Efficient exact batch prediction for label trees

Extended Abstract

Kalina Jasinska
Institute of Computing Science, PUT Poznan
kalina.jasinska@cs.put.poznan.pl

ABSTRACT

Label tree algorithms have recently gained increasing attention as they scale well to extreme classification problems. Examples of such algorithms are hierarchical softmax, probabilistic label trees or Parabel. Prediction for new test examples in these algorithms can be performed by following either uniform-cost search or beam search. While both of these approaches can be easily implemented in the online (i.e., example-by-example) mode, it is not entirely clear how to efficiently implement the uniform-cost search in the (mini-) batch mode. Below we show how to solve this problem.

KEYWORDS

extreme classification, label trees, multi-label classification, uniform-cost search, batch prediction

1 INTRODUCTION

Label tree algorithms, like HSM [5], PLT [4], or Parabel [6], are used in both multi-class and multi-label classification to create efficient predictive models. In general, they are specific trees of classifiers with labels assigned to leaves, where score (or a probability) of a leaf (label) is a function of scores of nodes in the path from the tree root to this leaf. Then, the prediction procedure, tailored for precision@k, is finding the k leaves with the highest probabilities. Finding the exact top k leaves requires searching the tree, for example with uniform-cost search. This however was used only for example-by-example prediction. In situatons where batch prediction is required the exact algorithm was not yet used. We show that exact batch prediction is not only possible, but also that it is competitive to heuristic search algorithms, such as beam search, in terms of both prediction time and memory requirements.

A label tree classifier consists of a number of underlying classifiers organized in a tree structure, such that each label is assigned to one and only one path from the root to a leaf. The classifier estimates the mariginal probability of a label by a product of the probability estimates delivered by the underlying classifiers on the corresponding path.

The exact prediction is searching the tree to retrieve the highest scoring leaves, using the intermediate scores in the inner nodes to guide the search procedure. This is usually accomplished by using a priority queue sorting the nodes on the frontier of the explored part of tree.

The WWW Workshop: XMLC for Social Media, 23 April 2018, Lyon, France

In many situations the easy to implement example-by-example prediction is efficient. This is so when, while test example features must be stored in a sparse format, a direct memory access to the models weights is possible. For example, if dense storage of all the models is possible, or feature hashing is applied [4]. However, when both example features and model weights are stored in a sparse format, the vector-vector product becomes expensive. To overcome the problem of sparse vector products one can convert the sparse model vector to a dense one, and process a batch of examples at once. Such approach can be applied to beam search.

We show that it is possible to apply this trick also to uniform-cost search. This way we can run the exact prediction in a batch mode. It can be shown that our algorithm visits exactly as many tree nodes as the uniform-cost search run examplewise. However, because it allows for benefits from the direct memory access it can proceed more efficiently.

Exact prediction is in general more computionally expensive than approximate. Exact search requires visiting as many nodes as necessary to ensure that the retrieved leaves are the highest scoring ones. Therefore, in the worst case the whole tree may be explored. This is not the case for approximate algorithms. However, as we show, in many real-word situations the exact prediction is feasible.

We first describe the batch prediction algorithm, that uses an additional data structure suited for the task, and then demonstrate empirically it's efficiency. The introduced technique can be applied not only to multi-label learning algorithms like PLT [4] or Parabel [6], but also to multi-class ones like HSM [5].

2 BATCH UNIFORM-COST SEARCH

The proposed prediction algorithm, given in Algorithm 1, runs n uniform-cost searches at once, and processes all n elements in the batch simultaneously. In uses an additional data structure, suited for such application, named MultiQueue. This structure stores n priority queues and keeps track of the frequency of top elements among them. The pseudocode of this data structure is given in Algorithm 2. We firstly briefly describe how the MultiQueue works, and secondly, give the details of the batch top-k prediction algorithm.

The MultiQueue is a data structure consisting of n priority queues q_i, and a master priority queue q, aggregating the top elements from all the n queues. Each queue q_i sorts the elements according to some criterion i. The master queue q sorts the elements from queues q_i according to the number of criteria on which they are ontop and is capable of providing a list of their indices.

In case of use of this data strucure as a part of the batch prediction algorithm, n is the number of examples in a batch, each q_i is related to a single test example and tracks the state of the uniform-cost search for this example. The elements in the queues are nodes of the label tree, and the priority is the intermediate probability product in given node. Therefore q sorts the nodes of the tree according

Algorithm 1 Batch uniform-cost search prediction

input: example features X, batch size n, number of nodes in the tree c, number of top labels to retrieve k
$\hat{Y}_{top-k} = [\emptyset]^n$
MQ = MultiQueue(n, c)
preds = densePredict(root.w, X, $[1:n]$])
for $i = 1, \ldots, n$ **do**
  MQ.push(i, (preds[i], 0))
end for
retrievedCount = $[0]^n$, removedInstances = $\emptyset$,
while |removedInstances| $\neq n$ **do**
  toDelete = $\emptyset$
  node, instances, values = MQ.topAndPop()
  if node.isLeaf **then**
    for $i \in$ instances **do**
      $\hat{Y}_{top-k}[i]$.add(node.label)
      retrievedCount[i] + +
      if retrievedCount[i] == k **then**
        toDelete.add(i)
      end if
    end for
  else
    for child $\in$ node.children **do**
      preds = densePredict(child.w, X, instances)
      for $i \in$ instances **do**
        p = values[i] $\cdot$ preds[i]
        MQ.push((i, (p, child)))
      end for
    end for
  end if
end while
return $\hat{Y}_{top-k}$.

Algorithm 2 MultiQueue

function init (n, t)
for $i = 1, \ldots n$ **do**
  q_i = priorityQueue()
end for
q = priorityQueueWithRandomAccess()
function push (i, v)
prevTop = q_i.top(), q_i.push(v), currTop = q_i.top()
decreasePriority(q, prevTop), increasePriority(q, currTop)
function pop (i)
prevTop = q_i.top(), q_i.pop(v), currTop = q_i.top()
decreasePriority(q, prevTop, i), increasePriority(q, currTop, i)
function topAndPop (i)
$j = q$.top(), q.pop()
for $i \in j$.queues **do**
  q_i.pop()
end for
function decreasePriority (q, v, i)
$q[v].s$.remove(i)
q.decrease(v) {according to $|q[v].s|$}
function increasePriority (q, v, i)
$q[v].s$.add(i)
q.increase(v) {according to $|q[v].s|$}
function remove (i)
delete q_i, remove i from q

to the number of examples with the highest intermediate product in this node at this step of the search. One can of course think of another aggregation criterion.

The exact batch inference works as follows. The batch size defines how many examples are processed at once. For each batch the prediction algorithm creates a MultiQueue with n equal to the batch size. Initially, the root node is added to the queues q_i of all the examples. Then, untill for all examples top k labels are retrieved, the prediction algorithm queries the MultiQueue for the node t to process and a list of examples (queues q_i) with this node ontop. Then, for each child of the node t, the inference algorithms creates a dense representation of the model of the child, calculates the product of example features (only from the list) and the dense model to get the scores, and in case of logistic models, may use the sigmoid transformation to get probabilities. Finally, the child nodes and their intermediate estimates are added to the relevant queues q_i in the MultiQueue. If the processed node is a leaf, the algorithm predicts the corresponding label for all the examples from the list. Notice that the implementation details may vary. For example, one may process the node model weights, not its children. In such case, when a leaf is processed, one should push the final predictions into the priority queues and predict the relevant labels once they get popped again.

It can be shown that such algorithm visits exactly as many nodes, or in other words calculates as many vector-vector products, as example-by-example exact prediction algorithm (as used in [3] and [2]). However, thanks to the use of MultiQueue, can be used in batch mode, and have the advantage of calculating the products in an efficient matrix mode.

3 EXPERIMENTS

We run the experiments on a selection of datasets of various sizes from [1]. We trained three types of models, each with a balanced binary tree, differing with the assignment fo labels to leaves. We used the 2-means++ assignment [6], the random assignment and assignment according to the indices of labels in the dataset. To measure the memory used by the process we use the unix $time - v$ command and report the maximum resident set size.

As Table 1 shows, precision@k of exact prediction algorithm is between beam search with window size 10 and 100. Notice that in case of beam search bigger window size not always gives higher precision, and sometimes beam search with smaller window outperforms exact predictions. Exact inference is better or equal to the beam search with medium window size. It's predictive performance is most similar to beam search with large window, however, as the next experiments show, exact batch prediction can be performed faster and with less memory.

Interestingly, the label partitioning influences not only the predictive performance, but also the computational one. This can be noticed on Figures 2a, 2b, 2c, 2d, which present results for 3 different

Table 1: Precision@k of predictions obtained by the beam search compared to exact predictions.

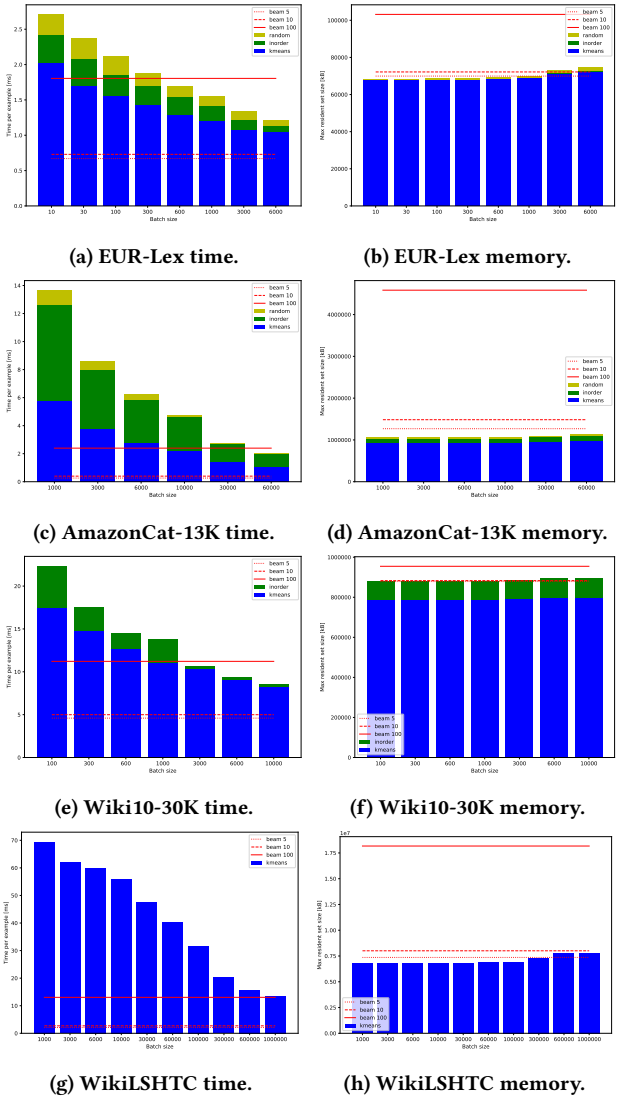
	Beam 5			Beam 10			Beam 100			Exact		
	p@1	p@3	p@5	p@1	p@3	p@5	p@1	p@3	p@5	p@1	p@3	p@5
EUR-Lex-4K	79.71	65.92	54.35	79.73	65.93	55.24	79.71	65.93	55.23	79.71	65.93	55.23
AmazonCat-13K	92.52	78.05	63.26	92.52	78.06	63.44	92.52	78.06	63.44	92.52	78.06	63.44
Wiki10-30K	82.42	71.62	61.64	82.42	72.16	63.88	82.42	72.16	63.90	82.42	72.16	63.89
WikiLSHTC-325K	59.23	38.43	27.49	59.28	38.71	28.43	59.28	38.72	28.45	59.28	38.72	28.45
Amazon-670K	42.28	37.28	33.50	42.97	38.37	34.90	43.19	38.82	35.60	43.19	38.81	35.59

tree structures. The precisions@k of all the models matched the following pattern: $p@k_{2-kmenas++} > p@k_{in-order} > p@k_{random}$. The prediction time with a better scoring tree is lower than with poorer scoring tree. This can be explained by the fact that the better is tree structure, the easier are the underlying binary problems, and the more accurate are the probability estimates provided by the node classifiers. Then, to find the exact top-k labels, less nodes must be explored.

Finally, the batch uniform-cost search algorithm proves in experiments to be efficient. The Figure 1 shows the prediction time per example and maximal memory used by the process of exact inference implementation (with growing batch size). The lines show the time and memory of the beam search with various window size. It can be seen that there is a time-memory tradeof: processing more examples at once requires more, and possibly longer, priority queues to be stored in memory, what has a visible impact on the memory consumption. However, the more examples are in the batch, the fewer times the sparse vectors are turned to dense vectors and the products are calculated more efficiently.

The following results show that, first, given a good batch size, the exact inference is feasible. Secondly, they show that for specific batch sizes of exact inference algorithm, and window sizes of beam search, the exact inference algorithm outperforms the approximate in terms of memory and time. Such phenomena is obseved because beam search is performed on the whole dataset. Then, the window size and numer of examples determine the memory use. Scaling down the memory use twice, results in, more or less, growth of prediction time twice. The exact batch prediction scales differently (see Figure 1), therefore with proper batch size, it is competitive in terms of computation requirements to the approximate one.

Figure 1: Prediction time per example [ms] and maximal memory used by the process of prediction with exact inference, with growing batch size, compared to prediction time and memory consumption of beam search with different window sizes for different tree structures.



(a) EUR-Lex time. (b) EUR-Lex memory.

(c) AmazonCat-13K time. (d) AmazonCat-13K memory.

(e) Wiki10-30K time. (f) Wiki10-30K memory.

(g) WikiLSHTC time. (h) WikiLSHTC memory.

ACKNOWLEDGEMENTS

The work was supported by the Polish National Science Centre under grant no. 2017/25/N/ST6/00747.

REFERENCES

[1] 2016. The Extreme Classification Repository. (2016). Retrieved Match 15, 2018 from http://manikvarma.org/downloads/XC/XMLRepository.html
[2] 2017. PLT in vowpal wabbit. (2017). Retrieved Match 15, 2018 from https://github.com/mwydmuch/extweme_wabbit
[3] Kalina Jasinska and Krzysztof Dembczynski. 2015. Consistent Label Tree Classifiers for Extreme Multi-Label Classification. In *XML Workshop 2015 at ICML*.
[4] Kalina Jasinska, Krzysztof Dembczynski, Robert Busa-Fekete, Karlson Pfannschmidt, Timo Klerx, and Eyke Hullermeier. 2016. Extreme F-measure Maximization using Sparse Probability Estimates. In *ICML 2016*.
[5] Frederic Morin and Yoshua Bengio. 2005. Hierarchical Probabilistic Neural Network Language Model. In *Aistats 2005*.
[6] Yashoteja Prahbu and Manik Varma. 2018. Parabel: Partitioned Label Trees for Extreme Classification with Application to Dynamic Search Advertising. In *WWW 2018*.