Extreme F-Measure Maximization using Sparse Probability Estimates

Kalina Jasinska KJASINSKA@CS.PUT.POZNAN.PL
Krzysztof Dembczyński KDEMBCZYNSKI@CS.PUT.POZNAN.PL
Institute of Computing Science, Poznań University of Technology, 60-965 Poznań, Poland

Róbert Busa-Fekete BUSAROBI@GMAIL.COM
Karlson Pfannschmidt KIUDEE@MAIL.UPB.DE
Timo Klerx TIMOK@UPB.DE
Eyke Hüllermeier EYKE@UPB.DE
Department of Computer Science, Paderborn University, 33098 Paderborn, Germany

Abstract

We consider the problem of (macro) F-measure maximization in the context of extreme multi-label classification (XMLC), i.e., multi-label classification with extremely large label spaces. We investigate several approaches based on recent results on the maximization of complex performance measures in binary classification. According to these results, the F-measure can be maximized by properly thresholding conditional class probability estimates. We show that a naïve adaptation of this approach can be very costly for XMLC and propose to solve the problem by classifiers that efficiently deliver *sparse probability estimates* (SPEs), that is, probability estimates restricted to the most probable labels. Empirical results provide evidence for the strong practical performance of this approach.

1. Introduction

Extreme classification refers to multi-class and multi-label learning problems that involve hundreds of thousands (Deng et al., 2009; Partalas et al., 2015) or even millions of labels (Agrawal et al., 2013). Applications of that kind can be found in image classification (Deng et al., 2011), text document classification (Dekel & Shamir, 2010), on-line advertising (Beygelzimer et al., 2009a), and video recommendation (Weston et al., 2013). Several approaches for efficient extreme classification have recently been proposed, including FastXML (Prabhu & Varma, 2014), LomTrees (Choromanska & Langford, 2015), SLEEC (Bhatia et al., 2015), robust Bloom filters (Cisse et al., 2013), label partitioning (Weston et al., 2013), and fast label embeddings (Mineiro & Karampatziakis, 2015).

These algorithms have been mainly evaluated in terms of ranking-based measures such as Precision@K or NDCG@K. In this work, we focus on extreme multi-label classification (XMLC) and turn our attention to the (macro) F-measure, which is a commonly used performance measure in multi-label classification as well as other fields, such as natural language processing. A variant of this measure has also been used in the Kaggle's LSHTC competition (Partalas et al., 2015); see `https://www.kaggle.com/c/lshtc`.

We tackle the learning problem within the Empirical Utility Maximization (EUM) framework for F-measure maximization. As shown recently, maximizing the F-measure in EUM can be accomplished by properly tuning a threshold τ on class probability estimates (CPEs), that is, using a suitably chosen threshold τ to separate between positive and negative predictions (Koyejo et al., 2014; Narasimhan et al., 2014; Kotłowski & Dembczynski, 2015). In the extreme learning scenario, both probability estimation and threshold tuning are challenging tasks; in particular, tuning the threshold for each label through exhaustive search is computationally infeasible. In this paper, we aim at extending existing threshold tuning methods for optimizing the F-measure in binary classification, so as to make them amenable to the optimization of the macro F-measure in the XMLC setting. To this end, we propose the idea of *sparse probability estimates* (SPEs).

Following a formal description of the macro F-measure and the EUM approach in Section 2 and 3, respectively, we adapt threshold tuning methods to the XMLC setting by exploting the idea of SPEs in Section 4. In Section 5, we discuss two concrete methods for delivering SPEs. The combination of these methods with threshold optimization approaches is evaluated experimentally in Section 6.

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48.

2. Macro-averaged F-measure

Consider an XMLC problem with m labels and denote the true and predicted label vector of the ith instance $\boldsymbol{x}_i$ by $\boldsymbol{y}_i = (y_{i,1}, \ldots, y_{i,m}) \in \{0,1\}^m$ and $\widehat{\boldsymbol{y}}_i = (\hat{y}_{i,1}, \ldots, \hat{y}_{i,m}) \in \{0,1\}^m$, respectively. For a test set $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ of size n, let $\boldsymbol{Y}$ be the $n \times m$ matrix with entries $y_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$, and denote by $\boldsymbol{y}_{\cdot j}$ the jth column of $\boldsymbol{Y}$; the same notation is used for the collection of estimates $\widehat{\boldsymbol{Y}}$. The macro F-measure (or F-score) is then defined as:

$$F_M\left(\boldsymbol{Y}, \widehat{\boldsymbol{Y}}\right) = \frac{1}{m}\sum_{j=1}^{m} F(\boldsymbol{y}_{\cdot j}, \widehat{\boldsymbol{y}}_{\cdot j}) = \frac{1}{m}\sum_{j=1}^{m} \frac{2\sum_{i=1}^n y_{i,j}\hat{y}_{i,j}}{\sum_{i=1}^n y_{i,j} + \sum_{i=1}^n \hat{y}_{i,j}} \quad (1)$$

To maximize this measure, it obviously suffices to maximize the (standard) F-measure $F(\boldsymbol{y}_{\cdot j}, \widehat{\boldsymbol{y}}_{\cdot j})$ for each label j separately. In principle, threshold tuning methods for F-measure maximization in binary classification can thus be used, simply by applying them independently for each label. However, a naïve adaptation of these methods can be very costly for problems with extremely large label spaces. This is because the tuning methods require CPEs for all labels and instances of the set at hand; for example, at least 10^{10} predictions have to be produced and possibly stored for $m > 10^5$ labels and $n > 10^5$ instances.

Fortunately, a significant reduction of complexity is possible thanks to important properties of the XMLC problem and the F-measure. First, the number of positive labels in XMLC is typically very small in comparison to the number of negative labels. Second, for computing the F-measure, only the true positive labels ($y_{i,j} = 1$) and the predicted positive labels ($\hat{y}_{i,j} = 1$) are needed, but neither the true negatives nor the predicted negatives. These properties motivate the idea of *sparse probability estimates* (SPEs), by which we mean probability estimates restricted to the top-labels, i.e., those labels with a sufficiently high probability—other probabilities are not important for tuning the threshold. As will be seen, SPEs will significantly reduce complexity of threshold tuning in extreme classification.

3. Expected Utility Maximization (EUM)

Suppose a finite set $\mathcal{D}_n = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ of labeled instances to be given. Moreover, denote by $\eta(\boldsymbol{x}, j)$ the probability that the jth label is positive for the instance $\boldsymbol{x}$, i.e., $\eta(\boldsymbol{x}, j) = \mathbf{P}(y_j = 1 \,|\, \boldsymbol{x})$, and that estimates $\widehat{\eta}(\boldsymbol{x}, j)$ of these posteriors are produced by a probabilistic classifier $\widehat{\eta} : \mathcal{X} \times [m] \longrightarrow [0,1]$. For the time being, we are not interested in the training of this classifier but focus on the F-measure optimization step.

The EUM principle is based on the empirical estimate of the F-score on the data $\mathcal{D}_n$. Consider the F-score obtained by the threshold classifier $\widehat{\eta}^{\boldsymbol{\tau}}(\boldsymbol{x}) = (\widehat{\eta}^{\tau_1}(\boldsymbol{x}, 1), \ldots, \widehat{\eta}^{\tau_m}(\boldsymbol{x}, m))$, where $\widehat{\eta}^{\tau_j}(\boldsymbol{x}, j) = [\![\widehat{\eta}(\boldsymbol{x}, j) \geq \tau_j]\!]$:[1]

$$F\left(\boldsymbol{\tau}; \widehat{\eta}, \mathcal{D}_n\right) = \frac{1}{m}\sum_{j=1}^{m} \frac{\sum_{i=1}^n y_{i,j}\widehat{\eta}^{\boldsymbol{\tau}}(\boldsymbol{x}_i, j)}{\sum_{i=1}^n y_{i,j} + \sum_{i=1}^n \widehat{\eta}^{\boldsymbol{\tau}}(\boldsymbol{x}_i, j)} \quad (2)$$

Obviously, with $\widehat{y}_{i,j} = \widehat{\eta}^{\boldsymbol{\tau}}(\boldsymbol{x}_i, j)$, we have $F(\boldsymbol{\tau}; \widehat{\eta}, \mathcal{D}_n) = F_M(\boldsymbol{Y}, \widehat{\boldsymbol{Y}})$, and the optimal threshold vector

$$\boldsymbol{\tau} \in \operatorname*{arg\,max}_{\boldsymbol{\tau} \in [0,1]^m} F(\boldsymbol{\tau}; \widehat{\eta}, \mathcal{D}_n) \quad (3)$$

can be found by searching for the τ_j in the finite set of candidates $\{\widehat{\eta}(\boldsymbol{x}_i, j)\}_{i=1}^n$, independently for each label.

The problem of F-measure optimization has received increasing attention in recent years and has been tackled with different methods, including EUM but also alternatives like the so-called decision-theoretic approach (Ye et al., 2012). Although most of the contributions so far refer to binary classification (i.e., the standard F-measure), many of the results can be readily extended to the case of the macro F-measure, thanks to the fact that the latter is an average of binary F-measures. More concretely, based on the result of Ye et al. (2012), $F(\boldsymbol{\tau}; \widehat{\eta}, \mathcal{D}_n) \xrightarrow{P} F(\widehat{\eta}^{\boldsymbol{\tau}})$ as $n \to \infty$ for any $\boldsymbol{\tau} \in (0,1)^m$, where $F(\widehat{\eta}^{\boldsymbol{\tau}})$ is the population level macro F-score of $\widehat{\eta}^{\boldsymbol{\tau}}$ defined as

$$F(\widehat{\eta}^{\boldsymbol{\tau}}) = \frac{1}{m}\sum_{i=1}^{m} \frac{\mathbf{E}\left[\eta(\boldsymbol{x}, i)\widehat{\eta}^{\tau_i}(\boldsymbol{x}, i)\right]}{\mathbf{E}\left[\eta(\boldsymbol{x}, i)\right] + \mathbf{E}\left[\widehat{\eta}^{\tau_i}(\boldsymbol{x}, i)\right]} \;,$$

and the expectation is taken with respect to the data distribution. Narasimhan et al. (2014) provide an even stronger result, which can be extended to the macro F-measure, too: If a classifier $\widehat{\eta}_{\mathcal{D}_n}$ is induced from $\mathcal{D}_n$ by an L_1-consistent learner for every label and a threshold $\boldsymbol{\tau}$ is obtained by maximizing (2) on an independent set $\mathcal{D}'_n$, then $F(\widehat{\eta}^{\boldsymbol{\tau}}_{\mathcal{D}_n}) \xrightarrow{P} F(\eta^{\boldsymbol{\tau}^*})$ as $n \longrightarrow \infty$ (under mild assumptions on the data distribution), where $\boldsymbol{\tau}^* = \arg\max_{\boldsymbol{\tau} \in [0,1]^m} F(\eta^{\boldsymbol{\tau}})$. Finally, based on (Ye et al., 2012) (see their Theorem 4), there is no classifier of the form $\mathcal{X} \times [j] \longrightarrow \{0,1\}$ with a population level macro F-score better than $F(\eta^{\boldsymbol{\tau}^*})$. This provides another justification for using threshold classifiers in the multi-label scenario. For a more elaborate discussion on consistency of multi-label classification with complex performance measures, see (Koyejo et al., 2015).

4. Macro F-measure maximization

This section presents three F-measure optimization methods, the first two of which directly build on the EUM framework. Each of these methods seeks to optimize the threshold values in (3), typically in a label-wise manner. Sorting-based

[1] $[\![P]\!] = 1$ if the predicate P is true and $= 0$ otherwise.

threshold optimization (STO) finds an optimal threshold on a validation set, while the fixed thresholds approach (FTA) validates only a restricted set of predefined candidates for the threshold. The third method, online F-measure optimization (OFO), tunes the threshold in an online setting. It can be used either in combination with an online learning method or, like the two previous methods, be applied on a validation set.

In the following, we denote the SPE of the posterior $\eta(\boldsymbol{x}) = (\eta(\boldsymbol{x}, 1), \ldots, \eta(\boldsymbol{x}, m))$ for an instance $\boldsymbol{x}$ and the corresponding index set by

$$\widehat{s}_{\boldsymbol{\kappa}}(\boldsymbol{x}) = \left\{\widehat{\eta}(\boldsymbol{x}, j) : j \in \widehat{\ell}_{\boldsymbol{\kappa}}(\boldsymbol{x})\right\} \ ,$$
$$\widehat{\ell}_{\boldsymbol{\kappa}}(\boldsymbol{x}) = \{j \in [m] : \widehat{\eta}(\boldsymbol{x}, j) > \kappa_j\} \ ,$$

where $\boldsymbol{\kappa} = (\kappa_1, \ldots, \kappa_m) \in [0, 1]^m$, and κ_j is the threshold used for the jth label. If the threshold is the same for each label, i.e., $\kappa_1 = \cdots = \kappa_m = \kappa$, we write $\widehat{s}_\kappa(\boldsymbol{x})$ and $\widehat{\ell}_\kappa(\boldsymbol{x})$ instead of $\widehat{s}_{\boldsymbol{\kappa}}(\boldsymbol{x})$ and $\widehat{\ell}_{\boldsymbol{\kappa}}(\boldsymbol{x})$. Concrete (tree-based) SPE techniques for computing these sets in a very efficient way will be discussed in Section 5.

4.1. Search-based threshold optimization (STO)

An exact solution of (3) on a finite set $\mathcal{D}_n$ calls for computing all CPEs for all labels and, moreover, the F-score for all CPEs as possible thresholds. This can be implemented by sorting the CPEs for each label first, and finding the threshold that yields the highest F-measure with a single pass over the sorted list afterward. Since $n \cdot m$ posterior estimates are needed, and sorting requires time $\mathcal{O}(n \log n)$, the overall computational complexity of this procedure is $\mathcal{O}(mn \log n)$.

To reduce the cost, we can solve (3) by using SPEs. We first compute and store the positive labels for which the posterior estimates are above fixed thresholds $\boldsymbol{\kappa}$, which is an input parameter of Algorithm 1. Next, the algorithm seeks to find the optimal threshold for each label j based on an exhaustive search by computing the F-score for each $\widehat{\eta}(\boldsymbol{x}_i, j)$ as a possible threshold. The cost of this search is much lower now, as it only needs to sort the SPE values—provided κ_j is not too close to 0, these should be much less than n.

Needless to say, the SPE thresholds κ_j have to be chosen carefully and definitely smaller than the optimal thresholds in (3); otherwise, the latter cannot be found anymore. On the other side, the larger κ_j, the more efficient the search procedure will be. So what is a reasonable range of values for an optimal threshold τ^*, and hence for κ_j?

To answer this question, we derive (theoretical) bounds for a threshold. Interestingly, the optimal solution on a population level (i.e., all probabilities are known) satisfies the following condition (Zhao et al., 2013):

$$F^* = F(\tau^*) = 2\tau^* \tag{4}$$

Algorithm 1 STO($\mathcal{D}_n, \widehat{\eta}, \boldsymbol{\kappa}$)

1: **for** $j = 1 \to m$ **do**
2: $\quad I_j = \emptyset$
3: **for** $i = 1 \to n$ **do**
4: $\quad$ Compute $\widehat{s}_\kappa(\boldsymbol{x}_i)$ and $\widehat{\ell}_\kappa(\boldsymbol{x}_i)$ ▷ SPE
5: $\quad$ **for each** $j \in \widehat{\ell}_\tau(\boldsymbol{x}_i)$ **do**
6: $\quad\quad I_j = I_j \cup \{i\}$
7: $\quad\quad p_{ij} = \widehat{\eta}(\boldsymbol{x}_1, j) (\in \widehat{s}_\kappa(\boldsymbol{x}_i))$
8: **for** $j = 1 \to m$ **do**
9: $\quad$ Find τ_j based on $\{p_{ij} : i \in I_j\}$ and $\{y_{i,j} : i \in I_j\}$ using sorting
10: **return** $\boldsymbol{\tau}$

That is, the optimal F-measure is twice the value of the optimal threshold. As an immediate consequence, the upper bound of the threshold is 0.5. In order to derive a lower bound, note that a classifier which always predicts positive (i.e., which uses a threshold $\tau = 0$) yields an F-measure of $F = 2\pi/(\pi + 1)$, where π is the (prior) probability of the label being positive. Since $\pi > 0$ can reasonably be assumed, $F > \tau = 0$, and (4) is violated. Therefore, F must be smaller than the optimal value F^*, and

$$\tau^* = \frac{1}{2}F^* > \frac{1}{2}F = \frac{\pi}{\pi + 1} \ .$$

Our analysis thus suggests that, for each label j, the threshold $\tau^* = \tau_j^*$ should be chosen from the range

$$(\pi_j/(\pi_j + 1), 0.5] \ . \tag{5}$$

Remark that this result assumes the posteriors $\eta(\boldsymbol{x}, j)$ (or at least perfect estimates $\hat{\eta}(\boldsymbol{x}, j)$ thereof) to be given.

4.2. Fixed thresholds approach (FTA)

Following the EUM principle, STO optimally tunes the threshold for each label on the data $\mathcal{D}_n$. By doing so, it is of course prone to overfitting. Therefore, one can consider a different implementation of line 9 in Algorithm 1. FTA follows a simple modification, in which a predefined set of possible threshold values within the range (5) is tried. This approach can be simplified even further, by using the same threshold for all labels, which turns out to produce more stable results. As a side remark, we note that a common threshold for all labels is provably optimal for micro-averaged and instance-averaged F-measures (Koyejo et al., 2015).

Implementing FTA like Algorithm 1 requires computing and storing all SPEs for a validation set and checking the F-measure for each predefined threshold. Alternatively, one can compute the F-measure for each threshold simultaneously by passing the validation set only once. In that case, SPEs do not need to be stored, but auxiliary variables for each of the predefined thresholds have to be kept.

4.3. Online F-measure optimization (OFO)

The OFO algorithm by Busa-Fekete et al. (2015) optimizes the binary F-measure by tuning the threshold in an online fashion. Assuming instances to be observed sequentially, the algorithm seeks to maximize the so-called *online F-measure*, which is defined for label j as

$$F_{i,j} = \frac{2\sum_{\ell=1}^{i} y_{\ell,j}\widehat{y}_{\ell,j}}{\sum_{\ell=1}^{i} y_{\ell,j} + \sum_{\ell=1}^{i} \widehat{y}_{\ell,i}} = \frac{2a_{i,j}}{b_{i,j}} \ , \qquad (6)$$

where we concisely write

$$a_{i,j} = \sum_{\ell=1}^{i} y_{\ell,j}\widehat{y}_{\ell,j}, \quad b_{i,j} = \sum_{\ell=1}^{i} y_{\ell,j} + \sum_{\ell=1}^{i} \widehat{y}_{\ell,j} \ . \qquad (7)$$

The OFO algorithm simply sets the threshold to

$$\tau_{i,j} = \frac{a_{i-1,j}}{b_{i-1,j}} \ , \qquad (8)$$

when processing the ith instance, and thus makes predictions $\widehat{y}_{i,j} = [\![\widehat{\eta}(\boldsymbol{x}_i, j) > \tau_j]\!]$. The threshold update is motivated by condition (4). Moreover, Busa-Fekete et al. (2015) have proven the consistency of this approach: The threshold and the online F-score computed by OFO converge in probability to the optimal F-score and threshold, respectively, as n goes to infinity, provided the posterior estimates are coming from an L_1 consistent classifier.

Thanks to its online nature, OFO can be readily adapted to a large-scale learning regime. In contrast to STO, it allows the data to be processed in a sequential manner, without the need to store any predicted scores or labels from previous iterations. The pseudo-code of the algorithm is shown in Algorithm 2. First, the initial threshold is set based on $\mathbf{a} = (a_1, \ldots, a_m)$ and $\mathbf{b} = (b_1, \ldots, b_m)$, which are input parameters (the impact of which will be investigated in our experimental study). Then, in each iteration, the set of predicted positive labels $\widehat{\ell}(\boldsymbol{x})$ is computed (line 4), and the thresholds are updated according to (8) for all labels in $\ell(\boldsymbol{x}_i) \cup \widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)$, where $\ell(\boldsymbol{x})$ is the set of positive labels for $\boldsymbol{x}$. Note that this update can be implemented in an efficient way, since, according to the definition of $a_{i,j}$ and $b_{i,j}$, if label j and the predicted label are both negative, then neither $a_{i,j}$ nor $b_{i,j}$ need to be updated. Finally, the memory requirement of the algorithm is $\mathcal{O}(m)$, since only two auxiliary arrays ($\mathbf{a}$ and $\mathbf{b}$) and the array of thresholds need to be stored. The computational complexity is $\mathcal{O}(nm')$, where $m' = \sum_{i=1}^{n} |\ell(\boldsymbol{x}_i) \cup \widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)|$, which can be orders of magnitude smaller than m if both the labels and the predictions are sparse. In general, OFO can be either applied on a validation set or run simultaneously with training of the class probability model. For large validation sets, a single pass over the data is enough to obtain an accurate estimate of the threshold.

Algorithm 2 OFO$(\mathcal{D}_n, \widehat{\eta}, \mathbf{a}, \mathbf{b})$

1: **for** $i = 1 \to m$ **do**
2: Set $\tau_i = a_i / b_i$ ▷ Initial value
3: **for** $i = 1 \to n$ **do**
4: Compute $\widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)$ ▷ Predicted positives
5: **for all** $j \in \ell(\boldsymbol{x}_i) \cup \widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)$ **do**
6: $a_j = a_j + [\![j \in \ell(\boldsymbol{x}_i) \cap \widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)]\!]$
7: $b_j = b_j + [\![j \in \ell(\boldsymbol{x}_i)]\!] + [\![j \in \widehat{\ell}_{\boldsymbol{\tau}}(\boldsymbol{x}_i)]\!]$
8: $\tau_j = \frac{a_j}{b_j}$
9: **return** $\boldsymbol{\tau}$

5. Efficient sparse probability estimators

In this section, we discuss extreme classification algorithms for computing sparse probability estimates. We mainly focus on Probabilistic Label Trees (PLTs), which ideally fit the idea of sparse probability estimation. Later, we recall FastXML (Prabhu & Varma, 2014) and explain why this algorithm can be treated as an efficient sparse probability estimator, too. We also discuss other large scale algorithms that do not deliver sparse probability estimates in an efficient way.

5.1. Probabilistic label trees

PLTs share similarities with conditional probability estimation trees (Beygelzimer et al., 2009a) and probabilistic classifier chains (Dembczyński et al., 2010), while being suited to estimate marginal posterior probabilities $\eta(\boldsymbol{x}, j)$. They are also similar to Homer (Tsoumakas et al., 2008), which transforms training examples in the same way but does not admit a probabilistic interpretation. Let us also remark that a similar concept is known in neural networks and natural language processing under the name of hierarchical softmax classifiers (Morin & Bengio, 2005); however, it has not been used in a multi-label scenario.

In a nutshell, PLTs are based on the label tree approach (Beygelzimer et al., 2009b; Bengio et al., 2010; Deng et al., 2011), in which each leaf node corresponds to one label. Classification of a test example relies on a sequence of decisions made by node classifiers, leading the test example from the root to the leaves of the tree. Since PLTs are designed for multi-label classification, each internal node classifier decides whether or not to continue the path by moving to the child nodes. This is different from typical left/right decisions made in tree-based classifiers. Moreover, a leaf node classifier needs to make a final decision regarding the prediction of a label associated to this leaf. PLTs use a class probability estimator in each node of the tree, such that an estimate of the posterior probability of a label associated with a leaf is given by the product of the probability estimates on the path from the root to that leaf. Prediction then relies on traversing the tree from the root to the leaf nodes. Whenever

the intermediate value of the product of probabilities in an internal node is less than a given threshold, the subtree below this node is not explored anymore. This pruning strategy leads to a very efficient classification procedure.

5.2. Formal description of PLTs

To introduce PLTs more formally, denote a tree by T and the root of the tree $r(T)$. In general, T can be of any form; here, we consider trees of height k and degree b. The leaves of T correspond to labels. We denote a set of leaves of a (sub)tree rooted in node t by $L(t)$. The parent node of t is denoted by $\mathrm{pa}(t)$, and the set of child nodes by $\mathrm{Ch}(t)$. The path from the root $r(T)$ to the jth leaf is denoted by $\mathrm{Path}(j)$.

PLTs use a path from a root to the jth leaf to estimate posteriors $\eta(\boldsymbol{x}, j)$. With each node t and training instance $\boldsymbol{x}$, we associate a label $z_t = [\![\sum_{j \in L(t)} y_j \geq 1]\!]$. In the leaf nodes, the labels z_j, $j \in L$, correspond to the original labels y_j.

Consider the leaf node j and the path from the root to this leaf node. Using the chain rule of probability, we can express $\eta(\boldsymbol{x}, j)$ in the following way:

$$\eta(\boldsymbol{x}, j) = \prod_{t \in \mathrm{Path}(j)} \eta_T(\boldsymbol{x}, t)\,, \tag{9}$$

where $\eta_T(\boldsymbol{x}, t) = \mathbf{P}(z_t = 1 \mid z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})$ for all non-root nodes t, and $\eta_T(\boldsymbol{x}, t) = \mathbf{P}(z_t = 1 \mid \boldsymbol{x})$ if t is the root node. The correctness of (9) follows from the observation that $z_t = 1$ implies $z_{\mathrm{pa}(t)} = 1$. A detailed derivation of the chain rule in this setup is given in Appendix A.

The training algorithm for PLTs is given in Algorithm 3. Let $\mathcal{D}_n = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ be a training set of multi-label examples. To learn classifiers in all nodes of a tree T, we need to properly filter training examples to estimate $\eta_T(\boldsymbol{x}, t)$ (line 5). Moreover, we need to use a learning algorithm A that trains a class probability estimator $\hat{\eta}_T(\boldsymbol{x}, t)$ for each node t in the tree. The training algorithm returns a set of probability estimation classifiers $\mathcal{Q}$.

The learning time complexity of PLTs can be expressed in terms of the number of nodes in which an original training example $(\boldsymbol{x}, \boldsymbol{y})$ is used. Since the training example is used in a node t only if t is the root or $z_{\mathrm{pa}(t)} = 1$, this number is upper bounded by $s \cdot b \cdot k + 1$, where b and k denote the degree and height of the tree, respectively, and s is the number of positive labels in $\boldsymbol{y}$. For sparse labels, this value is much lower than m. Note that learning can be performed simultaneously for all nodes, and each node classifier can be trained using online methods, such as stochastic gradient descent (Bottou, 2010).

Interestingly, in the case of sparse features, the space complexity can be significantly reduced as well. Admittedly, the number of models is the highest for binary trees and can be as high as $2m - 1$ (notice that the size of a tree with m leaves is upper bounded by $2m - 1$). This is twice the number of models in the simplest 1-vs-all approach. Paradoxically, the space complexity can be the lowest at this upper bound. This is because only the sibling nodes need to share the same features, while no other features are needed to build corresponding classifiers. Therefore, only those (few) features needed to describe the sibling labels have to be used in the models. If the space complexity still exceeds the available memory, one can always use feature hashing over all nodes (Weinberger et al., 2009).

Algorithm 3 PLT.TRAIN($T, A, \mathcal{D}_n$)

1: $\mathcal{Q} = \emptyset$
2: **for** each node $t \in T$ **do**
3: $\quad \mathcal{D}' = \emptyset$
4: $\quad$ **for** $i = 1 \to n$ **do**
5: $\quad\quad$ **if** t is **root** or $z_{\mathrm{pa}(t)} = 1$ **then**
6: $\quad\quad\quad z_t = [\![\sum_{j \in L(t)} y_{ij} \geq 1]\!]$
7: $\quad\quad\quad \mathcal{D}' = \mathcal{D}' \cup (\boldsymbol{x}_i, z_t)$
8: $\quad \hat{\eta}_T(\boldsymbol{x}, t) = A(\mathcal{D}')$, $\mathcal{Q} = \mathcal{Q} \cup \hat{\eta}_T(\boldsymbol{x}, t)$
9: **return** a set of probability estimation classifiers $\mathcal{Q}$.

Algorithm 4 PLT.PREDICT($\boldsymbol{x}, T, \mathcal{Q}, \tau$)

1: $\hat{\boldsymbol{y}} = \mathbf{0}_m$, $Q = \emptyset$, $Q.\mathrm{add}(r(T), \hat{\eta}_T(\boldsymbol{x}, r(T)))$
2: **while** $Q \neq \emptyset$ **do**
3: $\quad (t, p_t) = \mathrm{pop}(Q)$
4: $\quad$ **if** $p_t \geq \tau$ **then**
5: $\quad\quad$ **if** t is a leaf node **then**
6: $\quad\quad\quad \hat{y}_t = 1$
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ **for** $c \in \mathrm{Ch}(t)$ **do**
9: $\quad\quad\quad\quad Q.\mathrm{add}(c, p_t \cdot \hat{\eta}_T(\boldsymbol{x}, c))$
10: **return** $\hat{\boldsymbol{y}}$.

Prediction with probabilistic label trees relies on estimating (9) by traversing the tree from the root to the leaf nodes. However, if the intermediate value of this product in node t, denoted by p_t, is less than a given threshold τ, then the subtree starting in node t is no longer explored. For the sake of completeness, we shortly describe this procedure (see Algorithm 4). We start with setting $\hat{\boldsymbol{y}} = \mathbf{0}_m$. In order to traverse a tree, we initialize a queue Q to which we add the root node r_T with its posterior estimate $\hat{\eta}_T(\boldsymbol{x}, r(T))$. In the while loop, we iteratively pop a node from Q and compute p_t. If $p_t \geq \tau$, we either set $\hat{y}_j = 1$ if t is a leaf, or otherwise add its child nodes to Q with the value p_t multiplied by their posterior estimates $\hat{\eta}_T(\boldsymbol{x}, c)$, $c \in \mathrm{Ch}(t)$. If Q is empty, we stop the search and return $\hat{\boldsymbol{y}}$.

Traversing a label tree can be much cheaper than querying m independent classifiers, one for each label. If there is only one label exceeding the threshold, PLT ideally needs to call only $bk + 1$ classifiers (all classifiers on a path from

the root to a leaf plus all classifiers in siblings of the path nodes). Of course, in the worst-case scenario, the entire tree might be explored, but even then, no more than $2m - 1$ calls are required (with m leaves, the size of the tree is upper bounded by $2m - 1$). In the case of sparse label sets, PLTs can significantly speed up the classification procedure. The expected cost of prediction depends, however, on the tree structure and accuracy of node classifiers.

Note that PLTs can be used with any value as a threshold. Moreover, by considering a separate threshold τ_t in each node t, one can use a different threshold for each label j on posterior estimates $\hat{\eta}(\boldsymbol{x}, j)$. It is enough to set a threshold in the parent node t as follows: $\tau_t = \min_{j \in \mathrm{Ch}(t)} \tau_j$. In this way, PLTs can efficiently obtain SPEs for any $\boldsymbol{\kappa}$ in STO and FTA (Algorithm 1), and any $\boldsymbol{\tau}$ in OFO (Algorithm 2). PLTs can easily be tailored for Precision@K, too. To predict the top labels, it is enough to change Q to a priority queue and stop the prediction procedure after a given number of top labels.

Let us finally underline that PLTs obey strong theoretical guarantees. In Appendix A, we derive a surrogate regret bound showing that the overall error of PLTs is reduced by improving the node classifiers. For optimal node classifiers, we obtain optimal multi-label classifiers in terms of estimation of posterior probabilities $\eta(\boldsymbol{x}, j)$.

5.3. FastXML

As an example of another efficient sparse probability estimator, we recall FastXML, which adapts the idea of standard decision trees (Breiman et al., 1984) to the XMLC setting. To improve predictive performance, FastXML uses an ensemble of decision trees. Internal nodes in the trees contain sparse linear classifiers, which are trained to optimize an nDCG-based ranking loss. The authors show that this optimization can be performed very efficiently. The most costly step is L_1-regularized logistic regression solved by the newGLMNET algorithm (Yuan et al., 2012) implemented in the LibLinear package (Fan et al., 2008). The height of trees in FastXML scales logarithmically with the number of training examples (though it should be noted that different stopping rules for growing a tree can be used).

Sparse predictions of class probabilities, a major prerequistie for efficient macro F-measure maximization, are produced by FastXML in a quite natural way. This is because, during training, the nodes are recursively partitioned till each leaf contains only a small number of training examples. Thus, only a small number of active labels is assigned to each leaf. During prediction, a test example passes down the tree until it reaches a leaf node. The prediction procedure can then focus exclusively on the label distribution in the leaf node. FastXML uses a standard prediction procedure for decision trees that relies on the relative frequencies of labels in the leaf node reached by the test example. These relative frequencies can be treated as CPEs. Since FastXML is an ensemble, the leaf node label distributions are averaged over all trees in the ensemble. Eventually, CPEs are thus only delivered for a small number of labels for a given test example. For the remaining labels, the probabilities are assumed to be 0. By using thresholds $\boldsymbol{\kappa}$ and $\boldsymbol{\tau}$ in STO, FTA, and OFO, respectively, the CPEs returned by FastXML can be treated as SPEs.

5.4. PLT vs. FastXML

Both FastXML and PLTs achieve fast prediction time thanks to exploiting a tree-structure. A PLT is a single label tree with linear classifiers in each node, while FastXML makes use of an ensemble of regular trees with linear splits. Notice that the height of a PLT scales logarithmically with the number of labels, while the size of a tree in FastXML is logarithmic in the number of training examples. Therefore, depending on the problem, the trees may differ in size.

For a single tree, FastXML follows only a single path from the root to a leaf. Each leaf node corresponds to a region in the feature space and returns a label distribution for this region. By assuming that, in a certain region, only a small number of labels have non-zero conditional probability, FastXML is able to efficiently deliver SPEs. PLTs in turn may explore several paths from the root to the leaves. By using a threshold on probability estimates, each internal node decides whether to go down the tree or stop exploration. Therefore, PLTs can efficiently deliver all labels the CPEs of which exceed given thresholds.

Another difference is that, while FastXML is training the structure of the tree, PLT assumes a predefined structure (or uses a random tree). Of course, just like conditional probability trees (Beygelzimer et al., 2009a), PLTs could in principle be combined with an online tree learning procedure. Let us remark, however, that a predefined tree structure allows for training the node classifiers independently of each other, thereby making the whole training process amenable to massive parallelization.

In the XMLC setting, only a few methods suggest themselves for a combination with threshold tuning methods, such as STO, FTA and OFO. It seems that PLTs and FastXML are among the best options. Some algorithms are also able to deliver SPEs, but not always efficiently. For example, the 1-vs-all approach scales linearly with m, which is too expensive in many applications. Even methods that rely on fast training, like negative sampling (Collobert & Weston, 2008) or fast label embeddings (Mineiro & Karampatziakis, 2015), still exhibit linear cost for prediction. To deliver sparse probability estimates efficiently, additional structures, such as trees (Bengio et al., 2010; Weston et al., 2013) or locality sensitive hashing (Shrivastava & Li, 2014), are required.

6. Experiments

We carried out two sets of experiments. In the first, we verify the effectiveness of PLTs in handling a large number of labels by comparing its performance to that of FASTXML in terms of Precision@K. In the second experiment, we combine PLTs and FASTXML with the threshold tuning methods, namely with FTA, STO and OFO, for maximizing the macro F-measure. In both experiments we used six large-scale datasets taken from the Extreme Classification Repository[2] with predefined train/test splits (see main statistics of these datasets in Table 1).

To perform the experiments we implemented PLTs in Java.[3] We used random complete b-ary trees (where b is a hyperparameter) with L_2-logistic regression as a node classifier, trained by a variant of stochastic gradient descent introduced by Duchi & Singer (2009) (see Appendix B for details). To deal with a large number of weights, we used feature hashing shared over all tree nodes with 2^{30} hash values. In the first experiment, we tuned the hyperparameters of PLTs in a simple 80/20 validation on the training set. We applied the off-the-shelf hyperparameter optimizer SMAC (Hutter et al., 2011) with a wide range of parameters, reported in Appendix C. In the second experiment, we used the values of the hyperparameters found to be optimal in the first experiment. For FASTXML, we used the C++ code delivered by Prabhu & Varma (2014), as well as the hyperparameters suggested by them.

6.1. Sparse probability estimators

In the first experiment, we evaluate the algorithms in terms of Precision@K (for $K = 1, 3, 5$) and computational cost. The results are shown in Table 1. The performance of PLTs is on a par with that of FASTXML, getting better results on half of the datasets. It also seems that PLTs are more efficient in handling very sparse labels, because their training time is typically smaller for those datasets where the number of labels per instance is small (i.e., < 6). This can be explained by the fact that only a very small part of the label tree is updated in case of only a few positive labels (see Algorithm 4). Notice, however, that the comparison is not completely fair as both algorithms are implemented using different technologies and coding styles. In the next subsection, we more carefully analyze test time complexity, focusing on the number of computed inner products per test example.

6.2. Extreme macro F-measure optimization

In the second experiment, we test the threshold tuning algorithms described in Section 4. We use 80% of each dataset for training PLTs and FASTXML, and then run FTA, STO and OFO on the remaining 20%. The latter part of the training set is also used to validate the input parameters of the threshold tuning algorithms. For the vector $\boldsymbol{\kappa}$, the input parameter of STO and FTA, we first compute the lower bound ν_j of the optimal threshold according to (5), i.e., $\nu_j = \widehat{\pi}_j/(\widehat{\pi}_j + 1)$, with $\widehat{\pi}_j$ the prior probability estimate for label j. Then, each element of $\boldsymbol{\kappa}$ is set to $\max(1/c, \nu_j)$, where $c \in C = \{10000, 1000, 200, 100, 50, 20, 10, 7, 5, 4, 3, 2\}$. Similarly, the input parameter b of OFO is tuned over the same set C, while its other input parameter a is constantly set to 1. We additionally carried out experiments for assessing the impact of parameter a (see results in Appendix D), which slightly improves the results. We also control the thresholds in OFO to be greater than the lower bound ν_j.

In Figure 1, the validation and test macro F-scores of PLTs and FASTXML are plotted against various input parameter values in the form of $1/c$. As one would expect, STO outperforms both FTA and OFO in terms of F-score on the validation set for high values of c (see the dashed lines on the plots). This is not surprising, since as κ_j goes to 0, the probability estimates are getting less sparse. In particular, for $\kappa_j = 0$, STO finds the optimal threshold on the validation set. Therefore, $\boldsymbol{\kappa}$ can be used to trade computational complexity off against approximation quality.

More surprisingly, the difference between the validation and test performance of STO is substantial. This can be explained by the fact that only 20% of the training data was used for tuning the thresholds, and this data is not guaranteed to contain positive instances for every label. In case of no positive examples for a label and no predicted positives either the F-score is set to 1 as suggested by Lewis (1995). In turn, FTA and OFO seem to be more efficient in handling sparse probability estimate and there is no such difference in performance on validation and test sets.

The optimal parameters of the threshold tuning algorithms were chosen based on the validation performance (dashed lines in Figure 1). The corresponding test scores are shown in Table 2. Based on the results, STO is clearly outperformed by either FTA or OFO. We explain the poor performance of sorting-based optimization by its susceptibility to overfitting for sparse classes, whereas FTA and especially OFO seem to be more robust in the extreme learning regime.

In extreme classification, the computational requirements for evaluating the model on test examples are not less important than the predictive performance of the model. Therefore, we also study the average (per example) wall-clock test times and number of inner products computed by the trained models. We believe that the number of inner products is a more objective measure of the computational complexity, because PLTs and FASTXML can be viewed as an ensemble of linear classifiers, whose main computational effort

[2] http://research.microsoft.com/en-us/um/people/manik/downloads/XC/XMLRepository.html

[3] Get code at https://github.com/busarobi/XMLC.

Table 1. Main characteristics of the datasets used in the experiment and Precision@K scores with $K = \{1, 3, 5\}$ for PLTs and FASTXML along with their training time in minutes. The highest Precision@K values are in bold for every dataset.

	Main statistics of datasets						PLT				FASTXML			
	#labels	#features	#test	#train	inst./lab.	lab./inst.	P@1	P@3	P@5	Time	P@1	P@3	P@5	Time
RCV1	2456	47236	155962	623847	1218.56	4.79	90.46	72.4	51.86	64	**91.13**	**73.35**	**52.67**	78
AmazonCat	13330	203882	306782	1186239	448.57	5.04	91.47	75.84	61.02	96	**92.95**	**77.50**	**62.51**	561
Wiki10	30938	101938	6616	14146	8.52	18.64	**84.34**	**72.34**	**62.72**	290	81.71	66.67	56.70	16
Delicious	205443	782585	100095	196606	72.29	75.54	**45.37**	**38.94**	35.88	1327	42.81	38.76	**36.34**	458
WikiLSHTC	325056	1617899	587084	1778351	17.46	3.19	45.67	29.13	21.95	653	**49.35**	**32.69**	**24.03**	724
Amazon	670091	135909	153025	490449	3.99	5.45	**36.65**	**32.12**	**28.85**	54	34.24	29.3	26.12	422

Table 2. The macro F-measure on the test set for the best hyperparameter values obtained on a validation set and the average (per example) wall-clock test time (in milliseconds) and number of inner products computed by PLTs and FASTXML on the test sets. The numbers in bold indicate the best macro F-score achieved on each dataset.

	Macro F-score on test set						Test time in millisec.				Num. of inner products			
	PLT			FASTXML			PLT			FASTXML	PLT			FASTXML
Dataset	FTA	STO	OFO	FTA	STO	OFO	FTA	STO	OFO	–	FTA	STO	OFO	–
RCV1	20.41	21.16	**21.56**	17.04	19.58	18.93	0.33	0.78	0.19	0.96	252	354	212	747
AmazonCat	34.83	31.64	33.13	41.07	37.28	**42.46**	0.37	2.23	0.85	1.14	225	2272	711	871
Wiki10	29.98	24.02	**30.28**	29.88	28.26	29.51	1.69	2.18	1.87	3.00	534	13682	2518	541
Delicious-200K	11.12	10.96	**11.20**	11.18	10.83	11.19	0.61	17.41	4.11	3.86	216	29067	5610	739
WikiLSHTC	12.31	16.22	14.00	**21.24**	20.41	20.84	0.22	13.46	4.54	1.17	175	28448	2875	900
Amazon	51.77	46.94	51.28	**52.86**	47.53	50.44	0.29	5.54	0.54	1.39	132	5557	125	796

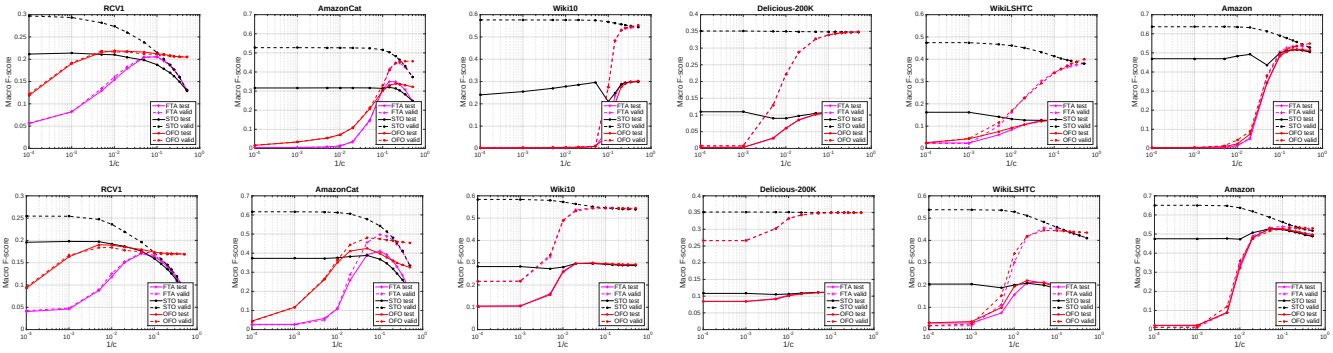

Figure 1. The macro F-measure obtained by STO, FTA and OFO with PLTs (top) and FASTXML (bottom). The dashed and solid lines represent the performance of the methods on the validation and test set for various initial parameters, respectively.

consists of computing inner products.

The average per test example wall-clock time and number of inner products are shown in Table 2. We used the set of thresholds that were found to be the best in the validation process described above. Remark that for FASTXML the number of inner products does not depend on a tunning method (i.e., for each method only one path from the root to a leaf node is visited). The results reveal some general trends. First, PLTs along with the thresholds found by FTA have the smallest computation time except on RCV1. Second, PLTs with thresholds found by OFO outperforms FASTXML on three datasets. Third, the computational time for STO is the highest in almost every case; since its performance is usually lower than that of OFO and FTA, this suggests that the thresholds found by STO are in general underestimated. Remark that FASTXML uses 50 trees, while in PLTs a single tree is explored. Moreover, the height of a tree in FASTXML scales with the number of training examples. Therefore, a FASTXML tree can in some cases be larger than a PLT.

7. Conclusion

We addressed the problem of macro-F measure maximization in XMLC and combined three tuning methods (STO, FTA, OFO) with two efficient sparse probability estimators (PLTs and FASTXML). It seems that online F-measure optimization (OFO) used with either of the two classifiers yields the most promising results. As a next step, we plan to generalize the results presented here to other complex performance measures and work further on different types of sparse probability estimators.

Acknowledgments

Kalina Jasinska and Krzysztof Dembczyński are supported by the Polish National Science Centre under grant no. 2013/09/D/ST6/03917.

References

Agarwal, S. Surrogate regret bounds for bipartite ranking via strongly proper losses. *JMLR*, 15:1653–1674, 2014.

Agrawal, R., Gupta, A., Prabhu, Y., and Varma, M. Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages. In *WWW*, pp. 13–24. ACM, 2013.

Bengio, S., Weston, J., and Grangier, D. Label embedding trees for large multi-class tasks. In *NIPS 24*, pp. 163–171, 2010.

Beygelzimer, A., Langford, J., Lifshits, Y., Sorkin, G., and Strehl, A. Conditional probability tree estimation analysis and algorithms. In *UAI*, pp. 51–58, 2009a.

Beygelzimer, A., Langford, J., and Ravikumar, P. Error-correcting tournaments. In *ALT*, pp. 247–262. Springer, 2009b.

Bhatia, K., Jain, H., Kar, P., Varma, M., and Jain, P. Sparse local embeddings for extreme multi-label classification. In *NIPS 29*, pp. 730–738, 2015.

Bottou, L. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, pp. 177–187. Springer, 2010.

Bottou, L. Stochastic gradient tricks. In *Neural Networks, Tricks of the Trade, Reloaded*, pp. 430–445. Springer, 2012.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

Busa-Fekete, R., Szörényi, B., Dembczyński, K., and Hüllermeier, E. Online F-measure optimization. In *NIPS 29*, pp. 595–603, 2015.

Choromanska, A. and Langford, J. Logarithmic time online multiclass prediction. In *NIPS 29*, pp. 55–63, 2015.

Cisse, M., Usunier, N., Artières, T., and Gallinari, P. Robust Bloom filters for large multilabel classification tasks. In *NIPS 26*, pp. 1851–1859, 2013.

Collobert, R. and Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, pp. 160–167, 2008.

Dekel, O. and Shamir, O. Multiclass-multilabel classification with more classes than examples. In *AISTATS*, pp. 137–144, 2010.

Dembczyński, K., Cheng, W., and Hüllermeier, E. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, pp. 279–286, 2010.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *CVPR*, pp. 248—255, 2009.

Deng, J., Satheesh, S., Berg, A. C., and Fei-Fei, L. Fast and balanced: Efficient label tree learning for large scale object recognition. In *NIPS 24*, pp. 567–575, 2011.

Duchi, J. and Singer, Y. Efficient online and batch learning using forward backward splitting. *JMLR*, 10:2899–2934, 2009.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874, 2008.

Hutter, F., Hoos, H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Kotłowski, W. and Dembczynski, K. Surrogate regret bounds for generalized classification performance metrics. In *ACML*, 2015.

Koyejo, S., Natarajan, N., Ravikumar, P., and Dhillon, I. Consistent binary classification with generalized performance metrics. In *NIPS 27*, pp. 2744–2752, 2014.

Koyejo, S., Natarajan, N., Ravikumar, P., and Dhillon, I. Consistent multilabel classification. In *NIPS 29*, pp. 3321–3329, 2015.

Lewis, D. Evaluating and optimizing autonomous text classification systems. In *SIGIR*, pp. 246–254, 1995.

Mineiro, P. and Karampatziakis, N. Fast label embeddings via randomized linear algebra. In *ECML/PKDD 2015*, pp. 37–51, 2015.

Morin, F. and Bengio, Y. Hierarchical probabilistic neural network language model. In *AISTATS*, pp. 246–252, 2005.

Narasimhan, H., Vaish, R., and S., Agarwal. On the statistical consistency of plug-in classifiers for non-decomposable performance measures. In *NIPS 27*, pp. 1493–1501, 2014.

Partalas, I., Kosmopoulos, A., Baskiotis, N., Artières, T., Paliouras, G., Gaussier, É., Androutsopoulos, I., Amini, M.-R., and Gallinari, P. LSHTC: A benchmark for large-scale text classification. *CoRR*, 2015.

Prabhu, Y. and Varma, M. FastXML: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *KDD*, pp. 263–272. ACM, 2014.

Shrivastava, A. and Li, P. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NIPS 27*, pp. 2321–2329, 2014.

Tsoumakas, G., Katakis, I., and Vlahavas, I. Effective and efficient multilabel classification in domains with large number of labels. In *Proc. ECML/PKDD 2008 Workshop on Mining Multidimensional Data*, 2008.

Weinberger, K.Q., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. Feature hashing for large scale multitask learning. In *ICML*, pp. 1113–1120. ACM, 2009.

Weston, J., Makadia, A., and Yee, H. Label partitioning for sublinear ranking. In *ICML*, pp. 181–189, 2013.

Ye, N., Chai, A., Lee, W., and Chieu, H. Optimizing F-measures: A tale of two approaches. In *ICML*, 2012.

Yuan, G.-X., Ho, C.-H., and Lin, C.-J. An improved GLMNET for L1-regularized logistic regression. *JMLR*, 13:1999–2030, 2012.

Zhao, M.-J., Edakunni, N., Pocock, A., and Brown, G. Beyond Fano's inequality: Bounds on the Optimal F-Score, BER, and Cost-Sensitive Risk and Their Implications. *JMLR*, 14:1033–1090, 2013.

A. Probabilistic label trees

An example of a probabilistic label tree (PLT) for 4 labels (y_1, y_2, y_3, y_4) is given in Fig. 2. To estimate posterior probability $\eta(\boldsymbol{x}, j) = \mathbf{P}(y_j = 1 \,|\, \boldsymbol{x})$, PLT uses a path from a root to the j-th leaf. In each node t, we associate with a training instance $(\boldsymbol{x}, \boldsymbol{y})$ a label z_t such that:

$$z_t = [\![\textstyle\sum_{j \in L(t)} y_j \geq 1]\!] \quad \text{(or equivalently } z_t = \textstyle\bigvee_{j \in L(t)} y_j \text{)}$$

Recall that $L(t)$ is a set of all leaves of a subtree with the root in the t-th node. In leaf nodes the labels z_j, $j \in L$, correspond to original labels y_j.

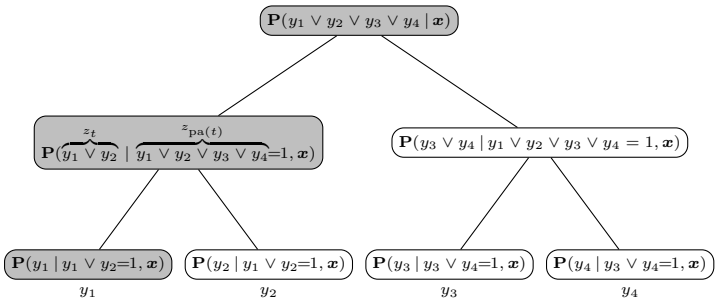

Figure 2. An example of a probabilistic label tree for 4 labels (y_1, y_2, y_3, y_4).

Consider the leaf node j and the path from the root to this leaf node. Using the chain rule of probability, we can express $\eta(\boldsymbol{x}, j)$ in the following way:

$$\eta(\boldsymbol{x}, j) = \prod_{t \in \mathrm{Path}(j)} \eta_T(\boldsymbol{x}, t)\,, \tag{10}$$

where $\eta_T(\boldsymbol{x}, t) = \mathbf{P}(z_t = 1 \,|\, z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})$, for all non-root nodes t, and $\eta(\boldsymbol{x}, t) = \mathbf{P}(z_t = 1 \,|\, \boldsymbol{x})$, if t is the root node (denoted by $r(T)$).

To see the correctness of (10) notice that $z_t = 1$ implies $z_{\mathrm{pa}(t)} = 1$. So, for non-root nodes t and $\mathrm{pa}(t)$ we have:

$$\begin{aligned}
\eta_T(\boldsymbol{x}, t)\eta_T(\boldsymbol{x}, \mathrm{pa}(t)) &= \mathbf{P}(z_t = 1 \,|\, z_{\mathrm{pa}(t)} = 1, \boldsymbol{x}) \mathbf{P}(z_{\mathrm{pa}(t)} = 1 \,|\, z_{\mathrm{pa}(\mathrm{pa}(t))} = 1, \boldsymbol{x}) \\
&= \frac{\mathbf{P}(z_t = 1, z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})}{\mathbf{P}(z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})} \frac{\mathbf{P}(z_{\mathrm{pa}(t)} = 1, z_{\mathrm{pa}(\mathrm{pa}(t))} = 1, \boldsymbol{x})}{\mathbf{P}(z_{\mathrm{pa}(\mathrm{pa}(t))} = 1, \boldsymbol{x})} \\
&= \frac{\mathbf{P}(z_t = 1, \boldsymbol{x})}{\mathbf{P}(z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})} \frac{\mathbf{P}(z_{\mathrm{pa}(t)} = 1, \boldsymbol{x})}{\mathbf{P}(z_{\mathrm{pa}(\mathrm{pa}(t))} = 1, \boldsymbol{x})} \\
&= \frac{\mathbf{P}(z_t = 1, \boldsymbol{x})}{\mathbf{P}(z_{\mathrm{pa}(\mathrm{pa}(t))} = 1, \boldsymbol{x})}\,.
\end{aligned}$$

In words, the probability associated with the parent node $\mathrm{pa}(t)$ cancels out and we can express $\eta_T(\boldsymbol{x}, t)\eta_T(\boldsymbol{x}, \mathrm{pa}(t))$ as the product of probabilities associated only with node t and its grandparent node $\mathrm{pa}(\mathrm{pa}(t))$. Applying the above rule consecutively to $\prod_{t \in \mathrm{Path}(j)} \eta_T(\boldsymbol{x}, t)$ and recalling that for the root note $\eta_T(\boldsymbol{x}, r(T)) = \mathbf{P}(z_{r(T)} = 1 \,|\, \boldsymbol{x})$, we finally get $\eta(\boldsymbol{x}, j)$.

Below we show that PLTs posses strong theoretical guarantees. We derive a relation between estimation error minimized by the node classifiers and estimation error of posterior probabilities $\eta(\boldsymbol{x}, j)$. This relation states that we can upperbound the latter error by the former. This also implies that for optimal node classifiers we get optimal multi-label classifier in terms of estimation of posterior probabilities.

We are interested in bounding the estimation error of posterior probabilities of labels at point $\boldsymbol{x}$

$$\ell(\eta(\boldsymbol{x}), \hat{\eta}(\boldsymbol{x})) = \frac{1}{m} \sum_{j=1}^{m} |\eta(\boldsymbol{x}, j) - \hat{\eta}(\boldsymbol{x}, j)|\,,$$

in terms of an estimation error of node classifiers

$$\ell(\eta_T(\boldsymbol{x}, t), \hat{\eta}_T(\boldsymbol{x}, t)) = |\eta_T(\boldsymbol{x}, t) - \hat{\eta}_T(\boldsymbol{x}, t)|\,.$$

Expressing $\eta(\boldsymbol{x}, j)$ and $\hat{\eta}(\boldsymbol{x}, j)$ by (10) and applying Lemma 2 from (Beygelzimer et al., 2009a), we get:

$$|\eta(\boldsymbol{x}, j) - \hat{\eta}(\boldsymbol{x}, j)| \leq \sum_{t \in \text{Path}(j)} |\eta_T(\boldsymbol{x}, t) - \hat{\eta}_T(\boldsymbol{x}, t)| \,. \tag{11}$$

Equation (11) already shows that minimization of the estimation error by node classifiers improves the overall performance of PLTs. We can show, however, even a more general result concerning surrogate regret bounds by referring to the theory of strongly proper composite losses (Agarwal, 2014).

Assume that a node classifier has a form of a real-valued function f_t. Moreover, there exists a strictly increasing (and therefore invertible) link function $\psi : [0, 1] \to \mathbb{R}$ such that $f_t(\boldsymbol{x}) = \psi(\hat{\eta}_T(\boldsymbol{x}, t))$. Recall that the regret of f_t in terms of a loss function ℓ at point $\boldsymbol{x}$ is defined as:

$$\text{reg}_\ell(f_t \,|\, \boldsymbol{x}) = L_\ell(f_t \,|\, \boldsymbol{x}) - L_\ell^*(\boldsymbol{x}) \,,$$

where $L_\ell(f_t \,|\, \boldsymbol{x})$ is the expected loss at point $\boldsymbol{x}$:

$$L_\ell(f \,|\, \boldsymbol{x}) = \mathbb{E}_{y_j \,|\, \boldsymbol{x}} \left[\ell(y_j, f_t(\boldsymbol{x}))\right] \,,$$

and $L_\ell^*(\boldsymbol{x})$ is the minimum expected loss at point $\boldsymbol{x}$.

If a node classifier is trained by a learning algorithm that minimizes a strongly proper composite loss, e.g., squared, exponential, or logistic loss, like in our implementation (see in Appendix B), then the bound (11) can be expressed in terms of the regret of this loss function:

$$\left|\eta_T(\boldsymbol{x}, t) - \psi^{-1}(f_t)\right| \leq \sqrt{\frac{2}{\lambda}} \sqrt{\text{reg}_\ell(f_t \,|\, \boldsymbol{x})}$$

where λ is a strong properness constant specific for a given loss function (for more detail, see (Agarwal, 2014)). By putting the above inequality into (11), we get

$$|\eta(\boldsymbol{x}, j) - \hat{\eta}(\boldsymbol{x}, j)| \leq \sum_{t \in \text{Path}(j)} |\eta_T(\boldsymbol{x}, t) - \hat{\eta}_T(\boldsymbol{x}, t)| = \sum_{t \in \text{Path}(j)} \left|\eta_T(\boldsymbol{x}, t) - \psi^{-1}(f_t)\right| \leq \sum_{t \in \text{Path}(j)} \sqrt{\frac{2}{\lambda}} \sqrt{\text{reg}_\ell(f_t \,|\, \boldsymbol{x})}$$

B. Training of node classifiers

In each node t we trained a linear classifier $f_t(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x}$, where $\boldsymbol{x} = (1, x_1, \ldots, x_p)$. To this end we used a variant of stochastic gradient descent to minimize logistic loss. Albeit successfully used in large scale learning, the optimization of empirical loss using stochastic gradient descent is a particularly challenging task when the number of features and labels is large. The step function should ensure a quick convergence in order to reduce the number of required training epochs. Furthermore, it should support sparse updates of the weights (i.e., only weights for non-zero features should be updated to ensure fast training time). Duchi & Singer (2009) propose a two phase gradient step:

$$\begin{aligned} \boldsymbol{w}_{t+\frac{1}{2}} &= \boldsymbol{w}_t - \gamma_t \boldsymbol{g}_t \\ \boldsymbol{w}_{t+1} &= \arg\min_{\boldsymbol{w}} \left\{ \frac{1}{2} \left\| \boldsymbol{w} - \boldsymbol{w}_{t+\frac{1}{2}} \right\|^2 + \lambda \gamma_t r(\boldsymbol{w}) \right\} \end{aligned}$$

where $\boldsymbol{w}_t$ is the weight vector at time step t, $r(\boldsymbol{w})$ is a regularization function, λ is regularization parameter, and γ_t is an adaptive step size, and $\boldsymbol{g}_t$ is the gradient vector at $\boldsymbol{x}_t$ of logistic loss applied to the linear model f_t.

For stochastic gradient descent with L_2^2 regularization, the step function reduces to

$$\boldsymbol{w}_{t+1} = \frac{\boldsymbol{w}_t - \gamma_t \boldsymbol{g}_t}{1 + \lambda \gamma_t}$$

By using

$$\Pi_t = \prod_{i=1}^{t} (1 + \lambda \gamma_t) \quad \text{and} \quad \tilde{\boldsymbol{w}}_t = \Pi_t \boldsymbol{w}_t \,,$$

we can rewrite the step function to the following form:

$$\tilde{\boldsymbol{w}}_{t+1} = \tilde{\boldsymbol{w}}_t - \Pi_t \gamma_t \boldsymbol{g}_t$$

Thanks to this transformation, we are able to make sparse updates by storing only the current value of Π_t (one value for each node classifier). This is because the i-th component of $\tilde{\boldsymbol{w}}$ does not change when x_i is zero. More formally,

$$\tilde{w}_{i,t+1} = \tilde{w}_{i,t}, \text{if } g_{i,t} = 0\,.$$

During prediction or computation of gradient $\boldsymbol{g}_t$, we use:

$$\boldsymbol{w}_t = \frac{\tilde{\boldsymbol{w}}_t}{\Pi_t}\,.$$

In our implementation we adapt the step size γ_t as suggested in (Bottou, 2012):

$$\gamma_t = \frac{\gamma}{1 + \lambda\gamma t}\,,$$

where γ is an initial parameter.

C. Tuning of hyperparameters

A PLT has only one global hyperparameter which is the degree of the tree denoted by b. The other hyperparameters are associated with the node classifiers. To tune the stochastic gradient descent described above we varied values of γ, λ, and the number of epochs. All hyperparameters were tuned by using the open-source hyperparameter optimizer SMAC (Hutter et al., 2011) with a wide range of parameters, which is reported in Table 3. The validation process was carried out by using a $80/20$ split of the training data for every dataset we used.

Table 3. The hyperparameters of the PLT method and their ranges used in hyperparameter optimization,

Hyperparameter	Validation range
b	$\{2, \ldots, 256\}$
λ	$[10 - 0.000001]$
γ	$[10 - 0.000001]$
Num. of epochs	$\{5, \ldots, 30\}$

D. F-scores by tuning the input parameters $\mathbf{a}$ and $\mathbf{b}$ of OFO algorithms

In our experimental study described in Section 6, we did not tune the input parameter $\mathbf{a}$ of the OFO algorithm but set all of its components to 1. We carried out experiments for assessing the impact of the input parameter $\mathbf{a}$ on the performance of OFO. Its optimal value was selected from the set $C = \{10000, 1000, 200, 100, 50, 20, 10, 7, 5, 4, 3, 2\}$ based on the same validation process like in case of $\mathbf{b}$ and we took into account the fact that a_i/b_i should be in range $(\hat{\pi}_j/(\hat{\pi}_j + 1), 0.5]$, as it was pointed out in (5). The macro F-scores computed for the test and validation set are shown in Table 4 along with the validated values of a_i and b_i. For sake of readability, we repeat here the scores achieved by STO and FTA reported earlier in Table 2. The macro F-scores achieved by OFO are slightly better thanks to the additional degree of freedom, and thus, the OFO algorithm outperforms FTA and STO algorithm on almost every datasets except the Amazon dataset in which case the OFO and FTA algorithms are tied.

Table 4. The test macro F-scores obtained by validating both input parameters a and b. The numbers in bold indicate the best score achieved on each dataset.

Algorithm	Dataset	a_i	b_i	OFO Valid. F-score	OFO Test F-score	FTA Test F-score	STO Test F-score
PLT	RCV1	300	20000	22.20	**22.00**	20.41	21.16
PLT	AmazonCat	700	5000	33.37	35.30	34.83	31.64
PLT	wiki10	100	200	55.27	**30.28**	29.98	24.02
PLT	Delicious-200K	100	200	34.88	**11.20**	11.12	10.96
PLT	WikiLSHTC	100	200	39.94	14.00	12.31	16.22
PLT	Amazon	100	200	54.84	51.28	51.77	46.94
FASTXML	RCV1	1000	100000	19.84	19.28	17.04	19.58
FASTXML	AmazonCat	10000	100000	50.21	**41.48**	41.07	37.28
FASTXML	wiki10	5000	100000	54.72	29.91	29.88	28.26
FASTXML	Delicious-200K	100	500	35.02	**11.20**	11.18	10.83
FASTXML	WikiLSHTC	5000	100000	45.78	**21.38**	21.24	20.41
FASTXML	Amazon	10000	100000	53.91	**52.86**	**52.86**	47.53