

# LibMultiLabel: a Library for Multi-label Classification

LibMultiLabel Project Authors\*

April 3, 2026

## Abstract

LibMultiLabel is an open source software for binary, multi-class and multi-label classification, supporting various neural network architectures and linear classifiers. LibMultiLabel can be found at <https://www.csie.ntu.edu.tw/~cjlin/libmultilabel/> This paper provides the mathematical formulations and implementation details of LibMultiLabel.

## Contents

<b>1</b>	<b>Metrics</b>	<b>3</b>
1.1	Precision and Recall at $K$ . . . . .	3
1.2	R-Precision at $K$ . . . . .	3
1.3	Normalized Discounted Cumulative Gains at $K$ . . . . .	4
1.4	F-measure . . . . .	5
1.5	Choosing the Suitable Metrics . . . . .	6
<b>2</b>	<b>Handling zero-shot labels</b>	<b>7</b>
2.1	The Default Behavior in LibMultiLabel . . . . .	7
2.2	When to Set the Option to be True? . . . . .	7
2.3	Remarks on Labels in Training/Validation Sets . . . . .	7
<b>3</b>	<b>Linear Methods</b>	<b>7</b>
3.1	One-vs-rest (Binary Relevance) . . . . .	8
3.1.1	Thresholding . . . . .	9
3.1.2	Cost-Sensitive . . . . .	9
3.1.3	Training Time Analysis . . . . .	9
3.1.4	Space Analysis . . . . .	10
3.1.5	Implementation Details for One-vs-rest Setting . . . . .	10
3.2	Tree-Based Methods . . . . .	11
3.2.1	Definition of a Tree . . . . .	11
3.2.2	Tree Construction . . . . .	12
3.2.3	Training . . . . .	12
3.2.4	Training Time Analysis . . . . .	13

---

\*See contributors at <https://github.com/ntumlgroup/LibMultiLabel/graphs/contributors>

3.2.5	Prediction . . . . .	13
3.2.6	Ensemble Training and Prediction . . . . .	14
3.2.7	Implementation Issue and Mitigation Strategy for Prediction in Python . . . . .	15
3.2.8	Sparse Format Selection . . . . .	17
3.2.9	Space Analysis . . . . .	18
3.3	Choice of Solvers in LIBLINEAR . . . . .	19
3.4	The -B Option in LIBLINEAR . . . . .	19
3.5	Cross-validation Data Splits . . . . .	19

# 1 Metrics

Metrics are functions that represent the performance of models during evaluation. When predicting, we use a model to calculate the scores for an instance associated with labels. For example, let  $w$  be the weight of a linear model for label  $l$ . Then for a given instance  $x$ , the score of  $x$  for label  $l$  is calculated by  $w^T x$ . This score will be used to decide whether this instance is associated with label  $l$ . For this reason, we called this score decision value. If a given instance  $x$  has the label  $l$ , we say that the label  $l$  is relevant to  $x$ .

For a given data instance, let  $L$  be the number of labels and

$$\begin{aligned} \mathbf{p} &= [p_1 \ p_2 \ \cdots \ p_L] \in \mathbb{R}^L, \\ \hat{\mathbf{y}} &= [\hat{y}_1 \ \hat{y}_2 \ \cdots \ \hat{y}_L] \in \{0, 1\}^L, \\ \mathbf{y} &= [y_1 \ y_2 \ \cdots \ y_L] \in \{0, 1\}^L \end{aligned} \tag{1}$$

be the decision values, the predictions, and the ground truths associated with the instance respectively. The value of 1 indicates a relevant label and 0 indicates an irrelevant label. Define  $I_p = \{i_1, i_2, \dots, i_L\}$  to be the sorted index of  $\mathbf{p}$  by decision values.

## 1.1 Precision and Recall at $K$

Precision@ $K$  aims to check that among the top- $K$  predictions for a given instance, how many labels are relevant to the instance. So, precision@ $K$  for the instance is defined as follows

$$\text{P@K} = \frac{\text{\#relevant labels in the top-}K \text{ predictions}}{K} = \frac{\sum_{s=1}^K y_{i_s}}{K}. \tag{2}$$

On the other hand, recall@ $K$  shows that among labels associated with the given instance, how many are in the top- $K$  predictions. Recall@ $K$  for the instance can be defined as follows

$$\text{R@K} = \frac{\text{\#relevant labels in the top-}K \text{ predictions}}{\text{\#relevant labels}} = \frac{\sum_{s=1}^K y_{i_s}}{\sum_{s=1}^L y_s}. \tag{3}$$

Note that, we set  $\text{R@K} = 0$  if the number of relevant labels associated with the instance is zero.

The values of  $\text{P@K}$  and  $\text{R@K}$  over the entire data set is the average of  $D$  instances, calculated as follows:

$$\begin{aligned} \text{P@K} &= \frac{1}{D} \sum_{j=1}^D \text{P@K for the } j\text{th instance,} \\ \text{R@K} &= \frac{1}{D} \sum_{j=1}^D \text{R@K for the } j\text{th instance.} \end{aligned}$$

## 1.2 R-Precision at $K$

For instances where the number of relevant labels is less than  $K$ , even with a perfect prediction,  $\text{P@K}$  will be smaller than 1. This is because

$$\text{P@K} = \frac{\text{\#relevant labels in the top-}K \text{ predictions}}{K} < \frac{K}{K} = 1.$$

On the other hand, when  $K$  is smaller than the number of relevant labels, then even with a perfect prediction,  $R@K$  will be smaller than 1. The reason is

$$R@K = \frac{\text{\#relevant labels in the top-}K \text{ predictions}}{\text{\#relevant labels}} < \frac{\text{\#relevant labels}}{\text{\#relevant labels}} = 1.$$

For example, if the instance associates with two labels, then the value of  $P@5$  for a perfect prediction is 0.4 and the value of  $R@1$  for a perfect prediction is 0.5. For this reason, we cannot ensure that the values of  $P@K$  and  $R@K$  over different data sets for perfect predictions are always 1. To ensure the maximum value of the metric is 1, R-Precision at  $K$  ( $RP@K$ ) may be used.

$RP@K$  is very similar to  $P@K$  and  $R@K$  as the only difference is in the denominator. The denominators of  $P@K$  and  $R@K$  are  $K$  and the number relevant labels respectively. Instead, the denominator of  $RP@K$  is

$$\min(K, \text{\#relevant labels of the instance}).$$

With this change, the maximum value of  $RP@K$  is always 1. The definition of  $RP@K$  for the instance is as follows

$$RP@K = \frac{\text{\#relevant labels in the top-}K \text{ predictions}}{\min(K, \text{\#relevant labels of the instance})} = \frac{\sum_{s=1}^K y_{i_s}}{\min(K, \text{\#relevant labels of the instance})}.$$

Similarly, the value of  $RP@K$  over the entire data set is the average of  $D$  instances, calculated as follows:

$$RP@K = \frac{1}{D} \sum_{j=1}^D RP@K \text{ for the } j\text{th instance}.$$

### 1.3 Normalized Discounted Cumulative Gains at $K$

When the number of relevant labels in the top- $K$  predictions are the same for two predictions, then by (2) and (3), these two predictions will have the same value of  $P@K$  and  $R@K$ . In this case, these metrics cannot discriminate between the two predictions. For example, consider the ground truth and two predictions for an instance as follows

$$\begin{aligned} \text{ground truth} &= [0, 1, 1, 0, 0], \\ \text{decision values of prediction 1} &= [0.1, 0.3, 1.0, -0.3, -0.7], \\ \text{decision values of prediction 2} &= [0.8, 0.2, 0.7, -0.1, -0.5]. \end{aligned}$$

In this case,  $P@5$  for these two predictions are both 0.4, but have different orders of labels.

To understand why this matters, consider a search engine. If these are the search results, we hope that positive labels appear first. That is, positive labels have higher ranks. From this perspective, prediction 1 is better than prediction 2 in the example above.

To solve this problem, we use another metric called normalized discounted cumulative gains at  $K$  ( $NDCG@K$ ). Before introducing how to compute  $NDCG@K$ , we need to understand what  $DCG@K$  and  $IDCG@K$  are.

Discounted cumulative gains at  $K$  ( $DCG@K$ ) measures the top- $K$  predictions by taking discounts for different ranks. With this metric, the above two predictions will have different values of  $DCG@K$  and we can use these values to compare which is better.  $DCG@K$  for the instance is defined as follows

$$DCG@K = \sum_{s=1}^K \frac{y_{i_s}}{\log_2(s+1)}.$$

A problem with DCG@K is that it is not comparable across instances with a different number of relevant labels. For example, consider these two instances

$$\begin{aligned} \text{ground truth 1} &= [0, 1, 0, 0, 0], \\ \text{decision values of prediction 1} &= [0.1, 1.2, -0.9, -0.7, -0.5], \\ \text{ground truth 2} &= [1, 0, 1, 0, 1], \\ \text{decision values of prediction 2} &= [0.3, 1.0, 0.4, -0.9, 0.1]. \end{aligned}$$

Then DCG@5 for these two instances will be 1 and 1.52 respectively. Despite the first instance having the best possible prediction, it has a lower DCG@5 than the second instance. To solve this problem, one way is to consider the ratio of DCG@K for a prediction and DCG@K for the best prediction. DCG@K for the best prediction is called ideal DCG@K (IDCG@K) and this ratio called normalized DCG@K (NDCG@K).

IDCG@K is the maximum value of DCG@K. The maximum value of DCG@K occurs when all of the relevant labels are ranked higher than irrelevant labels. In other words, let  $I = \min(K, \|\mathbf{y}\|_0)$ . Note that  $\|\mathbf{y}\|_0$  is the 0-norm of  $\mathbf{y}$ , which is the number of non-zero elements of  $\mathbf{y}$ . Then the maximum value of DCG@K occurs when the top- $I$  predictions for the given instance are all relevant. Thus, the expression of IDCG@K for the instance is defined as

$$\text{IDCG@K} = \sum_{i=1}^{\min(K, \|\mathbf{y}\|_0)} \frac{1}{\log_2(i+1)}.$$

NDCG@K shows how close the prediction is to the best possible prediction, calculated as follows:

$$\text{NDCG@K} = \frac{\text{DCG@K for the instance}}{\text{IDCG@K for the instance}}.$$

The value of NDCG@K over the entire data set is the average of each instance, calculated as follows:

$$\text{NDCG@K} = \frac{1}{D} \sum_{j=1}^D \text{NDCG@K for the } j\text{th instance}.$$

## 1.4 F-measure

In the above, we introduced some ranking measures. They only check top-K predictions and  $K$  is usually a small number. When we need to consider the whole predictions, a ranking metric may not be a good choice. Instead, we may choose some classification measures like the F-measure. F-measure is one of the most used performance measures for information retrieval systems. It is the harmonic means of precision ( $P$ ) and recall ( $R$ ).

Precision shows that among predictions for all instances, how many positive predictions are correct. Recall shows that among positive instances, how many are predicted. Precision and recall for label  $l$  are expressed as follows:

$$P_l = \frac{\#\text{TP for label } l}{\#(\text{TP} + \text{FP}) \text{ for label } l} \quad \text{and} \quad R_l = \frac{\#\text{TP for label } l}{\#(\text{TP} + \text{FN}) \text{ for label } l},$$

where TP, FP, and FN are defined in Table 1.

Table 1: Definition of TP, FP, FN, and TN.

		ground truth	
		True	False
prediction	True	TP (true positive)	FP (false positive)
	False	FN (false negative)	TN (true negative)

Then the F-measure for label  $l$  is

$$F_l = \frac{2 \cdot P_l \cdot R_l}{P_l + R_l} = \frac{2 \# \text{TP for label } l}{(2 \# \text{TP} + \# \text{FP} + \# \text{FN}) \text{ for label } l}$$

To extend the F-measure from single-label to multi-label, two approaches are developed in Tague (1981). The first is the macro-average F-measure, which is the unweighted mean of label F-measures,

$$\text{Macro-F1} = \frac{1}{L} \sum_{l=1}^L F_l = \frac{1}{L} \sum_{l=1}^L \frac{2 \# \text{TP for label } l}{(2 \# \text{TP} + \# \text{FP} + \# \text{FN}) \text{ for label } l}.$$

Some use a different way, denoted as Macro\*-F1, by calculating the average percision and recall over all labels first.

$$\begin{aligned} \bar{P} &= \frac{1}{L} \sum_{l=1}^L P_l = \frac{1}{L} \sum_{l=1}^L \frac{\# \text{TP for label } l}{\#(\text{TP} + \text{FP}) \text{ for label } l}, \\ \bar{R} &= \frac{1}{L} \sum_{l=1}^L R_l = \frac{1}{L} \sum_{l=1}^L \frac{\# \text{TP for label } l}{\#(\text{TP} + \text{FN}) \text{ for label } l}, \\ \text{Macro*-F1} &= \frac{2 \cdot \bar{P} \cdot \bar{R}}{\bar{P} + \bar{R}}. \end{aligned}$$

Opitz and Burst (2021) suggest that Macro\*-F1 is less suitable to use.

The other multi-label measure is the micro-average F-measure, which calculates total TP, FP, and FN first.

$$\text{Micro-F1} = \frac{\sum_{l=1}^L \# \text{TP for label } l}{\sum_{l=1}^L (2 \# \text{TP} + \# \text{FP} + \# \text{FN}) \text{ for label } l}.$$

## 1.5 Choosing the Suitable Metrics

The choice of metrics should be motivated by the use case of the model. No metric fits every scenario equally well.

For example, if the model is used as a large-scale search engine, then the number of labels will be enormous. In this case, only the first few dozens of search results are important because no user will read every one of the results. For this reason, multi-label problems with a large amount of labels are often only concerned about the top few predictions. In this case, we might use P@K or NDCG@K with a choice of  $K$  that reflects the use case well.

In contrast, multi-label problems with a small amount of labels are often concerned with predicting all the labels correctly. For example, illness prediction in medical data is usually concerned about every label. In such a case, we may choose to use Macro-F1 and Micro-F1.

## 2 Handling zero-shot labels

In some cases, there exist labels that only appear in the test data. These labels are called zero-shot labels. We provide an option `include_test_labels` in `LibMultiLabel` to handle these labels in evaluation. This option can be true or false to decide whether to include zero-shot labels for evaluation. In this section, we illustrate some details on how to choose a correct value of `include_test_labels`.

### 2.1 The Default Behavior in LibMultiLabel

The default value of `include_test_labels` is false because of the following reasons. Consider the case that models do not handle the zero-shot labels. If we include these labels for evaluation, the ranking measures are not affected. However, the classification measures such as Macro-F1 or Micro-F1 become different. In particular, because the F-measure of zero-shot labels is zero, the resulting Macro-F1, which is the unweighted mean of label F-measures, can be significantly different. In this situation, the zero-shot labels should not be included in the evaluation. Popular software such as scikit-learn does not include test labels for evaluation, and we hope to be consistent with them.

### 2.2 When to Set the Option to be True?

Sometimes, we may need to include zero-shot labels for evaluation. For example, if a paper experiments with approaches to handle zero-shot labels and report some classification measures, then to compare their results, the option `include_test_labels` should be true. For example, Chalkidis et al. (2019) propose the data set EURLEX57K, which contains zero-shot labels. In their experimental results, they include zero-shot labels for evaluation and report Micro-F1. To compare with their results, we must set `include_test_labels` to be true.

### 2.3 Remarks on Labels in Training/Validation Sets

No matter which value of `include_test_labels`, we always consider the combined label set of training and validation sets. The reason is that training and validation instances are considered as all available data, so those labels that only appear in validation sets should not be regarded as zero-shot labels. Further, it is possible that we conduct training/validation splits several times. For example, we adopt the cross-validation strategy in our linear solvers. Therefore, it is better to consider the same label set across splits.

## 3 Linear Methods

Linear methods are the methods based on linear classifiers trained with bag-of-words (BOW) features. Specifically, let  $\mathcal{D}$  be the set of documents and  $\mathcal{T}$  be the set of all terms appearing in  $\mathcal{D}$ . Given a term

$t \in \mathcal{T}$  and a document  $d \in \mathcal{D}$ , the associated BOW feature is the l2-normalized TF-IDF generated by

$$\begin{aligned} \text{normalized-tf-idf}(d, t) &= \frac{\text{tf-idf}(d, t)}{\sqrt{\sum_{s \in \mathcal{T}} \text{tf-idf}^2(d, s)}}, \\ \text{tf-idf}(d, t) &= \text{tf}(d, t) \cdot \text{idf}(t), \\ \text{tf}(d, t) &= \text{number of times } t \text{ occurs in } d, \\ \text{idf}(t) &= \log \left( \frac{1 + |\mathcal{D}|}{1 + \text{df}(t)} \right) + 1, \\ \text{df}(t) &= \text{number of documents containing } t. \end{aligned}$$

Consider a set of training instances  $\{(\mathbf{x}_j, \mathbf{y}_j)\}_{j=1}^D$  where  $D$  is the number of instances,  $L$  is the number of labels,  $n$  is the number of features,  $\mathbf{x}_j \in \mathbb{R}^n$  is BOW features, and  $\mathbf{y}_j \in \{-1, 1\}^L$  is a label vector such that

$$y_{jl} = \begin{cases} 1, & \text{if } \mathbf{x}_j \text{ is associated with the label } l, \\ -1, & \text{otherwise.} \end{cases}$$

Note that here we use  $+1/-1$  instead of  $+1/0$  in (1) to indicate relevant/irrelevant labels.

In `LibMultiLabel`, currently all linear methods except a tree-based setting aim to learn a  $f : \mathbb{R}^n \rightarrow \mathbb{R}^L$  which is composed of  $L$  decision functions.

$$f(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_L(\mathbf{x})).$$

In this section, we begin with introducing linear methods in the software and then discuss implementation details.

### 3.1 One-vs-rest (Binary Relevance)

The one-versus-rest setting, also referred to as binary relevance, trains a binary classification problem for each label on data with/without that label. That is, for the  $l$ -th label, we solve the corresponding  $l$ -th binary classification problem

$$\mathbf{w}_l = \arg \min_{\mathbf{w}} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{j=1}^D \xi(y_{jl} \mathbf{w}^T \mathbf{x}_j), \quad (4)$$

where  $C$  is a penalty parameter. For the loss function  $\xi$ , we support logistic regression and linear SVM through `LIBLINEAR` (Fan et al., 2008). In Section 3.1.5, we further introduce the implementation details of the one-vs-rest setting in `LibMultiLabel`. After the training process ends, for any test instance  $\mathbf{x}$ , the decision function of label  $l$  is  $f_l(\mathbf{x}) = \mathbf{w}_l^T \mathbf{x}$ . Various ways can be applied on decision values to make predictions. The most used method is to use  $f_l(\mathbf{x})$  as a binary classifier so that

$$\begin{cases} \mathbf{x} \text{ is predicted to have the label } l & \text{if } f_l(\mathbf{x}) > 0, \\ \text{otherwise} & \text{if } f_l(\mathbf{x}) \leq 0. \end{cases} \quad (5)$$

Alternatively, in some applications, labels corresponding to the largest  $K$  values of  $f_l(\mathbf{x}), \forall l$  are predicted to be associated with  $\mathbf{x}$ , where  $K$  is a number specified by users.

In multi-label classification, for rare labels, the binary problem (4) is highly imbalanced. However, the one-vs-rest method applies the same decision rules (5), for all labels  $l = 1, \dots, L$ , without considering class imbalance. To address this issue, we describe thresholding techniques and cost-sensitive learning strategies that extend the one-vs-rest formulation in the following two subsections.

### 3.1.1 Thresholding

Thresholding is a technique to address this class imbalance issue described above, and it is effective to optimize the Macro-F1 score (Lewis et al., 1996; Yang, 1999; Fan and Lin, 2007). Lin and Lin (2023) proposed the method that automatically decides a threshold  $\Delta_l$  for label  $l$  through a cross-validation procedure so that the decision function becomes

$$f_l(\mathbf{x}) = \mathbf{w}_l^T \mathbf{x} + \Delta_l.$$

Therefore, this method is more expensive than one-vs-rest. See Section 4.3 and supplementary D of Lin and Lin (2023) for details of the thresholding method.

### 3.1.2 Cost-Sensitive

Another scheme to solve the class imbalance problem is cost-sensitive learning, which uses a higher loss on positive training instances. Parambath et al. (2014) give some theoretical support showing that the F1 score can be optimized through cost-sensitive learning. For the label  $l$ , they extend problem (4) of one-vs-rest to

$$\mathbf{w}_l = \arg \min_{\mathbf{w}} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \left( \frac{2-t}{t} \right) \sum_{j:y_{jl}=1} \xi(y_{jl} \mathbf{w}^T \mathbf{x}_j) + C \sum_{j:y_{jl}=-1} \xi(y_{jl} \mathbf{w}^T \mathbf{x}_j), \quad (6)$$

where  $(2-t)/t$  is the cost of false negatives, and  $t \in (0, 1]$ . In `LibMultiLabel`, for each label, a pre-defined grid of  $(C, t)$  pairs are checked to find the one leading to the best validation F1 score. The best pair is then applied to the whole training set to get the final decision function of the corresponding label. Therefore, this method is more expensive than one-vs-rest.

### 3.1.3 Training Time Analysis

In general, the training time of the optimization problems (4) and (6) depends on the dimensions (number of instances and features), data values and parameters. While we may not be able to calculate the training time without knowing the data set, a rough comparison of the training time of different linear methods can be by counting the optimization problems solved. The reason is that all optimization problems solved (including those in the cross-validation procedures) have comparable dimensions.

- One-vs-rest: Because one problem (4) is solved for each label, the total number of optimization problems solved is  $L$ .
- Thresholding: We conduct three-folds cross-validation, so the number of optimization problems solved is  $3L$ .
- Cost-sensitive: This method finds the optimal  $C(2-t)/t$  value by a grid search on checking the cross-validation performance at each  $C(2-t)/t$  value. For each label, there are 7 pre-defined  $(C, t)$  pairs and 3 folds for each pair. The number of optimization problems solved is  $21L$ . For one-vs-rest and thresholding, by default the regularization parameter  $C = 1$  is used across all optimization problems, but in (6),  $C(2-t)/t$  is the regularization parameter on the training loss of positive data. It is known that the training time increases with a larger regularization parameter. With  $t \in (0, 1]$ , we have  $C(2-t)/t > C$ , so the training time of cost-sensitive may be longer than the count of optimization problems suggests.

### 3.1.4 Space Analysis

All linear methods under the binary relevance setting, including One-vs-rest, Thresholding, and Cost-Sensitive, require each label’s weights  $(w_1, \dots, w_L)$  to be stored. We store weights as dense vector, so the space consumption is  $O(nL)$ . More specifically, each weight entry is stored as a double-precision floating-point number, which requires 8B. Thus, the space consumption for storing the weights is  $8nLB$ .

### 3.1.5 Implementation Details for One-vs-rest Setting

The classification problems (4) and (6) are solved with LIBLINEAR (Fan et al., 2008). All two-class classification methods in LIBLINEAR are supported. We choose the coordinate descent method to solve the dual problem of L2-loss SVM as the default. The reason is that for large sparse data (e.g., documents), Table 15 in Lin et al. (2026) shows that this coordinate descent method is more memory efficient than others. Table 16 in that work also shows that the performance of L2-loss SVM is generally better than that of L1-loss SVM. Depending on the requirements, the solver can be changed by specifying the option `liblinear_options`. See Fan et al. (2008) for details of each solver.

The training time can be further reduced by utilizing the multi-core architecture of modern computers. In the one-vs-rest setting, for the optimization problems (4) that we need to solve, we have two parallelization strategies.

- **Strategy A:** Using a for-loop on the  $L$  problems (4) and performing parallelization on each problem (4) using Multi-core LIBLINEAR (Lee et al., 2015; Chiang et al., 2016; Zhuang et al., 2018).
- **Strategy B:** Solving the  $L$  problems (4) in parallel.

While **Strategy B** can independently process each parallel task, **Strategy A** cannot. This makes **Strategy B** a more efficient choice for the use of the cores.

However, to make the efficient parallelization of **Strategy B** viable, an obstacle we have to overcome is the memory usage. In Python, multi-threading and multi-processing are two different ways to achieve parallelization. We do not consider multi-processing because it duplicates the data for each core, and the memory usage would be  $T$  times larger if we use  $T$  cores. While multi-threading becomes a better choice, Python’s Global Interpreter Lock (GIL) restricts that Python can only run one thread at a time. This means we cannot achieve real parallel processing with multi-threading in Python. Fortunately, there is a chance for the GIL to be released. When a thread calls a C extension in Python, it will release the GIL and only go back to Python when it finishes its task in the C extension. That means, in Python, if there are other threads waiting for the GIL, they can compete for the released GIL and be able to run once they get it. In `LibMultiLabel`, since we call LIBLINEAR C extension to solve each problem (4), when each thread begins solving the optimization problem (4), it releases the GIL. Once the GIL is released, while the releaser keeps running its task in the C extension, the other threads can compete for the GIL and begin their training. As each thread spends a significant amount of time on training, most of the time they are processing in parallel.

Moreover, good news is that for both strategies, since  $\{\mathbf{x}_j\}_{j=1}^D$  is identical for all  $L$  problems (4), we can share the memory for  $\mathbf{x}$  across all loops in **Strategy A** and all threads in **Strategy B**. We achieve this by copying and sharing the `feature_node` array, which represents  $\{\mathbf{x}_j\}_{j=1}^D$  in (4), within each LIBLINEAR problem. This is easily achieved using the `copy` member function of the LIBLINEAR problem.

Table 2: Comparison of the training time in seconds between **Strategy A** and **Strategy B** with different numbers of threads on a 32-core machine, and the speed-up for **Strategy B** over **Strategy A**. We solve each binary problem using L2-regularized L2-loss SVM. The data set is AmazonCat-13K, with 16 classes,<sup>3</sup> 1, 186, 239 instances, and 203, 882 features.

	1 thread	2 threads	4 threads	8 threads	16 threads
<b>Strategy A</b>	112.59	88.44	72.56	66.72	96.74
<b>Strategy B</b>	113.23	60.18	34.05	24.61	22.72
Speed-up	0.99	1.47	2.13	2.71	4.26

To sum up, as Table 2 shows, on AmazonCat-13K,<sup>1</sup> we can see **Strategy B** achieves up to four times speed-up over **Strategy A** when the number of threads increases. Therefore, we implement Python-level parallelization with multi-threading on the  $L$  problems (4) for the one-vs-rest setting in LibMultiLabel. One thing we should notice is that even with **Strategy B**, which achieves parallelization in a more efficient way, more threads is not always better due to hardware limitations. When the number of threads is too large, the memory usage will become worse as more threads join the competition for memory and eventually hurt the training time. For the thresholding setting and the cost-sensitive setting, there is no implementation for the Python-level parallelization yet. We believe it is important to implement it due to the significant improvement in the training time for the one-vs-rest setting, and we will do it in the future.

## 3.2 Tree-Based Methods

Tree methods are divide-and-conquer techniques that recursively divide the label space in order to reduce the time complexity of training and prediction. At each divide step, labels are partitioned into disjoint sets. These sets form smaller multi-label problems, which are in turn recursively divided. This process is naturally described as a tree, with each node (including the root) representing a multi-label problem.

### 3.2.1 Definition of a Tree

Since we recursively partition labels into metalabels, we can also describe this process in terms of a tree as follows:

- A node is either a set of labels or a set of label subsets.
- A node is a leaf node if it is a set of labels.
- For each node  $S$ , its child nodes form a partition of  $S$ .
- If any two nodes  $T$  and  $S$  are with the same depth, then  $T$  and  $S$  are disjoint.
- The union of all nodes at the same depth is equal to the label space.

At the root node, we have a corresponding representation of the tree. We consider two ways:

- A recursive set of sets.

<sup>1</sup>[https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel/AmazonCat-13K\\_raw\\_texts\\_train.txt.bz2](https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel/AmazonCat-13K_raw_texts_train.txt.bz2)

<sup>3</sup>The original data set has 13, 330 classes, but we grouped the classes into 16 groups by the remainder of the label index divided by 16 for the experiment.

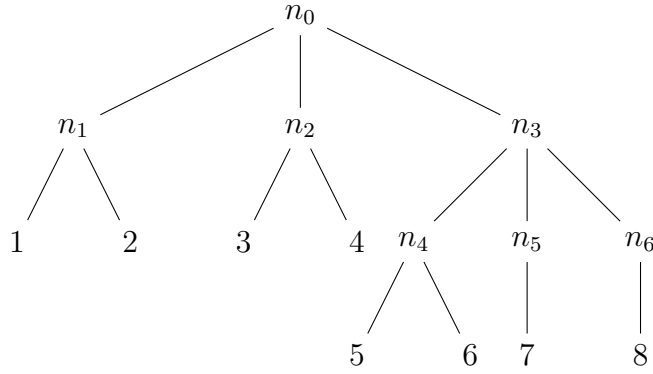


Figure 1: A possible tree with eight labels.

- A set of label subsets.

For example, the root node  $n_0$  of the tree in Figure 1 can be represented as

$$\begin{aligned} n_0 &= \{\{1, 2\}, \{3, 4\}, \{\{5, 6\}, \{7\}, \{8\}\}\} \\ &= \{\{1, 2\}, \{3, 4\}, \{5, 6, 7, 8\}\}. \end{aligned}$$

The first form depicts the structure of the tree, used when we want to emphasize a property of the entire tree. The second form shows the immediate children, used when we want to emphasize a property of a single depth. The second form also shows how we have a multi-label problem at each node.

Note that since a node covers a subset of labels, we say an instance  $\mathbf{x}$  is associated with node  $T$  if  $\mathbf{x}$  is associated with any label in  $T$ .

### 3.2.2 Tree Construction

One way to recursively divide the label space is by the  $K$ -means algorithm with label information. However, the label information is unavailable or meaningless in some data sets. To handle this problem, several methods are proposed in Khandagale et al. (2020) and Yu et al. (2022) to construct the label representations without knowing any information about labels. We chose a method that generates the representation of label  $l$  by aggregating the feature of instances associated with label  $l$ . For label  $l$ , its label representation is constructed as follows

$$\mathbf{z}_l = \frac{\sum_{j:y_{jl}=1} \mathbf{x}_j}{\|\sum_{j:y_{jl}=1} \mathbf{x}_j\|}$$

### 3.2.3 Training

As mentioned at the beginning, each node corresponds to a multi-label problem. The goal of training is to associate a model  $m_T$  to each node  $T$ . The model  $m_T$  is trained by a one-vs-rest strategy. For node  $T$  with  $K_T$  children, the training data set is constructed as follows:

$$\{(\tilde{\mathbf{y}}_j, \mathbf{x}_j) \mid \mathbf{x}_j \text{ is associated with node } T, j = 1, \dots, D\}.$$

where  $D$  is the number of instances and  $\tilde{\mathbf{y}}_j \in \{-1, 1\}^{K_T}$  is a metalabel or label vector defined by

$$\tilde{y}_{jl} = \begin{cases} 1, & \text{if } \mathbf{x}_j \text{ is associated with the } l\text{-th child node of } T \\ -1, & \text{otherwise} \end{cases}$$

Table 3: Example of the one-vs-rest setting at the node  $n_3$  in Figure 1. Note that only instances associated with any of labels 5, 6, 7, 8 are considered at  $n_3$ .

	positive	negative
classifier 1	instances associated with 5 or 6	instances not associated with 5 and 6
classifier 2	instances associated with 7	instances not associated with 7
classifier 3	instances associated with 8	instances not associated with 8

For example, the classifier of node  $n_3$  in Figure 1 considers three binary problems shown in Table 3.

### 3.2.4 Training Time Analysis

In Section 3.1.3, we explain why we compare the number of optimization problems solved, and we follow the same logic to analyze the training time. For each node  $u$ , we train a one-vs-rest model corresponding to the elements of the node  $u$ 's label set. However, the label set associated with non-leaf and leaf nodes exhibits different structural properties, so we discuss them separately and describe their properties below.

- Non-leaf node  $u$ : Each linear classifier represents an edge leading to one of the node  $u$ 's child nodes.
- Leaf node  $u$ : Each linear classifier is used to predict a label in the label space.

In a tree structure, the total number of edges is equal to the total number of nodes minus one. Let us assume the total number of nodes in the tree is  $N$ . Thus, the total number of classifiers in the non-leaf nodes is  $N - 1$ . For leaf nodes, because the label sets between any two leaf nodes are disjoint, the total number of classifiers in leaf nodes is  $L$ . In summary, the total number of optimization problems solved is  $L + (N - 1)$ .

### 3.2.5 Prediction

For node  $V$  and its child  $S$ , the model  $m_V$  can estimate the probability

$$\mathbb{P}(\mathbf{x} \text{ is associated with } S \mid \mathbf{x}, V) \quad (7)$$

via the transformation  $\sigma$  (Yu et al., 2022)

$$\sigma(\mathbf{w}_V^T \mathbf{x}) = \exp(-\max(1 - \mathbf{w}_V^T \mathbf{x}, 0)^2), \quad (8)$$

where  $\mathbf{w}_V$  is the weight of the model  $m_V$ . Note that  $\sigma(\mathbf{w}_V^T \mathbf{x})$  is a vector consisting of the probabilities (7) of all  $V$ 's child nodes. Let  $\{n_{p_0} = n_0, n_{p_1}, \dots, n_{p_d} = l\}$  be a path from root node  $n_0$  to label  $l$ . Then by the concept of conditional probability, we estimate the probability

$$\mathbb{P}(\mathbf{x} \text{ is associated with label } l \mid \mathbf{x}, n_{p_0}, \dots, n_{p_d}) = \prod_{i=1}^d \mathbb{P}(\mathbf{x} \text{ is associated with } n_i \mid \mathbf{x}, n_{i-1}). \quad (9)$$

To calculate (9) for all labels, we should not handle labels separately. The reason is that duplicated calculation occurs since the paths for two labels share some nodes in the beginning. To this end, we calculate  $\sigma(\mathbf{w}^T \mathbf{x})$  layer by layer according to the depth and multiply the results based on (9).

For a multi-label problem with a large label space, we usually only concern with the top few predictions. That means calculating the probability of these labels when predicting is enough. We use a beam

search algorithm to retain nodes with higher probabilities. Specifically, for nodes selected at the current layer of the tree, we calculate  $\sigma(\mathbf{w}^T \mathbf{x})$  of their children and select the top- $B$  child nodes for the next layer. This setting effectively reduces the prediction time.

Take Figure 1 as an example. For simplicity, we denote  $w_i$  as the weight of the model  $m_{n_i}$  and set  $B = 1$ . First, we calculate  $\sigma(\mathbf{w}_0^T \mathbf{x})$ . Assuming the third entry of  $\sigma(\mathbf{w}_0^T \mathbf{x})$  is the largest, we use node  $n_3$  for further calculation. Next, we calculate  $\sigma(\mathbf{w}_3^T \mathbf{x})$  and multiply with the third entry of  $\sigma(\mathbf{w}_0^T \mathbf{x})$ . If the first entry is the largest, we use node  $n_4$  for further calculation. Since  $n_4$  is a leaf node, the results are the probabilities of label 5 and label 6.

### 3.2.6 Ensemble Training and Prediction

While a single tree-based model offers scalability and efficiency for extreme multi-label classification tasks, its predictive performance can occasionally fall short compared to more expressive methods such as one-vs-rest classification. This is primarily because a single tree relies on a fixed recursive partitioning structure, which may not sufficiently capture the diversity and complexity of the label space, particularly in noisy or imbalanced scenarios.

To address the limitations of a single tree and enhance model robustness, we construct an ensemble of tree models. Each tree is trained independently using the same recursive construction method described in Section 3.2.3, but with a unique random seed used to initialize the clustering process at each tree. This seed-level randomness introduces structural diversity among trees, while all other hyperparameters—such as the number of clusters at each node, the maximum depth of the tree, the clustering algorithm, beam width, and the linear solver configuration—are kept fixed across ensemble members. As a result, the ensemble learns complementary decompositions of the label space from the same training data. During inference, the models’ output probabilities are averaged to produce final predictions, reducing variance and improving generalization.

Although training time increases approximately linearly with the number of trees  $M$ , the ensemble method remains efficient. Each model is constructed sequentially by default, but since training is independent across trees, users may manually parallelize the process if desired. In practice, even small ensembles (e.g.,  $M = 3$  or  $M = 10$ ) yield meaningful improvements over a single tree.

During prediction, each test instance  $\mathbf{x}$  is passed through all  $M$  trained tree models using the beam search method described in Section 3.2.5. Each model estimates the conditional probability that  $\mathbf{x}$  is associated with label  $l$  based on the path from the root to the leaf node corresponding to  $l$ , as in Equation (9). The final probability that instance  $\mathbf{x}$  is associated with label  $l$  is computed by averaging the estimates across all trees:

$$\mathbb{P}(\mathbf{x} \text{ is associated with label } l) = \frac{1}{M} \sum_{m=1}^M \mathbb{P}_m(\mathbf{x} \text{ is associated with label } l),$$

where  $\mathbb{P}_m$  denotes the probability output from the  $m$ -th tree. After computing these averaged probabilities, we select the top- $k$  labels with the highest probabilities as the final prediction. As in the single-tree case, beam search is employed at each level of each tree to efficiently prune the candidate paths.

To evaluate the effectiveness of the ensemble approach, we conduct experiments on five benchmark datasets. For each tree, we use the same training configuration, including feature normalization and a linear classifier trained with LIBLINEAR. Table 4 reports Precision at 1, 3, and 5 (P@1, P@3, P@5) for single-tree and ensemble models using 3, 10, and 15 trees.

These results demonstrate that ensemble methods consistently outperform single tree models across datasets of varying scale and complexity. While the performance gains from 3 to 15 trees are modest, even

Table 4: Benchmark Results for Single and Ensemble Tree Models (P@K in %)

Dataset	Model	P@1	P@3	P@5
EURLex-4k	Single Tree	82.35	68.98	57.62
	Ensemble-3	82.38	69.28	58.01
	Ensemble-10	82.74	69.66	58.39
	Ensemble-15	82.61	69.56	58.29
EURLex-57k	Single Tree	90.77	80.81	67.82
	Ensemble-3	91.02	81.06	68.26
	Ensemble-10	91.23	81.22	68.34
	Ensemble-15	91.25	81.31	68.34
MIMIC	Single Tree	84.67	77.87	71.47
	Ensemble-3	85.44	78.14	71.79
	Ensemble-10	85.23	78.44	72.08
	Ensemble-15	85.44	78.44	72.05
Wiki-31k	Single Tree	84.40	74.47	65.40
	Ensemble-3	84.27	74.80	65.75
	Ensemble-10	84.27	74.71	65.93
	Ensemble-15	84.39	74.80	65.99
AmazonCat-13k	Single Tree	93.34	79.51	64.60
	Ensemble-3	93.79	79.90	65.04
	Ensemble-10	93.95	80.04	65.16
	Ensemble-15	93.98	80.06	65.17

a small ensemble offers tangible improvements, particularly on large datasets such as EURLex-57k and AmazonCat-13k.

The ensemble approach improves prediction accuracy and robustness by averaging outputs from diverse tree structures. It helps reduce overfitting and handles label imbalance more effectively than a single tree.

The main cost is increased training time and memory usage proportional to the number of trees  $M$ . Prediction is slightly slower due to averaging, but remains efficient. In practice, using  $M = 3$  to  $M = 10$  trees often achieves a good balance between performance and efficiency.

### 3.2.7 Implementation Issue and Mitigation Strategy for Prediction in Python

In prediction, we should calculate  $\sigma(\mathbf{w}^T \mathbf{x})$  on-demand only for the specific nodes required by the beam search algorithm. This way, the algorithm evaluates  $\sigma(\mathbf{w}^T \mathbf{x})$  node-by-node, resulting in a series of small and frequent inner-product function calls. However, in Python, frequent function calls would lead to significant overheads. To prevent this, we first collect the classifier weights of the OVR model into matrix  $W_u$  for each node  $u$ , and then concatenate all nodes' weights into one large matrix

$$W = [W_1 \quad W_2 \quad \cdots \quad W_N] \in \mathbb{R}^{n \times (N+L-1)}, \tag{10}$$

where  $N + L - 1$  is the total number of linear classifiers mentioned in Section 3.2.4. We then perform a single multiplication  $W^T \mathbf{x}$  before the beam search. We call the matrix  $W$  the flattened model. This approach reduces the frequent calls, but it introduces unnecessary computation on decision values for many nodes that will be discarded. The unnecessary computation becomes more severe in deep and large

tree structures. During the beam search algorithm, we evaluate the top- $B$  nodes at each level in a level-by-level manner. As the tree deepens, unnecessary computations accumulate along pruned paths. For example, in an example of running a large-scale data set such as Amazon-670K, which has approximately 150,000 nodes and maximum tree depth 10, the beam search algorithm traverses roughly 100 nodes in total across the entire tree for each instance with  $B = 10$ . Consequently, this approach results in unnecessary computation for almost all nodes.

To balance the trade-off between unnecessary computation and frequent function calls, we consider two separate cases depending on the relative size of the beam width  $B$  and the number of the root’s children  $K$ .

- When  $B \geq K$ , the number of unnecessary computations is relatively small, so we use a single multiplication  $W^T \mathbf{x}$ .
- When  $B < K$ , we propose a mitigation strategy to reduce the overhead.

Next, we discuss the strategy when  $B < K$ . Our method begins by splitting the flattened model into subtrees based on the children of the root node,

$$W = [W_{\text{root}} \quad W_{\text{subtree}_1} \quad \cdots \quad W_{\text{subtree}_K}], \quad (11)$$

where  $K$  is the number of root’s children. This can be done as a preprocessing step before prediction. For a given batch of instances  $X = \{\mathbf{x}_i \mid i = 1, \dots, \text{batch\_size}\}$ , by using only the root weight  $W_{\text{root}}$ , we perform the multiplications  $W_{\text{root}}^T \mathbf{x}_i, \forall \mathbf{x}_i \in X$  to determine the top- $B$  subtrees for each  $\mathbf{x}_i$ . Then, all instances that share the top- $B$  subtree  $j$  are grouped into a subset of the batch

$$X_j = \{\mathbf{x}_i \mid \text{subtree } j \in \text{top-}B \text{ subtrees of } \mathbf{x}_i, i = 1, \dots, \text{batch\_size}\}.$$

For each subtree  $j$  and the corresponding data subset  $X_j$ , we continue the beam search for prediction by considering  $W_j$  as the flattened model. That is, for every  $\mathbf{x}_i \in X_j$ , we perform the  $W_j^T \mathbf{x}_i$  to calculate the decision values at all nodes of subtree  $j$ . This strategy effectively balances frequent function calls and the unnecessary computation by the following aspects.

- **Limited function calls.**

To determine the top- $B$  subtrees for each instance  $\mathbf{x}_i \in X$ , we need to compute  $W_{\text{root}}^T \mathbf{x}_i$ . By representing  $X$  as a matrix  $\text{MAT}(X)$ , we can perform a single matrix multiplication with  $W_{\text{root}}$ , avoiding repeated computations for each instances. Similarly, for each subtree  $j$ , we can represent  $X_j$  as a matrix  $\text{MAT}(X_j)$  and perform a single matrix multiplication with  $W_j$ . However, since some subtrees might not be among the top- $B$  subtree candidates for all instances  $\mathbf{x}_i \in X$ , the total number of function calls is at most  $K + 1$ .

- **Significant reduction on unnecessary computation.** By pruning irrelevant first-level subtrees for each instance, this approach effectively reduces some unnecessary computation.

We demonstrate the effectiveness of this balanced strategy with prediction time on several common benchmark data sets, as shown in Table 5.

Please note that the efficiency of sparse matrix multiplication in SciPy is affected by the properties of two matrices involved, such as the instance matrix  $\text{MAT}(X)$  and the flattened model weights  $W$  (or subtrees’ weights  $W_j$ ). SciPy ensures that both sparse matrices are in the same sparse format. If the formats differ, SciPy will automatically convert one matrix to match the other. This conversion introduces

Table 5: Comparison of prediction time in seconds between the “single multiplication” approach and the approach of “pruning first-level subtrees”.

Data set	Single multiplication	Pruning first-level subtrees
EUR-Lex	4.41	2.37
Wiki10-31K	106.98	68.58
Amazon-670K	4291.12	1648.39

Table 6: Storage requirements for different sparse formats. We consider the weight matrix  $W_u$  of any node  $u$ .

Format	Values	Row Indices (or Pointers)	Column Indices (or Pointers)
COO	$\text{NNZ}(W_u)$	$\text{NNZ}(W_u)$	$\text{NNZ}(W_u)$
CSR	$\text{NNZ}(W_u)$	$n + 1$	$\text{NNZ}(W_u)$
CSC	$\text{NNZ}(W_u)$	$\text{NNZ}(W_u)$	$K + 1$

significant overhead if it occurs for every multiplication. Therefore, we ensure that the formats of both matrices are the same before prediction. Additionally, since different sparse formats favor different shapes of two matrices for matrix multiplication, the shapes of the two sparse matrices also influence the choice of sparseformat. In our tests, we observe that matrix multiplication between two CSR matrices is generally faster than multiplication between two CSC matrices. For these reasons, we make sure that the instance matrix  $\text{MAT}(X)$ , the flattened model  $W$  and subtrees’ weights  $W_j$  are in the CSR format in the preprocessing step.

### 3.2.8 Sparse Format Selection

In Section 3.2.7, we discuss that we save the weight matrix  $W$  in (10). Before concatenation, we need to collect all node weights  $W_1, \dots, W_N$ . Lin et al. (2024) show that training a tree method with sparse data can return sparse model weights. The reason is that nodes in deeper levels involve subsets of data and some features are not used. Thus, we should store  $W$  as a sparse matrix. In practice, the selection of a sparse format significantly affects the memory consumption of the weight matrix. To illustrate the consideration of sparse formats, we use Amazon-670K as a demonstration, with labels  $L = 670,091$ , features  $n = 135,909$ , and the default tree setting  $K = 100$  in LibMultiLabel.

We first introduce the three commonly used sparse formats, namely Coordinate (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC). For the weight matrix  $W_u$  of any node  $u$ , we show the storage requirement of a sparse matrix with dimensions  $n \times K$  and  $\text{NNZ}(W_u)$ , the number of non-zero elements of  $W_u$ , in Table 6.

In general, the memory usage of sparse matrix  $W_u$  is dominated by  $\text{NNZ}(W_u)$ . However, as pointed out in Lin et al. (2024), some features are not used at all in  $W_u$ , meaning that some rows are all zero. In this situation,  $n$  may be even bigger than  $\text{NNZ}(W_u)$ , making the storage of row indices an important factor for the memory efficiency of  $W_u$ . Therefore, the selection of sparse format on  $W_u$  is important for the memory efficiency. After excluding terms needed by all features, we only need to compare the sizes of

$$\text{Row Indices for COO, Row Pointers for CSR, and Columns Pointers for CSC.} \quad (12)$$

In the case of Amazon-670K, each node involves a  $K$ -label multi-label problem. To store the  $K$  weight vectors from the one-vs-rest training we use a matrix of of size  $n \times K = 135,909 \times 100$ . The terms singled

out in (12) respectively takes the following memory consumption.

$$8.7 \text{ KB (COO)}, 543 \text{ KB (CSR)}, \text{ and } 404 \text{ B (CSC)}.$$

Because we have approximately 150,000 nodes in Amazon-670K, the overall differences are roughly

$$1.3 \text{ GB (COO)}, 81.45 \text{ GB (CSR)}, \text{ and } 60.6 \text{ MB (CSC)}. \quad (13)$$

The significant difference between the CSC and CSR is due to the fact of  $K \ll n$  in general. Further, in the  $n \times K$  matrix  $W_u$ , as pointed out in Lin et al. (2024), some features are not used at all due to the involvement of a small subset of data, meaning that some rows are empty. In this situation,  $n$  may be even bigger than  $\text{NNZ}(W_u)$ , so we waste space to have a long row-pointer array. Note that for a sparse matrix, typically the number of non-zero entries is bigger than the numbers of rows and columns. This condition may not hold for our matrices because of the many empty rows.

Therefore, based on CSC’s significant advantage in memory usage, we select the CSC format to store  $W_1, \dots, W_N$ . Now consider the scenario of using a single weight matrix  $W$  in (10). We compare the storage requirements of using different sparse formats. After the concatenation, the large sparse matrix  $W$  in (10) takes the following space for the problem Amazon-670K,

$$27.1 \text{ GB (COO)}, 20.33 \text{ GB (CSR)}, \text{ and } 20.33 \text{ GB (CSC)}.$$

Using the COO format consumes more space because of the need to store row and column indices. On the other hand, CSR no longer needs the large space shown in (13), because, after concatenation, there are no (or few) empty rows. Moreover, unlike the previous discussion, for the flattened model  $W$ , the  $\text{NNZ}(W)$  is much larger than  $n$  or  $L + N - 1$ , resulting in that the space consumption is dominated by the  $\text{NNZ}(W)$ . We observe that the space consumption of CSC and CSR are nearly identical, which is consistent with the theoretical analysis.

Beyond space efficiency, another important consideration of sparse formats for storing  $W$  is the prediction speed. As discussed in Section 3.2.7, if we consider the approach in (11) to split the flattened model for prediction at the root node, we need to do column-wise slicing on  $W$  to obtain  $W_{\text{root}}, \dots, W_{\text{subtree}_K}$ . However, we also need each  $W_u$  to be in the CSR format for faster matrix multiplication. These two requirements conflict because the column-wise slicing is very slow in the CSR format. In our observations, the column-wise slicing in the CSR format consumes approximately one-third to one-half of the overall prediction time. To address this issue, we initially save flattened model  $W$  in the CSC format. Then, we defer the column-wise slicing (if required) and the subsequent CSR format conversion in the preprocessing step of prediction.

### 3.2.9 Space Analysis

Following Section 3.2.8, we analyze the practical space consumption of tree-based methods. We begin with checking the needed space for storing the flattened model  $W$  in (10) in the CSC format. We store values in 8-byte floating-point numbers, and row indices and column pointers in 4-byte integers. The space consumption of the flattened model  $W$  is

$$\text{NNZ}(W) \times 12 + (N + L - 1) \times 4 \text{ bytes}.$$

However, when  $\text{NNZ}(W)$  exceeds the 32-bit integer limit, SciPy automatically switches to using 64-bit integers for storing row indices. Then, the memory consumption of the flattened model  $W$  becomes

$$\text{NNZ}(W) \times 16 + (N + L - 1) \times 4 \text{ bytes}.$$

one-versus-rest	ECtHR (A)		ECtHR (B)		UNFAIR-ToS		EUR-LEX		LEDGAR		SCOTUS	
	$\mu$ -F1	m-F1	$\mu$ -F1	m-F1	$\mu$ -F1	m-F1	$\mu$ -F1	m-F1	$\mu$ -F1	m-F1	$\mu$ -F1	m-F1
Default dual solver	54.5	69.6	68.9	75.5	51.7	59.3	56.7	72.5	79.4	86.1	68.8	78.0
Default primal solver	54.4	69.5	68.7	75.2	51.7	59.3	56.7	72.6	79.4	86.1	68.8	78.0

Table 7: Micro-F1 ( $\mu$ -F1) and Macro-F1 (m-F1) scores for the default dual solver and the default primal solver.

However, the peak memory use is twice of the above amount because the concatenation process to aggregate weight matrices of all nodes doubles the memory consumption. Thus, our memory bottleneck occurs in the end of the training procedure.

### 3.3 Choice of Solvers in LIBLINEAR

Note that in LIBLINEAR, the default solver is the primal solver. However, in LibMultiLabel, we set it to the dual solver. See Table 7 to check the comparison of these two types of solvers.

### 3.4 The -B Option in LIBLINEAR

LIBLINEAR (Fan et al., 2008) has the option -B for adding a regularized bias term. Given a parameter value  $B$  and a bias term  $b$ , the weights  $\mathbf{w}$  and features  $\mathbf{x}$  are augmented with an additional dimension:

$$\mathbf{w}' = \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} \quad \mathbf{x}' = \begin{bmatrix} \mathbf{x} \\ B \end{bmatrix}$$

Problem (4) is then modified as

$$\begin{aligned} \mathbf{w}'_l &= \arg \min_{\mathbf{w}'} \frac{1}{2} \mathbf{w}'^T \mathbf{w}' + C \sum_{j=1}^D \xi(y_{jl} \mathbf{w}'^T \mathbf{x}'_j) \\ &= \arg \min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{1}{2} b^2 + C \sum_{j=1}^D \xi(y_{jl} \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}^T \begin{bmatrix} \mathbf{x}_j \\ B \end{bmatrix}), \end{aligned}$$

We note the following implementation details. Since the prediction of LibMultiLabel is not performed through LIBLINEAR, it is convenient to acquire  $b$  in both training and prediction processes. To that end, if users specify the -B option, we augment the value  $B$  as an additional feature of the data in LibMultiLabel and strip -B from the options before passing training data to LIBLINEAR.

By default, we use -B 1 because empirically this seems to be useful.

### 3.5 Cross-validation Data Splits

In cost-sensitive, a cross-validation procedure is performed for each value of  $C(2-t)/t$ . The data splits, i.e. the subsets of data chosen for training or validation, in each cross-validation procedure may be the same or different for each  $C(2-t)/t$  value. The supplementary of Lin et al. (2022) showed that the difference of having the same or different data splits is insignificant. We chose to have the same data splits for each  $C(2-t)/t$  value.

## References

- I. Chalkidis, E. Fergadiotis, P. Malakasiotis, and I. Androutsopoulos. Large-scale multi-label text classification on EU legislation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 6314–6322, 2019. doi: 10.18653/v1/P19-1636.
- W.-L. Chiang, M.-C. Lee, and C.-J. Lin. Parallel dual coordinate descent method for large-scale linear classification in multi-core environments. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016. URL [http://www.csie.ntu.edu.tw/~cjlin/papers/multicore\\_cddual.pdf](http://www.csie.ntu.edu.tw/~cjlin/papers/multicore_cddual.pdf).
- R.-E. Fan and C.-J. Lin. A study on threshold selection for multi-label classification. Technical report, Department of Computer Science, National Taiwan University, 2007.
- R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: a library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. URL <http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf>.
- S. Khandagale, H. Xiao, and R. Babbar. Bonsai: diverse and shallow trees for extreme multi-label classification. *Machine Learning*, 109:2099–2119, 2020. doi: 10.1007/s10994-020-05888-2.
- M.-C. Lee, W.-L. Chiang, and C.-J. Lin. Fast matrix-vector multiplications for large-scale logistic regression on shared-memory systems. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2015. URL [http://www.csie.ntu.edu.tw/~cjlin/papers/multicore\\_liblinear\\_icdm.pdf](http://www.csie.ntu.edu.tw/~cjlin/papers/multicore_liblinear_icdm.pdf).
- D. D. Lewis, R. E. Schapire, J. P. Callan, and R. Papka. Training algorithms for linear text classifiers. *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 298–306, 1996.
- H.-Z. Lin, C.-H. Liu, and C.-J. Lin. Exploring space efficiency in a tree-based linear model for extreme multi-label classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 16245–16260, 2024. URL [https://www.csie.ntu.edu.tw/~cjlin/papers/multilabel\\_tree\\_model\\_size/multilabel\\_tree\\_model\\_size.pdf](https://www.csie.ntu.edu.tw/~cjlin/papers/multilabel_tree_model_size/multilabel_tree_model_size.pdf).
- H.-Z. Lin, Z.-B. Lu, S.-W. Chen, C.-H. Liu, and C.-J. Lin. On the weight density of l2-regularized linear classification and regression. In *Proceedings of the Twenty-Ninth Annual Conference on Artificial Intelligence and Statistics (AISTATS)*, 2026. URL <https://www.csie.ntu.edu.tw/~cjlin/papers/dual-space/dual-space.pdf>.
- L.-C. Lin, C.-H. Liu, C.-M. Chen, K.-C. Hsu, I.-F. Wu, M.-F. Tsai, and C.-J. Lin. On the use of unrealistic predictions in hundreds of papers evaluating graph representations. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*, pages 7479–7487, 2022. URL [https://www.csie.ntu.edu.tw/~cjlin/papers/multilabel-embedding/multilabel\\_embedding.pdf](https://www.csie.ntu.edu.tw/~cjlin/papers/multilabel-embedding/multilabel_embedding.pdf).
- Y.-J. Lin and C.-J. Lin. On the thresholding strategy for infrequent labels in multi-label classification. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*

- (*CIKM*), pages 1441–1450, 2023. doi: 10.1145/3583780.3614996. URL [https://www.csie.ntu.edu.tw/~cjlin/papers/thresholding/smooth\\_acm.pdf](https://www.csie.ntu.edu.tw/~cjlin/papers/thresholding/smooth_acm.pdf).
- J. Opitz and S. Burst. Macro F1 and Macro F1, 2021. arXiv preprint arXiv:1911.03347.
- S. A. P. Parambath, N. Usunier, and Y. Grandvalet. Optimizing F-measures by cost-sensitive classification. In *Advances in Neural Information Processing Systems*, volume 27, 2014.
- J. M. Tague. Information retrieval experiment. In K. S. Jones, editor, *The pragmatics of information retrieval experimentation*, chapter 5, pages 59–102. Butterworths, London, 1981.
- Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1/2): 69–90, 1999.
- H.-F. Yu, K. Zhong, J. Zhang, W.-C. Chang, and I. S. Dhillon. PECOS: Prediction for enormous and correlated output spaces. *Journal of Machine Learning Research*, 23(98):1–32, 2022.
- Y. Zhuang, Y. Juan, G.-X. Yuan, and C.-J. Lin. Naive parallelization of coordinate descent methods and an application on multi-core l1-regularized classification. In *Proceedings of the 27th ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, 2018. URL [http://www.csie.ntu.edu.tw/~cjlin/papers/l1\\_multicore\\_cikm.pdf](http://www.csie.ntu.edu.tw/~cjlin/papers/l1_multicore_cikm.pdf).