IEICE TRANS. INF. & SYST., VOL.E101–D, NO.11 NOVEMBER 2018

PAPER

Speeding up Extreme Multi-Label Classifier by Approximate Nearest Neighbor Search*

Yukihiro TAGAMI[†,††a)], *Member*

SUMMARY Extreme multi-label classification methods have been widely used in Web-scale classification tasks such as Web page tagging and product recommendation. In this paper, we present a novel graph embedding method called "AnnexML". At the training step, AnnexML constructs a k-nearest neighbor graph of label vectors and attempts to reproduce the graph structure in the embedding space. The prediction is efficiently performed by using an approximate nearest neighbor search method that efficiently explores the learned k-nearest neighbor graph in the embedding space. We conducted evaluations on several large-scale real-world data sets and compared our method with recent state-of-the-art methods. Experimental results show that our AnnexML can significantly improve prediction accuracy, especially on data sets that have a larger label space. In addition, AnnexML improves the trade-off between prediction time and accuracy. At the same level of accuracy, the prediction time of AnnexML was up to 58 times faster than that of SLEEC, a state-of-the-art embedding-based method.

key words: *extreme multi-label classification, k-nearest neighbor graph, approximate nearest neighbor search, learning-to-rank*

1. Introduction

Extreme multi-label classification has recently been receiving much attention. Its objective is to learn a classifier that can automatically annotate a data point with the most relevant subset from an extremely large label set (10^4 to 10^6) [2], [3]. In one example, an extreme multi-label classifier is learned to tag a new Wikipedia article by using a subset of the most relevant Wikipedia categories [4]. In another example, a classifier is built to display a subset of online advertisements to Web users, given the users' Web browsing histories and search keywords.

When a label space is extremely large, a traditional baseline approach that builds a one-versus-rest classifier for each label is computationally expensive [5], [6]. More specifically, this naive approach needs to train an extremely large number of binary classifiers. Thus, some methods of dealing with this problem attempt to reduce the effective number of labels. Some "embedding-based" approaches rely on the low-rank label matrix assumption [7]–[9]. This assumption means that there are correlations between labels, so these approaches learn a small number of "latent" factors of labels. Regressors are learned to perform prediction on these "latent" factors of labels with features and project them back onto the original high-dimensional label space. However, this assumption is violated in many real-world data sets because of the number of "tail" labels that only a few data points have [3], [9].

To address this problem, Sparse Local Embeddings for Extreme Classification (SLEEC) [3] was developed. SLEEC is also an embedding-based method, but is free of any low-rank label matrix assumptions. This method first partitions data points by using k-means clustering and then learns a projection matrix (or regressor) for each partition by preserving distances from a relatively small number of nearest neighbors in the label space. In other words, SLEEC reduces the effective number of labels by converting a multi-label classification problem into a set of regression problems by using nearest neighbors in the label space. The number of these regression problems is independent from the number of labels, whereas the above naive one-versus-rest approach needs to train the same number of classifiers as labels. Prediction is performed by using the labels of training points close to the test point in the embedding space. In other words, SLEEC uses a k-nearest neighbor classifier in the embedding space.

However, SLEEC also has three problems. The first problem is learning to partition data. SLEEC partitions training points with k-means clustering before learning embeddings. This means SLEEC only uses feature vectors and does not access label information in this procedure. Therefore, data points that have similar label vectors are not guaranteed to be assigned to the same partition. This partitioning could affect the quality of embeddings learned in subsequent steps. The second is learning embeddings. In the prediction step, SLEEC predicts labels of the test point by using the nearest training points in the embedding space, as described above. Hence, the order of distance plays a crucial role, whereas the values themselves are not very important. Thus, the objective function of SLEEC is somewhat indirect for this purpose. In addition, SLEEC's optimization process for learning regressors is slightly complicated because of sparsity-induced and rank-constraint regularization terms. The third problem is prediction speed. The authors [3] reported that SLEEC made predictions in 8 milliseconds per test point compared with 0.5 milliseconds for the tree-based FastXML [2] on a WikiLSHTC-325K data set, but SLEEC made predictions much more accurately than FastXML. Al-

Manuscript received March 15, 2018.
Manuscript revised July 3, 2018.
Manuscript publicized August 6, 2018.

[†]The author is with Yahoo Japan Corporation, Tokyo, 102–8282 Japan.
[††]The author is with Department of Intelligence Science and Technology, Kyoto University, Kyoto-shi, 606–8501 Japan.

*This study was presented at KDD2017 [1].

a) E-mail: yutagami@yahoo-corp.jp

DOI: 10.1587/transinf.2018EDP7096

though this prediction time would be acceptable for most real-world applications, much faster prediction is preferable for scaling up to solve Web-scale classification problems.

In this paper, we present a novel graph embedding method named "AnnexML[†]," which copes with the all three problems in a comprehensive and more direct way based on the k-nearest neighbor graph (KNNG). The key idea of AnnexML is reproducing the KNNG of label vectors in the embedding space to improve both the prediction accuracy and speed of the k-nearest neighbor classifier. The KNNG consists of training points as vertices. A directed edge connects from the i-th vertex to the j-th one if the j-th point is included in the set of the nearest neighbors of the i-th point in a certain metric space.

More specifically, AnnexML tackles the above three problems as follows. For the first problem, AnnexML learns a multiclass classifier, which partitions the approximate KNNG of the label vectors in order to preserve the graph structure as much as possible. Then, for the second problem, AnnexML projects the each divided subgraph into an individual embedding space by formulating this problem as a ranking problem instead of regression one. This objective function is easily optimized with simple stochastic gradient descent. For the third problem, AnnexML efficiently retrieves approximate nearest neighbors of a test point by exploring the learned KNNG in the embedding space. This technique improves the trade-off between prediction time and accuracy.

To summarize, our main contributions are as follows.

- We present a novel method of learning to partition data points by using an approximate KNNG as weak supervision, instead of unsupervised k-means clustering (Sect. 3.1)
- For learning embeddings, we formulate this problem as a ranking one and optimize the objective function by using simple stochastic gradient descent (Sect. 3.2)
- For faster prediction, we use an approximate k-nearest neighbor search technique by efficiently traversing the learned KNNG in the embedding space (Sect. 3.3)
- We conducted experiments on several large-scale real-world data sets and compared our method with recent state-of-the-art methods in terms of prediction accuracy and time (Sect. 4)

2. Problem Formulation

In this paper, we consider a data set $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, which consists of N training points, where $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^M$ is an M-dimensional feature vector and $\boldsymbol{y}_i \in \mathcal{Y} \subseteq \{0,1\}^L$ is the corresponding L-dimensional label vector. $y_{ij} = 1$ if the i-th sample has the j-th label, and $y_{ij} = 0$ otherwise. Multi-label learning aims to build a classifier, $f : \mathbb{R}^M \to \{0,1\}^L$, that accurately predicts the label vector for a given sample.

[†]**A**pproximate **N**earest **N**eighbor Search for **EX**treme **M**ulti-**L**abel Classification

Algorithm 1 Overview of training single learner

```
Require: Training data: $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$
1: Partition $\mathcal{D}$ into $K$ subsets $\mathcal{D}_1, \dots, \mathcal{D}_K$        ▷ Section 3.1
2: for each parition $c$ do
3:     Learn projection matrix $\boldsymbol{V}_c$ using $\mathcal{D}_c$        ▷ Section 3.2
4:     $\boldsymbol{Z}_c \leftarrow \boldsymbol{V}_c \boldsymbol{X}_c$
5: end for
6: return $\{(\mathcal{D}_1, \boldsymbol{V}_1, \boldsymbol{Z}_1), \dots, (\mathcal{D}_K, \boldsymbol{V}_K, \boldsymbol{Z}_K)\}$
```

Algorithm 2 Overview of prediction with single learner

```
Require: Test point: $\boldsymbol{x}_t$, number of nearest neighbors: $n$
1: $c_t \leftarrow$ partition closest to $\boldsymbol{x}_t$
2: $z_t \leftarrow \boldsymbol{V}_{c_t} \boldsymbol{x}_t$
3: $\tilde{\mathcal{N}}^{(t)}_{\boldsymbol{Z}_{c_t}} \leftarrow$ approximate $n$-nearest neighbors of $z_t$ in $\boldsymbol{Z}_{c_t}$        ▷ Section 3.3
4: $\hat{\boldsymbol{y}}_t \leftarrow$ empirical label distribution for points in $\tilde{\mathcal{N}}^{(t)}_{\boldsymbol{Z}_{c_t}}$
5: return $\hat{\boldsymbol{y}}_t$
```

In the context of extreme multi-label learning, the number of labels L is large and in the same order as the numbers of training points N and features M (see Table 2 for the statistics of the data sets used in the experiments). For example, a naive approach, which uses the one-versus-rest technique where an independent classifier is learned for each label, needs to train a massive number of binary classifiers. In addition, at the prediction time of this approach, all binary classifiers are applied to each test point. Thus, this naive approach might be computationally expensive in terms of both training and prediction time.

To overcome the above problem, some methods developed for extreme multi-label classification attempt to reduce the effective number of labels. We briefly describe these methods in Sect. 5.

3. Proposed Method

In this section, we introduce our proposed method, AnnexML.

Hereafter, we represent an index set of entire training points as $\mathcal{I} = \{1, 2, \dots, N\}$. Let $\boldsymbol{X} = [\boldsymbol{x}_1; \dots; \boldsymbol{x}_N] \in \mathbb{R}^{M \times N}$ be the data matrix and $\boldsymbol{Y} = [\boldsymbol{y}_1; \dots; \boldsymbol{y}_N] \in \mathbb{R}^{L \times N}$ be the label matrix.

First, we describe training and prediction overviews. These high-level overviews are similar to those of SLEEC [3]. However, we stress that AnnexML is a graph embedding method that copes with the three problems of SLEEC in a comprehensive and straightforward way based on KNNG, which consists of training points as vertices.

Algorithm 1 shows an overview of the training procedure. First, AnnexML splits the data points into K partitions and then learns a linear map $\boldsymbol{V}_c \in \mathbb{R}^{d \times M}$ that projects the data points to a low-dimensional subspace for each partition $c \in \{1, \dots K\}$, instead of a global projection map. The embedding vector $z_i \in \mathbb{R}^d$ is mapped from the feature vector $\boldsymbol{x}_i$ by using $\boldsymbol{V}_c$ for each $i \in \mathcal{I}_c$. Here, $\mathcal{I}_c$ is the index set of data points in the partition c. The data matrix of partition c is denoted as $\boldsymbol{X}_c$. In a similar way, $\boldsymbol{Y}_c$ and $\boldsymbol{Z}_c$ represent the

Table 1 Label frequency of training data used in experiments. For example, more than 174,000 labels occur in at most five training points in WikiLSHTC-325K data set.

Frequency	WikiLSHTC-325K		Amazon-670K	
1	79,732	(24.82%)	71,817	(10.76%)
≤ 2	112,788	(35.11%)	309,976	(46.45%)
≤ 3	137,596	(42.84%)	435,442	(65.25%)
≤ 4	157,541	(49.04%)	509,203	(76.31%)
≤ 5	174,341	(54.27%)	555,905	(83.30%)
≤ 10	226,956	(70.65%)	637,379	(95.51%)
All	321,222	(100.00%)	667,317	(100.00%)

label and embedding matrix, respectively. AnnexML is able to improve its prediction accuracy by learning an ensemble of multiple learners with different random initializations, as described later (Sect. 3.1).

Algorithm 2 shows an overview of the prediction procedure. AnnexML first determines the partition to which a test point $\boldsymbol{x}_t$ belongs, finds approximate k-nearest neighbors (training points) in the low-dimensional subspace corresponding to the partition c_t, and finally predicts labels on the basis of the labels of neighbors. To make predictions with an ensemble of multiple learners, first, all sets of nearest neighbors obtained by learners are aggregated; then, AnnexML outputs an empirical label distribution of points in the aggregated nearest neighbors.

In the following subsections, we give details on the training and prediction procedures. The key point of AnnexML is reproducing the structure of the KNNG of the label vectors as much as possible in the embedding space. In Sect. 3.1, we represent a novel method of learning to partition data. AnnexML learns a multiclass classifier to divide the KNNG of the label vectors into subgraphs. Then, in Sect. 3.2, we learn a projection matrix in order to preserve the edge connections of each subgraph in an individual embedding space. Regarding this problem as ranking one, we apply a learning-to-rank technique that uses cosine similarity. Finally, in Sect. 3.3, we present an approximate nearest neighbor search method that efficiently explores the learned KNNG in the embedding space. Replacing the naive brute-force calculation with this method, AnnexML successfully speeds up the prediction time without a noticeable drop in accuracy.

3.1 Learning to Partition Data Points

AnnexML partitions data points before learning embeddings for efficient training and prediction, like SLEEC does. While SLEEC simply uses k-means clustering for this purpose, AnnexML aims to allocate the data points that have similar label vectors to the same partition. This means that AnnexML utilizes label information $\boldsymbol{y}$ but SLEEC only accesses feature vectors $\boldsymbol{x}$. The label frequency of an extreme multi-label classification data set follows a "heavy tailed" distribution [3], [9]. Table 1 shows the label frequency in the training data of WikiLSHTC-325K and Amazon-670K data sets. About 54% and 83% of the labels occur in at most five training points on each data set. Without label information, the data points that have the same tail label might be allocated to different partitions as the number of partitions increases. Thus, this difference between AnnexML and SLEEC could affect the quality of the embeddings learned in the subsequent steps and the final prediction results.

From the perspective of reproducing the KNNG of the label vectors, AnnexML divides the graph into K subgraphs while keeping the structure as much as possible. Hence, this problem can be regarded as finding the minimum K-way graph cut [10]. However, in contrast with the common minimum K-cut problem, we need to learn a multi-class classifier to predict the partition of an unknown test point. Thus, we use a novel sequential maximization procedure for learning the classifier.

To construct the KNNG of the label vectors, we find the nearest neighbors $\mathcal{N}_{\boldsymbol{Y}}^{(i)}$ for each i-th data point from the indices of all training points $\mathcal{I}$. The set of nearest neighbors $\mathcal{N}_{\boldsymbol{Y}}^{(i)}$ is defined by using the inner product between normalized label vectors $\boldsymbol{y}/|\boldsymbol{y}|$ as:

$$\mathcal{N}_{\boldsymbol{Y}}^{(i)} := \underset{S:S\subseteq\mathcal{I},|S|=n,i\notin S}{\arg\max} \sum_{j\in S} \frac{\boldsymbol{y}_i^{\mathrm{T}}\boldsymbol{y}_j}{|\boldsymbol{y}_i||\boldsymbol{y}_j|}, \tag{1}$$

where S is the index set in which the number of elements equals n and $|\boldsymbol{y}_i| = \sum_j y_{ij}$ is the number of labels that a data point has.

The computational cost of naively finding $\mathcal{N}_{\boldsymbol{Y}}^{(i)}$ for all data points is $O(N^2)$. Thus, this naive approach is infeasible for a large $N \sim 10^6$. If the label vectors $\boldsymbol{y}$ are sufficiently sparse, we can efficiently find the nearest neighbors $\mathcal{N}_{\boldsymbol{Y}}^{(i)}$ by using an inverted index. In this case, we first construct a list of references of data points that have each label, and then we just have to evaluate pairs of data points in each list. This procedure is similar to calculating the dot product between two sparse vectors, which only considers dimensions where both vectors have nonzero elements and sums up the products of their values. The estimated computational cost is $\sum_{j=1}^{L} n_j(n_j-1)/2$, where n_j is the number of data points that have the j-th label or the size of the j-th list. However, if a few n_j corresponding to "head" labels are near N, which means almost all data points have the same label, the above cost reaches $O(N^2)$. Efficient algorithms for top-k retrieval on an inverted index [11] or finding approximate k-nearest neighbors [12] can be applied to this situation. However, we focus on tail labels and ignore head labels in some cases. In other words, we expect that the number of data points corresponding to head labels in each partition is enough to not affect the subsequent step, even if head labels are ignored in this step. Thus, we only consider the l tail labels and their corresponding lists under the conditions $\sum_{j=1}^{l} n_j(n_j-1)/2 \leq \alpha N$, $n_1 \leq n_2 \leq \ldots \leq n_L$ to find approximate nearest neighbors $\tilde{\mathcal{N}}_{\boldsymbol{Y}}^{(i)}$ in order to keep the computational cost within $O(N)$ by using the adjusting parameter $\alpha \ll N$. Note that it is easy to determine l that satisfies the above condition by sorting n_js and accumulating $n_j(n_j-1)/2$ in ascending order.

After approximate nearest neighbors $\tilde{\mathcal{N}}_{\boldsymbol{Y}}^{(i)}$ are obtained for all data points, we learn the weight vector $\boldsymbol{w}_c$ for each partition c in order that the data points having the same tail labels are allocated in the same partition while the data points are divided almost equally among partitions. Using a sequential maximization procedure like stochastic k-means clustering [13], we sequentially maximize the following objective function for each i-th sample.

$$\max_{\boldsymbol{w}_{c_i}} \sum_{j \in \tilde{\mathcal{N}}_{\boldsymbol{Y}}^{(i)}} \log \sigma(\boldsymbol{w}_{c_i}^{\mathrm{T}} \boldsymbol{x}_j) + \sum_{k \in S^-} \log \sigma(-\boldsymbol{w}_{c_i}^{\mathrm{T}} \boldsymbol{x}_k) - \lambda |\boldsymbol{w}_c|_1, \quad (2)$$

where $c_i = \arg\max_c \boldsymbol{w}_c^{\mathrm{T}} \boldsymbol{x}_i$ is the partition to which the i-th point belongs at this time step, S^- is the set of indices randomly selected from $\mathcal{I}$, $\sigma(z) = 1/(1+\exp(-z))$ is a sigmoid function, and λ is a regularization parameter. We learn the linear classifier $\boldsymbol{w}_c$'s by using the FTRL-Proximal algorithm [14] with AdaGrad [15] learning rate adjustment.

The first term of the above objective function is intended to assign the approximate nearest neighbors $\tilde{\mathcal{N}}_{\boldsymbol{Y}}^{(i)}$ to the same partition c_i to which the i-th point belongs. The second term aims for the randomly selected points S^- to not be included in the partition c_i. Since some partitions that have a lot of data points cause the training and prediction time to become long, this term prevents a lot of data points from being allocated in a single partition. The last term is the L_1-regularization term to make $\boldsymbol{w}_c$'s sparse. Sparse $\boldsymbol{w}_c$'s are also preferable for faster prediction.

The above L_1-regularization term reduces the final model size as well as the prediction time. At training time, $\boldsymbol{w}_c$'s are favored to to be stored in dense vectors for faster training. These vectors use $O(KM)$ space. Note that the number of partitions K depends on the number of entire training points N (see Sect. 4). Thus, this space complexity is infeasible for large N and M. To remedy this problem, we use a hashing trick [16] for storing $\boldsymbol{w}_c$'s in 24-bit space (approximately 16.7 millions of bins) instead of $O(KM)$ space. In practice, we did not notice a drop in accuracy when using this technique.

Since the above objective function is non-convex (when c_i is not fixed), AnnexML is also able to improve its prediction accuracy by learning an ensemble of multiple learners with different random initializations of $\boldsymbol{w}_c$, like the k-means clustering of SLEEC.

At the prediction step (line 1 in Algorithm 2), the partition c_t to which a test point $\boldsymbol{x}_t$ belongs is determined by using the learned $\boldsymbol{w}_c$'s as $c_t = \arg\max_c \boldsymbol{w}_c^{\mathrm{T}} \boldsymbol{x}_t$.

3.2 Learning Embeddings

For learning embedding vector z_i and projection matrix $\boldsymbol{V}_c$, AnnexML employs a pairwise learning-to-rank approach. This is because the learning objective of AnnexML is to reconstruct the KNNG of label vectors in the embedding space. In other words, arranging the k-nearest neighbors of the i-th sample is regarded as a ranking problem if we consider an i-th point and other points as a query and items, respectively. Thus, we use the pairwise learning-to-rank approach similar to S2Net [17] and DSSM [18] in order to optimize k-nearest neighbors in the embedding space more directly.

To represent the objective function of AnnexML, we first define the relevance score between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ as cosine similarity between embedding vectors $z_i = \boldsymbol{V}_c \boldsymbol{x}_i$ and $z_j = \boldsymbol{V}_c \boldsymbol{x}_j$:

$$R(\boldsymbol{x}_i, \boldsymbol{x}_j) := \cos(z_i, z_j) = \frac{z_i^{\mathrm{T}} z_j}{\|z_i\| \|z_j\|} = \frac{\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{V}_c^{\mathrm{T}} \boldsymbol{V}_c \boldsymbol{x}_j}{\|\boldsymbol{V}_c \boldsymbol{x}_i\| \|\boldsymbol{V}_c \boldsymbol{x}_j\|}.$$

We also represent the conditional probability by using the above relevance score and softmax function as follows:

$$\begin{aligned} P(\boldsymbol{x}_j \mid \boldsymbol{x}_i) &:= \frac{\exp(\gamma R(\boldsymbol{x}_i, \boldsymbol{x}_j))}{\exp(\gamma R(\boldsymbol{x}_i, \boldsymbol{x}_j)) + \sum_{k \in S_c^-} \exp(\gamma R(\boldsymbol{x}_i, \boldsymbol{x}_k))} \\ &= \frac{\exp(\gamma \cos(z_i, z_j))}{\exp(\gamma \cos(z_i, z_j)) + \sum_{k \in S_c^-} \exp(\gamma \cos(z_i, z_k))}, \end{aligned}$$

where γ is a scaling parameter that magnifies $\cos(z_i, z_j)$ from $[-1, +1]$ to the range of larger values and $S_c^- \subseteq \mathcal{I}_c$ is the set of indices randomly selected from data points in the corresponding partition c. Then, we minimize the negative log likelihood:

$$\min_{\boldsymbol{V}_c} \sum_{i \in \mathcal{I}_c} \sum_{j \in \mathcal{N}_{\boldsymbol{Y}_c}^{(i)}} -\log P(\boldsymbol{x}_j \mid \boldsymbol{x}_i).$$

Here, the set of nearest neighbors $\mathcal{N}_{\boldsymbol{Y}_c}^{(i)}$ is almost the same as $\mathcal{N}_{\boldsymbol{Y}}^{(i)}$ (see Eq. (1)) but is selected from $\mathcal{I}_c$ rather than $\mathcal{I}$. Note that the computational cost of finding exact $\mathcal{N}_{\boldsymbol{Y}_c}^{(i)}$ for all i is not very large since the number of data points in a cluster $|\mathcal{I}_c|$ is much smaller than that of all training points $N = |\mathcal{I}|$ (the number of clusters to which K should be set to fulfill this condition). Thus, we can use the exact $\mathcal{N}_{\boldsymbol{Y}_c}^{(i)}$ for learning embeddings. The above negative log likelihood is transformed as:

$$\begin{aligned} &-\log P(\boldsymbol{x}_j \mid \boldsymbol{x}_i) \\ &= -\log\left(\frac{\exp(\gamma \cos(z_i, z_j))}{\exp(\gamma \cos(z_i, z_j)) + \sum_{k \in S_c^-} \exp(\gamma \cos(z_i, z_k))}\right) \\ &= -\log\left(\frac{1}{1 + \sum_{k \in S^-} \exp(\gamma(\cos(z_i, z_k) - \cos(z_i, z_j)))}\right) \\ &= \log\left(1 + \sum_{k \in S_c^-} \exp(-\gamma \Delta_{ijk})\right), \end{aligned}$$

where $\Delta_{ijk} = \cos(z_i, z_j) - \cos(z_i, z_k)$ is the difference between two cosine values. Hence, this objective function aims to increase the difference between these cosine values.

We learn the projection matrix $\boldsymbol{V}_c$ by using stochastic gradient descent with AdaGrad [15]. Let a, b, and c be $z_i^{\mathrm{T}} z_j$, $1/\|z_i\|$, and $1/\|z_j\|$, respectively. The gradient of the $\cos(z_i, z_j)$ is derived as:

$$\frac{\partial \cos(z_i, z_j)}{\partial \boldsymbol{V}_c} = \frac{\partial}{\partial \boldsymbol{V}_c}\left(\frac{z_i^{\mathrm{T}} z_j}{\|z_i\| \|z_j\|}\right)$$

Algorithm 3 ANNSearch(Query $\boldsymbol{q}$, Ball tree T, KNNG G)

```
1: $H \leftarrow$ empty heap
2: TreeSearch($\boldsymbol{q}$, $T$.root, $H$)
3: GraphSearch($\boldsymbol{q}$, $G$, $H$)
4: return $H$
```

$$= -abc^3 z_i \boldsymbol{x}_i^{\mathrm{T}} - acb^3 z_j \boldsymbol{x}_j^{\mathrm{T}} + bc(z_i \boldsymbol{x}_j^{\mathrm{T}} + z_j \boldsymbol{x}_i^{\mathrm{T}}).$$

Using cosine similarity instead of the inner product has some advantages. First, the objective function is regularized by the normalizing factor of a cosine. Hence, the explicit regularization term of $\boldsymbol{V}_c$ is not needed, so this learning procedure is simple. Second, we can easily apply an approximate nearest neighbor search technique that uses the learned KNNG for speeding up the prediction. We give the details of this technique in Sect. 3.3.

3.3 Faster Prediction using Approximate Nearest Neighbor Search on KNNG

The prediction of AnnexML mostly relies on the k-nearest neighbor search in the embedding space (see Algorithm 2). Thus, for faster prediction, we need to speed up this nearest neighbor search task. Instead of SLEEC's naive brute-force search, we apply an approximate k-nearest neighbor search method to this task. This method efficiently explores the learned KNNG in the embedding space by using an additional tree structure and a pruning technique via the triangle inequality. Note that we construct the KNNG precisely in the training phase and perform an approximate and fast search on the graph in prediction phase.

Representing the nearest neighbor search task more concretely, we find the following index set $\mathcal{N}_{\boldsymbol{Z}_c}^{(t)}$ from training points $\mathcal{I}_c$ in a certain partition c with cosine similarity:

$$\mathcal{N}_{\boldsymbol{Z}_c}^{(t)} = \underset{S:S\subseteq\mathcal{I}_c,|S|=n}{\arg\max} \sum_{j\in S} \cos(z_t, z_j) = \underset{S:S\subseteq\mathcal{I}_c,|S|=n}{\arg\max} \sum_{j\in S} \frac{z_t^{\mathrm{T}} z_j}{\|z_j\|},$$

where $z_t = \boldsymbol{V}_c \boldsymbol{x}_t$ is the embedding vector that corresponds to a test point $\boldsymbol{x}_t$.

For efficient indexing in a metric space, the triangle inequality is a key element, as suggested by Sugawara et al. [19]. We also want to utilize the triangle inequality for efficient retrieval, but the cosine distance $1 - \cos(z_t, z_j)$ cannot satisfy this inequality. Fortunately, if the indexed vectors z_j are normalized in advance, that is, $\|z_j\| = 1$ for all j, the above $\mathcal{N}_{\boldsymbol{Z}_c}^{(t)}$ becomes the same as the following index set of nearest neighbors by using Euclidean distance:

$$\underset{S:S\subseteq\mathcal{I}_c,|S|=n}{\arg\min} \sum_{j\in S} \|z_t - z_j\|^2 = \underset{S:S\subseteq\mathcal{I}_c,|S|=n}{\arg\min} \sum_{j\in S} \left(\|z_j\|^2 - 2z_t^{\mathrm{T}} z_j\right).$$

Thus, we use Euclidean distance between normalized vectors as metrics for searching. Note that this transformation does not change the structure (or edge connections) of the learned KNNG.

Algorithms 3, 4, 5, and 6 are the pseudo codes that represent the approximate k-nearest neighbor search procedure using the KNNG. In these pseudo codes, the embedding of a test point z_t is represented as query vector $\boldsymbol{q}$. To efficiently find reasonable starting points from the graph, our method combines the KNNG with a ball tree. Please refer to the papers [20], [21] for the indexing method of the ball tree.

Algorithm 4 TreeSearch(Query $\boldsymbol{q}$, Tree node N, Heap H)

```
1: if $N$ is a leaf node then
2:     LineSearch($\boldsymbol{q}$, $N.S$, $H$)
3: else
4:     ldist $\leftarrow d(\boldsymbol{q}, N$.left.center)
5:     rdist $\leftarrow d(\boldsymbol{q}, N$.right.center)
6:     if ldist < rdist then
7:         TreeSeach($\boldsymbol{q}$, $N$.left, $H$)      ▹ follow the left child node
8:     else
9:         TreeSeach($\boldsymbol{q}$, $N$.right, $H$)     ▹ follow the right child node
10:    end if
11: end if
```

Algorithm 5 LineSearch(Query $\boldsymbol{q}$, Index set S, Heap H)

```
1: for $i \in S$ do
2:     dist $\leftarrow d(\boldsymbol{q}, z_i)$
3:     if dist < LargestDistance($H$) then
4:         PopAndPushHeap($H$, $i$, dist)
5:     end if
6: end for
```

Algorithm 6 GraphSearch(Query $\boldsymbol{q}$, KNNG G, Heap H)

```
1: $C \leftarrow$ index of $H$                  ▹ a queue of candidates
2: $D \leftarrow$ empty set                     ▹ a set of already evaluated
3: while $C$ is not empty do
4:     $i \leftarrow$ pop from $C$
5:     if $i$ in $D$ then
6:         continue
7:     end if
8:     $D \leftarrow D \cup \{i\}$
9:     dist $\leftarrow d(\boldsymbol{q}, z_i)$
10:    if dist < LargestDistance($H$) then
11:        PopAndPushHeap($H$, $i$, dist)
12:        for $j$ in $\mathcal{N}_{\boldsymbol{Z}_c}^{(i)}$ do       ▹ nearest neighbors in $G$
13:            push $j$ to $C$
14:        end for
15:    end if
16: end while
```

A query $\boldsymbol{q}$ first traverses from the root to the leaf nodes of the ball tree (Algorithm 4). At each internal node, the test point determines which child node (left or right) is to be followed by using the distances from the centers of the balls. After the test point reaches a leaf node, the distance from each indexed point corresponding to the node is calculated and pushed into the heap (Algorithm 5). Using these data points as seeds, the KNNG is explored (Algorithm 6). In the exploration step, if a data point is satisfied with the condition and pushed into the heap, the nearest neighbors are also pushed into the queue of candidates for the subsequent evaluation.

To summarize, we first obtain a reasonable set of approximate nearest neighbors by using a ball tree and then improve the approximation quality by exploring the KNNG on the basis of local search. Since we only need to calculate the distances to a small number of balls' centers and subset of training points, we can speed up this search task.

3.4 Comparison with SLEEC

In this subsection, we clarify the difference between AnnexML and SLEEC.

As noted in the beginning part of this section, the training and prediction procedures of AnnexML are similar to those of SLEEC. In other words, Algorithms 1 and 2 also show overviews of the training and prediction procedures of SLEEC, respectively. However, there are three improvements of AnnexML compared to SLEEC, which are described in the above subsections.

For partitioning data points (line 1 in Algorithm 1), SLEEC simply applies usual k-means clustering to feature vectors of data points whereas AnnexML learns the classifiers by considering both feature and label vectors. Therefore, AnnexML is more likely to allocate the data points that have similar label vectors to the same partition and improve the quality of embeddings learned in subsequent steps.

For learning embeddings (line 3 in Algorithm 1), SLEEC utilizes the following objective function:

$$\min_{\boldsymbol{V}_c} \sum_{i \in \mathcal{I}_c} \sum_{j \in \mathcal{N}_{Y_c}^{(i)}} \left\| \boldsymbol{y}_i^{\mathrm{T}} \boldsymbol{y}_j - \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{V}_c^{\mathrm{T}} \boldsymbol{V}_c \boldsymbol{x}_j \right\|^2 + \lambda \sum_{i \in \mathcal{I}_c} |\boldsymbol{V}_c \boldsymbol{x}_i|_1 + \mu \|\boldsymbol{V}_c\|_F^2 .$$

The first term indicates the sum of squared errors between the inner product of label vectors and that of embedding vectors. The second term is an L_1-regularization term for embedding vectors $z_i = \boldsymbol{V}_c \boldsymbol{x}_i$, which leads to sparse embeddings for reducing the size of models and the prediction time as well as avoiding overfitting. The last is the L_2-regularization term of $\boldsymbol{V}_c$. λ and μ are regularization parameters corresponding to the second and last terms, respectively. This objective function is non-convex and non-differentiable. Thus, the optimization process is divided into two phases and done using singular value projection and ADMM.

Obviously, the above objective function of SLEEC is a regression problem of minimizing the sum of squared errors. This means SLEEC learns the projection matrix $\boldsymbol{V}_c$ to reproduce the inner product values of label vectors by using embedding vectors. However, at the prediction step, these learned values are used only for retrieving k-nearest neighbors. In contrast, the objective function of AnnexML focuses on whether a data point is included in the set of k-nearest neighbors or not. Thus, the learning procedure of AnnexML is more intuitive and consistent with the prediction procedure.

Instead of the approximate nearest neighbors search of AnnexML (line 3 in Algorithm 2), SLEEC just performs an exact and brute-force search for prediction. Note that the brute-force calculation of SLEEC cannot be directly replaced with our approximate search procedure because it does not use cosine similarity (or Euclidean distance) but rather the inner product as the metric in the embedding space. Some Maximum Inner-Product Search (MIPS) methods [21]–[23] might be applicable to SLEEC for faster prediction. However, such a study is beyond the scope of this paper. We leave this direction as future work.

4. Experiments

In this section, we evaluated our method on six large scale multi-label data sets. These data sets were provided by the Extreme Classification Repository [24] and had already been pre-processed and separated into training and test sets. We did not use any additional meta data. The statistics for the data sets are summarized in Table 2.

We compared AnnexML with several state-of-the-art methods: SLEEC [3], FastXML [2], PfastreXML [25], PLT [26], and PD-Sparse [5]. To represent the prediction difficulty of each data set, we also show the performance of a naive baseline that makes predictions by using the k-most common labels in the training data.

We evaluated the performance of the methods with precision at k ($k \in \{1, 3, 5\}$), which is a widely used metric for extreme multi-label classification and ranking tasks:

$$\mathrm{P}@k := \frac{1}{kN_{test}} \sum_{i=1}^{N_{test}} \sum_{l=1}^{k} y_{i\pi(l)}.$$

Here, $\pi(k) = j$ means that the j-th label is ranked in the k-th position by the predicted score. We also evaluated the performances with normalized Discounted Cumulative Gain (nDCG) at k. However, since the results showed the same tendency, we only report those with P@k. At least, by definition, the value of nDCG@1 is the same as that of P@1.

We implemented AnnexML in C++. In all experiments, we used ten default hyper-parameters for training

Table 2 Statistics of datasets used in experiments.

Dataset	#Train N	#Test N_{test}	#Features M	#Labels L	Avg. points per label	Avg. labels per point
AmazonCat-13K	1,186,239	306,782	203,882	13,330	448.57	448.57
Wiki10-31K	14,146	6,616	101,938	30,938	8.52	18.64
Delicious-200K	196,606	100,095	782,585	205,443	72.29	75.54
WikiLSHTC-325K	1,778,351	587,084	1,617,899	325,056	17.46	3.19
Wikipedia-500K	1,813,391	783,743	2,381,304	501,070	24.75	4.77
Amazon-670K	490,449	153,025	135,909	670,091	3.99	5.45

Table 3 Experimental results. Unavailable values are denoted as "–" (see text for details).

Dataset		AnnexML	SLEEC	FastXML	PfastreXML	PLT	PD-Sparse	Most common
AmazonCat-13K	P@1	**0.9355**	0.8919	0.9310	0.8994	0.9147	0.8931	0.2988
	P@3	**0.7838**	0.7517	0.7818	0.7724	0.7584	0.7403	0.1878
	P@5	0.6332	0.6109	0.6338	**0.6353**	0.6102	0.6011	0.1486
Wiki10-31K	P@1	**0.8650**	0.8554	0.8295	0.8263	0.8434	0.7771	0.8079
	P@3	**0.7428**	0.7359	0.6756	0.6874	0.7234	0.6573	0.5050
	P@5	**0.6419**	0.6310	0.5770	0.6006	0.6272	0.5539	0.3675
Delicious-200K	P@1	0.4666	**0.4703**	0.4320	0.3762	0.4537	0.3437	0.3873
	P@3	0.4079	**0.4167**	0.3868	0.3562	0.3894	0.2948	0.3675
	P@5	0.3764	**0.3888**	0.3621	0.3403	0.3588	0.2704	0.3552
WikiLSHTC-325K	P@1	**0.6336**	0.5557	0.4975	0.5810	0.4567	0.6126	0.1588
	P@3	**0.4066**	0.3306	0.3310	0.3761	0.2913	0.3948	0.0603
	P@5	**0.2979**	0.2407	0.2445	0.2769	0.2195	0.2879	0.0380
Wikipedia-500K	P@1	**0.6386**	0.5839	0.4934	0.5891	–	–	0.1529
	P@3	**0.4269**	0.3788	0.3351	0.3937	–	–	0.0583
	P@5	**0.3237**	0.2821	0.2586	0.3005	–	–	0.0368
Amazon-670K	P@1	**0.4208**	0.3505	0.3697	0.3919	0.3665	0.3370	0.0028
	P@3	**0.3665**	0.3125	0.3332	0.3584	0.3212	0.2962	0.0027
	P@5	0.3276	0.2856	0.3053	**0.3321**	0.2885	0.2684	0.0023

AnnexML: the number of partitions in a learner: $K = \lfloor N/6000 \rfloor$, the embedding dimension: $d = 50$, the number of learners: 15, the number of (approximate) nearest neighbors and randomly sampled points used in learning both partitionings and embeddings: $n = 10$, the number of epochs for learning both partitionings and embeddings: 10, the initial learning rate of AdaGrad: $\eta_0 = 0.1$, the L_1-regularization parameter in Eq. (2): $\lambda = 4$, the scaling parameter for cosine: $\gamma = 10$, the adjustment parameter for finding approximate nearest neighbors: $\alpha = 5000$, and the number of edges in the KNNG for prediction: 50. These default values of hyper-parameters were determined in preliminary experiments with small-scale data sets. Thus, we avoided hyper-parameter tuning for the large-scale data sets and significantly reduced the total training time.

For the other baseline methods, if P@k on each data set is reported in the original papers, we used those values for fair comparison. Otherwise, we used the C++ and MATLAB implementations for SLEEC, FastXML, PfastreXML and PD-Sparse, provided by the authors [24]. In this case, the suggested hyper-parameters were used. Since we use ordinary (not propensity scored) precision at k as the evaluation metrics, the propensity scores of PfastreXML were set to the same value for all labels. For PLT, we referred only to the values reported in the original paper because hyper-parameters were tuned by an off-the-shelf optimizer in the experiments.

4.1 Results

The experimental results are summarized in Table 3. The **bold** elements indicate the best performance of the methods. For PLT, since we referred only to the values reported in the original paper [26], as mentioned above, the scores on Wikipedia-500K data set are not available. The scores of PD-Sparse on three data sets are also unavailable because of excessive memory usage and training time in our experimental setting. Thus, we referred to the scores on Delicious-200K and WikiLSTHC-325K data sets reported in the Extreme Classification Repository [24].

Table 4 Comparing results of our three improvements proposed in Sect. 3.

		E	E+P	E+A	E+P+A
Learning **E**mbeddings		✓	✓	✓	✓
Learning to **P**artition data		✗	✓	✗	✓
ANN search for prediction		✗	✗	✓	✓
AmazonCat-13K	P@1	**0.9381**	0.9353	0.9374	0.9355
	P@3	0.7835	**0.7853**	0.7809	0.7838
	P@5	0.6320	**0.6353**	0.6289	0.6332
Wiki10-31K	P@1	**0.8703**	0.8655	0.8690	0.8650
	P@3	**0.7448**	0.7431	0.7446	0.7428
	P@5	**0.6454**	0.6435	0.6447	0.6419
Delicious-200K	P@1	**0.4670**	0.4660	0.4669	0.4666
	P@3	0.4018	0.4038	0.4077	**0.4079**
	P@5	0.3705	0.3717	**0.3770**	0.3764
WikiLSHTC-325K	P@1	0.6108	**0.6378**	0.6024	0.6336
	P@3	0.3907	**0.4102**	0.3836	0.4066
	P@5	0.2875	**0.3008**	0.2819	0.2979
Wikipedia-500K	P@1	0.6323	**0.6410**	0.6215	0.6386
	P@3	0.4156	**0.4299**	0.4048	0.4269
	P@5	0.3144	**0.3261**	0.3056	0.3237
Amazon-670K	P@1	0.3670	**0.4248**	0.3631	0.4208
	P@3	0.3190	**0.3698**	0.3161	0.3665
	P@5	0.2862	**0.3309**	0.2834	0.3276

AnnexML performed the best in almost all cases. There were especially large improvements for data sets that have larger label spaces. For example, AnnexML improved over SLEEC by about 8% and 6% in absolute terms of P@1 and P@5 on WikiLSHTC-325K data set. On Amazon-670K data set, AnnexML was also superior to SLEEC by approximately 7% and 4% in absolute terms of P@1 and P@5. These substantial improvements indicate AnnexML is not just a minor updated version of SLEEC. There is still much room to improve the prediction accuracy of AnnexML by tuning the hyper-parameters for each data set, like SLEEC does. However, we leave this as future work.

Table 4 compares the results of the three improvements proposed in Sect. 3. Method `E+P+A` equals AnnexML in Table 3. If we did not use the proposed partitioning or approximate nearest neighbor search, we used k-means clus-

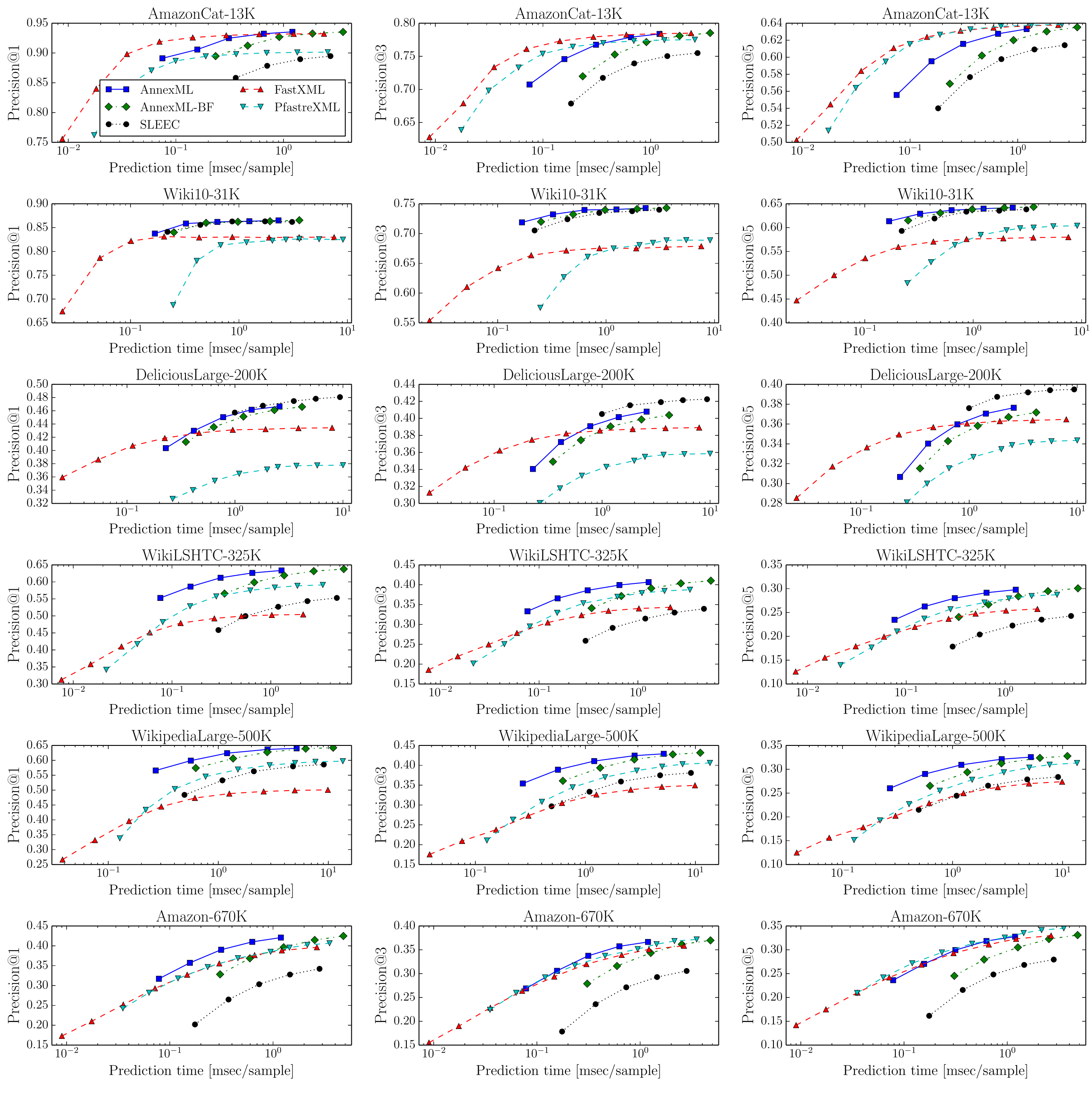


Fig. 1 Precision versus prediction time when the number of learners changes. Number of learners was {1, 2, 4, 8, 15} for AnnexML, AnnexML-BF, and SLEEC, and {1, 2, 4, . . . , 128, 256} for FastXML and PfastreXML. This experiment was conducted with C++ implementations on single CPU thread of the same machine for fair comparison.

tering and a brute-force cosine calculation, like SLEEC does. Method E outperformed SLEEC on almost all data sets. Thus, our learning procedure that uses the learning-to-rank approach successfully improved the embedding quality. In comparison with E and E+P, the proposed partitioning method significantly improved P@k on the data sets that have a larger label space. In comparison with E and E+A, the approximate nearest neighbor search technique for faster prediction had slightly poorer accuracy on all data sets except Delicious-200K. These results are consistent with our objective of proposing method A, which is not to make prediction more accurate, but rather faster. One possible explanation for the unexpected improvement on Delicious-200K was that the number of nearest neighbors retrieved for prediction was small (set to be 10 for all data sets). The number suggested by SLEEC's authors is 70 for this data set. Hence, the approximate nearest neighbor search retrieved more diverse samples, and these samples could fortunately stabilize the prediction result. Next, we present the speed-up effect of the approximate nearest neighbor search.

Figure 1 plots the prediction time and performances of AnnexML, AnnexML-BF, SLEEC, FastXML, and PfastreXML when the number of learners changes. A higher precision at the same prediction time (upper left line) indicates better results. This experiment was conducted by using a single CPU thread on a machine with two Xeon E5-2680v3 processors and 128 GB of RAM that ran the Linux operating system. AnnexML-BF is the same method as `E+P` in Table 4, which uses a brute-force cosine calculation for prediction. Note that AnnexML is the same as `E+P+A`. We chose SLEEC, FastXML and PfastreXML for comparison since these methods are also ensemble methods, and we can easily control the trade-off between prediction time and accuracy by just changing the number of learners. To conduct a fair comparison, we used our own C++ implementation for SLEEC's prediction, instead of the provided MATLAB codes. This is almost the same as the implementation of AnnexML-BF. For FastXML and PfastreXML, we used the provided C++ implementations with careful coding optimizations for improving efficiency.

Comparing AnnexML and AnnexML-BF, we see that the approximate nearest neighbor search technique successfully sped up prediction time with a slight drop in accuracy. On WikiLSHTC-325K data set, AnnexML made predictions more than four times faster than AnnexML-BF when using the same number of learners. In other words, the prediction time with an ensemble of four AnnexML learners was almost the same as that of a single AnnexML-BF model. In this case, AnnexML made predictions in approximately 0.34 milliseconds per test point and achieved an about 5% higher P@1 than that of AnnexML-BF in absolute terms (0.6121 vs. 0.5657). Compared with FastXML and PfastreXML, AnnexML also achieved a higher P@k at a 1 millisecond budget per test point in almost all cases. AnnexML achieved the same prediction accuracy as SLEEC with an ensemble of at most four learners, which also made predictions within 1 millisecond per test point. In particular, on WikiLSHTC-325K data set, a single model of AnnexML achieved almost the same P@1 as an ensemble of 15 SLEEC learners. This AnnexML's prediction time was about 58 times faster than that of SLEEC (0.08 milliseconds vs. 4.66 milliseconds).

5. Related Work

Extreme multi-label learning typically follows two major types of approaches: tree based [2], [25], [26] and embedding based [3], [7]–[9].

Tree-based methods are common in extreme multi-label classification because of their fast prediction. FastXML [2] is a state-of-the-art tree-based classifier. At the training phase, FastXML recursively partitions the feature space corresponding to the internal node by using a linear classifier optimized for nDCG-based ranking loss. A test point traverses the tree from the root node to a leaf node, and FastXML then makes predictions by using the labels of training points that correspond to the leaf node. PfastreXML [25] is an improved version of FastXML, made by replacing the nDCG loss with its propensity scored variant and using additional classifiers designed for tail labels. Jasinska et al. [26] developed PLT, which is a tree-based classifier that maximizes the F-measure. More recently, we gave a brief overview of a simple tree-based approach in a work-in-progress paper [27].

Most embedding-based approaches reduce the effective number of labels on the basis of the low-rank label matrix assumption. LEML [7] learns a low-rank projection matrix, which maps features to labels, by using a generic empirical risk minimization framework. Mineiro and Karampatziakis developed another embedding-based method, named "Rembrandt" [8], by using techniques of randomized linear algebra. REML [9] decomposes the label matrix into a low-rank structure and sparse component, which represents label correlations and outliers, respectively. Si et al. proposed Goal-directed Inductive Matrix Completion (GIMC) [28] and also applied this method to multi-label classification. In this paper, since SLEEC was reported to achieve better or comparable performance to the above embedding-based methods on larger data sets [3], [8], [9], [28], we only compared our method with SLEEC.

PD-Sparse [5] uses primal and dual-sparse formulation, which consists of the dual-sparse loss and the primal-sparse regularizer. Using these two types of sparsity and hashing techniques, PD-Sparse can be efficiently learned and achieves fast prediction.

Distributed learning is another approach to extreme multi-label classification. Babbar and Schölkopf proposed DiSMEC [6], which is a large-scale distributed framework for learning one-versus-rest linear classifiers. Using a distributed framework, DiSMEC learns a L_2-regularized L_2-loss SVM for each label in parallel. To reduce the model size and make prediction faster, they pruned *ambiguous* weights in the region near zero after training. They reported that the model for an entire WikiLSHTC-325K data set can be trained in approximately 6 hours on 400 cores and 3 hours on 1,000 cores. In this paper, we did not compare our method with DiSMEC because we focus on non-distributed approaches. Note that the ensemble of 15 AnnexML models on the aforementioned data set were trained within 4 hours by using 24 cores on the single machine described in Sect. 4.1.

In an approximate similarity search task for neural word embeddings, Sugawara et al. [19] compared hash-based, tree-based, and graph-based algorithms. They reported that a graph-based indexing method (NGT [29]) outperformed other methods. NGT is an indexing method that combines a variant of a vantage point tree [30] with an approximate KNNG. In our experiments, the KNNG combined with the ball tree also successfully accelerated the prediction speed without a noticeable drop in accuracy.

6. Conclusion

In this paper, we presented AnnexML by dealing with three problems affecting SLEEC. The key idea of AnnexML is reproducing the KNNG of label vectors in the embedding space to improve both the prediction accuracy and time of the k-nearest neighbor classifier. Experimental results on several large-scale data sets showed that AnnexML can significantly improve prediction accuracy, especially on larger data sets. In addition, the prediction time of AnnexML was up to 58 times faster than that of SLEEC at the same level of accuracy.

We released our implementation of AnnexML on the `github.com`[†]. Our code would be useful for both researchers who compare their results with ours and practitioners who try to solve real-world Web-scale classification problems.

Acknowledgements

We would like to thank Professor Hisashi Kashima, the members of his laboratory at Kyoto University, and our colleagues at the Yahoo Japan Corporation for the many insightful discussions and helpful comments.

[†]https://github.com/yahoojapan/AnnexML

References

[1] Y. Tagami, "AnnexML: Approximate nearest neighbor search for extreme multi-label classification," Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.455–464, 2017.

[2] Y. Prabhu and M. Varma, "FastXML: A fast, accurate and stable tree-classifier for extreme multi-label learning," Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.263–272, 2014.

[3] K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain, "Sparse local embeddings for extreme multi-label classification," Proceedings of the 28th International Conference on Neural Information Processing Systems, pp.730–738, 2015.

[4] I. Partalas, A. Kosmopoulos, N. Baskiotis, T. Artieres, G. Paliouras, E. Gaussier, I. Androutsopoulos, M.R. Amini, and P. Galinari, "LSHTC: A benchmark for large-scale text classification," arXiv preprint arXiv:1503.08581, 2015.

[5] I.E. Yen, X. Huang, K. Zhong, P. Ravikumar, and I.S. Dhillon, "PD-Sparse: A primal and dual sparse approach to extreme multiclass and multilabel classification," Proceedings of The 33rd International Conference on Machine Learning, pp.3069–3077, 2016.

[6] R. Babbar and B. Schölkopf, "Dismec: Distributed sparse machines for extreme multi-label classification," Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, pp.721–729, 2017.

[7] H.F. Yu, P. Jain, P. Kar, and I.S. Dhillon, "Large-scale multi-label learning with missing labels," Proceedings of The 31st International Conference on Machine Learning, pp.593–601, 2014.

[8] P. Mineiro and N. Karampatziakis, "Fast label embeddings via randomized linear algebra," Joint European Conference on Machine Learning and Knowledge Discovery in Databases, vol.9284, pp.37–51, 2015.

[9] C. Xu, D. Tao, and C. Xu, "Robust extreme multi-label learning," Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.1275–1284, 2016.

[10] K. Aydin, M. Bateni, and V. Mirrokni, "Distributed balanced partitioning via linear embedding," Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, pp.387–396, 2016.

[11] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes," The 37th International Conference on Very Large Databases, pp.1213–1224, 2011.

[12] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," Proceedings of the 20th International Conference on World Wide Web, pp.577–586, 2011.

[13] L. Bottou, Y. Bengio, et al., "Convergence properties of the k-means algorithms," Advances in neural information processing systems, pp.585–592, 1995.

[14] H.B. McMahan, "Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization," Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp.525–533, 2011.

[15] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," J. Mach. Learn. Res., vol.12, pp.2121–2159, 2011.

[16] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," ICML, 2009.

[17] W.T. Yih, K. Toutanova, J.C. Platt, and C. Meek, "Learning discriminative projections for text similarity measures," Proceedings of the Fifteenth Conference on Computational Natural Language Learning, pp.247–256, 2011.

[18] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, pp.2333–2338, 2013.

[19] K. Sugawara, H. Kobayashi, and M. Iwasaki, "On approximately searching for similar word embeddings," Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp.2265–2275, 2016.

[20] S.M. Omohundro, "Five balltree construction algorithms," tech. rep., No.562, International Computer Science Institute, 1989.

[21] P. Ram and A.G. Gray, "Maximum inner-product search using cone trees," Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.931–939, 2012.

[22] Y. Bachrach, Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice, and U. Paquet, "Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces," Proceedings of the 8th ACM Conference on Recommender Systems, pp.257–264, 2014.

[23] A. Shrivastava and P. Li, "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)," Advances in Neural Information Processing Systems, pp.2321–2329, 2014.

[24] K. Bhatia, H. Jain, Y. Prabhu, and M. Varma, "The extreme classification repository," 2016.

[25] H. Jain, Y. Prabhu, and M. Varma, "Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications," Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.935–944, 2016.

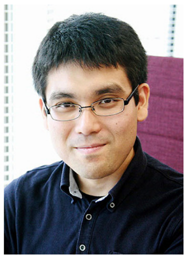
[26] K. Jasinska, K. Dembczyński, R. Busa-Fekete, K. Pfannschmidt, T. Klerx, and E. Hüllermeier, "Extreme f-measure maximization using sparse probability estimates," Proceedings of The 33rd International Conference on Machine Learning, pp.1435–1444, 2016.

[27] Y. Tagami, "Learning extreme multi-label tree-classifier via nearest neighbor graph partitioning," Proceedings of the 26th International Conference on World Wide Web Companion, pp.845–846, 2017.

[28] S. Si, K.-Y. Chiang, C.-J. Hsieh, N. Rao, and I.S. Dhillon, "Goal-directed inductive matrix completion," Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.1165–1174, 2016.

[29] M. Iwasaki, "Pruned bi-directed k-nearest neighbor graph for proximity search," International Conference on Similarity Search and Applications, vol.9939, pp.20–33, 2016.

[30] P.N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp.311–321, 1993.



Yukihiro Tagami has been a research engineer at Yahoo Japan Corporation from April 2010. He has worked in the area of applied machine learning, including click-through rate prediction for display advertising. Since October 2015, he has also been participating in a doctoral course at Kyoto University.