getFeatureInfo`, which should gather and contain information about a feature vector. This could be expanded to handle the

gradients and other information, as well.
We also would need to bin after each learned rule, so we could simplify filtering by not filtering the bins themselves, but the whole data instead. This has the advantage that each example is still stored with each of its feature values. So it is easy to find all feature values which need to be filtered, since we do not have to search them in each bin.
We did not develop algorithms for feature bundling. A future work would need to start from scratch, but it could be beneficial to enable the handling of sparse data.

6.2.3 Accounting for Sparsity

In the beginning of this thesis we ruled out sparse data to reduce complexity. In future work it is possible to expand the discussed approach to support sparse data. One possible solution could be ignoring missing values while binning. We would have to reduce the total number of elements in a given feature vector by one for each sparse value we would be skipping before sorting other values into bins. To filter correctly later on, we would have to keep a list of all sparse values for each feature vector and the examples they originate from, so we do not have to look at the entire vector. As pointed out before, Exclusive Feature Binning could be useful to reduce sparsity and ensure better results ones this approach is fit to handle sparse data.

Bibliography

[AAM20] Ahmad Alimadadi, Sachin Aryal, and Ishan Manandhar. “Artificial intelligence and machine learning to fight COVID-19”. In: *AI and Machine Learning for Understanding Biological Processes* (2020).

[Bah21] MJ Bahmani. *Understanding LightGBM Parameters (and How to Tune Them)*. Mar. 19, 2021. URL: https://neptune.ai/blog/lightgbm-parameters-guide.

[Bou+04] Matthew R. Boutell et al. “Learning multi-label scene classification”. In: *Pattern Recognition* 37.9 (2004), pp. 1757–1771.

[Bro20] Tim Brooks. *An Executive’s Guide to Demystifying AI and Machine Learning*. July 1, 2020. URL: https://www.wwt.com/article/executives-guide-to-demystifying-ai-and-machine-learning.

[CG16] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794.

[EW02] André Elisseeff and Jason Weston. “A Kernel Method for Multi-Labelled Classification”. In: *Advances in Neural Information Processing Systems 14 (NIPS-01)*. 2002, pp. 681–687.

[FGL12] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač. *Foundations of Rule Learning*. Springer, 2012.

[Gru15] Joel Grus. *Data science from scratch: first principles with Python*. O’Reilly, 2015.

[HS99] Geoffrey Hinton and Terrence J. Sejnowski. *Unsupervised Learning: Foundations of Neural Computation*. MIT Press, 1999.

[Knu98] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Adison-Wesley, 1998.

[Lit20] Nico Litzel. *Was ist Big Data?* Feb. 1, 2020. URL: https://www.bigdata-insider.de/was-ist-big-data-a-562440/.

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

[Pod20] Sidhantha Poddar. *Binning in Data Mining*. Sept. 28, 2020. URL: https://www.geeksforgeeks.org/binning-in-data-mining/.

[Pow89] David Powers. *Machine learning of natural language*. Springer, 1989.

[PS10] Santhosh Pathical and Gursel Serpen. “Comparison of subsampling techniques for random subspace ensembles”. In: *International Conference on Machine Learning and Cybernetics*. 2010.

[Rap+20] Michael Rapp et al. “Learning Gradient Boosted Multi-label Classification Rules”. In: *Proceedings of the European Conference of Machine Learning (ECML-PKDD)*. 2020, pp. 124–140.

[Rus10] Stuart Russell. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[Say21a] Saed Sayad. *Supervised Binning*. Mar. 31, 2021. URL: https://www.saedsayad.com/supervised_binning.htm.

[Say21b] Saed Sayad. *Unsupervised Binning*. Mar. 31, 2021. URL: https://www.saedsayad.com/unsupervised_binning.htm.

[Seb05] Nicu Sebe. *Machine learning in computer vision*. Springer, 2005.

[Sha18] Abhishek Sharma. *What makes LightGBM lightning fast?* Oct. 15, 2018. URL: https://towardsdatascience.com/what-makes-lightgbm-lightning-fast-a27cf0d9785e.

[Sno+06] Cees Snoek et al. "The challenge problem for automated detection of 101 semantic concepts in multimedia". In: *Information and Computation/Information and Control - IANDC*. 2006, pp. 421–430.

[Sri18] Harshita Srivastava. *What Is Sparse Data?* May 31, 2018. URL: https://magoosh.com/data-science/what-is-sparse-data/.

[Vis19] Venkat Anurag Setty Vishal Morde. *XGBoost Algorithm: Long May She Reign!* Apr. 8, 2019. URL: https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d.

[Yil19] Soner Yildirim. *Understanding the LightGBM*. Sept. 15, 2019. URL: https://towardsdatascience.com/understanding-the-lightgbm-772ca08aabfa.

[Zho12] Zhi-Hua Zhou. *Ensemble methods: Foundations and algorithms*. Taylor & Francis, 2012.