DXML: Distributed Extreme Multilabel Classification[⋆]

Pawan Kumar[1][0000−0001−5632−6964]

International Institute of Information Technology, Hyderabad, 500032, India
`pawan.kumar@iiit.ac.in`

Abstract. As a big data application, extreme multilabel classification has emerged as an important research topic with applications in ranking and recommendation of products and items. A scalable hybrid distributed and shared memory implementation of extreme classification for large scale ranking and recommendation is proposed. In particular, the implementation is a mix of message passing using MPI across nodes and using multithreading on the nodes using OpenMP. The expression for communication latency and communication volume is derived. Parallelism using work-span model is derived for shared memory architecture. This throws light on the expected scalability of similar extreme classification methods. Experiments show that the implementation is relatively faster to train and test on some large datasets. In some cases, model size is relatively small.
Code: `https://github.com/misterpawan/DXML`

Keywords: extreme multilabel classification · distributed memory · multithreading.

1 Introduction

Extreme multi-label learning is an active research problem in big data applications with several applications in tagging, recommendation, and ranking. Consider a features matrix of n examples $X \in \mathbb{R}^{n \times d}$ with d-dimensional features with their corresponding labels matrix $Y \in \mathbb{R}^{n \times \ell}$, the aim of multilabel classification problem is to assign some relevant labels out of a total of ℓ-labels to new data sample. The extreme multilabel classification problem refers to the setting when n, d, and ℓ quickly scale to large numbers often upto several millions. Moreover, number of data samples also are in several millions. This is one of the classic challenge problems of big data analytics.

There are several challenges in designing algorithms for extreme multilabel classification. It is found that the average number of labels per data points is usually very small, moreover, it is know that label frequencies follow the so-called Zipf's law, which means that there are only small number of labels which are found in large number of samples, such labels are called head labels. On the other hand, there are large number of lables that occur less frequently, and such labels are called tail labels. Such a distribution creates a bias in the classifier, because since head labels occur more frequently, the classifier may learn more robustly about predicting head labels compared to tail labels, thereby, leading to a classifier that is biased towards predicting head labels better. Despite all these challenges, one of the major concerns is designing scalable algorithms for modern day hardwares.

Looking at the increasing trend of number of labels going upto millions, a hybrid parallel design and implementation of extreme classification algorithms is essential that can exploit shared as well as distributed memory architectures [10,12,3,13,11]. In this paper, we show a parallel design and implementation for a hybrid distributed memory and shared memory implementation using MPI and OpenMP. To the best of our knowledge, this is the first time a hybrid parallel implementation has been shown for extreme classification. We derive the communication bounds for distributed memory, and parallelism bound for shared memory implementation. Our preliminary numerical experiments suggest that we have fastest training and test times when compared to some of the existing parallel implementations in C/C++.

The rest of the paper is organized as folows. In section 2, we discuss previous related work. In section 3, we discuss distributed memory implementation. We derive expressions for communication volume and latency. Finally, in section 4, we discuss numerical experiments on multilabel datasets.

2   Previous Work

Some papers for multilabel classification were proposed during 2006-2014 [17,24]. Some of these papers explored k nearest neighbours [25]. The idea of using random forest were also proposed [9]. With the recent demand for scaling the algorithms to millions of labels, new class of methods have been proposed, where scalability is achieved by either parallelism [21,22], dimension reduction [4,16,19,23], or by hierarchical embedding [6,7,14,20,8].

3   Distributed Memory Implementation

We show a hybrid parallel (MPI+OpenMP) implementation of CRAFTML [15], and call it DXML. During training, DXML computes a forest F of m_F k-ary instance trees, which are constructed by recursive partitioning. The training algorithm is shown in Algorithm 1. In line 1, the input is a feature matrix X and a label matrix Y. We wish to build a label-tree with nodes denoted by v. We then apply the termination condition. The termination condition of the recursive partitioning are the following

1. cardinality of the node's instance subset is less than a given threshold n_{leaf}
2. all the given instances have the same features
3. all the given instances have the same labels

If the stop condition is false, and the current node v is not a leaf as in line 4, then a multi-class classifier is built using Algorithm 2.

The sequential node training stage in DXML will be decomposed into following three consecutive steps:

- a random projection into lower dimensional spaces of the features and label vectors corresponding to the node's instances. That is, in Algorithm 2, in line 2, we first sample X_s and Y_s from X_v and Y_v with a sample size n_s. Then in lines 3 and 4, we do the random projection using projection feature matrix P_x, and a random label projection matrix P_y.
- from projected labels, partitioning of of the corresponding instances into k temporary subsets using $k-$means.
- A multiclass classifier is trained to assign each instance to the relevant temporary subset (that is, the cluster index found at step 2 above) from the corresponding feature vector. The classifier then partitiones the instances into k final subsets or child nodes. This is achieved in by first calling a splitting (line 6, Algorithm 1) of the instances into child nodes using output of k-means of Algorithm 2. Then trainTree function is called recursively on the child nodes.

Similar to FastXML, the nodes partitioning objective of DXML is to regroup instances with common labels in a same subset, but the computation is different. Finally, once a tree has been trained, its leaves store the average label vector of their associated instances. The partition strategy is driven by two constraints: partition computation must be based on randomly projected instances to ensure diversity, and must perform low complexity operations for scalability. The training algorithm is given below.

3.1   Some More Detail on Training

Step 1: Random projections of the instances of v**:** The feature and the label vectors x and y of each instance in v are projected into a space with a lower dimensionality: $x^{'} = XP_x$ and $y^{'} = yP_y$

Algorithm 1 trainTree

1: **Input:** Training set with a feature matrix X and a label matrix Y.
2: Initialize node v.
3: v.isLeaf $\leftarrow$ testStopCondition(X, Y)
4: if v.isLeaf $=$ false then
5:     v.classif $\leftarrow$ trainNodeClassifier(X, Y)
6:     $(X_{child_i}, Y_{child_i})_{i=0,\cdots,k-1} \leftarrow$ split$(v$.classif$, X, Y)$
7:     for i from 0 to $k - 1$ do
8:         $v.child_i \leftarrow$ trainTree$(X_{child_i}, Y_{child_i})$
9:     end for
10: else
11:     $v.\hat{y} \leftarrow$ computeMeanLabelVector(Y)
12: end if
13: **Output:** node v

Algorithm 2 trainNodeClassifier

1: **Input:** feature matrix (X_v) and label matrix (Y_v) of the instance set of the node v.
2: $X_s, Y_s \leftarrow$ sampleRows(X_v, Y_v, n_s) $\triangleright n_s$ is the sample size
3: $X_s' \leftarrow X_s P_x$ $\triangleright$ random feature projection
4: $Y_s' \leftarrow Y_s P_y$ $\triangleright$ random label projection
5: $c \leftarrow k$-means(Y_s', k) $\triangleright c \in \{0, \cdots, k-1\}^{\min(n_v, n_s)}$
6:     $\triangleright c$ is a vector where the jth component c_j denotes the cluster idx of the jth instance associated to $(X_s')_{j,.}$ and $(Y_s')_{j,.}$.
7: for i from 0 to $k - 1$ do
8:     $(classif)_{i,.} \leftarrow$ computeCentroid$((X_s')_{j,.}|c_j = i)$
9: end for
10: **Output:** Classifier $classif (\in \mathbb{R}^{k \times d_x'})$

---

where P_x and P_y are random projection matrices of $\mathbb{R}^{d_x \times d_x'}$ and $\mathbb{R}^{d_y \times d_y'}$ respectively, and d_x' and d_y' are the dimensions of the reduced feature and label spaces resp. The projection matrices are kept different from one tree to another. The random projection considered is a sparse orthogonal projection matrix [18] with values of -1 or $+1$ on each row. The sparsity of the projections lead to faster computations, hence, faster projections. To retain the sparsity we use the hashing; we describe it next.

In this so-called hashing trick algorithm, high dimensional dataset is projected into a lower dimension. Projection is done row after row in this algorithm. In a row of original matrix each element index is considered as a key and each element index of corresponding row in the projected matrix is considered as the bucket. Each index of original row (key) is mapped to index of lower dimension projected row (bucket) using the hash function. Similarly each key is also mapped to one of the two signs ($+$ or $-$) using another hash function. Multiple keys may be mapped to the same bucket, in that case elements in the same bucket are multiplied by their respective signs (obtained from second hash function) and added. Below is the algorithm of Hashing Trick.

Algorithm 3 Hashing Trick

1: **Input:** X_s, projectedSpaceDimension, S_x, SS_x
2: $X'_s \leftarrow 0$ ▷ initialize projected data matrix
3: $p \leftarrow$ projectedSpaceDimension
4: $nR \leftarrow X_s$.numberOfRows()
5: $nC \leftarrow X_s$.numberOfCols()
6: **for** $i \leftarrow 0$ to nR :
7:     **for** key $\leftarrow 0$ to nC :
8:         Index $\leftarrow hash1(key, S_x)\%p$
9:         Sign $\leftarrow 2 \times (hash2(key, SS_x)\%2) - 1$
10:         X'_s[i,Index] $\leftarrow X'_s$[i,Index] + (Sign* X_s[i,key])
11: **Output:** X'_s

This Algorithm 3 takes X_s (or Y_s) and the projectedSpaceDimension, and S_x, SS_x (seeds for hash1, hash2 functions respectively), and returns projected samples X'_s (or Y'_s). In line 4, nR is the number of rows in X_s, and nC is the number of columns in X_s. The lines 6 and 7 are loops with i iterating over the row indices of X_s, and **key** iterating over the column indices in the row (with index i). In line 8, the Column Index (key) is hashed using hash1 function and if its more than the projectedSpaceDimension, then remainder when divided by p is considered to map it in the range$(0, p)$. In line 9, $2 \times hash2(key)\%2 - 1$ maps key to $+1$ or -1, and finally in line 10, the element $X_s[i, key]$ is multiplied by the sign, and then added to already present element at $X'_s[i, Index]$.

We consider the case where feature and label projections are the same in each node of T. We may have tried different projections per node, but we don't consider that in this paper.

Step 2: Partitioning of Instance into k Temporary Subsets : Let Y_s be the label matrix of a sample drawn without replacement. Let the sample size be at most n_s. This sample is partitioned with a spherical k-means applied on $Y_s P_y$. The use of spherical k-means is motivated by the facts that it is well-adapted to sparse data, moreover, the cosine metric is fast to compute.

The cluster centroids are initialized using the k-means++ strategy to ameliorate the cluster stability, and to improve the algorithm performance against a random initialization.

The $k$means++ initialization strategy is as follows:

1. Among the label centers, choose one center uniformly at random.
1. Choose one center uniformly at random from among the label vectors.
2. For each label vector y, compute the distance $D(y)$, defined to be the distance between y and the closest center that has already been chosen above.
3. Choose one new data point at random as a new center, using a weighted probability distribution where a point y is chosen with probability proportional to $D(y)^2$.
4. Repeat steps 2 and 3 above until all centers have been chosen.
5. After all the centers have been chosen, proceed with the spherical k-means clustering.

Step 3: Assigning a subset from the projected features: In each temporary subset the centroid of the projected feature vectors is computed. During the prediction phase, if the centroid of the subset is closest to the projected feature vector, then the classifier assigns this subset. For computing the closeness, the cosine measure is used.

Prediction: In the prediction phase, for each tree, the input sample goes from root to leaf, which is determined by the successive decisions of the classifier. The prediction is the average label vector stored in the leaf reached. The forest then aggregates the tree predictions with the average operator.

Algorithm Analysis Let $s_x(s_y)$ denote the average number of non-zero elements in the feature (label) vectors of the instances. Due to the hashing trick, the projected feature and label vectors have less than s_x and s_y non-zero elements in average. For a node v of a tree T, let n_v denote the number of instances of the subset associated to v. Let i be the number of iterations of the spherical k-means algorithm.

Lemma 1. *For a node v of a tree T, the time complexity C_v is bounded by $O(n_v \times C)$ where*

$$C = k \times (i \times s_y + s_x)$$

is the complexity per instance.

Proof. See [15].

Let T be a strictly k-ary tree and ℓ_T be its number of leaves. Let $m_T = \dfrac{\ell_T - 1}{k - 1}$ be its number of nodes and $\bar{n}_T = \dfrac{\sum_{v \in T} n_v}{m_T}$ be the average number of instances in its nodes.

Proposition 1. *If the tree T is balanced, its training time complexity C_T is bounded by*

$$O\left(\log_k\left(\frac{n}{n_{leaf}}\right) \times n \times C\right).$$

Otherwise, C_T is equal to

$$O\left(\frac{\ell_T - 1}{k - 1} \times \bar{n}_T \times C\right).$$

Proof. See [15].

It can be observed that the time complexities are independent of the projection dimensions d'_x and d'_y. The clustering is done after sampling from the instances, thus the training and clustering complexity are further reduced.

Proposition 2. *The memory complexity of a tree T is bounded by*

$$O\left(n \times s_y + m_T \times k \times d'_x\right).$$

Proof. See [15].

3.2   Hybrid MPI and OpenMP Parallel Implementation

We use message passing interface MPI [1] to train for each learner. Each learner or process reads the data, and calls trainTree in Algorithm 1. The classification for each node (child) at a given level is processed by multiple threads. Each learner stores their own mean label vector computed in line 11 of Algorithm 1. The model parameters for each tree is sent to master node for faster prediction. Let n_t be the model size for each tree, and there are m_F trees, then the communication volume from the worker nodes to the master node is $O(m_F n_t)$. The latency cost is the number of messages passed. In this case, each processor communicates once to master node. Hence the latency cost is $O(m_F)$. Let P denote the number of processors, then since there are as many processors as number of trees in the forest, $P = m_F$. We have the following results.

Proposition 3. *Let there be P processors and $m_F = P$, we have the following communication costs*

$$communication\_volume = O(Pn_t)$$
$$latency = O(P)$$

We also exploit shared memory parallelism using OpenMP [2] when training each tree. This follows a spawn-sync model. At root, a master thread launches k child processes, and each of the k child process calls trainNodeClassifier. We use work-span model [5] to do parallel complexity analysis of trainTree. The span denoted by T_∞, which corresponds to critical path, i.e., the longest path from root node to a leaf node. Now we calculate the work denoted by T_1, which is the total work done to train the classifier for all the nodes. We define the parallelism to be T_1/T_∞. We have the following proposition.

Proposition 4. *Let T be a strictly $k-ary$ tree, ℓ_T be its number of leaves. Let $m_T, \bar{n}_T$, and C be defined as above, then the parallelism for a tree is bounded by*

$$T_1/T_\infty = O\left(\frac{1}{\log_k m_T} \log_k\left(\frac{n}{n_{leaf}}\right) \times n \times C\right).$$

Proof. From 1, the total work done which is denoted by C is given by the following bound

$$T_1 = C = O\left(\log_k\left(\frac{n}{n_{leaf}}\right) \times n \times C\right).$$

We also have $T_\infty = \log_k m_T$. This gives the required parallelism.

4   Numerical Experiments

We did our MPI+OpenMP experiments on Intel Xeon architecture with 10 nodes with 120GB RAM. We used 10 MPI processes and 5 threads per processes. We choose $m_F = 50$ and $n_{leaf} = 10$. The feature and label projection dimensions are $d'_x = \min(d_y, 10000)$ and $d'_y = \min(d_y, 10000)$. We show the precision scores in Table 2. The best precision scores among the tree-based classifiers are indicated in bold. For example, the DXML has highest P@1 precision scores for Mediamill, EURLex-4K, Delicious-200K among the tree based classifiers. In other cases, it is close to the precisions of other classifiers. In Table 1, we show the train time, test time, and model size. The train times for DXML was best among all methods for all the datasets. The model size for DXML is same as the one for CRAFTML. The model size for DXML/CRAFTML was best for Amazon-670 and Delicious-200K. The model size for PPDSp is very large for Amazon-670. For DXML, the learned model parameters remain distributed, hence, there is an additional cost of reduction operation.

Language		C/C++				C++
Machine		50 cores	1 core			100 cores
Algorithm		DXML	FastXML	PFastReXML	SLEEC	DISMEC
EURLex-4K	Train	**38.01**	315.9	324.4	4543.4	76.07
	Test (ms)	**1.29**	3.65	5.43	3.67	2.26
	Model (MB)	30	384	NA	121	**15**
Delicious-200K	Train	**2929.0**	8832.46	8807.51	4838.7	38814
	Test (ms)	10.40	1.28	7.4	2.685	311.4
	Model (GB)	**0.346**	1.3	20	2.1	18
Amazon-670K	Train	**752.65**	5624	6559	20904	174135
	Test (ms)	3.65	**1.41**	1.98	6.94	148
	Model (GB)	**0.494**	4.0	6.3	6.6	8.1
AmazonCat-13K	Train	**1164.13**	11535	13985	119840	11828
	Test (ms)	8.98	1.21	1.34	13.36	0.2
	Model (GB)	**0.659**	9.7	11	12	2.1

Table 1. Train Time, Test Time, and Model Size. Here NA means not available.

5   Conclusion

For large scale recommendation problem using extreme multilabel classification, a scalable recommender model is essential. We proposed a hybrid parallel implementation of extreme classification using MPI and OpenMP that can scale to arbitrary number of processors. The best part is that this does not involve any loss function or iterative gradient methods. Our preliminary results show that our training time is fastest for some datasets. We derived the communication complexity analysis bounds for both the shared and distributed memory implementations. With more cores, our parallel analysis suggests that the proposed implementation has the potential to scale further.

Method Type		Tree based				Other	
Algorithm	Scores	DXML	PFastReXML	FastXML	LPSR	SLEEC	DISMEC
Mediamill	P@1	**87.20**	83.98	84.22	83.57	87.82	84.83
	P@3	**71.52**	67.37	67.33	65.78	73.45	67.17
	P@5	**57.56**	53.02	53.04	49.97	59.17	52.80
Delicious	P@1	67.28	67.13	**69.61**	65.01	67.59	NA
	P@3	61.64	62.33	64.12	58.96	61.38	NA
	P@5	57.17	58.62	59.27	53.49	56.56	NA
EURLex-4K	P@1	**78.20**	75.45	71.36	76.37	79.26	82.40
	P@3	64.2	62.70	59.90	63.36	64.30	68.50
	P@5	53.26	52.51	50.39	52.03	52.33	57.70
Wiki-10	P@1	**84.68**	83.57	83.03	72.72	85.88	85.20
	P@3	72.54	68.61	67.47	58.51	72.98	74.60
	P@5	62.47	59.10	57.76	49.50	62.70	65.90
Delicious-200K	P@1	**47.8**6	41.72	43.07	18.59	47.85	45.50
	P@3	41.25	37.83	38.66	15.43	42.21	38.70
	P@5	38.00	35.58	36.19	14.07	39.43	35.50
Amazon-670K	P@1	37.31	**39.46**	36.99	28.65	35.05	44.70
	P@3	33.30	35.81	33.28	24.88	31.25	39.70
	P@5	30.50	33.05	30.53	22.37	28.56	36.10
AmazonCat-13K	P@1	92.69	91.75	**93.11**	NA	90.53	93.40
	P@3	78.43	77.97	78.20	NA	76.33	79.10
	P@5	63.56	63.68	63.41	NA	61.52	64.10

Table 2. Precision Scores. Here NA stands for not available.

6 Acknowledgement

References

1. Open mpi: Open source high performance computing, https://www.open-mpi.org/, https://www.open-mpi.org/
2. Openmp, https://www.openmp.org/, https://www.openmp.org/
3. High performance solvers for implicit particle in cell simulation (special issue). Procedia Computer Science **18**, 2251–2258 (2013). https://doi.org/https://doi.org/10.1016/j.procs.2013.05.396, https://www.sciencedirect.com/science/article/pii/S1877050913005395, 2013 International Conference on Computational Science
4. Bhatia, K., Jain, H., Kar, P., Varma, M., Jain, P.: Sparse local embeddings for extreme multi-label classification. In: Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1. p. 730–738. NIPS'15, MIT Press, Cambridge, MA, USA (2015)
5. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (Aug 1995). https://doi.org/10.1145/209937.209958, https://doi.org/10.1145/209937.209958
6. Jain, H., Prabhu, Y., Varma, M.: Extreme multi-label loss functions for recommendation, tagging, ranking and other missing label applications. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 935–944. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939756, https://doi.org/10.1145/2939672.2939756
7. Jasinska, K., Dembczynski, K., Busa-Fekete, R., Pfannschmidt, K., Klerx, T., Hullermeier, E.: Extreme f-measure maximization using sparse probability estimates. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. p. 1435–1444. ICML'16, JMLR.org (2016)
8. Jayadev Naram, Tanmay Sinha, P.K.: A riemannian approach for constrained optimization problem in extreme classification problems. CoRR **abs/2109.15021** (2021), https://arxiv.org/abs/2109.15021

9. Kocev, D., Vens, C., Struyf, J., Džeroski, S.: Ensembles of multi-objective decision trees. In: Kok, J.N., Koronacki, J., Mantaras, R.L.d., Matwin, S., Mladenič, D., Skowron, A. (eds.) Machine Learning: ECML 2007. pp. 624–631. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

10. Kumar, P.: Communication optimal least squares solver. In: 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS). pp. 316–319 (2014). https://doi.org/10.1109/HPCC.2014.55

11. Kumar, P.: Multithreaded direction preserving preconditioners. In: 2014 IEEE 13th International Symposium on Parallel and Distributed Computing. pp. 148–155 (2014). https://doi.org/10.1109/ISPDC.2014.23

12. Kumar, P.: Multilevel communication optimal least squares (special issue). Procedia Computer Science **51**, 1838–1847 (2015). https://doi.org/https://doi.org/10.1016/j.procs.2015.05.410, `https://www.sciencedirect.com/science/article/pii/S1877050915012181`, international Conference On Computational Science, ICCS 2015

13. Kumar, P., Meerbergen, K., Roose, D.: Multi-threaded nested filtering factorization preconditioner. In: Manninen, P., Öster, P. (eds.) Applied Parallel and Scientific Computing. pp. 220–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

14. Prabhu, Y., Varma, M.: Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. p. 263–272. KDD '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2623330.2623651, `https://doi.org/10.1145/2623330.2623651`

15. Siblini, W., Meyer, F., Kuntz, P.: Craftml, an efficient clustering-based random forest for extreme multi-label learning. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 4671–4680. PMLR (2018), `http://proceedings.mlr.press/v80/siblini18a.html`

16. Tagami, Y.: Annexml: Approximate nearest neighbor search for extreme multi-label classification. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 455–464. KDD '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3097983.3097987, `https://doi.org/10.1145/3097983.3097987`

17. Tsoumakas, G., Katakis, I.: Multi-label classification: An overview. Int. J. Data Warehous. Min. **3**, 1–13 (2007)

18. Weinberger, K.Q., Dasgupta, A., Attenberg, J., Langford, J., Smola, A.J.: Feature hashing for large scale multitask learning. CoRR **abs/0902.2206** (2009), `http://arxiv.org/abs/0902.2206`

19. Weston, J., Bengio, S., Usunier, N.: Wsabie: Scaling up to large vocabulary image annotation. p. 2764–2770. IJCAI'11, AAAI Press (2011)

20. Weston, J., Makadia, A., Yee, H.: Label partitioning for sublinear ranking. In: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28. p. II–181–II–189. ICML'13, JMLR.org (2013)

21. Yen, I.E.H., Huang, X., Zhong, K., Ravikumar, P., Dhillon, I.S.: Pd-sparse: A primal and dual sparse approach to extreme multiclass and multilabel classification. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. p. 3069–3077. ICML'16, JMLR.org (2016)

22. Yen, I.E., Huang, X., Dai, W., Ravikumar, P., Dhillon, I., Xing, E.: Ppdsparse: A parallel primal-dual sparse method for extreme classification. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 545–553. KDD '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3097983.3098083, `https://doi.org/10.1145/3097983.3098083`

23. Yu, H.F., Jain, P., Kar, P., Dhillon, I.S.: Large-scale multi-label learning with missing labels. In: Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32. p. I–593–I–601. ICML'14, JMLR.org (2014)

24. Zhang, M., Zhou, Z.: A review on multi-label learning algorithms. IEEE Transactions on Knowledge and Data Engineering **26**(8), 1819–1837 (2014). https://doi.org/10.1109/TKDE.2013.39

25. Zhang, M.L., Zhou, Z.H.: Ml-knn: A lazy learning approach to multi-label learning. Pattern Recognition **40**(7), 2038–2048 (2007). https://doi.org/https://doi.org/10.1016/j.patcog.2006.12.019, `https://www.sciencedirect.com/science/article/pii/S0031320307000027`