Dynamic Multilabel Classification using Gradient Boosted Trees

Bachelor-Thesis von Simon Peter Bohlender
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Dynamic Multilabel Classification using Gradient Boosted Trees

Vorgelegte Bachelor-Thesis von Simon Peter Bohlender

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den

(Simon Peter Bohlender)

Abstract

Besides the classical machine learning challenges classification and regression, also multilabel-problems, where each instance can belong to several classes, continue to have a gain in importance. Most of these problems are solved by applying problem-transformation methods which split up the problem into several smaller tasks where each one can be individually solved by using one of the classic single-label approaches. However, a drawback of these approaches is that possible dependencies between labels are completely ignored or only respected partially. In this thesis, a novel approach is introduced that is based on a chain of classifiers which is capable of predicting labels dynamically by detecting and exploiting these potential dependencies. Therefore, each classifier in the chain maximizes the probability of a single label for a test-instance and propagates this label to the following classifier. The distinctive feature thereby is that each classifier is not forced to predict the same label for each label, but can assign a different label to different instances. By propagating these predictions gradually along the chain this additional knowledge can be used by later classifiers to exploit label dependencies and predict further labels based on this information. The foundation therefore is a modified version of a XGBoost classifier (short for Extreme Gradient Tree Boosting) which was extended to be capable of predicting a probability for each target label. This classifier is already used for classic regression and classification tasks where it provides state-of-the art results. This introduced dynamic chain provides especially for Hamming Loss good results which are in some cases even better than the compared problem transformation methods.

Zusammenfassung

Neben den klassischen Machine Learning Aufgaben Klassifizierung und Regression, haben mittlerweile auch Multilabel-Probleme, wo jede Instanz verschiedenen Klassen zugeordnet werden kann, stark an Bedeutung gewonnen. Zumeist werden diese durch Anwendung von Transformations-Methoden gelöst, die das Problem in individuelle und kleinere Aufgaben zerteilen, welche dann jeweils mit klassischen single-label Ansätzen gelöst werden können. Ein Nachteil dieser Ansätze ist allerdings, dass mögliche Abhängigkeiten zwischen den Labels ignoriert oder nur zum Teil beachtet werden. In dieser Arbeit wird ein neuer Ansatz vorgestellt, der auf einer Kette von Klassifizierern basiert, die die Fähigkeit besitzt verschiedene Labels dynamisch und unter Beachtung möglicher Abhängigkeiten vorherzusagen. Dafür versucht jeder Klassifizierer der Kette die Wahrscheinlichkeit für ein Label einer Test-Instanz zu maximieren und gibt dieses Label anschließend an die nachfolgenden Klassifizierer weiter. Die Besonderheit dabei ist, dass die einzelnen Klassifizierer nicht gezwungen werden ein bestimmtes Label vorherzusagen, sondern für verschiedene Instanzen auch verschiedene Labels vorhersagen können. Indem diese Vorhersagen dann schrittweise durch die Kette weitergereicht werden, können dieses zusätzlichen Informationen von späteren Klassfizieren genutzt werden, um mögliche Label Abhängigkeiten zu entdecken und um weitere Labels basierend auf diesem Wissen vorherzusagen. Die Grundlage dafür stellt ein modifizierter XGBoost-Klassifizierer (kurz für: Extreme Gradient Tree Boosting) dar. Dieser findet bereits Anwendung in klassischen Klassifizierungs und Regressionsproblemen und liefert dort sehr gute state-of-the-art Ergebnisse. Die hier eingeführte dynamische Kette liefert besonders für Hamming Loss gute Ergebnisse, die in einigen Fällen auch die verglichenen Problem-Transformations-Ansätze schlagen.

Contents

1 Introduction

1.1 Motivation

Most machine learning algorithms deal with regression or classification tasks where the goal is to predict a unique class from a set of disjoint class labels or to predict a continuous variable. Multilabel classification is an extension to this problem where each instance can be associated with more than one class (Tsoumakas and Katakis, 2006). A well known example for such a problem is automatic text-categorization. If we want to tag books from a library, we have a large set of distinct categories that can be assigned. Thus, a single book can belong to different categories at the same time like, for example, *Adult* and *Sci-Fi*. Another area of application is image classification where an image matches to more than one class, such as *Beach, Sunset* and *Island*.
Most of the current approaches address these problems with label-transformation-methods that split up this multilabel-task into smaller subtasks where each one of these is solved individually with an independent model. Often these attempts provide good results, but there is one problem all these approaches share: By solving individual tasks, global dependencies between labels are completely ignored or disregarded. But especially these dependencies can be used to further improve the results. As an example: If we want to tag books, we know that if we categorize it as *scientific work*, it cannot be assigned to *comedy* or *fantasy* at the same time. So a classifier should be able to detect such a connection and should predict other documents in accordance with this.
A popular approach to address this problem is called *Classifier Chains* where the labels are predicted by linked classifiers. Each classifier of the chain receives all predictions of previous ones and can then use this information to exploit label dependencies. However, a problem of this approach is that all labels are predicted in a static order, this means that for example the first classifier always attempts to predict the first label even if another label is obviously more likely. Another problem of this static chain is that we have to evaluate different orders for predicting labels in order to find the best structure. This motivated us to extend this approach and remove these *static* factors ad build a dynamic chain. This dynamic chain can predict different labels at each stage and is not forced to predict always the same. Therewith we get rid of the problem of finding an optimal chain order and get the ability to always predict the most probable label for an instance. It also supports the detection of label dependencies because after the first round we will most likely get predictions for different labels on which dependencies can be detected.

1.2 Goal

The goal of this thesis is to develop a dynamic chain where each node contains a classifier that can dynamically predict labels for each instance. To use XGBoost for the base classifiers we have to modify the current algorithm and make it capable of solving multilabel classification problems. Therefore, a new tree representation is elaborated, which returns a prediction score for each possible label. The novel approach should also be able to detect dependencies between multiple labels and exploit them to build a robust model. Afterwards this extended algorithm is compared to current state-of-the-art problem transformation methods.

1.3 Structure

Section 2 starts with some basic definitions to provide the basic knowledge about classification and multi-label problems which is required to understand the following approaches. Also some current problem-transformation methods used for solving multilabel classification tasks are being discussed.
In **section 3** a detailed overview of extreme gradient tree boosting and especially the XGBoost algorithm is given. It also explains how the individual trees are constructed and how the prediction values are represented.
Section 4 introduces to the new chaining approach for dynamic multilabel classification with the capability of detecting and exploiting label dependencies. Furthermore the extension of the current XGBoost

algorithm is discussed, as well as the structure of the dynamic chain. Afterwards some refinement steps are shown to address some basic problems of this new concept.
Section 5 introduces to the evaluation datasets and shows the setups for the experiments with the dynamic and the baseline approaches.
In **section 6** the results of these experiments are analyzed and compared.
Section 7 then shows some related work with approaches that also try to exploit label dependencies.
Section 8 gives some prospects for future work to further improve dynamic-multilabel classification and **section 9** will draw some final conclusions and summarizes results.

2 Multilabel Classification

In this first chapter we give a short introduction to supervised classification tasks and especially multilabel classification. Afterwards an overview over different approaches, to deal with suchlike problems, is given.

2.1 Definitions and Notations

The basic goal of supervised classification tasks is to learn an association of objects to classes. For the notations we follow (Loza Mencía, 2013). We want to find an output vector y given an input vector x. In order to learn this mapping, we have to train a classifier h on a given dataset S. A dataset contains n data instances where each instance I is a tuple of a feature vector x, in the feature space $\mathcal{X}$ with m elements, and a corresponding label vector y, in the label space $\mathcal{Y}$ with size k.

$$x = (x_0, x_1, ..., x_m) \in \mathcal{X}$$
$$y = (y_0, y_1, ..., y_k) \in \mathcal{Y}$$
$$I = (x, y) \in S$$

Individual features x_i can be represented as continuous, categorical or binary value. The output vector y consists of binary values that denote the relevant classes of an instance. So we can define the feature space $\mathcal{X}$ and the label space $\mathcal{Y}$ as follows.

$$\mathcal{X} \subseteq \mathbb{R}^m$$
$$\mathcal{Y} := \{0, 1\}^k$$
$$y_i = \begin{cases} 1 & \text{if } \lambda_i \text{ is a relevant class label} \\ 0 & \text{otherwise} \end{cases}$$

where λ_i denotes a single unique class.
A classifier h is then trained on a training-dataset S_{train} in order to learn the mapping between input features and output label vector. Afterward it is used to predict the label vector $\hat{y}$ of an unknown instance x_{test} where we do not know the true target labels y.

$$h : \mathcal{X} \to \mathcal{Y} \qquad \hat{y} = h(x)$$

Additionally we also we define $\mathcal{L}$ as the finite set of k unique classes, which can be assigned to an instance.

$$\mathcal{L} = (\lambda_0, \lambda_1, ..., \lambda_k) \qquad k = |\mathcal{L}|$$

If the label space $\mathcal{L}$ only consists of one class ($|\mathcal{L}| = 1$) the problem is called **binary classification**. In this case the output vector y consists of only one value that is 1 if an instance belongs to this class and 0 otherwise.
If $\mathcal{L}$ consists of more than one class ($|\mathcal{L}| > 1$) the problem is called **multiclass classification** with k unique classes $\{\lambda_1, ..., \lambda_k\}$. The goal is now to predict a single relevant class label for each instance. So we have to add the restriction $|y| = 1$.

For **multilabel classification** we also have more than one possible class label that can be assigned to an instance. But instead of being restricted to assign only one label, it allows to assign multiple labels to the same instance.

$$|\mathcal{L}| = k \qquad 0 \leq |y| \leq k$$

We also introduce some additional definitions: Relevant class labels that are assigned to an instance ($y_i = 1$) are called **positive labels** and class labels that are not assigned as relevant labels ($y_i = 0$) are called **negative labels**. Table 1 shows an example for how labels are represented. Therefore, we assume

that we have four books, numbered in the first column from one to four. These books can belong to up to four categories *Adult, Youth, Fantasy* and *Thriller*. If a category is assigned to an instance and therefore a positive label, it is represented by a tick (✓). If a category is not assigned and a negative label, the corresponding cell is empty. The last column then shows how the labels for an instance are represented by the previous introduced vector $y^{(i)}$, where $y^{(i)} = \{y_{Adult}, y_{Youth}, y_{Fantasy}, y_{Thriller}\}$. So if we take for example the second instance, we can see that it has the first label *Adult* and last label *Thriller* marked as positive. Therefore the output vector $y^{(2)}$ contains 1 for the values y_{Adult} and $y_{Thriller}$.

	λ_{Adult}	λ_{Youth}	$\lambda_{Fantasy}$	$\lambda_{Thriller}$	Representation
1	✓		✓	✓	$y^{(1)} = (1,0,1,1)$
2	✓			✓	$y^{(2)} = (1,0,0,1)$
3		✓	✓		$y^{(3)} = (0,1,1,0)$
4					$y^{(4)} = (0,0,0,0)$

Table 1: Example: multilabel dataset for book tagging

2.2 Baseline Approaches

Most of the existing methods for multilabel classification belong to one of these two categories: **problem transformation** and **algorithm adaption**.(Tsoumakas and Katakis, 2006)
Problem transformation divides the problem into one or more single label, multiclass-classification or regression tasks. The advantage of such a transformation is that single-label-classifiers, which provide good and robust results in their domain, can be applied to solve these smaller subtasks. The idea is basically that a combination of some state-of-the-art single-label classifiers should also lead to some good results for multilabel problems.
The second approach attempts to address the problem in its original form without any transformations or preprocessing steps. This is achieved by modifying existing algorithms to make them capable of handling multilabel problems. This chapter will give a small overview over existing multilabel algorithms from both categories. They will later be taken as baseline approaches to compare current state-of-the-art approaches to the dynamic-chain algorithm, presented in this thesis.

2.2.1 Label Powerset

The first baseline approach is the label powerset method (Tsoumakas and Katakis, 2006). This algorithm belongs to the category of problem transformation methods. The original multi-label problem is transformed into a single-label multi-class classification problem. The idea ist to replace every possible and unique subset of labels, that occurs in the given dataset, with a single class label. For a new instance the label powerset algorithm gives the most probable class as an output, which can then be mapped back to a subset of labels. Table 2 illustrates this idea.

	Y		Y_{LP}
$x^{(1)}$	$y^{(1)} = (1,0,1,1)$		$y_{1,3,4}$
$x^{(2)}$	$y^{(2)} = (1,0,0,1)$		$y_{1,4}$
$x^{(3)}$	$y^{(3)} = (0,1,1,0)$		$y_{2,3}$
$x^{(4)}$	$y^{(4)} = (1,0,1,1)$		$y_{1,3,4}$
$x^{(5)}$	$y^{(5)} = (0,1,0,0)$		y_2

Table 2: Example: Label Powerset Transformation

Column Y shows the relevant label set for an instance from the dataset, while column Y_{LP} shows the same instances but with their label-set transformed into a *label power set*. The class $y_{i,j,...,k}$ means that

the instance belongs to the respective label subset with labels $\{y_i, y_j, ..., y_k\}$. The table also shows that instances with the same label subset are mapped to the same class-label (in this example instances 1 and 4).
Although every multiclass-algorithm can be applied to this transformation, the task can easily become rather challenging if there is a tremendous number of unique label sets and therefore possible Label Powerset classes. The computational complexity is upper bounded by $min(n, 2^k)$ with n is the number of data instances and k the number of possible labels of the multilabel dataset.
Another important drawback of Label Powerset methods is called the *class imbalance problem* (Guo et al., 2008). This means that we have many instance assigned to one class-label, but only a few number of instances assigned to another class-label. With such an imbalance it is very hard to train a model that produces good results, because most of the classifiers tend to predict mostly the major occurring class-label and ignore the minor one.

2.2.2 Binary Relevance

Another approach, which is also one of the most popular ones, for problem-transformation methods is Binary Relevance (Cherman et al., 2011). This method decomposes the multilabel problem into a set of k binary classification problems where k is the number of class-labels ($k = |\mathcal{L}|$). Each of these problems takes the same datasets for training but adds a different single class-label as the target value, which is positive if the instance belongs to class λ_j (with $l \leqslant j \leqslant k$) and negative otherwise. After the dataset is transformed, a binary classifier for each set is constructed and trained. Table 3 illustrates this basic idea of transforming the dataset.

	Y
$x^{(1)}$	$y^{(1)} = (1,0,1,1)$
$x^{(2)}$	$y^{(2)} = (1,0,0,1)$
$x^{(3)}$	$y^{(3)} = (0,1,1,0)$
$x^{(4)}$	$y^{(4)} = (1,0,1,1)$
$x^{(5)}$	$y^{(5)} = (0,1,0,0)$

$\Longrightarrow$

	Y_1
$x^{(1)}$	1
$x^{(2)}$	1
$x^{(3)}$	0
$x^{(4)}$	1
$x^{(5)}$	0

	Y_2
$x^{(1)}$	0
$x^{(2)}$	0
$x^{(3)}$	1
$x^{(4)}$	0
$x^{(5)}$	1

	Y_3
$x^{(1)}$	1
$x^{(2)}$	0
$x^{(3)}$	1
$x^{(4)}$	1
$x^{(5)}$	0

	Y_4
$x^{(1)}$	1
$x^{(2)}$	1
$x^{(3)}$	0
$x^{(4)}$	1
$x^{(5)}$	0

Table 3: Example for Binary Relevance Transformation

A big drawback of Binary Relevance is that it does not take label dependencies into account (Luaces et al., 2012). If there are obvious dependencies the prediction of specific or obvious combinations may fail because each label is independently predicted by a classifier which has zero knowledge about other labels.
Besides this problem there are also some advantages of this method. The first one is that any binary classifier can be taken as a base classifier. So a specific classifier which is known to perform well on a special structured dataset can be taken and even different classifiers can be easily swapped for comparisons.
Another advantage is that the computational complexity is linear with respect to the number of labels. But even on datasets with a large number of labels it can perform fast because, due to the fact that the classifiers are independent of one another, the training-process can be easily parallelized.

2.2.3 Classifier Chains

The last example for label-transformation-methods are classifier chains (Read et al., 2011). In contrast to binary relevance this approach attempts to model and exploit statistical dependencies between labels. Similar to binary relevance, $|L|$ binary classifiers are involved. These classifiers are linked along a chain where each classifier deals with a binary problem which is associated to a single label. In order to learn dependencies between labels, the feature space of each classifier is extended with the 0/1 label predictions of all previous classifiers of the chain. Hence a chain $C_1, ..., C_k$ of binary classifiers is formed where

each classifier C_j, with $1 \leqslant j \leqslant k$, is learning and predicting a binary label λ_j.
For predicting the labels of a new instance, the chain begins with classifier C_1, that predicts the first label λ_1 where l_1 denotes the prediction for the given instance: $l_1 = h_{l_1}(x)$. This result is then propagated along the chain and the second classifier predicts the second label λ_2, given the predictions of the first classifier: $l_2 = h_{l_2}(x, l_1)$. This step is repeated for each link and at the end the last classifier will predict the last label λ_k with $l_k = h_{l_k}(x, l_1, l_2, ..., l_{k-1})$. Table 4 depicts this propagation process for predicting new instances.

X	Y
$\mathbf{x}$	$\mathbf{h_{l_1}(x)}$
$x^{(1)}$	1
$x^{(2)}$	1
$x^{(3)}$	0
$x^{(4)}$	1
$x^{(5)}$	0

$\Rightarrow$

X		Y
$\mathbf{x}$	$\mathbf{l_1}$	$\mathbf{h_{l_2}(x, l_1)}$
$x^{(1)}$	1	0
$x^{(2)}$	1	0
$x^{(3)}$	0	1
$x^{(4)}$	1	0
$x^{(5)}$	0	1

$\Rightarrow$

X			Y
$\mathbf{x}$	$\mathbf{l_1}$	$\mathbf{l_2}$	$\mathbf{h_{l_3}(x, l_1, l_2)}$
$x^{(1)}$	1	0	1
$x^{(2)}$	1	0	0
$x^{(3)}$	0	1	1
$x^{(4)}$	1	0	1
$x^{(5)}$	0	1	0

Table 4: Propagating label information along the classifier chain

By passing the label information through the chain, possible label dependencies can be taken into account, which helps to overcome the problem of the label-independent binary relevance model. Besides this improvement classifier chains also retain most of the advantages from binary relevance models (Read et al., 2011): The number of binary classifiers stays the same so the computational complexity is similar and the base-learner can easily be changed to adapt best to the problem. Although classifier chains can not be parallelized, because each classifier depends on the one further ahead in the chain, the problem is serializable so that only one binary classifier has to be kept in memory at a time, which is beneficial compared to some other approaches.
Nevertheless, one problem remains: The order of the chain decides the order of label dependencies. If the chain predicts the labels in the sequence $\lambda_1 \rightarrow \lambda_2 \rightarrow \lambda_3$ the assumptions, that label 2 depends on label 1 and label 3 depends on label 1 and 2, are made. This order may be wrong on a specific dataset. So the need arises to fit the order of label classifiers to the real label-dependencies of the dataset, which are not known in advance. Therefore, often some heuristics can be applied for selecting the chain order. Another approach to address this problem is the application of ensemble models. Therefore multiple classifier chains with random chain orders are trained and the predictions are determined by voting the predictions from all chains.

2.2.4 Multi-Target Regression Tree

The last baseline method belongs to the category of algorithm adaption methods and is based on regression trees (Kocev et al., 2009). These trees consist of inner nodes, branches and leaves. The inner nodes always contain tests on the input features and the branches correspond to the outcome of a test and connect the nodes with child nodes. Each leaf of a regression tree contains a numeric value as a prediction for the target label. In order to obtain the predictions for a test-instance we pass it, starting at the root-node, along the tree. This is done by applying the test of a tree-node and propagate the instance, depending on the outcome of the test, to the corresponding child node or leaf. The only difference of multi-target trees is that their leaves do not just contain a single value, but a vector with a prediction for each target label. Figure 6 shows a example for a multi-target tree and section 4.1 gives a further introduction to the structure and construction of these trees.
The main advantages of multi-target trees over multiple single-target trees (Struyf and Džeroski, 2005) is that the final model is much smaller than the size of an ensemble that contains a single tree for each label. The second and more important advantage is that a multi-target tree can explicitly represent

dependencies between the target labels which is a great benefit compared to the label-transformation approaches.

3 Extreme Gradient Tree Boosting

This chapter gives an introduction to extreme gradient tree boosting algorithms and especially XGBoost[1]. Thereby we follow (Chen and Guestrin, 2016) for the description of this algorithm. XGBoost is a current state-of-the-art end-to-end tree boosting system which is mostly used to solve machine learning tasks for regression, multiclass and ranking problems. It is widespread and especially used in machine-learning and data-mining challenges on websites like Kaggle[2], where it can often be found applicated in top solutions.
This is mainly due to the fact that XGBoost is highly scalable in different scenarios. Because of its efficient and optimized implementation it can deal with up to billions of examples on a single machine and is also applicable on distributed systems. In addition to that interfaces for most of the major programming-languages are supplied, which make it fast and easy to setup and work with the algorithm.

3.1 Model and Learning Objective

Basics of Supervised Learning

We start with the basic idea of supervised learning and follow. (Chen, 2014)
The basic goal is to find a model that generates a prediction $\hat{y}_i$ given a feature vector x_i.
The simplest case to do this is a linear model, where the prediction is a linear combination of weighted input features:

$$\hat{y}_i = \sum_j \theta_j x_{ij} \tag{1}$$

θ denotes the parameters which need to be learned from the data. $\theta = \{w_j | j = 1, ..., d\}$
In order to find the best parameters, we have to evaluate a model and measure its performance for a given set of parameters. Therefore, a so-called objective-function has to be defined.

$$Obj(\theta) = L(\theta) + \Omega(\theta) \tag{2}$$

This objective-function consists of two parts:

- **Training Loss L** measures how well a model fits on data
- **Regularization Term Ω** measures the complexity of a model

Tree Ensemble Model

These basics of supervised learning can now be applied to XGBoost. The internal model of XGBoost is a tree ensemble model, or more specific a set of classification and regression trees (short: CART). Each tree consists of nodes, leaves and branches which connect these elements. Figure 1 shows two examples for CART decision trees. A node is always connected with two child nodes (or leaves) and contains a decision test to decide to which child a specific instance will be forwarded. The leaves of a tree always contain a score that represents the prediction of the tree, if an instance ends up in this leaf. In order to predict the final output value the scores of all trees are summed up, this means the model uses C additive functions:

$$\hat{y}_i = \phi(x_i) = \sum_{c=1}^{C} f_c(x_i), \quad f_c \in F \tag{3}$$

$$\text{where } F = \{f(x) = w_{q(x)}\}(q : \mathbb{R}_m \to T, w \in \mathbb{R}^T) \tag{4}$$

[1] https://github.com/dmlc/xgboost
[2] https://www.kaggle.com/

Each f_c is an individual tree structure. F is the space of all possible regression trees and q represents the structure of a tree, that maps an example to the corresponding leaf index, where T is the number of leaves in the tree. Each tree contains a continuous score in its leaves (w_i is the score for the i-th leaf). In order to calculate a new prediction for an instance, the decision rules q, inside the inner nodes of each tree, are used and the scores of corresponding leaves are summed up (shown in Figure 1).

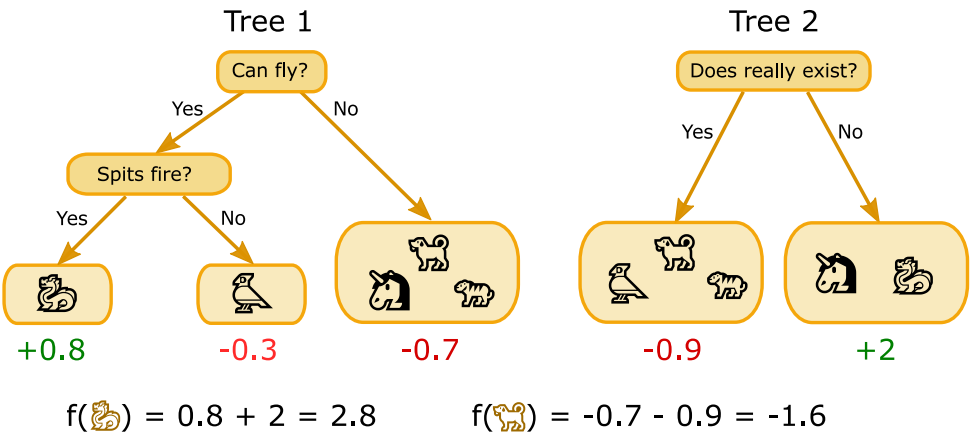


Figure 1: Tree Ensemble Model

To learn this additive model, we get this regularized objective function to optimize:

$$L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_c) \tag{5}$$

$$\text{where} \quad \Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \tag{6}$$

l is a convex differentiable loss function that measures difference between prediction $\hat{y}_i$ and target y_i value. Ω penalizes the complexity of the model in order to avoid overfitting.

3.2 Additive Training and Tree Boosting

Due to the fact that the objective function contains functions as parameters, it cannot be optimized with traditional methods. So the model is trained in an additive manner. The prediction value at each iteration t is then calculated based on the prediction value from the previous iteration:

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{c=1}^{t} f_k(x_i) = \hat{y}_i^{(t-i)} + f_t(x_i)
\end{aligned} \tag{7}$$

$\hat{y}_i^t$ denotes the prediction of the i-th instance at the t-th iteration. This additive calculation can then be inserted into the objective function:

$$L^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \tag{8}$$

In order to minimize this objective, we add an additional tree structure f_t that improves the model most. For optimizing this additive objective, we take the second-order approximation of it:

$$\begin{aligned} L^{(t)} &\simeq \sum_{i=1}^{n}[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t) \\ \text{where } & g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \\ \text{and } & h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \end{aligned} \tag{9}$$

This can again be simplified by removing all constant terms, so we get the objective at iteration t

$$\tilde{L}^{(t)} = \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t) \tag{10}$$

This is now the goal for optimizing a new tree. The big advantage is that we got rid of the explicit loss function inside the objective. It now depends only on the values of g_i and h_i that are derived from the loss function. This reveals another big advantage of XGBoost: Every loss function can be used as a training goal and can easily be optimized. The only requirement is the need to calculate its first and second order derivatives.

3.3 Tree Construction and Leaf Weights

Now there is a simpler objective to optimize, but the questions, how to construct a new tree and how to determine the leaf weights, still remain. Therefore, two points have to be considered: On the one hand side a new tree has to optimize the training loss, and on the other hand it has to maintain a low complexity to get no large penalties. So we take again the objective function, but with inserted regularization term:

$$\tilde{L}^{(t)} = \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{11}$$

After constructing a new tree, the most important parts are the leaves, because they determine the final predictions. So we define $I_j = \{i|q(x_i) = j\}$ as the instance set of leaf j. With this definition we can again rewrite the objective function

$$\tilde{L}^{(t)} = \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2] + \gamma T \tag{12}$$

The summation also changed because all datapoints in the same leaf also get the same scores. This expression can be even further compressed by defining $G_j = \sum_{i\in I_j} g_j$ and $H_j = \sum_{i\in I_j} h_j$

$$obj^{(t)} = \sum_{j=1}^{T}[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T \tag{13}$$

Now we are given a quadratic term, which allows us to compute the optimal weights $\star w_j$ of leaf j of a given tree structure $q(x)$:

$$\begin{aligned} w_j^* &= -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda} \\ &= -\frac{G_j}{H_j + \lambda} \end{aligned} \tag{14}$$

It also allows to measure the quality of the complete tree structure q

$$\tilde{L}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T$$
$$obj^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \tag{15}$$

Figure 2 shows an example for calculating the leaf-scores. Basically, all instances of the dataset are assigned to their corresponding leaves, together with their statistics g_j and h_j. These statistics are then summed up to calculate the overall score. At the same time this score also takes the complexity of the tree into account by adding γT.

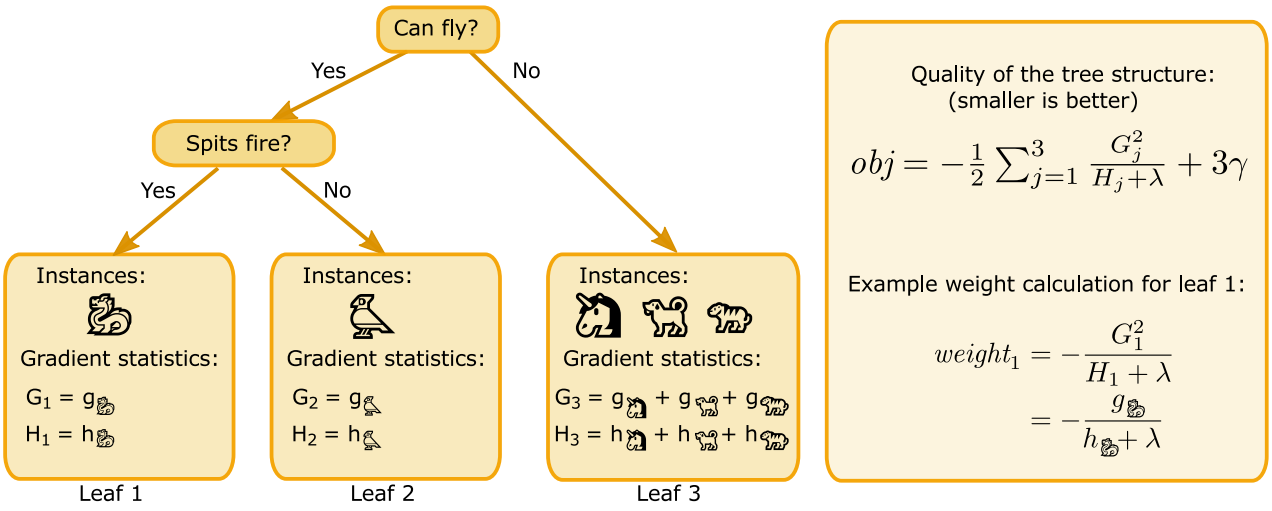


Figure 2: Example for leaf weight and tree quality score calculation

3.3.1 Split Finding

Now we can measure the quality of trees. The ideal idea would be to enumerate all possible tree structures, evaluate them and choose the best one. In practice this approach is not possible and highly inefficient. Instead a greedy algorithm is used that starts with a single leaf and tries iteratively to split it up and add branches. So a possible split candidate consists of four parts:

- I_L: Instance set of left leaf after splitting
- I_R: Instance set of right leaf after splitting
- I: Instance set of the original leaf before splitting with $I = I_L \cup I_R$
- γ: Regularization for adding a new leaf

$$L_{split} = \frac{1}{2}[I_L + I_R - I] - \gamma$$
$$L_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \tag{16}$$

For choosing the best splits an *exact greedy algorithm* is used. The algorithm is shown in Alg. 1 and taken from (Chen, 2014). It enumerates over all possible splits on all features. In order to reduce the computational complexity the data is sorted according to the evaluated feature. Thereby the algorithm starts with the original leaf and a new empty leaf. All possible splits are evaluated by removing instances from the original leaf and adding them in their ordered sequence to the new leaf. At the end the split with the best gain-score is chosen and the decision test can be easily be set by choosing the first element of the new leaf. Figure 3 also shows an example for selecting this decision test.

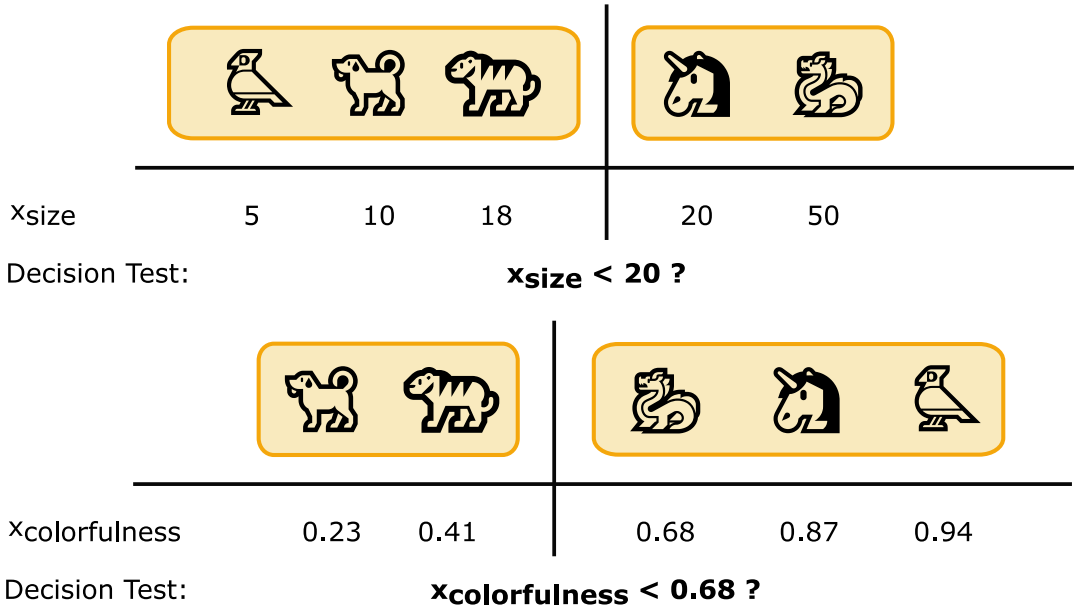


Figure 3: Linear scan over presorted columns to find best split

In this example we have proposed two different splits. Since the candidates are chosen by ordering the instance by a certain attribute, we have also sorted the instances by an attribute. The first attribute that is used for sorting is x_{size} which represents the total size of an animal. So the bird is the smallest one with a size of 5 and the dragon is the biggest animal and has a size of 50. For the second candidate we used $x_{colorfulness}$ for sorting. This example attribute is meant to describe how colorful an animal is. So the bird with a feathering that can have any color, gets the highest score. If we now assume that for both examples the score for the displayed split is the highest, we can easily determine the decision test. For x_{size} the first element in the right child is the unicorn with a size of 20. So the corresponding decision test is: $x_{size} < 20$ which means: *Is the size of an animal smaller than 20 ? If yes, go to the left leaf, and if no, go to the right leaf.*
One problem of this approach occurs when the data does not fit into memory and an efficient exact greedy split finding becomes impossible. In this case an approximation algorithm is used. In a first step split candidates, according to percentiles of the feature distribution, are proposed. Afterwards all continuous features are mapped to one split candidate, determined by the proposed split-point, and the scores are calculated in order to find the best split. This proposal can be global, where it is only performed once for all splits in the tree, or it can be local, where new candidates are proposed for each split individually.

Missing Values

In many real world datasets we have the case that not all instances contain values for each attribute. These values are called *missing values* and create the problem that we cannot sort them to find the best split and that the decision tests of the nodes cannot be performed. Therefore the basic split-finding algorithm can be further extended to give it the ability to deal with these missing feature values. This can be achieved by adding a small extension to the split finding: Besides the two links to the left and right

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input : I, instance set of current node
Input : d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i,\ H \leftarrow \sum_{i \in I} h_i;$
for $k = 1$ ***to*** m **do**
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted(I, by* x_{jk}*)* **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_L \leftarrow H - H_L$
    $score \leftarrow max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

child node, a third connection is learned which represents the *default direction* that points to one of the nodes. Whenever a decision test cannot be applied because the value is missing, this default direction is taken. Therefore each split-candidate evaluation is divided into two cases. The first one assumes that all instances with missing labels are added to the left child, and the second one assumes all these instances to be added to the right node. Before ordering the instances by their feature value, the missing labels are sorted out and added to the corresponding candidate. The gain-score calculation then continues as before.

4 Learning a Dynamic Chain of Boosted Tree Classifiers

The basic idea of this thesis is to overcome the problems of current baseline approaches and to develop a model that produces more accurate label predictions. The most important point is the ability to detect and exploit label dependencies. Instead of using a static chain which predicts labels in a rigid order, a new chain approach is developed that predicts labels in an individual order for each data-instance. Therewith each instance is only assigned to the labels which are most probable to be a solution in this case.
More specific this means that each classifier inside the chain is allowed to assign one label to an instance. Classifiers further along the chain can then use this additional information of previous set labels to learn possible label-dependencies. Therefore each base classifier has to maximize the probability of a single label in order to make sure that later classifiers base their result only on labels with high probabilities of being correct.
Because of the really good performance of XGBoost classifiers, this algorithm has been chosen to be used for the base classifiers inside each chain-node. In order to deal with multiple labels the original XGBoost implementation has to be modified. These new XGBoost models differ from the original algorithm by using an adapted tree structure that is able to predict an arbitrary number of labels at once.

4.1 Multi-Target Trees

The first step is to adapt XGBoost to make it capable of solving multilabel tasks. So far we have to fall back on label-transformation strategies where each label is learned by an individual tree structure. Afterward the predictions are combined into a vector where each label is assigned to a score. Therefore we introduce, in the following sections, an adaption of the current XGBoost tree structure that allows us to predict multiple labels simultaneously.

4.1.1 Tree Model

Up until now XGBoost is especially used for solving regression problems. If we take a look at this approach, we can see that a single tree is learned for each boosting round. Inside of inner tree-nodes there is always a decision test that determines which path to take next and inside the leaf-nodes there is always a score which represents the prediction. Figure 4 shows an example for a XGBoost regression tree that was visualized using *Graphviz* (Ellson et al., 2001). The caption for each node thereby shows the decision test that is performed to determine where to proceed. The second line of a node always shows the gain score that represents the quality of the given node which we want to maximize in the split evaluations. The last line corresponds to the number covered instances and is calculated as the sum of the hessian values for all instances covered by the node. The connections between the nodes are the branches and show where to pass an instance after the decision test has been applied. If the test instance contains a missing value for the tested attribute, the *missing* path is taken. The leaves finally contain the scores which represent the predictions for this particular tree and a cover value that is calculated as before in the inner nodes. Besides regression, XGBoost is also capable of dealing with multiclass problems. This approach is also called *one-against-all* (Fürnkranz, 2001) and basically the same as *binary relevance*, where a single classifier is learned per class and not per label. So in this case an individual tree is learned for each possible class. If we then want to get the prediction for a new instance, we propagate this instance through every one of these trees. At the end each tree returns a score for assigning its class to the given instance. These classes are then mapped into a vector, where we get a probability for each class, which sum up to one. Figure 5 depicts this process for three classes *01, 02* and *03*. Each of these trees is trained for a single label. If we now pass the given test instance through each tree, we get a prediction for each class. The path that the test instance takes and the corresponding predictions are thereby highlighted green. At the end these predictions are then scaled to sum up to one.
With this approach a basic idea could be to represent each label as an individual class and then use XGBoost multiclass-learning to solve the task. But this method again shows some problems that we

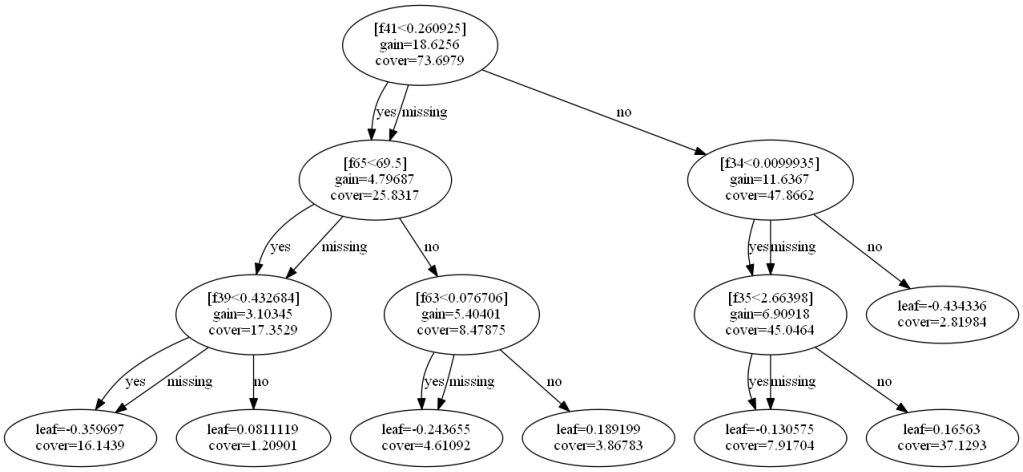


Figure 4: Example for a XGBoost Regression Tree

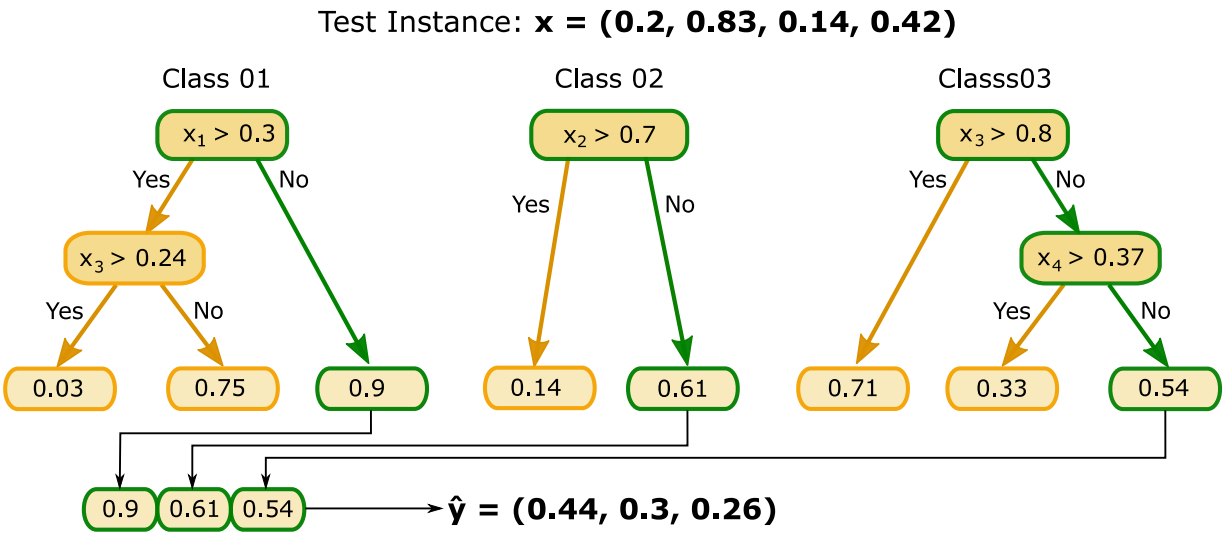


Figure 5: XGBoost multiclass prediction process

have to address. The first one is that this method generates predictions that sum up to one, but for multilabel classification we need absolute probabilities, which are not scaled, for each label. This is mainly a technical problem of the current XGBoost implementation which we could solve with some modifications. But the second problem is the reason why we cannot use this approach. Since the trees are independently learned for each label, we again do not respect dependencies between label. As mentioned before the way this multiclass problem is addressed is the same as *binary relevance* which is also not able to detect label dependencies.

What we now want to achieve is a single tree that can predict a probability for each label. We do not enforce these probabilities to sum up to one, so the tree can predict high probabilities for multiple labels. Additionally each tree should try to maximize the probability of a particular relevant label. The basic idea is to evaluate split candidates among all labels and take the one that increases the probability of a single label most. Figure 6 shows a tree that fulfills these requirements. The basic structure is same as before and the only differences are the leaves where each now contains a vector of probabilities. The probabilities thereby are the probabilities that a label is a positive one. So a high probability for λ_1 means that this label is very likely a positive one, whereas a low probability is an indicator for a negative label.

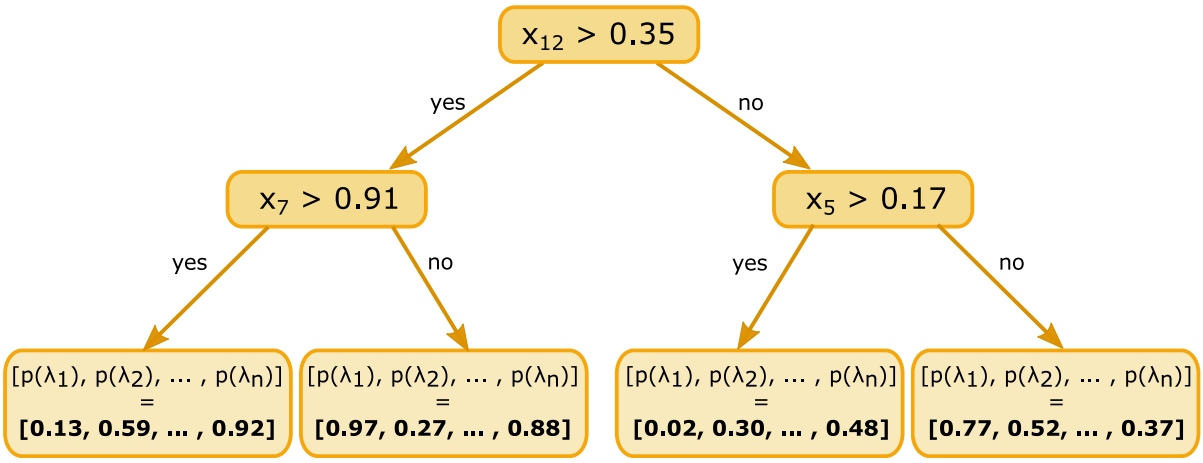


Figure 6: Multilabel Tree Structure

4.1.2 Tree Construction

specified the tree structure, we have to determine how the tree is constructed. The goal is, to split up each tree node to maximize at least one of the label probabilities. As we have seen before (in Section 3.3), the gain of a leaf is calculated by

$$\frac{G^2}{H+\lambda} \tag{17}$$

So G and H are the sums of gradient and hessian values of the instances contained in the leaf. To calculate them we need to know the objective function. We decided to use cross-entropy for our loss function

$$l(y,\hat{y}) = -y\log(\hat{y}) + (1-y)\log(1-\hat{y}) \tag{18}$$

where y denotes the real target label and $\hat{y}$ is the prediction for the label. This loss function is commonly used for logistic regression and even the standard loss for XGBoost regression tasks. The raw-predictions p for an instance are calculated by summing up the cross entropy results of all boosting trees for each label. But since we need to get a prediction between 0 and 1 for each label, we apply a sigmoid transformation on p to get the final predictions $\hat{y}$.

$$\hat{y} = \frac{1}{1+e^{-p}} \tag{19}$$

$$l(y,\hat{y}) = -y\log(\frac{1}{1+e^{-p}}) + (1-y)\log(1-\frac{1}{1+e^{-p}}) \tag{20}$$

If we now take into account that the final prediction $\hat{y}$ is the sigmoid transformation of the raw leaf weights p we can easily calculate the first order derivative of the cross-entropy to get G:

$$\begin{aligned} g = \frac{\partial L}{\partial p} &= -\frac{e^p(y-1)+y}{e^p+1} \\ &= -\frac{e^p y e^{-p} - e^p e^{-p} + y e^{-p}}{e^p e^{-p} + e^{-p}} \\ &= -\frac{y-1+ye^{-p}}{1+e^{-p}} \\ &= \frac{1}{1+e^{-p}} - \frac{y\cdot(1+e^{-p})}{1+e^{-p}} \\ &= p-y \end{aligned} \tag{21}$$

We also need the sum of the second derivatives H which is calculated equally to the first derivative:

$$\begin{aligned} h = \frac{\partial^2 L}{\partial p} &= \frac{e^p}{(1+e^p)^2} \\ &= \frac{e^{-p}}{(1+e^{-p})^2} \\ &= \frac{1}{1+e^{-p}} \cdot \frac{e^{-p}}{1+e^{-p}} \\ &= \frac{1}{1+e^{-p}} \cdot \left(\frac{1+e^{-p}}{1+e^{-p}} - \frac{1}{1+e^{-p}} \right) \\ &= p\cdot(1-p) \end{aligned} \tag{22}$$

In order to deal with multiple labels, we have to extend the basic gain function (18). Therefore we introduce the notations G_l which denotes the sum of all gradient values of instances in a leaf for label λ_l and H_l for the sum of the hessian values. The new gain function now has to take all G_l and H_l (for $1 \leqslant l \leqslant k$) into account. So we have evaluated different methods to find the best split candidate:

1) **Maximum default gain over all labels:** The first and simplest idea is to calculate the gain in the same way as default XGBoost did before, but take the maximum over all labels. So we can make sure to take the split candidate that maximizes one label-score, and therefore also the corresponding probability, most:

$$\text{gain}_1 = \max_{1\leqslant l\leqslant k} \left(\frac{G_l^2}{H_l+\lambda} \right)$$

The condition $1 \leqslant l \leqslant k$ thereby means that we perform the calculation for each label to select the one with the highest score. This approach causes also a problem. The final weights inside the leaf are calculated with $-\frac{G}{H+\lambda}$ where H will be always ≥ 0. So if G is negative, we will get a prediction near 1 after applying the sigmoid, and for positive G a prediction near 0. So if we now calculate the gain by taking the squared G value, we do not only favor probabilities near 1, but also probabilities near 0. So this approach also encourages to predict labels with low probabilities which is the same as not assigning a label to an instance or predicting a negative label.

2) **Maximum leaf score over all labels:** In order to prevent the minimization of probabilities to predict negative labels, we can modify the gain calculation by not squaring G and instead directly try to maximize the leaf weights, which represent the final predictions of a tree.

$$\text{gain}_2 = \max_{1\leqslant l\leqslant k} \left(\frac{-G_l}{H_l+\lambda} \right)$$

Now we only optimize positive labels and do no longer also take negative labels into account. But both of the presented gain-calculations share a property that may be a drawback: Their final decision, which split to take, is only based on one single label, which is actually the label with the highest probability among all labels. In this case we ignore all remaining labels. So a situation may occur where we take a split that indeed maximizes one label, but at the same time the remaining labels probabilities become worse. At this point taking a split where a single label may be increased by a lower amount, but at the same time the remaining labels also get higher probabilities, can be a better decision.

3) **Sum of default gains over all labels:** To consider all labels for evaluating a new split candidate, this approach takes the sum of all gain-scores over all labels.

$$\text{gain}_3 = \sum_{l=1}^{k} \left(\frac{G_l^2}{H_l + \lambda} \right)$$

This method is just an extension of the first approach for split-finding. Again, maximizing the score can result in either a label with a high probability which can then be seen as a positive label, or in a label with a low probability which then represents a negative label. The basic idea is that if all labels have high scores for being positive (or in this case negative) labels, the sum of these scores is also a higher value than for labels with non-optimized probabilities. So now we can take a split where we favor all labels to have high probabilities, instead of the one where only one label has a very high probability.

4) **Sum of leaf scores over all labels:** The next step is to apply this approach to the the second method. We can now again take the sum over all labels, but this time negative labels will reduce the score, which will lead to a split that maximizes only scores for positive labels.

$$\text{gain}_4 = \sum_{l=1}^{k} \left(\frac{-G_l}{H_l + \lambda} \right)$$

5) **Maximum absolute leaf score over all labels:** The last idea is to allow score-maximizing for negative labels on purpose and find a split which is a good compromise of positive labels with high probabilities and negative labels with low probabilities. Therefore we take the absolute value of the default calculated weight and maximize it again over all labels.

$$\text{gain}_5 = \max_{1 \leqslant l \leqslant k} \left(\left| \frac{-G_l}{H_l + \lambda} \right| \right)$$

Now the highest score can be assigned to a label with a low probability, which then can be set as a negative label.

6) **Sum of absolute leaf scores from all labels:** The last method now is based on the same idea and uses the absolute value of the calculated leaf-weights. This is then combined with the approach of summing up the scores of all possible labels to get a high score over all labels.

$$\text{gain}_6 = \sum_{l=1}^{k} \left(\left| \frac{-G_l}{H_l + \lambda} \right| \right)$$

After the whole tree has been constructed, the last task is to assign the weights for each label inside the leaves. This is done the same way as before. Since we know all the instances inside a leaf and their corresponding gradient and hessian values per label, we can calculate the weight for label l, just as before:

$$\text{weight}_l = \frac{-G_l}{H_l}$$

4.1.3 Example for Split Finding

To give a better understanding of how an actual tree is constructed we give an example. Therefore we will use the `Extended Weather` dataset (Loza Mencía and Janssen, 2016) shown in table 5. We propose two possible splits and evaluate both to determine which one to choose for getting the best prediction results.

index	outlook	temperature	humidity	windy	play	icecream	tea	lemonade	dontplay
1	rainy	65	70	yes	0	0	1	0	1
2	rainy	71	90	yes	0	0	1	0	1
3	sunny	85	85	no	0	1	0	1	1
4	sunny	80	90	yes	0	1	0	1	1
5	sunny	72	95	no	0	1	0	1	1
6	sunny	69	70	no	1	0	0	1	0
7	sunny	75	70	yes	1	0	0	1	0
8	overcast	83	86	no	1	0	0	1	0
9	overcast	64	65	yes	1	0	0	1	0
10	overcast	72	90	yes	1	0	0	1	0
11	overcast	81	75	no	1	0	0	1	0
12	rainy	70	96	no	1	0	0	1	0
13	rainy	68	80	no	1	0	0	1	0
14	rainy	75	80	no	1	0	0	1	0

Table 5: Extended Weather Dataset

The first step is to calculate the gradient and hessian of the cross-entropy loss function for each instance and label. We also assume that this is the first round and no previous trees exist and therefore state that every initial prediction for an instance-label-combination is 0.5. This assumption makes it easy to calculate the corresponding derivative values. The hessian only depends on the prediction and is therewith the same for each instance: $h = 0.5 \cdot (1 - 0.5) = 0.25$. The gradient is also easy to calculate and is -0.5 if the true label is 1 and 0.5 otherwise. Table 6 shows the results of this first calculation step.

Now that we have a dataset and the gradient / hessian values for each instance and label combination, we need to define two splits to evaluate. So for the first split we simply take all instances for the negative label `play` into the right leaf-candidate and all instances for the positive label into the left leaf-candidate. For the second split we take a distribution that has no obvious structure and take all instances with an odd index into the left leaf and the remaining instances with an even index into the right one. Figure 07 illustrates both split candidates.

These two splits should now be evaluated according to the first proposed method of gain calculation. This process can be easily divided into six steps:

1. Calculate the sums $G = \sum_k g_l$ and $H \sum_k h_l$ of all hessian and gradient values for each label
2. Calculate the gain value for each label by applying $gain = \frac{G^2}{H}$ (The normalization term λ is ignored in this example, due to simplification.)
3. Compare the gain values of the labels and take the one with the highest score and save it as the gain-score for this particular leaf $gain_{leaf} = max(gain_l)$
4. Repeat steps 1-4 for the second leaf

index	play		icecream		tea		lemonade		dontplay	
	g	h	g	h	g	h	g	h	g	h
1	0.5	0.25	-0.5	0.25	-0.5	0.25	0.5	0.25	-0.5	0.25
2	0.5	0.25	-0.5	0.25	-0.5	0.25	0.5	0.25	-0.5	0.25
3	0.5	0.25	-0.5	0.25	0.5	0.25	-0.5	0.25	-0.5	0.25
4	0.5	0.25	-0.5	0.25	0.5	0.25	-0.5	0.25	-0.5	0.25
5	0.5	0.25	-0.5	0.25	0.5	0.25	-0.5	0.25	-0.5	0.25
6	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
7	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
8	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
9	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
10	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
11	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
12	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
13	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25
14	-0.5	0.25	0.5	0.25	0.5	0.25	-0.5	0.25	0.5	0.25

Table 6: Gradient and Hessian Values for Weather Dataset

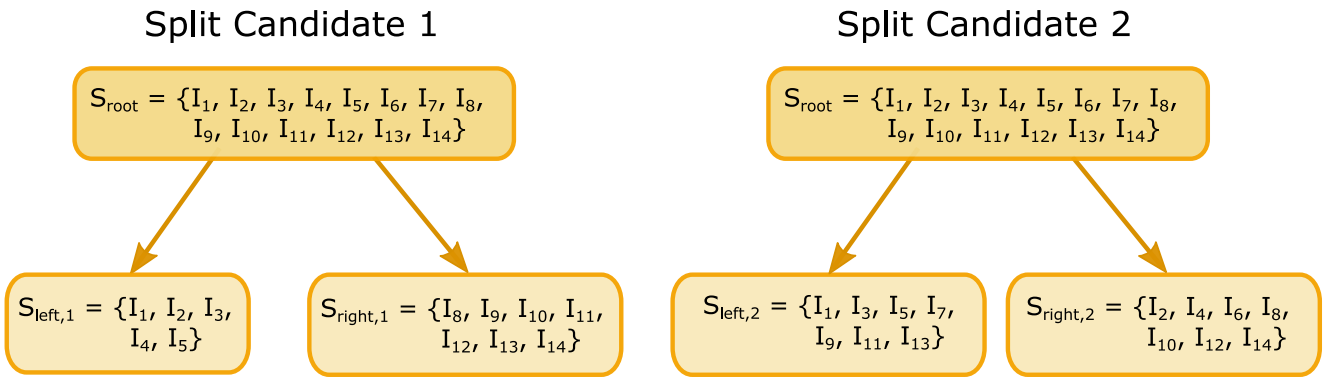


Figure 7: Candidates for split evaluation

5. Take the sum of the gain-scores from both leafs to get the final gain-score for this split candidate $gain_{total} = gain_{left} + gain_{right}$

6. Compare the scores of different candidates to find the best split

	Candidate 1	
	$leaf_{1,left}$	$leaf_{1,right}$
Instances	$[1,2,3,4,5]$	$[6,7,...,14]$
G_l	$[2.5,-2.5,0.5,-0.5,-2.5]$	$[-4.5,4.5,4.5,-4.5,4.5]$
H_l	$[1.25,1.25,1.25,1.25,1.25]$	$[2.25,2.25,2.25,2.25,2.25]$
$gain_l$	$[5,5,0.2,0.2,5]$	$[9,9,9,9,9]$
$gain_{leaf}$	5	9
$gain_{total}$	14	
$weight_l = -G_l/H_l$	$[-2,2,-0.4,0.4,2]$	$[2,-2,-2,2,-2]$

	Candidate 2	
	$leaf_{2,left}$	$leaf_{2,right}$
Instances	$[1,3,5,7,9,11,13]$	$[2,4,6,8,10,12,14]$
G_l	$[-0.5,0.5,2.5,-2.5,0.5]$	$[-1.5,1.5,2.5,-2.5,1.5]$
H_l	$[1.75,1.75,1.75,1.75,1.75]$	$[1.75,1.75,1.75,1.75,1.75]$
$gain_l$	$[0.14,0.14,3.57,3.57,0.14]$	$[1.29,1.29,3.57,3.57,1.29]$
$gain_{leaf}$	3.57 x	3.57
$gain_{total}$	7.14	
$weight_l = -G_l/H_l$	$[0.29,-0.29,-1.43,-0.29]$	$[0.86,-0.86,-1.43,1.43,-0.86]$

Table 7: Example for Gain Calculation of two Split Candidates

In table 7 we have performed these steps for both candidates. We also added an additional step at the end of the table, where we calculate the leaf weights which represent the prediction values.

Now that we have determined the gain-scores for both labels, we can clearly see that the score for candidate 1 is much higher than the score for candidate 2. So in this example we would decide to continue the tree construction by applying the first split candidate. For both new leafs this process can then be repeated until the whole tree is constructed.
We can now recheck the selected split by taking an instance and compare the real labels with the predicted ones. So if we take instance 3, it will end up in the left leaf where the predicted weights are $[-2,2,-0.4,0.4,2]$. As mentioned before, we now have to apply the sigmoid function to these weights in order to get the probabilities. This leads to the final predictions $[0.12,0.88,0.40,0.60,0.88]$. We can take the default decision boundary at 0.5, that means if a probability is ≥ 0.5, the label is a positive label for this instance, and a negative label otherwise. So our predicted labels for this instance are $[0,1,0,1,1]$. If we now compare these predictions with the real ones in Table 5, we can see that they match exactly.
But let us repeat this check again with the first instance. As before we end up in the left leaf which leads to the same predictions as before $[0,1,0,1,1]$. But this time we can see that these predictions differ from the real labels. Only the first and the last label have been set correctly. This may mean that we have to split up this leaf again. The table shows that the first two instances share the same label distribution and the next three instances also have the same labels set. So a possible outcome for the next split may be a leaf with instances [1,2] and another one with instances [3,4,5].

4.2 Dynamic Classifier Chains

The main concept of Dynamic Chains is based on the same idea as classifier chains. The chain consists of nodes and links. Each node contains a base classifier that learns a model on the given dataset. This model is then used to generate label predictions for every instance in the given set. The links represent the propagation of these predictions to the next classifier in the chain. Before training a new model, the additional knowledge is added, as new features, to the train set. The next model can then take this additional information into account, to learn dependencies between previous set labels and labels yet to be set. So far this is a very static approach. The labels are predicted and propagated in an fixed order. Thus bad predictions from early classifiers can so corrupt the chain and lead to worse results.
At this point the dynamic chain approach comes into place. Figure 8 depicts the base idea for a dynamic chain. The main difference is that we do not force the models to predict the labels in a fixed order anymore. Instead only safe labels with high probabilities are set and propagated to the next chain-node. This is also the reason why we had to develop a new tree structure for XGBoost. With this new tree-representation, each classifier returns its predictions as a vector of probabilities, so we can easily choose the one with the highest value. Where classifier-chains have to predict a predetermined label as positive or negative for each instance, a single base-classifier has now the ability to set every label individually for an instance. So it is possible for a single classifier to assign label λ_3 to the first instance, label λ_5 to the second instance. Another difference is that we now focus on positive labels. Instead of evaluating the same label for each instance and decide whether it is a positive or a negative one, we try to assign only positive labels with high probabilities.

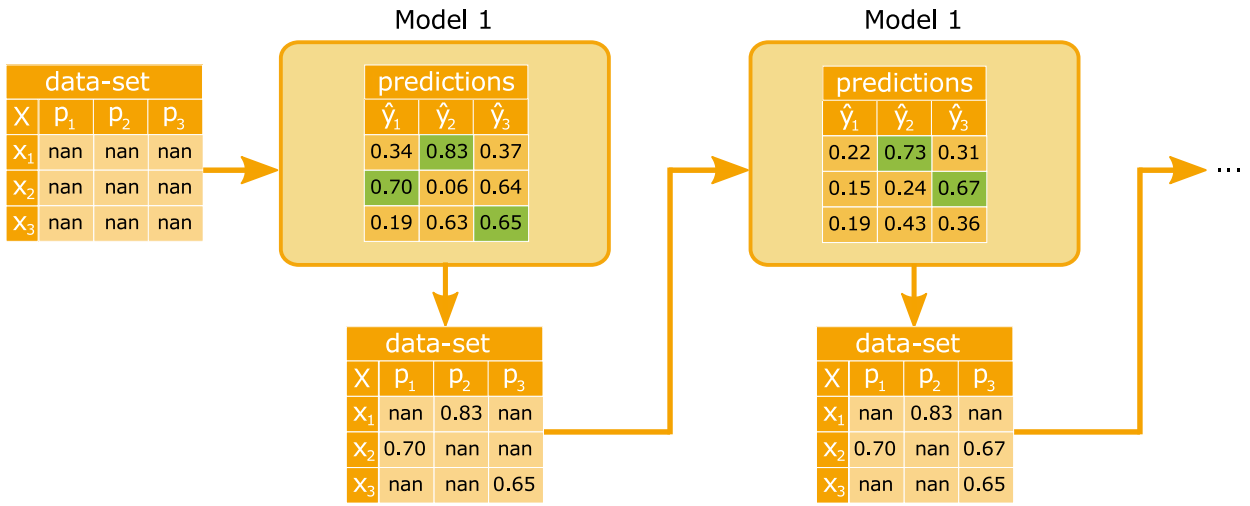


Figure 8: Dynamic Chain - base concept

4.2.1 Training Process

Before training the first classifier, we have to make a change to the training dataset. For each label we add a new placeholder feature which is initialized as *nan*. While proceeding inside the chain, these *nan* values are then replaced with predicted label probabilities. Since the values are not set at the beginning, the first classifiers will ignore them, because they do not contain any information about the dataset. But if these feature columns begin to be filled with values, later classifiers can take them into account and exploit the dependencies between them. Moreover the ability of XGBoost to deal with missing values allows to use these features, even if they are only partially filled with values.
Now the first classifier can be trained. Therefore, we pass the extended training-dataset and the corresponding real-label-vectors to a new XGBoost model. After the training is completed we start predicting

on the same dataset where the training process took place. These predictions are then propagated to the next classifier of the chain. For this step we have determined different approaches how to deal with the predictions and how to process them:

1) **Propagate only positive labels:** The first method is to only propagate a positive label with the highest probability. Since our base classifier already aims at maximizing a single label probability, we only want this information to be passed to the next classifier. Therefore, we take a look at the prediction-vector of each instance to find the maximum value. Now we can get two different outcomes. The desired one is that the prediction vector contains one or more probabilities above 0.5. In this case we can take the highest one, add it to the placeholder features and propagate it to the next classifier.
The second case is that all probabilities are below 0.5. Then we know that the classifier was not able to optimize the trees to get a higher result and therefore assume that this particular instance does not have another positive label assigned. So we do not modify the placeholder features for this instance and just pass them to the next node, as they were before.

$$p_l^i = \begin{cases} \hat{y}_l & \text{if } \hat{y}_l = max(\hat{y}_n | 1 \leq n \leq k) \geq 0.5 \text{ and } p_l^i = nan \\ p_l^i & \text{otherwise} \end{cases}$$

p_l^i denotes the placeholder feature for label λ_l of an instance i, which is not changed unless the predicted probability for feature l is the highest probability among all labels and greater than 0.5.

2) **Propagate negative labels, if no positive ones are predicted:** The next idea is similar to the first one but has an important extension. Same as before we aim for propagating especially positive labels, but this time we also allow to propagate a negative label, if there is no positive one predicted. Therefore, we can also use another split-calculation for our tree (see section 4.1.2), where we favor maximizing high probabilities, and minimizing low probabilities at the same time. Again, we have two cases for an outcome prediction of an instance. The first one, if there is at least one probability above 0.5 predicted, is treated as before and only the highest probability is propagated. The second case where all probabilities are below 0.5 is now treated differently. Instead of propagating no additional information, we take the probability with the lowest value, which represents the most probable non-relevant or negative label, from the prediction-vector and pass it to the next chain-node.
In both cases we do not allow later classifiers to change these predictions. So if a later classifier will predict a high probability, above 0.5, for a label that has already replaced a placeholder features, with a probability below 0.5, it cannot overwrite this information. This restriction is necessary because later classifiers tend to have a higher error rate. This is due to the fact that each classifier depends, at least partially, on the predictions of the previous one. If there is only one node in the chain that produces bad results, the following classifiers are affected by this. So we assume that the predictions of early classifiers are more robust than predictions of later ones and do therefore not allow them to overwrite previous predictions. The calculation for the propagated label gets an additional case:

$$p_l^i = \begin{cases} \hat{y}_l & \text{if } \hat{y}_l = max(\hat{y}_n | 1 \leq n \leq k) \geq 0.5 \text{ and } p_l^i = nan \\ \hat{y}_l & \text{if } \hat{y}_l = min(\hat{y}_n | 1 \leq n \leq k) \text{ and } max(\hat{y}_n | 1 \leq n \leq k) \leq 0.5 \text{ and } p_l^i = nan \\ p_l^i & \text{otherwise} \end{cases}$$

3) **Propagate the positive or negative label with the highest absolute probability:** The last method is an approach where positive and negative labels are treated equally. Instead of only propagating probabilities for negative labels if no positive labels are predicted, we now propagate the label with

the overall highest probability. That means we take the absolute distance of the probability to the decision boarder of 0.5 and pass the label where this value is maximized. For an example we have a look at following probability vector: $[0.1, 0.6, 0.7]$. If we now calculate the absolute distance to 0.5 we get $[0.4, 0.1, 0.2]$. So in this case we would propagate the probability 0.1 of the first (negative) label, although there are two positive predicted label with probabilities above 0.5.

$$p_l^i = \begin{cases} \hat{y}_l & \text{where } |\hat{y}_l - 0.5| = max(abs(\hat{y}_n - 0.5) | 1 \leq n \leq k) \text{ and } p_l^i = nan \\ p_l^i & \text{otherwise} \end{cases}$$

After all the predictions have been propagated, the whole process is repeated. Each repetition creates one node of the chain. The number of chain-classifiers also defines the number of labels which can be predicted, because each classifier only propagates one label-probability to the next one. So a chain with ten classifiers is able to predict up to 10 labels per instance. For datasets with a large number of labels, it may be appropriate to shorten the dynamic chain in order to obtain a low computational complexity. If each instance of a dataset with 200 labels is only assigned to an average of 3 labels, we do not have to build a chain with 200 classifiers.

Figure 9 shows a schematic view for training a chain with three classifier-nodes. The process is split up into four basic parts. At first we start with the default train set where three placeholder columns for the labels are added and initialized with *nan*. The second step is training the model. Therefore we pass the train set together with the corresponding real labels y to the XGBoost algorithm that learns *XGBoost model 1*. In the third step we again pass the train set to this new model in order to get predictions for each train instances. In the predictions table we have now highlighted the best predictions above 0.5 in green. For this example we use the propagation method where only positive labels are propagated and therefore the next train set has replaced some *nan* values with the corresponding predictions. Only x_4 does not get a propagated value since all probabilities are below 0.5. We then have an updated train-set and repeat the whole process until the third model has been trained. In the second propagation step we have the case that we want to propagate the prediction 0.52 for x_2. Since this label has already been predicted by the first classifier and has been added to the train set, this new prediction is ignored.

4.2.2 Prediction Process

The prediction process for getting predictions of unknown data-instances works similar to the training process. We start by also adding placeholder features, that are initialized with *nan* values, for each label to the test-data. These instances are then just passed through the chain. Each classifier generates predictions for all labels and, same as before, only replaces the placeholder value for the label with the highest probability. This is done according to the selected method used for training (only positive labels, only negative labels if no positive one is predicted, positive or negative label with highest absolute probability). After propagating an instance through all classifiers we get a prediction vector that can still contain nan values. Due to our main focus on positive labels, we overwrite these remaining nan-values with a probability of zero and assume that they are negative labels, because otherwise they would have been predicted as a positive label. In the last step we compare all probabilities with our stated decision boundary of 0.5 and assign all positive labels, which are the ones with probabilities above this boundary, to an instance.

This process is depicted in figure 10 and we can again split up the pipeline into several parts for predicting all labels of an instance. Similar to the training example we start with an empty dataset with the added placeholder columns, but this time it is the test set. In the first step the previous trained *XGBoost Model 1* is used to generate predictions on this test set. Again we can see them in the connected *predictions* table where the best predictions above 0.5 are green highlighted for each instance. These predictions are then again propagated and replace some of the *nan* placeholders of the test-set. This

process is then being repeated for the next XGBoost model. Same as before we do not allow to overwrite previews predictions which is the reason why the value 0.52 of the second prediction table is not propagated. After all models have generated their predictions we split the placeholder columns off the test set. These are then transformed to the *final predictions* by assigning a positive label to an instance if the corresponding prediction is above 0.5 and assigning a negative label if the prediction is below 0.5 or equal to nan. We have marked positive labels green and negative labels red in the final predictions table.

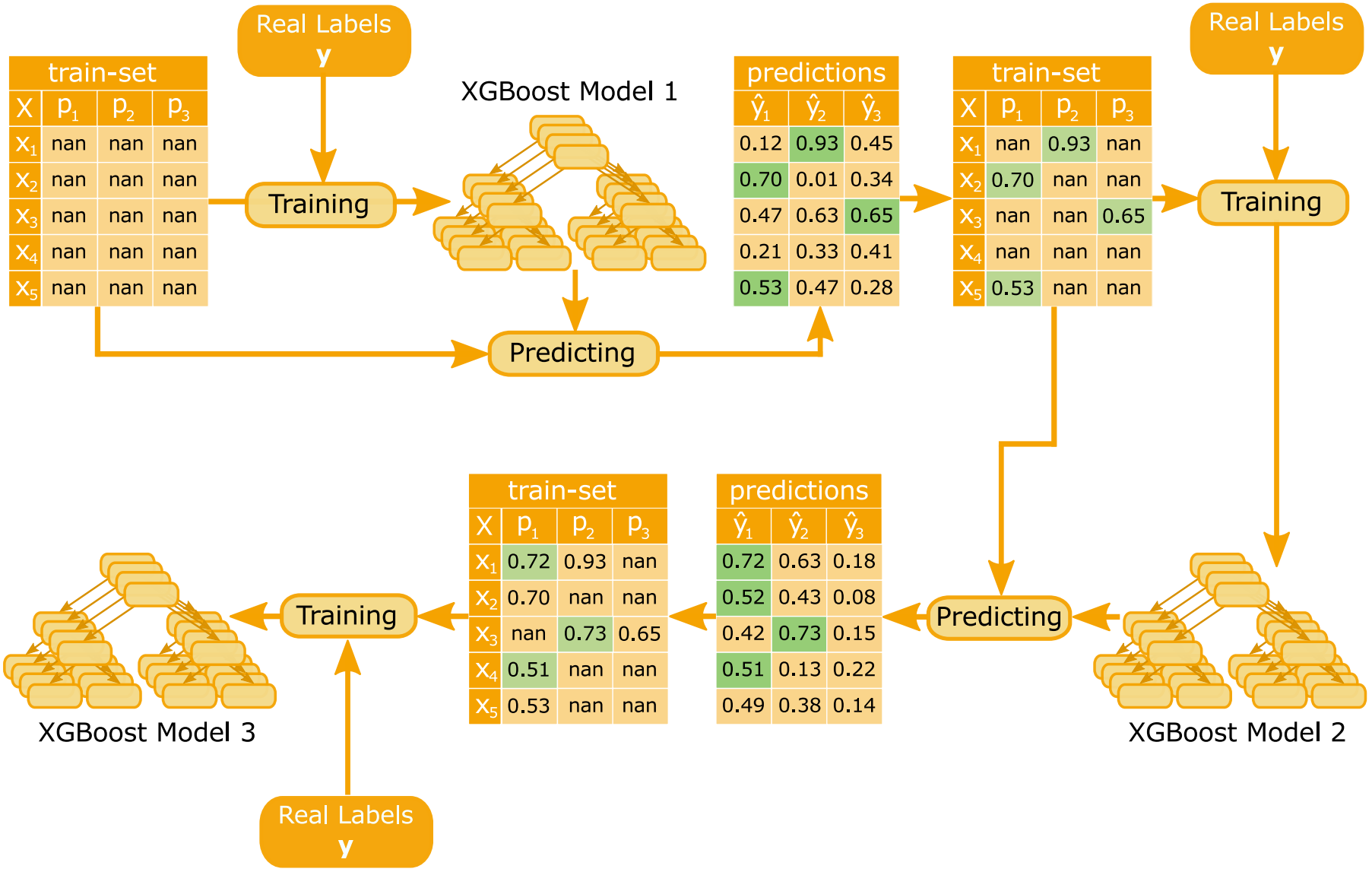


Figure 9: Dynamic Chain: Training Pipeline

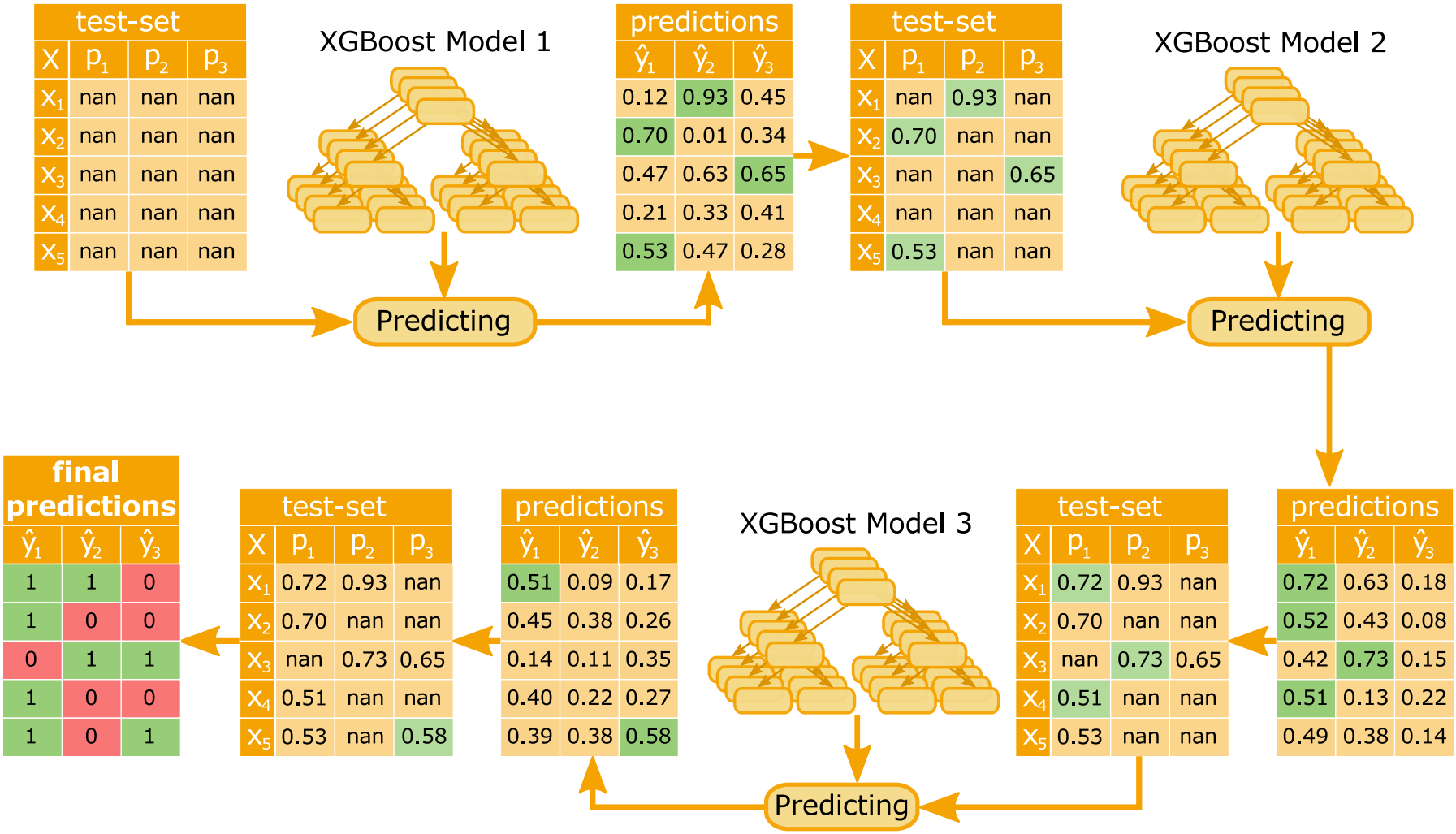


Figure 10: Dynamic Chain: Prediction Pipeline

4.3 Refinement Approaches

After the first application of the dynamic chain to several small datasets, some large drawbacks quickly occurred. Therefore we had to develop some modification to the chain in order to address these problems.

4.3.1 Separate and Conquer

The first issue was that all classifiers of the chain learned a very similar model and hence predicted also nearly the same labels. Some further investigation showed that this was owed to the new tree construction. The goal is that the split calculation of classifiers, later in the chain, should be influenced by the added features which contain the predictions of earlier models. So the first classifier learns a model only based on the default training-dataset. All splits are calculated over these features to get a maximum gain. Later classifiers can then take also previous predicted labels into account for their split calculation. The problem now was that the score for taking the same split as the classifier before was much higher than taking another split, otherwise the first classifier would have chosen it. The only difference can now be achieved by taking the previous predictions into account. But since we do only propagate a single previous label per instance, the propagated feature vectors are sparse and thus a higher gain score only rarely occurs.
We now have to prevent later classifiers to choose splits that always result in the same predictions. So we choose a approach similar to separate-and-conquer rule learning (Fürnkranz, 1999). The basic idea is to learn a rule on some training instances, then remove all instances that are covered by this rule and learn a new rule on the remaining data.
This approach can be easily transferred to the dynamic chain. We start by learning the first classifier and generating predictions on the train-set, where we take the ones with the highest probability per instance and pass them to the next node. At this point we can now apply *separating*. Since we have not yet predicted all labels of an instance, we cannot remove it completely from the dataset. If we now remember the tree construction, we know that split-finding only depends on the gradient and hessian values of every predicted label per instance. So we just do not want labels, that are already predicted, to affect the split calculation, by increasing the score value which would favor predicting them again. In order to remove their influence from the gain calculation, we set their corresponding gradient and hessian values to zero. After that follows the *conquering* step where a new classifier is trained. Taking the same split as before would then result in a lower gain-score, so the tree construction is forced to find a new optimal split. Warning: This does not mean that we now remove the possibility to learn label dependencies. The dependencies are respected by choosing a split over one of the added feature columns that contain previous labels (e.g. $\hat{y}_2 \geq 0.34$). Table 8 depicts an example how separate and conquer

x	label 01		label 02	
	g	h	g	h
1	0.5	0.25	-0.5	0.25
2	0.5	0.25	-0.5	0.25
3	-0.5	0.25	0.5	0.25
4	-0.5	0.25	-0.5	0.25
5	0.5	0.25	0.5	0.25

$\Longrightarrow$

x	$\hat{y}_1$	$\hat{y}_2$
1	0.2	0.6
2	0.7	0.8
3	0.1	0.2
4	0.9	0.4
5	0.4	0.6

$\Longrightarrow$

x	label 01		label 02	
	g	h	g	h
1	0.5	0.25	0	0
2	0.5	0.25	0	0
3	-0.5	0.25	0.5	0.25
4	0	0	-0.5	0.25
5	0.5	0.25	0	0

Table 8: Example for Separate and Conquer approach

works. The first table shows all gradient and hessian values at the beginning of the chain. Then a first classifier is trained and predicts on the training-data. The table in the middle shows these predictions. We then propagate only the highest, green marked, probabilities above 0.5 to the next classifier. Before starting the next training, all gradient and hessian values for the already predicted labels are set to zero, to ignore these instances for learning the next tree-model. This is done in the third table.

4.3.2 Cumulated Predictions

Another problem that showed up in early experiments is that the final predictions, after the chain has been traversed, contain too little positive labels. As soon as we replace all remaining nan values with a probability of zero, the number of false negative labels strongly goes up. So even if the chain contains as many classifiers as labels exist in the dataset, this problem still occurs. So we need to increase the number of positive predictions. So far each classifier only propagates a single label per instance, although early classifiers often have high probabilities for several labels. But if we now allow to propagate more than one label we also raise the potential error rate of later classifier, because they may then learn dependencies to these less probable labels. Since we do not want to worsen the stagewise predictions, we use another strategy that we call *cumulated predictions*. This approach does not make any changes to the current training or prediction process of the chain. We only add one step to each node in the prediction pipeline. At the point where one label is propagated and replaces a value in the placeholder features, we also pass the full predictions as a matrix to the next classifier. This additional information does not affect the further predictions, it is just taken along. All following classifiers then merge their complete predictions into this matrix. We thereby allow to overwrite lower predictions with higher ones. By doing this we obtain the same predictions as before, but now we also get all predictions for positive labels which were not propagated and are lost otherwise. This approach has greatly enhanced the final results.

Table 9 depicts this process once more. We start with the test set and its added placeholder columns

test-set		
X	l_1	l_2
x_1	nan	nan
x_2	nan	nan
x_3	nan	nan
x_4	nan	nan
x_5	nan	nan

$\Longrightarrow$

predicts 1	
$\hat{y}_1$	$\hat{y}_2$
0.2	0.6
0.7	0.8
0.1	0.3
0.9	0.4
0.4	0.6

$\rightarrow$

test-set		
X	l_1	l_2
x_1	nan	0.6
x_2	nan	0.8
x_3	nan	nan
x_4	0.9	nan
x_5	nan	0.6

$\Longrightarrow$

predicts 2	
$\hat{y}_1$	$\hat{y}_2$
0.2	0.1
0.4	0.6
0.6	0.1
0.3	0.4
0.7	0.6

$\rightarrow$

test-set		
X	l_1	l_2
x_1	nan	0.6
x_2	nan	0.8
x_3	0.6	nan
x_4	0.9	nan
x_5	0.7	0.6

$\downarrow$

Cumulated Predictions	
$\hat{y}_1$	$\hat{y}_2$
0.2	0.6
0.7	0.8
0.1	0.2
0.9	0.4
0.4	0.6

$\Longrightarrow$

$\downarrow$

Cumulated Predictions	
$\hat{y}_1$	$\hat{y}_2$
0.2	0.6
0.7	0.8
0.6	0.3
0.9	0.4
0.7	0.6

Table 9: Example for Cumulated Predictions

l_1 and l_2 for labels λ_1 and λ_2. The first classifier generates its predictions *predicts 1* for this test set, and merges its highest probabilities above 0.5 into the test-set's placeholder columns. These propagated probablities are highlighted blue in the second test-set table. At the same time we store the complete predictions into the cumulated predictions table below. After that we continue with the modified test set and the second classifier generates its predictions *predicts 2* thereon. Same as before we insert these predictions with highest probabilities into the test set and also merge the complete prediction matrix into the cumulated predictions. The colored table cells respectively show the updated values. This process can then be repeated for all classifiers of the chain. In order to get the final predictions afterwards, we just have to take the latest *Cumulated Predictions* table and map predictions above 0.5 to positive labels and probabilities below 0.5 to negative labels.

5 Experimental Setup

This chapter shows the results of the dynamic chain with XGBoost base classifiers on several datasets. The different methods for split finding and label propagation are evaluated and compared to the presented baseline approaches from chapter 2.2.

5.1 Datasets

The following datasets in table 10 were used for evaluation. They are taken from the mulan project (Tsoumakas et al., 2011) that provides a collection of various multilabel datasets: These datasets cover

name	domain	instances	labels	attributes	cardinality
birds	audio	645	19	260	1.014
emotions	music	593	6	72	1.869
flags	images	194	7	19	3.392
genbase	biology	662	27	1186	1.252
medical	text	978	45	1449	1.245
scene	images	2407	6	294	1.074
yeast	biology	2417	14	103	4.237

Table 10: Datasets used for the experiments

a wide variety of application areas for multilabel classification. We also have selected datasets with different structures. There are small sets with a low number of labels, like *emotions*, but also some with a higher number, like *medical* and other ones with high cardinalities, like *yeast*, which also has a larger number of instances. The *cardinality* is thereby the average number of positive labels which is assigned to an instance. Additionally we have chosen datasets with significant label dependencies that were detected in (Loza Mencía and Janssen, 2016). Especially *yeast* has some high degree of label dependencies, but also the smaller sets *emotions* and *scene* show dependencies. In particular the ones in *scene* are mostly partially dependencies which means that the dependencies also depend on some instance features. All provided datasets consist of a training and testing dataset that we used for the final evaluations.

5.2 Evaluation Measures

In contrast to multiclass or single-label regression tasks, we now have more than one target value. This means we cannot apply the same measures for multilabel evaluation (Maimon and Rokach, 2009) anymore. Bipartition measures for multilabel classification can be divided into two groups of *example-based* and *label-based* evaluation metrics.
Example-based measures are calculated based on the average differences of the real target values and the predicted labels over all test instances.
Label-based measures are calculated per label and afterward averaged over all labels.
We use measures from both groups for evaluation. To calculate them we need to know the proportions of correct and incorrect classified labels. Therefore we define a confusion matrix:

	predicted	not predicted
relevant / positive	true positive	false negative
irrelevant / negative	false positive	true negative

Table 11: Confusion matrix for predictions

Example Based

- **Hamming Loss** denotes the percentage of missclassified labels. This means *false positive* and *false negative* labels. Therefore we calculate the missclassification rate for all labels of an instance and sum these rates up over all n instances.

$$\text{Hamming-Loss} = \frac{1}{n}\sum_{i=1}^{n}\frac{\hat{y}_i \oplus y_i}{k} = \frac{1}{n}\sum_{i=1}^{n}\frac{fp_i + fn_i}{k} \tag{23}$$

  where fp_i and fn_i denote the number of false positive and false negative labels by comparing the vectors $\hat{y}_i$ and y_i.

- **Subset accuracy** (sometimes also called classification accuracy) denotes the percentage of instances where the predictions match exactly with the true labels.

$$\text{Subset-Accuracy} = \frac{1}{n}\sum_{i=1}^{n} I(\hat{y}_i = y_i) \tag{24}$$

  where $I(true) = 1$ if the prediction vector $\hat{y}_i$ matches with the real label vector y_i and $I(false) = 0$ if these vectors differ.

Label Based

Here we can now apply any measure for binary evaluation. To get a result for all labels we have to average over them. Therefore two variants exist, called *macro averaging* and *micro averaging*. Let $E(tp, fp, tn, fn)$ denote an evaluation function, we can define macro-averaging as

$$E_{\text{macro}} = \frac{1}{k}\sum_{i=1}^{k} E(tp^{(i)}, fp^{(i)}, tn^{(i)}, fn^{(i)}) \tag{25}$$

where we calculate the macro-score as the sum of the evaluations per label. In this case $tp^{(i)}$ denotes the number of true positive predictions over all instances for label λ. The calculation of $fp^{(i)}$, $tn^{(i)}$ and $fn^{(i)}$ is performed analogical for false positive, true negative and false negative predictions. Similarly we can define micro-averaging as

$$E_{\text{micro}} = E\left(\sum_{i=1}^{k} tp^{(i)}, \sum_{i=1}^{k} fp^{(i)}, \sum_{i=1}^{k} tn^{(i)}, \sum_{i=1}^{k} fn^{(i)}\right) \tag{26}$$

where the evaluation score is calculated on the summed up confusion matrices for all labels. We use both of these methods and apply them to different evaluation functions. Therefore we introduce a new notation $\hat{z}_\lambda$ for the prediction vector and z_λ for the real label vector. In contrast to y_i and $\hat{y}_i$ which denote all labels for a single instance, z_λ now denotes the real labels of all instances for a single label λ and $\hat{z}_\lambda$ denotes the predictions of all instances for this label (with $\hat{z}_\lambda, z_\lambda \in \{0,1\}^n$).

- **Precision** denotes the percentage of correct predictions for positive labels given the number of all positive predicted labels.

$$precision = \frac{||\hat{z}_\lambda \cdot z_\lambda||_1}{||\hat{z}_\lambda||_1} = \frac{tp^{(i)}}{tp^{(i)} + fp^{(i)}} \tag{27}$$

  If both vectors for predictions and ground truth are empty ($\hat{z}_\lambda$ and z_λ only contain negative labels), the precision is considered equal to 1. If $\hat{z}_\lambda$ is empty, but z_λ is not empty and contains also positive labels, we consider the precision equal to 0.

- **Recall** denotes the percentage of correct predictions for positive labels given the true number of positive label.

$$recall = \frac{||\hat{z}_\lambda \cdot z_\lambda||_1}{||z_\lambda||_1} = \frac{tp^{(i)}}{tp^{(i)} + fn^{(i)}} \tag{28}$$

  Similar as before we consider the recall equal to 1 if $\hat{z}_\lambda$ and z_λ are empty and we consider it equal to 0 if the the prediction vector is empty but the ground truth vector is not empty.

- **F1-measure** denotes the harmonic mean of precision and recall cf. e.g. (Lipton et al., 2014)

$$F1 = \frac{2 \cdot ||\hat{z}_\lambda \cdot z_\lambda||_1}{||z_\lambda||_1 + ||\hat{z}_\lambda||_1} = \frac{2 \cdot tp^{(i)}}{2 \cdot tp^{(i)} + fp^{(i)} + fn^{(i)}} \tag{29}$$

  If $\hat{z}_\lambda$ and z_λ are both empty we consider the F1 measure equal to 1 and if only one of both vectors is empty the measure is considered equal to 0. We use this F1 measure also as an example-based measure, where the score is firstly calculated over a single instance with all labels and then averaged over all instances.

5.3 Baseline Setup

All experiments were performed using the MULAN Java library (Tsoumakas et al., 2011) that offers a large collection of tools to support the work with multilabel-problems. It allows to easily process multilabel-datasets and apply different pre-implemented algorithms for problem transformations, including *Binary Relevance*, *Classifier Chains* and *Label Powerset*, to them. Besides this MULAN also provides an evaluation framework to compute a wide variety of evaluation measures. Since many multilabel algorithms are based on label transformation methods, we need base classifiers to use them. Therefore the MULAN library is built on top of WEKA (Hall et al., 2009). WEKA is a widespread open source machine-learning toolbox that provides a vast collection of state-of-the-art supervised learning algorithms. These learning algorithms can be used for classical problems like classification and regression tasks, but in order to deal with multilabel problems, we have to combine them with the MULAN library. For all tests we have selected *Subset Accuracy* and *Hamming Loss* as our main optimization goals.

5.3.1 Baseline Algorithms

The problem-transformation baseline methods are taken directly from the MULAN library, so we were able run these tests without the need to implement the methods from scratch.

Label Powerset, Binary Relevance and Classifier Chains

Since these three algorithms are all pre-implemented in MULAN, the only requirement was to pass a binary classifier to the algorithm, that generates binary predictions or probabilities between zero and one. This classifier was then trained on the automatically transformed dataset and the predictions were mapped back to a vector of probabilities for each label of an instance. For classifier chains the default configuration was used that predicts labels in the static order of their occurrence in the dataset and we did not evaluate other chain orders. So the first model of the chain predicts label λ_1 and the last model predicts the last label λ_k. In order to obtain the best results on the test-set, we performed a grid search, for each baseline-method, on the train-set to find the best parameters for the base classifiers. We evaluated the different parameter-sets by using a full cross-validation with 3 folds on the train-set. The best parameters were then used to train the corresponding classifier on the full train-set and to generate the final predictions on the test-set afterwards.

Multi-Target Tree

To get the baseline results for an algorithm adaption method, we decided to use a single multitarget tree, that was introduced in section 4.1. Therefore we trained our modified XGBoost with only one boosting round and a fixed maximum depth of 50, in order to obtain a single tree only. This helps us to analyze the quality of a single multilabel-tree in order to determine how much the ensemble version improves the results. For generating the results on the test-set we evaluated the six different split methods from in section 4.1.2 on a hold-out validation set. This validation set contained 25% of the instances from the train-set and the remaining 75% of the instances were used to train the model. The split that generated the best results on the validation set was then used to train the last classifier which predicted on the test-set.

5.3.2 XGBoost Base Classifier

In order to get a proper comparison of the modified XGBoost with the dynamic chain approach and the baseline methods, we decided to use the default XGBoost as a base classifier. Therewith we can easily detect if our modifications can improve the results on multilabel datasets. The problem with this idea was that all MULAN problem-transformation algorithms require a Weka classifier. But at the beginning of this work there were no appropriate Weka wrappers for XGBoost available and thus we had to implement this wrapper ourselves. Fortunately, Weka classifiers are implemented in Java and XGBoost provides an Java interface which made it not too difficult to solve this problem.
As mentioned before we performed a grid search with a cross-validation on the train-set to get the best parameters. In order to make a fair comparison we only tuned the parameters, which were also tuned for the dynamic chain approach. Since the dynamic chain also tunes a variety of other parameters, we decided to limit the XGBoost parameter tuning to *max_depth*, that represents the maximum depth each tree is allowed to grow, and *num_round*, that denotes the number of boosting rounds per model. Both parameters were evaluated with all combinations of the values $[10, 25, 50, 100, 150]$. For the training objective we choose *logistic regression* which is also used in the dynamic approach. The remaining parameters kept their default values.

5.3.3 J48 Base Classifier

The second base classifier we used is the Weka J48 algorithm which is also based on decision trees. But instead of XGBoost where we learn a tree forest, J48 only learns a single C4.5 decision tree (Korting, 2006). The biggest difference to CART trees from XGBoost is that C4.5 splits are not forced to be binary which means that every tree node can have an arbitrary number of child nodes. These trees are also called *univariate trees* because they contain one attribute in a inner node to test where to pass a test-instance.
For training we used the default parameters and tuned only pruning confidence threshold C with ten linear steps between 0.05 and 0.5.

5.4 Dynamic Chain Setup

We decided to implement the dynamic chain algorithm into the MULAN library. This helps us to deal with the multilabel dataset and allows us to evaluate the results with the built-in evaluation library. The modifications of the XGBoost algorithm were made in the C++ implementation. Since our changes were compatible to the existing interfaces we were able to use the previous implemented Weka wrapper to combine the modified algorithm with the dynamic chain.
For the first evaluation we initialized the number of chain-nodes with the number of labels of the dataset. The trees were initialized with a *max_depth* of 50 and the number of boosting rounds *num_round* with 50. The dynamic chain was trained on 75% of the train-set and evaluated on the split off 25% of the train set. This setup was used to evaluate the six different split-calculations for the tree construction in combination with the three different methods of propagating the predictions along the chain. We

also compared the results of the default prediction pipeline with the cumulated predictions, from section 4.3.2, for each combination. Since these cumulated predictions are calculated parallel to the chain propagation and do not affect the training process, we get them together with the default predictions by only using a single model. So all in all we trained 18 dynamic chain models per dataset for this first evaluation. We also analyzed the gain of information by propagating the test instances along the chain. So if a chain with ten nodes does not make any new predictions after the fifth node, we could shorten the training process and train only a dynamic chain with five nodes.
After this first evaluation, we took the best two split-calculation and propagation-method combinations for each dataset and evaluated them with a different number of boosting rounds and tree depths. For this grid search we used the same parameters as for training the XGBoost models for the baseline models. Both parameters were taken from $[10, 25, 50, 100, 150]$. At the end we initialized the dynamic chain for each dataset with the corresponding best parameter set, trained it on the full train-set and generated the final predictions on the test-set.

6 Evaluation

In this section we will now compare and discuss the different setups and parameter-sets of the dynamic chain. Afterwards we will compare these results to the baseline multilabel classifiers. The results on each dataset were generated on the validation sets which contain 25% of the instances split off the train set.

6.1 Comparison of Split Calculation and Label Propagation methods

We used the first evaluation round to determine the best split calculation and label propagation methods for each dataset. **Tables 12-18** show the results of training the chain where each XGBoost classifier does 50 boosting rounds with trees of maximum size 50. We only observe **Hamming Loss** and **Subset Accuracy** in the result tables because these are the measures we want to optimize. The first column always shows the propagation methods which correspond to the ones proposed in section 4.2.1. Method 0 means that only positive labels with probabilities above 0.5 are propagated, method 1 means that we again propagate positive labels with the highest probability, but also pass negative labels with probabilities below 0.5 if no positive ones are predicted and method 3 means propagating positive and negative labels with the highest absolute probability. The second column *Prediction Type* denotes the way the predictions were generated. *DEF* stands for *default* and means that we used the prediction pipeline from section 4.2.2. *CUM* stands for *cumulated* and means that we used the modified prediction pipeline from section 4.3.2. Furthermore each block of the table belongs to a single split method that is denoted in the row above the block and corresponds directly to the split calculation methods from section 4.1.2. For each of these blocks we highlighted the best measures green, the overall best result for the main optimization goals, hamming loss and subset accuracy, yellow and the second best ones red. We also marked the two splits with the best and second best scores for both measures blue.
For the datasets emotions, flags, medical and yeast we can see that the best parameters for hamming loss also provide the best result for subset accuracy. In the scene dataset the best subset accuracy result belongs to the second best hamming loss result and vice versa. The genbase dataset in table 15 seems to be the only exception, but if we take a look at the different subset accuracies we notice that the best subset accuracies are very close together in contrast to the other datasets. The difference between the best one with 0.9460 and the second best with 0.9438 is only 0.0022 which can be neglected.
If we now take a closer look at the different propagation types we can clearly see that propagation method 2 never returns really good results. For all datasets it is outperformed by the other methods and especially the subset accuracy is often equal to zero. This phenomenon can be explained by taking a look at the predictions. The problem was that since we treated positive and negative labels equally, most of the classifiers only predicted negative labels, because they are the majority of occurring labels, and passed them along the chain. So in the end the predictions only contained negative labels and therefore have a very low subset accuracy. Again we have an exception for this case in the genbase dataset with split 3 where the results of all three propagation methods are the same for the cumulated predictions. This can be explained by taking a look at the prediction pipeline and the structure of the dataset. The dataset has a low cardinality and each instance is assigned to an average of 1.245 labels. A deeper look at the prediction pipeline showed that propagation method 0 and 1 predicted and passed most of the true positive labels, which results in a good subset accuracy for the DEF method, whereas method 2 only propagated negative labels and hence got a subset accuracy of 0. The reason why the CUM results are the same is that the first model of the chain was the same for all propagation methods. Although it could only propagate a single label per instance for the DEF predictions, it was already able to predict all labels which were otherwise added from later classifiers. The CUM method now takes all predictions of this first classifier and since later classifiers do not predict additional labels, we already got the final predictions after the first round. Figure 14, which will be further discussed in section 6.2, depicts this process. Thus the CUM predictions did not change by going through the chain, remained the same for all propagation methods and hence got the same scores.
After this analysis, we made the first decision to **ignore propagation method 2** for further tests because

it was always outperformed by the other approaches.
For the next aspect we have a look at the impact of using the *cumulated predictions* for evaluation. In most cases the best results per split method are the CUM predictions and if they beat the DEF results, they surpass them in both measures, hamming loss and subset accuracy. But there are also some examples where the DEF results are better than the CUM results. This applies especially for the datasets *birds, medical* and *scene*. In these cases the results of both methods are very close together and sometimes even equal. If we take a look at the structure of these datasets we can see that all of them have a very low label cardinality where mostly only one positive label is assigned to an instance. Figures 11, 15 and 16 also depict this fact by showing that the first model of the chain is the only model to predict new labels. The advantage of the cumulated predictions, where we allow each model to predict more than one positive label per instance, is lost if there exists only one positive label per instance.

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.0508	0.4318
	CUM	0.0502	0.4318
1	DEF	0.0858	0.1852
	CUM	0.1884	0.0000
2	DEF	0.0699	0.3421
	CUM	0.4418	0.0000
Split 2			
0	DEF	0.0532	0.5000
	CUM	0.0520	0.5000
1	DEF	0.0573	0.4177
	CUM	0.0740	0.2911
2	DEF	0.0658	0.4375
	CUM	0.3487	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.0498	0.5000
	CUM	0.0498	0.5000
1	DEF	0.0862	0.2048
	CUM	0.2175	0.0000
2	DEF	0.0829	0.3250
	CUM	0.6020	0.0000
Split 4			
0	DEF	0.0584	0.3659
	CUM	0.0584	0.3659
1	DEF	0.0905	0.1011
	CUM	0.2496	0.0000
2	DEF	0.0652	0.3864
	CUM	0.4689	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.0395	0.4861
	CUM	0.0409	0.4861
1	DEF	0.0774	0.3529
	CUM	0.1912	0.0000
2	DEF	0.0628	0.3974
	CUM	0.4486	0.0000
Split 6			
0	DEF	0.0490	0.5205
	CUM	0.0497	0.5205
1	DEF	0.0854	0.1948
	CUM	0.2064	0.0000
2	DEF	0.0627	0.4286
	CUM	0.5808	0.0000

Table 12: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on birds dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.2259	0.2000
	CUM	0.1889	0.3333
1	DEF	0.2195	0.1683
	CUM	0.2013	0.3069
2	DEF	0.2697	0.1011
	CUM	0.5131	0.0000
Split 2			
0	DEF	0.2378	0.2396
	CUM	0.2274	0.2604
1	DEF	0.2340	0.1515
	CUM	0.2222	0.1818
2	DEF	0.3093	0.1000
	CUM	0.5593	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.2044	0.2267
	CUM	0.1911	0.2667
1	DEF	0.2233	0.2400
	CUM	0.1983	0.3200
2	DEF	0.2891	0.0612
	CUM	0.5646	0.0102
Split 4			
0	DEF	0.2412	0.1647
	CUM	0.2451	0.1647
1	DEF	0.2484	0.1471
	CUM	0.2696	0.1275
2	DEF	0.3384	0.0202
	CUM	0.5673	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.2375	0.1264
	CUM	0.2107	0.2874
1	DEF	0.2211	0.1735
	CUM	0.2041	0.2551
2	DEF	0.2812	0.0938
	CUM	0.5937	0.0000
Split 6			
0	DEF	0.2289	0.1928
	CUM	0.2269	0.2169
1	DEF	0.2228	0.2247
	CUM	0.2285	0.2022
2	DEF	0.3163	0.0682
	CUM	0.5606	0.0114

Table 13: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on emotions dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.3086	0.0400
	CUM	0.2743	0.1600
1	DEF	0.4152	0.0625
	CUM	0.3571	0.0313
2	DEF	0.3609	0.0526
	CUM	0.4361	0.0263
Split 2			
0	DEF	0.3297	0.0385
	CUM	0.2308	0.1923
1	DEF	0.3532	0.0556
	CUM	0.3175	0.0833
2	DEF	0.3918	0.0000
	CUM	0.3714	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.3290	0.0606
	CUM	0.2900	0.0909
1	DEF	0.3512	0.0833
	CUM	0.3274	0.1250
2	DEF	0.3680	0.0000
	CUM	0.5108	0.0303
Split 4			
0	DEF	0.3824	0.0294
	CUM	0.2857	0.1176
1	DEF	0.3143	0.0400
	CUM	0.3029	0.0800
2	DEF	0.3498	0.0345
	CUM	0.4138	0.1034

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.3398	0.1081
	CUM	0.2741	0.1351
1	DEF	0.3333	0.0833
	CUM	0.3214	0.1389
2	DEF	0.3502	0.0000
	CUM	0.5115	0.0000
Split 6			
0	DEF	0.3771	0.0400
	CUM	0.2686	0.1600
1	DEF	0.2808	0.1034
	CUM	0.3103	0.1379
2	DEF	0.3687	0.0323
	CUM	0.4839	0.0000

Table 14: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on flags dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.0053	0.9028
	CUM	0.0030	0.9309
1	DEF	0.0056	0.8834
	CUM	0.0039	0.9071
2	DEF	0.0467	0.0000
	CUM	0.0082	0.8272
Split 2			
0	DEF	0.0173	0.6631
	CUM	0.0167	0.6760
1	DEF	0.0181	0.6609
	CUM	0.0181	0.6609
2	DEF	0.0467	0.0000
	CUM	0.0182	0.6739

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.0038	0.9352
	CUM	0.0022	0.9438
1	DEF	0.0038	0.9352
	CUM	0.0022	0.9438
2	DEF	0.0467	0.0000
	CUM	0.0022	0.9438
Split 4			
0	DEF	0.0294	0.4406
	CUM	0.0292	0.4471
1	DEF	0.0299	0.4579
	CUM	0.0295	0.4622
2	DEF	0.0467	0.0000
	CUM	0.0334	0.4147

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.0037	0.9244
	CUM	0.0024	0.9374
1	DEF	0.0031	0.9287
	CUM	0.0021	0.9438
2	DEF	0.0467	0.0000
	CUM	0.0042	0.9201
Split 6			
0	DEF	0.0041	0.9287
	CUM	0.0028	0.9460
1	DEF	0.0038	0.9352
	CUM	0.0026	0.9430
2	DEF	0.0466	0.0000
	CUM	0.0024	0.9438

Table 15: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on genbase dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.0214	0.2805
	CUM	0.0214	0.2805
1	DEF	0.0611	0.0000
	CUM	0.4820	0.0000
2	DEF	0.0280	0.0000
	CUM	0.6981	0.0000
Split 2			
0	DEF	0.0107	0.6437
	CUM	0.0100	0.6782
1	DEF	0.0111	0.6220
	CUM	0.0106	0.6341
2	DEF	0.0269	0.0000
	CUM	0.0123	0.6667

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.0225	0.2099
	CUM	0.0225	0.2099
1	DEF	0.0801	0.0000
	CUM	0.5907	0.0000
2	DEF	0.0302	0.0543
	CUM	0.8273	0.0000
Split 4			
0	DEF	0.0174	0.3750
	CUM	0.0172	0.3864
1	DEF	0.0204	0.3611
	CUM	0.0210	0.3472
2	DEF	0.0274	0.0000
	CUM	0.0214	0.3580

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.0146	0.3947
	CUM	0.0143	0.4079
1	DEF	0.0368	0.1778
	CUM	0.4165	0.0000
2	DEF	0.0285	0.0000
	CUM	0.7911	0.0000
Split 6			
0	DEF	0.0209	0.1860
	CUM	0.0209	0.1860
1	DEF	0.0948	0.0000
	CUM	0.5676	0.0000
2	DEF	0.0267	0.0714
	CUM	0.7553	0.0000

Table 16: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on medical dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.1130	0.3926
	CUM	0.1130	0.3926
1	DEF	0.1050	0.4429
	CUM	0.1044	0.4464
2	DEF	0.1519	0.2935
	CUM	0.2895	0.2765
Split 2			
0	DEF	0.1348	0.4130
	CUM	0.1348	0.4130
1	DEF	0.1330	0.4263
	CUM	0.1335	0.4199
2	DEF	0.1534	0.1981
	CUM	0.2700	0.1821

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.1020	0.4898
	CUM	0.1020	0.4864
1	DEF	0.1115	0.4808
	CUM	0.1121	0.4774
2	DEF	0.1501	0.3509
	CUM	0.4156	0.2019
Split 4			
0	DEF	0.1614	0.1533
	CUM	0.1620	0.1498
1	DEF	0.1689	0.1815
	CUM	0.1733	0.1716
2	DEF	0.1865	0.1190
	CUM	0.4399	0.0782

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.1076	0.4636
	CUM	0.1076	0.4603
1	DEF	0.1104	0.4570
	CUM	0.1093	0.4570
2	DEF	0.1536	0.4510
	CUM	0.4276	0.0980
Split 6			
0	DEF	0.1176	0.4940
	CUM	0.1181	0.4911
1	DEF	0.1030	0.5574
	CUM	0.1025	0.5608
2	DEF	0.1529	0.4868
	CUM	0.5883	0.1258

Table 17: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on scene dataset

Propagation	Pred. Type	Hamming	Subset Acc.
Split 1			
0	DEF	0.2199	0.1041
	CUM	0.2078	0.1269
1	DEF	0.2464	0.0542
	CUM	0.2462	0.0678
2	DEF	0.2570	0.0197
	CUM	0.5778	0.0000
Split 2			
0	DEF	0.2265	0.0296
	CUM	0.2101	0.1506
1	DEF	0.2351	0.0260
	CUM	0.2189	0.1302
2	DEF	0.2769	0.0082
	CUM	0.5995	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 3			
0	DEF	0.2241	0.1013
	CUM	0.1987	0.1620
1	DEF	0.2333	0.0705
	CUM	0.2236	0.1220
2	DEF	0.2732	0.0165
	CUM	0.6091	0.0055
Split 4			
0	DEF	0.2455	0.0208
	CUM	0.2466	0.0519
1	DEF	0.2598	0.0173
	CUM	0.2715	0.0202
2	DEF	0.2919	0.0056
	CUM	0.6210	0.0000

Propagation	Pred. Type	Hamming	Subset Acc.
Split 5			
0	DEF	0.2229	0.0546
	CUM	0.2121	0.1284
1	DEF	0.2321	0.0769
	CUM	0.2165	0.1333
2	DEF	0.2783	0.0079
	CUM	0.6523	0.0000
Split 6			
0	DEF	0.2212	0.0940
	CUM	0.2057	0.1462
1	DEF	0.2259	0.0613
	CUM	0.2143	0.1333
2	DEF	0.2807	0.0260
	CUM	0.6399	0.0000

Table 18: First evaluation with 50 boosting rounds and maximum tree depth 50 per base XGBoost model on yeast dataset

6.2 Prediction Pipeline

In this section we want to analyze the label propagation along the chain and use it to optimize the total length of the dynamic chain. Therefore we took the best parameters from the first evaluation round of section 6.1, reran the experiments and recorded the label propagation pipeline. **Figures 11-17** depict this process. They count the number of true-positive (TP), true negative (TN), false positive (FP) and false negative (FP) labels predicted in each label round, where label round i stands for the i-th classifier node of the chain.

6.2.1 Comparison of Default and Cumulated Predictions

We start with comparing the course of the graphs from the default (*DEF*) predictions to the graphs from the cumulated (*CUM*) predictions. The first thing to notice is that the datasets *birds, genbase, medical* and *scene* seem to predict most of their labels in the first round and have later on only minor changes. For the CUM and DEF predictions of *birds* and *scene* this observation is strongest, where for *birds* only one true negative switches to a false positive in round two and for *scene* actually no predictions change. *Genbase* predicts between the forth and the fifth model 110 new true positive labels and *medical* predicts up to round three 12 more true positive labels which is not very much compared to the absolute number of label-instance combinations. The same applies to the CUM predictions where *genbase* predicts 87 new true positive labels until round 4 and *medical* 2 true positive labels until round 3. The reason that the first chain-classifier predicts nearly all positive labels is again connected to the label cardinalities of the datasets. For *birds* and *scene* it is nearly one, which explains why the first model is the only one to predict positive labels, and for *genbase* and *medical* the cardinality is also very low with 1.25, so the first model is not able to predict all positive labels alone.
The three remaining sets are the more interesting ones. All of their true negative counts for DEF start very high and then start to slightly decline. This is due to the fact that, as the label cardinality shows, most of the true target labels are negative labels. So if a classifier predicts no positive label for an instance, we assume it to be a negative label. When we then go along the chain the classifiers begin to predict more and more positive labels. On the one hand side this raises the true positive rate, but on the other side the number of true negative labels declines because, since the classifiers are not perfect, they also start to predict false positive labels. Figure 13a for the *flags* dataset show this very well. We can see the number of true negative labels declining from 80 to 73, while the number of false positives increases from 8 to 15, and at the same time the number of true positive labels raising from 18 to 49 while simultaneous the number of false negatives decreases from 76 to 45. All in all the number of correctly predicted labels raises from 98 to 122 and the number of falsely predicted labels declines from 84 to 60. This effect can also be observed in figures 12a and 17a for the *emotions* and *yeast* datasets.
Although we can see that the number of true positive labels is raising there is still a problem with the DEF prediction pipeline. If we have a look at the number of false negative labels we can see that it is indeed decreasing, but remains, even on the lowest points, very high and above the number of true positives. This is where the idea for the CUM predictions comes in. Due to the fact that we now allow each model of the chain to predict more than one positive label per instance, the number of true positive labels starts now with a much higher value. We take again figure 13 as an example where the true positive count in 13b starts even higher than the maximum value of the true positives in 13a. This causes that also the positive labels start with a higher count and the true negatives are lower than before. We can now compare the total number of correctly and falsely predicted labels of the CUM predictions to the DEF predictions. The number of correct predictions starts at 137 and raises to 140 while the number of false predictions decline from 45 to 42. So even after the first label round, the CUM results are better than the DEF results after 7 label rounds. For the remaining and larger datasets this effect is even stronger. The number of correct DEF predictions raises for *yeast* from 4291 to 4431 correct CUM predictions and for the *emotions* dataset from 418 to 438. All in all, we have **between 2.2% and 4.8% more correct**

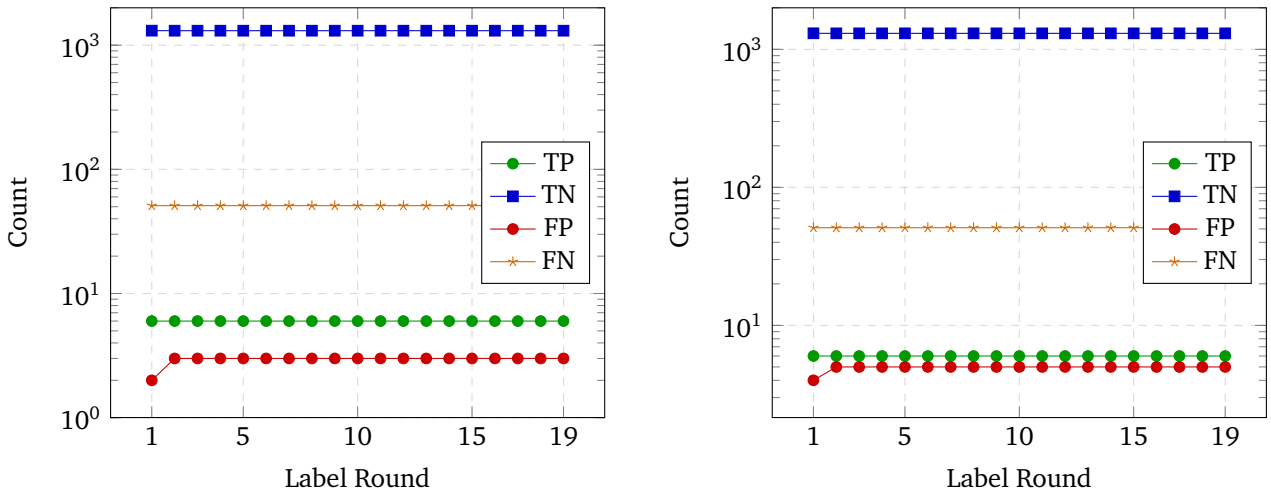
predictions for these first experiments and as tables 12-18 show this difference can often significantly increase the end results.

6.2.2 Dynamic Chain length Optimization

Besides this comparison of the two prediction methods, we also use this prediction pipeline analysis to determine the optimal length of the dynamic chain. The experiments depicted in table 12-18 were all performed by a chain with the length equal to the number of labels. So for the *emotions* dataset we used a chain with 6 classifier-nodes and for the *medical* dataset a chain with 45 classifiers. One problem of this is the current implementation of the dynamic chain and the modified XGBoost algorithm. Because of some compromises that were made during the practical implementation, the training of the chain is currently not very efficient and depends linearly on the number of labels. So in order to speed up further testing, we want to optimize the length of the chain in order to decrease the computational time.
In order to find the optimal lengths we took again a look at figures 11-17 and determined the label round from where the predictions did not change anymore. If we take for example the *yeast* dataset, we see in figure 17 that after label round five the TP, TN, FP and FN counts stay the same. So for this example we could choose a chain with five classifiers. We finally chose a chain with length six for this dataset in order to add some buffer. Table 19 shows the selected chain lengths for each dataset. For some of the smaller sets like *birds* we decided to keep the original full length because the training was already very fast, but especially for *genbase, medical* and *yeast* it was important to reduce the length, since these were the datasets with the highest computational time.

Dataset	Chain Length
birds	19
emotions	6
flags	7
genbase	4
medical	4
scene	6
yeast	6

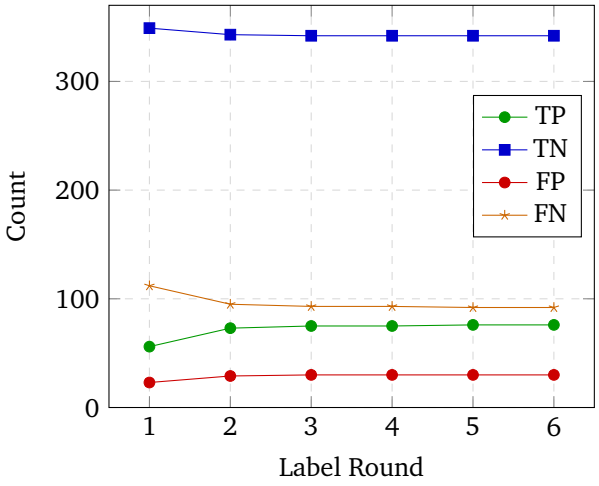
Table 19: Dynamic chain lengths used for further testing



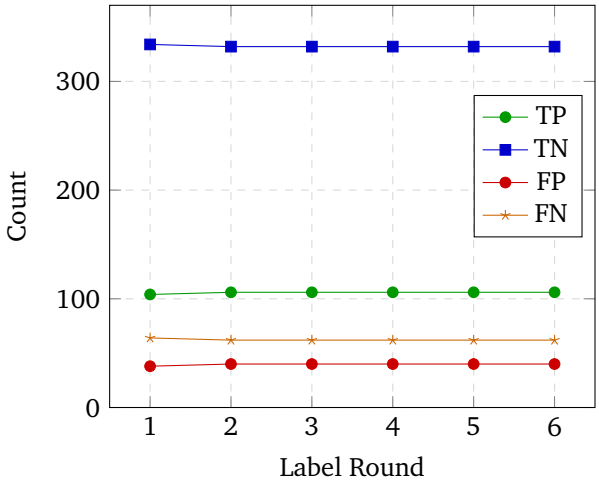
(a) Default Predictions

(b) Cumulated predictions

Figure 11: Predicted labels per label round - birds

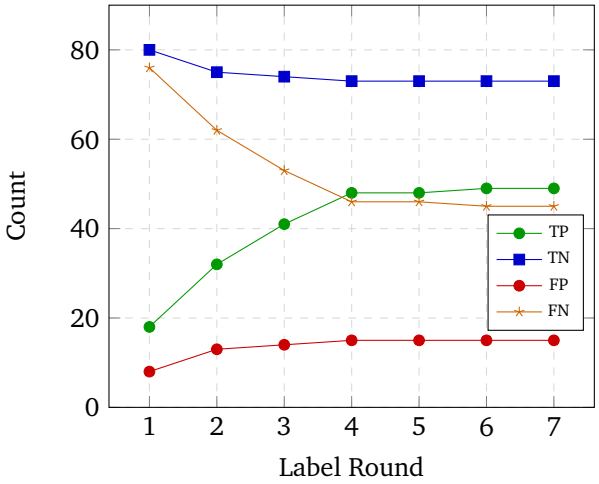


(a) Default Predictions

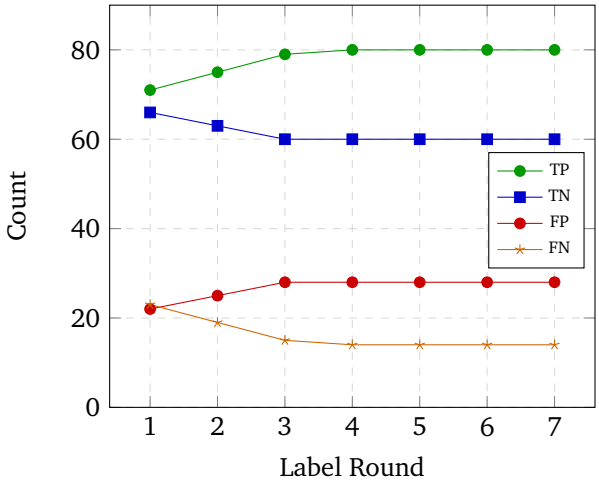


(b) Cumulated predictions

Figure 12: Predicted labels per label round - emotions

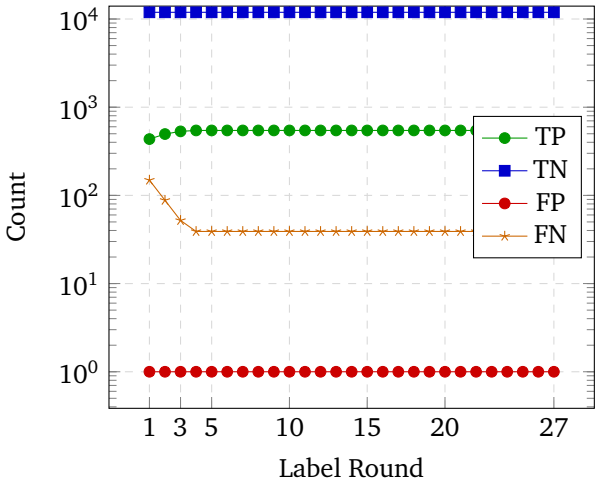


(a) Default Predictions

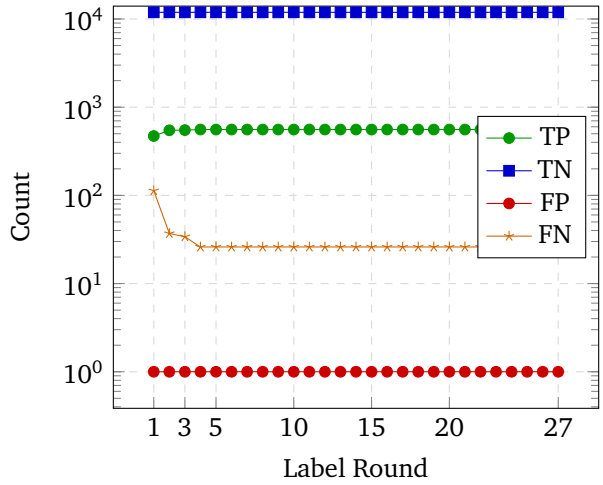


(b) Cumulated predictions

Figure 13: Predicted labels per label round - flags

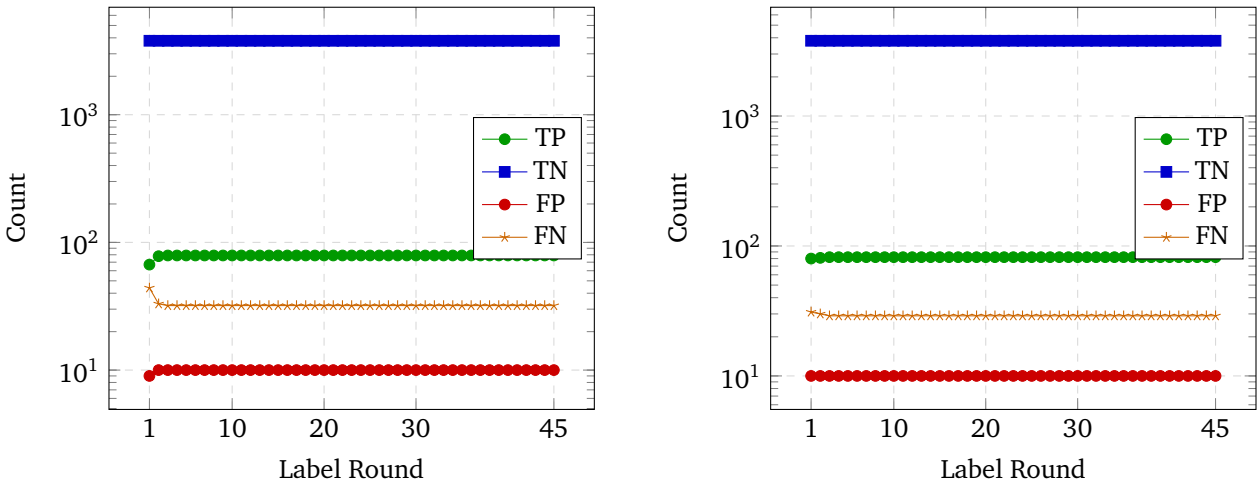


(a) Default Predictions



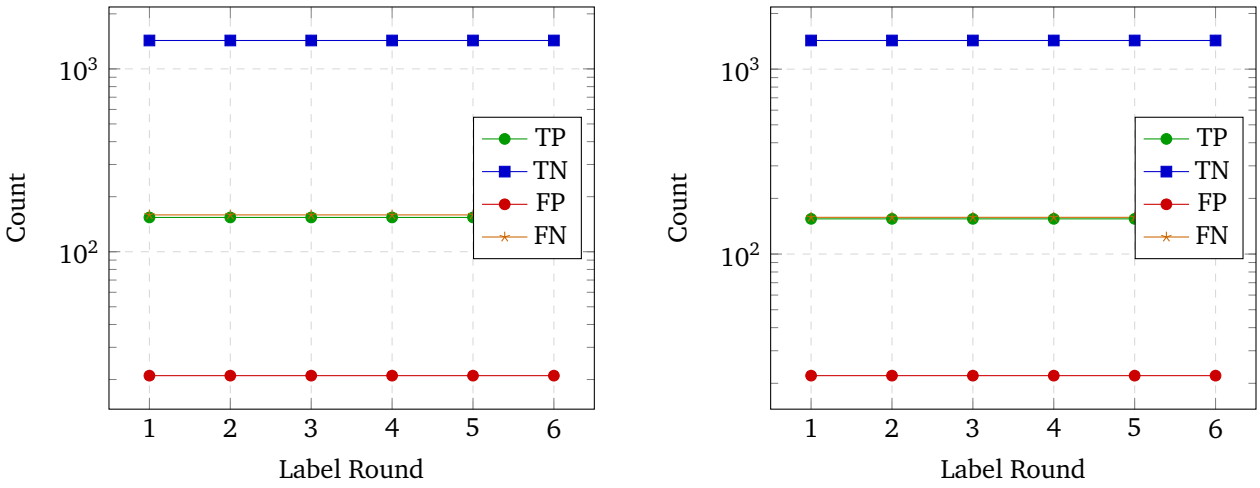
(b) Cumulated predictions

Figure 14: Predicted labels per label round - genbase



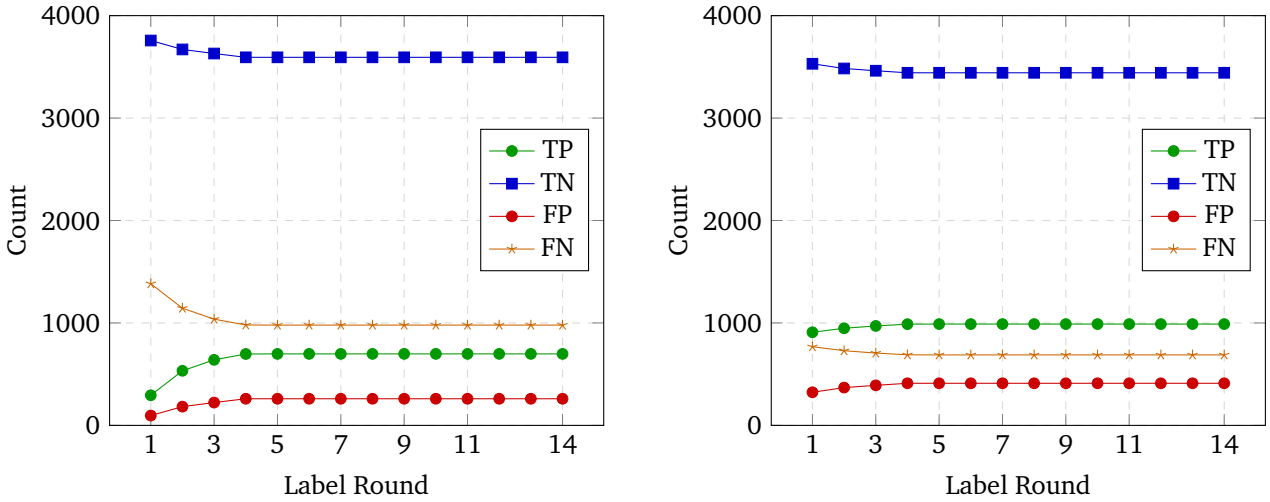
(a) Default Predictions **(b)** Cumulated predictions

Figure 15: Predicted labels per label round - medical



(a) Default Predictions **(b)** Cumulated predictions

Figure 16: Predicted labels per label round - scene



(a) Default Predictions **(b)** Cumulated predictions

Figure 17: Predicted labels per label round - yeast

6.3 Comparison of Dynamic Chain and Baseline Approaches

After the chain parameters for split-calculation, label-propagation and prediction-type have been evaluated, the next step was to tune the XGBoost base classifiers. Therefore, we took for each dataset the two splits with the best results (blue highlighted in tables 12-18) and tuned the XGBoost parameters for maximum tree depth and number of boosting rounds. The parameter set with the best results on the validation set was then used to train the dynamic chain on the complete train set and generate the final predictions on the test set. Note that one especialness that occurred during this evaluation was that the parameter sets for the best *Subset Accuracy* also corresponded to the best *Hamming Loss* results. The final prediction results on the test sets and their corresponding parameter initialization can be found in figure 21. Except for the *scene* dataset we always used the CUM predictions for this final evaluation. This can again be justified by the low label cardinality of this dataset. Thereby the scores for the DEF predictions were slightly better than the ones for the CUM predictions, because the CUM predictions seemed to have added some more false positives. For the *birds* dataset with even a lower cardinality we got the same scores for DEF and CUM and therefore chose the CUM ones since they provided better results in the majority of all cases.
Let us now look at the final results on the test sets. The results for the baseline methods can be found in table 20. In table 22 we compare the results of these baselines to the ones of the dynamic chain, but we only regard our main optimization goals *Hamming Loss* and *Subset Accuracy*. Thereby *LP* denotes the *Label Powerset* method, *BR* is short for *Binary Relevance*, *CC* means *Classifier Chains*, *MTRT* stands for the single *Multi Target Regression Tree* and *DC* denotes our *Dynamic Chain* approach. For some easier analysis we also highlighted the best results for each dataset. Additionally, we added figures 18 - 24 to give some more intuitive understanding of how the different approaches performed. The x-axis thereby shows the *Hamming Loss*, which we want to minimize, and the y-axis shows the *Subset Accuracies*, which need to be maximized. The different colors and forms then show the different algorithms. Red are the *J48* approaches, blue the *XGBoost* approaches, the *Multi Target Tree* is green and the *Dynamic Chain* results are orange. At first, we will focus on these tables and the measures *Hamming Loss* and *Subset Accuracy*. For *Hamming Loss* the dynamic chain provides the best results for the datasets *emotions, medical, scene* and *yeast* and also in the remaining datasets the results are close to the best one, but in *Subset Accuracy* they are always outperformed by *LP* and *CC* methods combined with XGBoost. The success of LP methods is due to the fact that the datasets are not very big and therefore the number of unique label combinations is rather low. The *scene* set for example contains only 14 unique label combinations. If the label powerset method then chooses the correct class, all labels of this instance are set correctly which then boosts both measures. Especially in the *flags* dataset these *Subset Accuracy* results are worst among all classifiers. Therefore, we can now also consider the precision results in tables 20 and 21. Although DC has the best results for *Recall*, which means a lower false negative rate, especially the *micro averaged precision* results are lower than for the other approaches. This means that the other approaches have a lower number of false positive predictions. Since the dataset only contains 194 instances, this higher number of false positives heavily worsens the *Subset Accuracy*. Another thing we can clearly detect is that the transformation methods in combination with XGBoost mostly outperform the corresponding approaches combined with J48. An explanation therefore can be found in (Wyner et al., 2015) where it is shown that random forest approaches, where also XGBoost belongs to, generate better results than single tree classifiers like J48. We also notice that the dynamic chain approach always exceeds the results of single multi-target-regression-trees and that the MTRT mostly generate bad results. In figure 21 they are not even shown since the corresponding *subset accuracy* is far below the other ones. We can therefore give two reasons. One is similar as before that a single tree is always outperformed by a forest of trees. The other one is that the results of the remaining methods, except for LP that transforms the whole problem, are generated by an ensemble or a chain of classifiers. So where the MTRT has to learn all label in a single tree, BR or CC predict each label with an own classifier.
Besides this, figure 21 also shows some anomalies. Both LP results perform bad in *Subset Accuracy* and

the CC and BR results are nearly the same for their corresponding base classifiers. But looking at the axes shows that the *hamming loss* results are, except for MTRT which we have already shown, nearly the same and only differ at most $1 \cdot 10^{-3}$ from each other. Also for the *subset accuracy* we can see that they are very close together. So already a single different predicted label can make the difference.
By having a look at the other evaluation measures, we detect that DC has often really good results on Micro averaged precision, which shows that the they have the best rate of true positive classified labels among all labels that have been predicted positive. Since the *subset accuracies* are still lower compared to the other approaches, we can conclude that we still have a high number of negative predicted labels where the true target label is positive. We can also see this by comparing the *Micro Averaged Recall* results where they are mostly low for datasets with high *Micro Averaged Precision* values. For *flags* and *yeast* we have the opposite case for these measures and we predict to much false positives. Even the *F1* measures confirm this observation and are mostly low compared to the other approaches. Again there is an exception for *flags* and *yeast* and both of them have really good *F1* results. This is due to the fact that the corresponding *Recall* values which influence the *F1* measures, are a whole lot better than for the other methods.

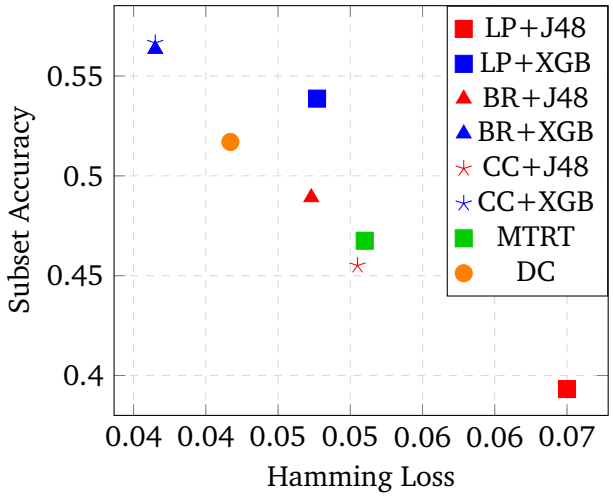


Figure 18: Model comparison - birds

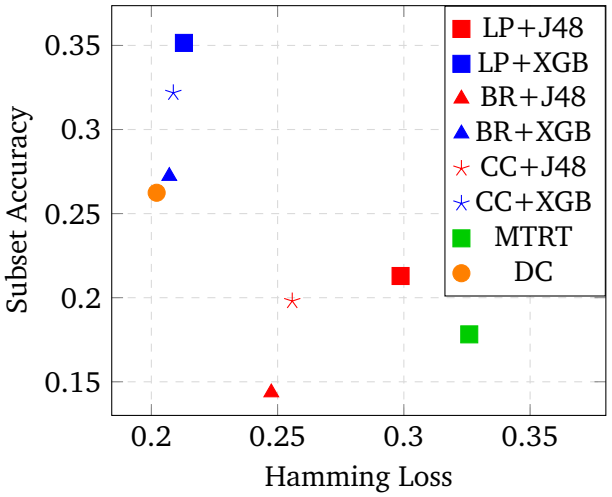


Figure 19: Model comparison - emotions

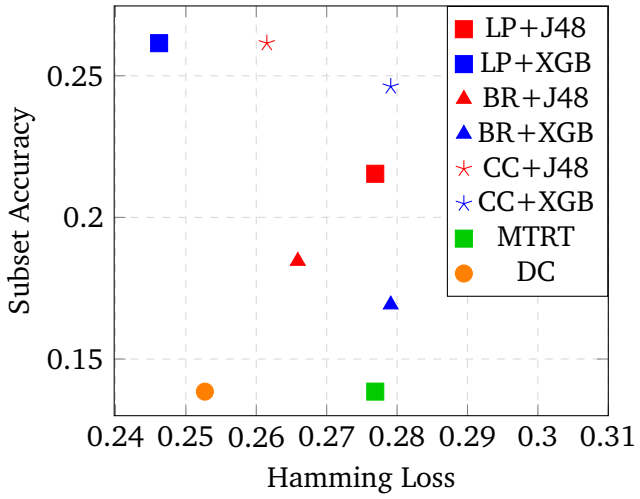


Figure 20: Model comparison - flags

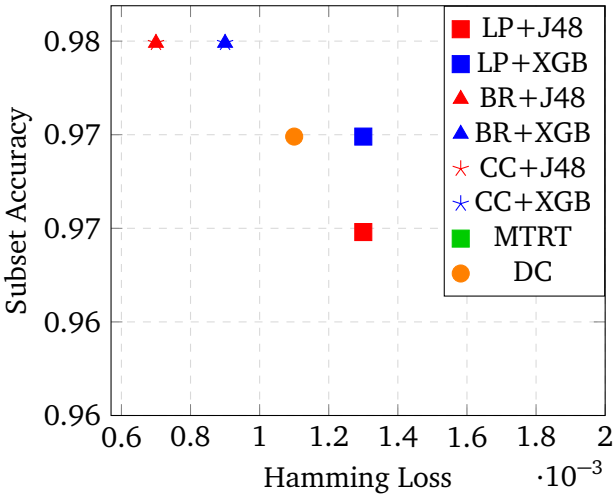


Figure 21: Model comparison - genbase

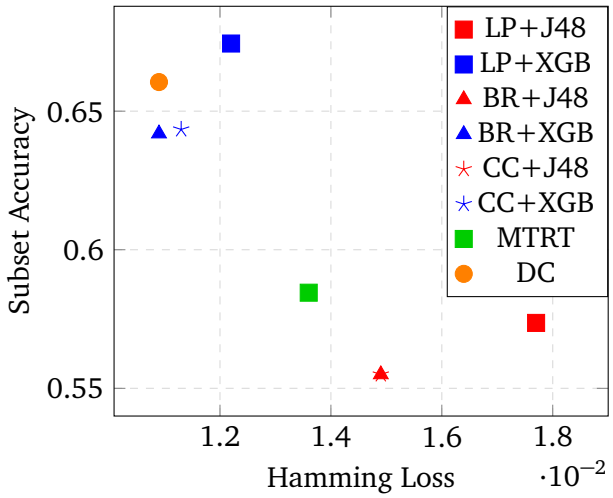


Figure 22: Model comparison - medical

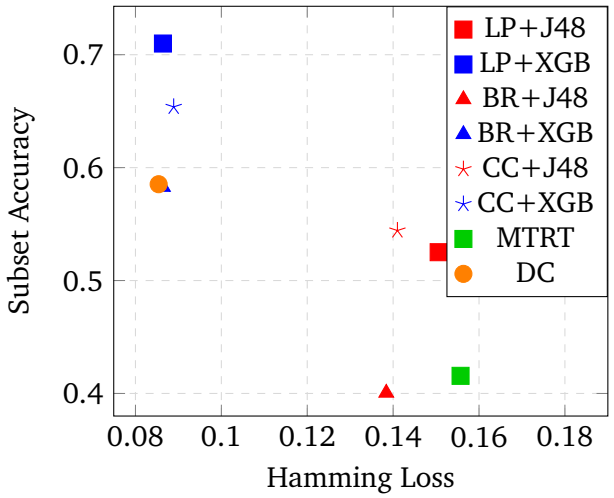


Figure 23: Model comparison - scene

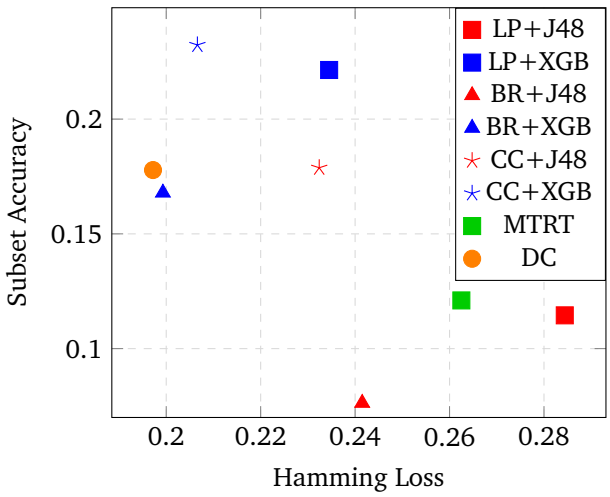


Figure 24: Model comparison - yeast

Dataset	Classifier	Hamming	Subset	Macro F1	Micro F1	Example F1	Macro Prec	Macro Rec	Micro Prec	Micro Rec
birds	LP+XGB	0.0477	0.5387	0.3601	0.4604	0.6500	0.4101	0.3287	0.5435	0.3994
	LP+J48	0.0650	0.3932	0.2951	0.4036	0.5337	0.2828	0.3306	0.3792	0.4313
	BR+XGB	0.0365	0.5635	0.3217	0.5109	0.6695	0.6085	0.2428	0.8069	0.3738
	BR+J48	0.0473	0.4892	0.1997	0.3750	0.5864	0.2913	0.1646	0.5762	0.2780
	CC+XGB	0.0365	0.5666	0.3228	0.5109	0.6699	0.6066	0.2435	0.8069	0.3738
	CC+J48	0.0505	0.4551	0.1790	0.3648	0.5521	0.2505	0.1549	0.5086	0.2843
	MTRT	0.0510	0.4675	0.0000	0.0000	0.4675	0.0000	0.0000	0.0000	0.0000
emotions	LP+XGB	0.2129	0.3515	0.6655	0.6718	0.6408	0.6746	0.6605	0.6822	0.6617
	LP+J48	0.2987	0.2129	0.5076	0.5347	0.4985	0.5259	0.4961	0.5488	0.5213
	BR+XGB	0.2071	0.2723	0.6441	0.6557	0.5886	0.7249	0.5902	0.7242	0.5990
	BR+J48	0.2475	0.1436	0.6156	0.6173	0.5680	0.6361	0.6029	0.6286	0.6065
	CC+XGB	0.2087	0.3218	0.6600	0.6735	0.6309	0.6867	0.6430	0.6941	0.6541
	CC+J48	0.2558	0.1980	0.5790	0.5921	0.5558	0.6175	0.5631	0.6233	0.5639
	MTRT	0.3259	0.1782	0.5029	0.5117	0.4802	0.5017	0.5145	0.5049	0.5188
flags	LP+XGB	0.2462	0.2615	0.6452	0.7431	0.7166	0.6862	0.6370	0.7397	0.7465
	LP+J48	0.2769	0.2154	0.6257	0.7123	0.6953	0.6405	0.6338	0.7059	0.7189
	BR+XGB	0.2791	0.1692	0.6317	0.7107	0.6806	0.6306	0.6409	0.7027	0.7189
	BR+J48	0.2659	0.1846	0.5765	0.7056	0.6870	0.6976	0.5492	0.7474	0.6682
	CC+XGB	0.2791	0.2462	0.6461	0.7107	0.6815	0.6390	0.6570	0.7027	0.7189
	CC+J48	0.2615	0.2615	0.6224	0.7252	0.7051	0.6540	0.6116	0.7269	0.7235
	MTRT	0.2769	0.1385	0.5949	0.7083	0.6816	0.6218	0.5886	0.7116	0.7051
genbase	LP+XGB	0.0013	0.9698	0.8342	0.9856	0.9910	0.8426	0.8333	0.9958	0.9755
	LP+J48	0.0013	0.9749	0.8758	0.9856	0.9889	0.8882	0.8667	0.9958	0.9755
	BR+XGB	0.0009	0.9799	0.8494	0.9897	0.9925	0.8472	0.8519	0.9959	0.9837
	BR+J48	0.0007	0.9799	0.8885	0.9918	0.9946	0.8882	0.8889	0.9918	0.9918
	CC+XGB	0.0009	0.9799	0.8494	0.9897	0.9925	0.8472	0.8519	0.9959	0.9837
	CC+J48	0.0007	0.9799	0.8885	0.9918	0.9946	0.8882	0.8889	0.9918	0.9918
	MTRT	0.0516	0.1709	0.1888	0.2063	0.1759	0.2348	0.1723	0.3462	0.1469
medical	LP+XGB	0.0122	0.6744	0.4261	0.7686	0.7752	0.4643	0.4127	0.8055	0.7350
	LP+J48	0.0177	0.5736	0.4415	0.6743	0.6857	0.4569	0.4419	0.6852	0.6638
	BR+XGB	0.0109	0.6419	0.4368	0.7929	0.7521	0.4555	0.4309	0.8333	0.7563
	BR+J48	0.0149	0.5550	0.3669	0.7251	0.6849	0.3760	0.3699	0.7368	0.7138
	CC+XGB	0.0113	0.6434	0.4384	0.7881	0.7507	0.4760	0.4374	0.8183	0.7600
	CC+J48	0.0149	0.5550	0.3718	0.7265	0.6878	0.3767	0.3757	0.7344	0.7188
	MTRT	0.0136	0.5845	0.3881	0.7227	0.7207	0.4290	0.3732	0.8217	0.6450
scene	LP+XGB	0.0864	0.7099	0.7577	0.7534	0.7595	0.7828	0.7356	0.7794	0.7290
	LP+J48	0.1506	0.5251	0.5871	0.5792	0.5832	0.6005	0.5764	0.5858	0.5727
	BR+XGB	0.0864	0.5828	0.7235	0.7254	0.6380	0.8494	0.6357	0.8540	0.6305
	BR+J48	0.1384	0.4005	0.6156	0.6070	0.5484	0.6407	0.5957	0.6246	0.5905
	CC+XGB	0.0889	0.6538	0.7393	0.7337	0.6990	0.8120	0.6833	0.8013	0.6767
	CC+J48	0.1410	0.5443	0.6164	0.6044	0.6062	0.6401	0.5992	0.6140	0.5951
	MTRT	0.1557	0.4156	0.5190	0.5200	0.4738	0.5800	0.4743	0.5885	0.4657
yeast	LP+XGB	0.2345	0.2214	0.4193	0.6009	0.5763	0.4483	0.4034	0.6218	0.5814
	LP+J48	0.2844	0.1145	0.3594	0.5277	0.4997	0.3633	0.3581	0.5324	0.5232
	BR+XGB	0.1993	0.1679	0.3948	0.6397	0.6106	0.5616	0.3598	0.7095	0.5825
	BR+J48	0.2415	0.0763	0.3761	0.5873	0.5650	0.3965	0.3670	0.6106	0.5658
	CC+XGB	0.2066	0.2323	0.4053	0.6288	0.5927	0.4866	0.3701	0.6921	0.5760
	CC+J48	0.2324	0.1788	0.3573	0.5810	0.5546	0.3977	0.3349	0.6419	0.5306
	MTRT	0.2625	0.1210	0.3515	0.5504	0.5143	0.3777	0.3388	0.5735	0.5291

Table 20: Results of the baseline methods on the test sets

Dataset	Split Method	Propagation Method	Label Rounds	Boosting Rounds	Max Depth	Prediction Type	Hamming	Subset Accuracy	Macro F1	Micro F1	Example F1	Macro Precision	Macro Recall	Micro Precision	Micro Recall
birds	5	1	19	100	50	CUM	0.0417	0.5170	0.1936	0.3600	0.5921	0.4519	0.1379	0.8276	0.2300
emotions	3	0	6	100	100	CUM	0.2021	0.2624	0.6610	0.6720	0.6089	0.7227	0.6212	0.7213	0.6291
flags	2	0	7	100	100	CUM	0.2527	0.1385	0.6989	0.7609	0.7430	0.6377	0.7796	0.6932	0.8433
genbase	3	0	4	100	10	CUM	0.0011	0.9749	0.7778	0.9876	0.9924	0.7778	0.7778	1.0	0.9755
medical	2	1	4	25	25	CUM	0.0109	0.6605	0.3904	0.7826	0.7580	0.4157	0.3773	0.8737	0.7088
scene	3	1	6	150	150	DEF	0.0854	0.5853	0.7142	0.7158	0.6237	0.8919	0.6020	0.8998	0.5943
yeast	3	1	6	150	25	CUM	0.1972	0.1778	0.4002	0.6519	0.6274	0.5481	0.3747	0.7025	0.6081

Table 21: Final parameter setup and prediction results of the dynamic chain on the test sets

	birds		emotions		flags		genbase		medical		scene		yeast	
	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy	Hamming Loss	Subset Accuracy
LP+J48	0.065	0.3932	0.2987	0.2129	0.2769	0.2154	0.0013	0.9698	0.0177	0.5736	0.1506	0.5251	0.2844	0.1145
LP+XGB	0.0477	0.5387	0.2129	0.3515	0.2462	0.2615	0.0013	0.9749	0.0122	0.6744	0.0864	0.7099	0.2345	0.2214
BR+J48	0.0473	0.4892	0.2475	0.1436	0.2659	0.1846	0.0007	0.9799	0.0149	0.555	0.1384	0.4005	0.2415	0.0763
BR+XGB	0.0365	0.5635	0.2071	0.2723	0.2791	0.1692	0.0009	0.9799	0.0109	0.6419	0.0864	0.5828	0.1993	0.1679
CC+J48	0.0505	0.4551	0.2558	0.198	0.2791	0.2462	0.0007	0.9799	0.0149	0.555	0.141	0.5443	0.2324	0.1788
CC+XGB	0.0365	0.5666	0.2087	0.3218	0.2615	0.2615	0.0009	0.9799	0.0113	0.6434	0.0889	0.6538	0.2066	0.2323
MTRT	0.051	0.4675	0.3259	0.1782	0.2769	0.1385	0.0516	0.1709	0.0136	0.5845	0.1557	0.4156	0.2625	0.121
DC	0.0417	0.517	0.2021	0.2624	0.2527	0.1385	0.0011	0.9749	0.0109	0.6605	0.0854	0.5853	0.1972	0.1778

Table 22: Comparison of the final predictions from the baseline methods and the dynamic chain on the test sets

7 Related Work

In this section we give a short insight to some other methods for multilabel classification and multi-target trees.

In (Schapire and Singer, 2000) two modified version of the AdaBoost algorithm (Freund and Schapire, 1995), AdaBoost.MH and AdaBoost.MR, are used for the task of text categorization. AdaBoost.MH maintains a set of weights and passes it each round to a set of weak classifiers. The sign of the output of these weak classifiers is then used to predict a label. Afterwards the weight distribution is updated and the process repeats. This version is used to minimize Hamming Loss. AdaBoost.MR is also based on weak classifiers where their output is used to find a ranking that tries to put the correct labels to the top.

Another tree classifier for extreme multilabel problems is FastXML (Prabhu and Varma, 2014) which scales very well on datasets with a tremendous number of labels and solves these problems efficiently with a low computational time complexity.

Besides these approach for tree models, there also exist some other methods to directly detect and exploit dependencies between labels. One of these proposes a modified SVM (Godbole and Sarawagi, 2004) that is able to exploit co-occurences of labels. Therefore, a binary classifier is trained for each label and the predictions of these classifiers are then used to create an extended dataset by adding them as features to the train-set. Afterwards a second ensemble of classifiers is trained on this extended dataset. For predicting a new test instance, the first classifier ensemble generates predictions for each label, adds them as features to this instance and the second ensemble generates the final predictions.

Two other approaches for exploiting label dependencies can be found in (Loza Mencía and Janssen, 2016). The first one combines rule learning with bootstrapped stacking where for each label a separate ruleset is learned on a train set which includes the remaining labels as attributes. The second one combines multilabel rule learning with a separate and conquer approach. Therefore, covered examples with predicted labels are re-included to allow learning of subsequent rules. Especially the stacking approach is very effective at learning rules that depend on other labels.

Another method that tries to incorporate label dependencies to improve multilabel classification can be found in (Zhang and Zhang, 2010). There a Bayesian network is used to efficiently encode the label dependencies. The problem is then decomposed into single-label tasks where parental labels are added as additional features. Unseen instances are then recursively predicted by the network. A similar approach can also be found in (Guo and Gu, 2011) where a cyclic directed graphical model is used to represent label dependencies.

8 Future Work

The dynamic chain proposed in this thesis still has some potential for improvements. In this section we will shortly introduce some points that may improve the performance, usability and the prediction results.

8.1 Performance Improvements

The first point, that was already mentioned before, concerns the computational time and performance of the algorithm. The current implementation and especially the XGBoost modifications are a compromise that allowed us to experiment with the dynamic chain. The main problem is the computation complexity that is linear with the number of labels. So the current implementation is not able to run on really large datasets like the *mediamill* or *bookmarks* datasets which are both available in the MULAN library. Therefore, the need arises to make some more profound changes to the XGBoost source code which adapt the fundamental tree representation and add the ability to directly deal with multilabel data. Another recommendation is to reduce the number of frameworks the data is passed through. At the moment we use MULAN to process the multilabel data. This data is then forwarded to a WEKA classifier that passes the data to the XGBoost Java interface which finally propagates it to the actual XGBoost core that learn the model.

8.2 Refinement of Tree Construction

For now, we have only focused on the two XGBoost parameters maximum tree depth and number of boosting rounds. Besides tuning the high number of remaining parameters, we assume that there is still space for improvement in the multitarget tree construction. Up until now we have combined the cross entropy objective with various kinds of split calculation approaches. Since the objective function can be easily swapped out, we could evaluate the performance of several other objective functions like a linear regression. Another idea is to analyze different split calculations. So far, they are based on the gain calculation for the default regression task or based on the leaf weight calculation. Some other split calculations could aim for a compromise of maximizing the probability for a single label while also obtaining decent results for the remaining ones. Furthermore, we could adapt the tree structure and allow the tree to connect nodes to more than two child-nodes or leaves.

8.3 Early Stopping and Self Correction

The last attempt for improvement concerns the structure of the dynamic chain. The evaluation showed that often a very short chain is sufficient to provide good results. So a idea could be to combine the chain with some early stopping criterion. We could therefore introduce a new label, called *stopping label*, in order to build a chain with a dynamic length. If this *stopping label* is predicted during the training or predicting process it means that this particular instance has no more positive labels missing. This instance could then be removed from the set and if no more instances are propagated along the chain the process stops.
The second idea for improving the prediction results is to add the ability for the chain to correct itself. If the current version of the chain predicts a label, this label is fixed and cannot be changed later on. Even if all following classifiers would predict it differently, they have no chance to update it. An idea of self-correction is to add another label, called the *correction label*. If this label is predicted it means that the lastly propagated label is very likely a wrong prediction. In this case the label could be removed or updated.

9 Conclusion

In this thesis we have developed a novel approach to solve multilabel problems by predicting the labels dynamically and individually per instance in order to exploit potential dependencies between them. In a first step we have therefore proposed a new tree structure that builds on top of the popular XGBoost algorithm for gradient tree boosting and adds the capability to predict multiple labels simultaneously. In a second step this modified XGBoost algorithm was then used as a base classifier for a dynamic chain of classifiers. This dynamic chain gradually trains these tree classifiers and propagates their most probable predicted labels to the next classification-node which can use this additional information to detect and exploit dependencies to previous predicted labels. Afterwards we introduced the additional refinement approaches *separate and conquer* and *cumulated predictions* which massively increased the predictive performance and robustness of the dynamic chain.
Besides the idea of detecting and exploiting label dependencies, the main advantage of this approach is that we no longer have to determine a static order of the dynamic chain as it was required for classifier chains. Each test instance, that is now passed along, gets its labels predicted in an individual order where only labels with high probabilities are assigned. Nevertheless, the current implementation has some drawbacks which do not allow an evaluation on larger datasets. Furthermore, the we have some pretty high computational times which make it hard to tune the high number of parameters for the chain and its base classifiers.
However, the final comparison with the baseline methods showed that the dynamic chain can compete with current state-of-the-art algorithms for multilabel-classification. Especially the results for *Hamming Loss* are pretty good and we were able to generate the best scores on four out of seven test datasets. So all in all, we can say that the proposed dynamic chain in combination with extreme gradient tree boosting shows some great potential, but still needs some further work to overcome its weaknesses and boost its predictive performance.

List of Figures

List of Tables

References

Tianqi Chen. Introduction to boosted trees. *University of Washing Computer Science. University of Washington*, 22, 2014.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

Everton Alvares Cherman, Maria Carolina Monard, and Jean Metz. Multi-label problem transformation methods: a case study. *CLEI Electronic Journal*, 14(1):4–4, 2011.

John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz-open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

Johannes Fürnkranz. Round robin rule learning. In *Proceedings of the 18th International Conference on Machine Learning (ICML-01): 146–153*. Citeseer, 2001.

Shantanu Godbole and Sunita Sarawagi. Discriminative methods for multi-labeled classification. *Advances in knowledge discovery and data mining*, pages 22–30, 2004.

Xinjian Guo, Yilong Yin, Cailing Dong, Gongping Yang, and Guangtong Zhou. On the class imbalance problem. In *Natural Computation, 2008. ICNC'08. Fourth International Conference on*, volume 4, pages 192–201. IEEE, 2008.

Yuhong Guo and Suicheng Gu. Multi-label classification using conditional dependency networks. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1300, 2011.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

Dragi Kocev, Sašo Džeroski, Matt D White, Graeme R Newell, and Peter Griffioen. Using single-and multi-target regression trees and ensembles to model a compound index of vegetation condition. *Ecological Modelling*, 220(8):1159–1168, 2009.

Thales Sehn Korting. C4. 5 algorithm and multivariate decision trees. *Image Processing Division, National Institute for Space Research–INPE Sao Jose dos Campos–SP, Brazil*, 2006.

Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. Optimal thresholding of classifiers to maximize f1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 225–239. Springer, 2014.

Eneldo Loza Mencía. *Efficient Pairwise Multilabel Classification*. PhD thesis, Technische Universität, 2013.

Eneldo Loza Mencía and Frederik Janssen. Learning rules for multi-label classification: a stacking and a separate-and-conquer approach. *Machine Learning*, 105(1):77–126, 2016.

Oscar Luaces, Jorge Díez, José Barranquero, Juan José del Coz, and Antonio Bahamonde. Binary relevance efficacy for multilabel classification. *Progress in Artificial Intelligence*, 1(4):303–313, 2012.

Oded Maimon and Lior Rokach. Introduction to knowledge discovery and data mining. In *Data Mining and Knowledge Discovery Handbook*, pages 677–679. Springer, 2009.

Yashoteja Prabhu and Manik Varma. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 263–272. ACM, 2014.

Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine learning*, 85(3):333–359, 2011.

Robert E Schapire and Yoram Singer. Boostexter: A boosting-based system for text categorization. *Machine learning*, 39(2-3):135–168, 2000.

Robin Senge, Juan José del Coz Velasco, and Eyke Hüllermeier. Rectifying classifier chains for multi-label classification. *Space, 2 (8)*, 2013.

Jan Struyf and Sašo Džeroski. Constraint based induction of multi-objective regression trees. In *International Workshop on Knowledge Discovery in Inductive Databases*, pages 222–233. Springer, 2005.

Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3), 2006.

Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research*, 12(Jul):2411–2414, 2011.

Abraham J Wyner, Matthew Olson, Justin Bleich, and David Mease. Explaining the success of adaboost and random forests as interpolating classifiers. *arXiv preprint arXiv:1504.07676*, 2015.

Min-Ling Zhang and Kun Zhang. Multi-label learning by exploiting label dependency. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 999–1008. ACM, 2010.