Training a multi-label FastXML classifier on the OpenImages dataset

David Marlon Gengenbach david_marlon.gengenbach@stud.tu-darmstadt.de
Eneldo Loza Mencía (Supervisor) eneldo@ke.tu-darmstadt.de

Praktikum aus Maschinellem Lernen und Data Mining

TECHNISCHE UNIVERSITÄT DARMSTADT

Contents

Abstract

Multi-label image classification is the task of assigning a set of one or more labels to a given image. For this task we built a multi-label classifier by using so called transfer learning: we used a pre-trained deep neural network called VGG16 to extract a set of features to subsequently train a FastXML classifier with these features.

1 Introduction

The goal was to train a classifier on the *OpenImages* dataset in a tree-based and supervised way.

For this, we first extracted the features from the images with a pre-strained neural network, then we trained a classifier with these features and the corresponding labels. For an overview of the setup, see Figure 1.

After training, to predict a label for a new image, one has to extract the features from the image the same way with the neural network and then run the FastXML classifier on these features. The classifier then predicts the probabilities of each label for that image.

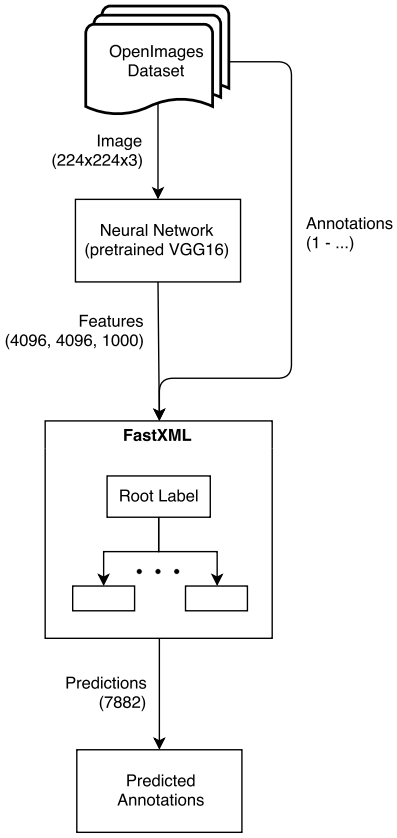

Figure 1: Classifier setup. The numbers in brackets are the dimensions of the inputs to the next step. The images are resized to fit 224x224 pixels and have 3 channels (RGB). The features are the outputs of the fc6 (4096 features), fc7 (4096 features) and fc8 (1000 features) layers of the VGG16 network [1] - for a detailed explanation of the VGG16 network and the layers, see Chapter 3.1.3. Those are the three layers before the softmax layer at the end of the network. We trained a separate FastXML classifier for each of these layers. The number of predicted labels is 7882.

In the next chapter, Chapter 2 (Dataset), we explore the *OpenImages* dataset and explain its different subsets. In Chapter 3 (Methodology) we then present our setup. We also introduce the algorithms and the implementations used to build the classifier. In Chapter 4 (Results and Learnings) we collect the results from training the classifier and discuss possible interpretations for these results. Finally we summarize our approach and key take-aways from this work in Chapter 5 (Conclusions). Possible further work is also discussed in that chapter.

2 Dataset

Google recently released the *OpenImages* image dataset [2]. Each of the images in this dataset has a list of one or more labels attached - the annotations. The dataset consists of approximately 9 million images and more than 7000 different

labels. With this big number of distinct labels the task of training a classifier on this dataset falls under the large-scale multi-label classification task [3].

2.1 Annotations

All annotations have been added by machines. Each annotation has a confidence/probability attached. Google did not disclose how they obtained these annotations.

On a small subset of images, humans either confirmed or rejected the annotations given by the machine. A non-neglectable amount of approximately 31% of annotations got rejected by humans which indicates the noisiness of the machine annotations.

Notice that while the manual confirmation/rejection of the machine annotations reduces the number of false positives (ie. labels that got assigned by the machine, but aren't suitable), the number of false negatives (ie. the missing labels) is not reduced by this approach.

We used this subset of human-moderated images, consisting of more than 150.000 images, to train and test our classifier.

There is a problem of skewed labels [4] in the dataset, meaning that the number of images that have a certain label is not the same for every label. More than half of the labels have less than 100 image instances. Circa 15% of the labels have less than 10 images assigned. Compare that with the number of 25% of all annotations that are assigned to only the top 50 labels (ie. the labels with the most images attached).

The average number of images per label is 237. The average number of annotations per image is 10. See Figure 2 for a graph of images per label.

The normal approach to obtain the dataset is to download the images individually from the URLs given in the *Open-Images* [2] repository. Instead we downloaded the complete dataset from a torrent mentioned in an issue on the dataset GitHub repository [5]. The images in this torrent have been scaled down to 420 pixel on the small side to reduce the file size.

Set	# Images	Size
train	9.011.220	18.3 TB
validation	167.057	309.9 GB

Table 1: Dataset size. The *OpenImages* dataset itself is split into two sets: train and validation. We used a subset of the validation set.

Set	# Annotations
human/validation	1.741.385
machine/validation	2.060.221
machine/train	79.196.416

Table 2: Annotations in the dataset. Over 1.7 million annotations were reviewed by humans. 31% of these annotations were rejected by humans. For training we only used *human/validation* set.

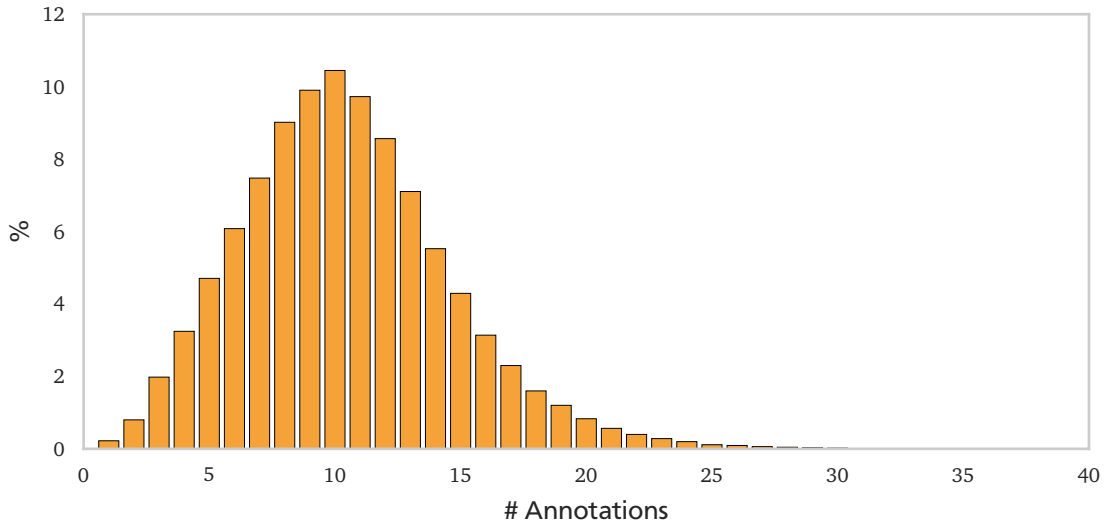

Figure 2: Images per number of annotations. The x-axis shows the number of annotations per image and the y-axis the percentage of images with that number of annotations. On average an image has 10.4 annotations.

3 Methodology

3.1 Setup

3.1.1 Dataset splitting

At first we used a fixed split into *training* and *test* set. The test set consisted of less than 5% of all training examples. This split was done randomly and is only for an overview.

To obtain more reliable results, we next used stratified k-fold cross-validation [6] (k = 3). For this we implemented the algorithm in Python [7].

3.1.2 Transfer learning

To extract features from images we use the wide known technique of transfer learning [8, 9]: a pre-trained network gets an image with the individual pixels as input and the outputs of some layer of this network are taken as the features. These features are subsequently used to train a classifier, in our case the FastXML tree classifier [10]. Transfer learning reduces the training time and computational cost of training a classifier since it does not need to be trained from scratch - instead the new classifier relies on the capability of the pre-trained neural network to extract meaningful features from the images.

In the next two subsections we explain the setup and components of the setup more closely. For an overview of the setup, see Figure 1. In Chapter 3 we introduce the implementations we used for these components.

3.1.3 Feature extraction (VGG16)

The pre-trained neural network we used was VGG16 by Simonyan et al [1] that won the first and second place on the ImageNet ILSVRC-2014 competition [11] in classification and localization. VGG16 was trained on a dataset of 60.000 images with 1.000 labels. For an overview of the VGG16 architecture see Figure 3.

We used the output of the last three fully connected layers (named fc6, fc7, fc8) to extract our features from.

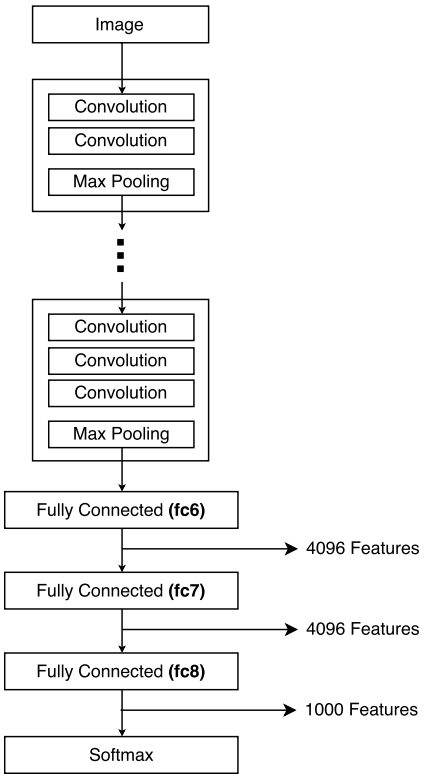

Figure 3: Neural network VGG16. The output of the last three fully-connected layers (fc6, fc7, fc8) are taken to train new classifiers. The weights and the model definition were downloaded from the website of the VGG16 authors.

3.1.4 Training the classifier (FastXML)

After we extracted the features with the VGG16 net, we trained a FastXML [10] tree classifier in a supervised way by feeding it the outputs from the pre-tained network together with the corresponding labels. An overview of the FastXML algorithm can be found in the next Chapter 3.2.1.

3.2 Algorithms

3.2.1 FastXML

FastXML [10] is a tree-based supervised machine learning algorithm. It is especially suitable for massive multi-label classification. The following two paragraphs introduce the principles of the FastXML algorithm.

Training

At training time a root node is created and all labels are assigned to it. Then a node partitioning algorithm is executed to partition the labels on this node. This algorithm splits the labels on the root node into two or more sets. Two or more child nodes are created and the labels are assigned to them according to the node partitioning algorithm. This is done recursively for each child node until all leaf nodes contain only a fixed number of labels - this number of labels per leaf is a hyperparameter that can be given to the FastXML algorithm at training time. The number of labels the node partitioning algorithm has to split decreases with increasing depth of the tree. The process of splitting can be parallelized because it only depends on the set of labels assigned to a given node and the datapoints corresponding to these labels.

The node partitioning algorithm is crucial for the FastXML algorithm since it is responsible for finding sets of similar labels and putting them in the same child node. See the original paper [10] for a more detailed explanation.

A given number of FastXML trees are trained in parallel and used in an ensemble to increase the prediction performance of the classifier. This process is also parallizable since there are no dependencies between the trees in the ensemble.

Prediction

During prediction a datapoint is fed to the root node and passes down the tree until it reaches the leaf nodes. Probabilities are calculated on every node for all the child nodes and their corresponding labels. If the probability of a given datapoint falls below a given threshold on a node, all child nodes of that node can be ignored because the labels on this

part of the tree are not relevant for this datapoint and the probability won't increase further down that subtree. This pruning technique of ignoring irrelevant parts of the tree greatly reduces the prediction time and computation effort. The calculation of the probabilities in the subtrees can also be parallelized since they don't depend on each other.

3.3 Technical Implementations

Following is a small summary of the used implementations. The VGG16 and FastXML implementations can also be found in the GitHub repository of this report [7].

3.3.1 VGG16

For the VGG16 we tried a number of different implementations. Some of them had minor to major flaws and it quickly became apparent that cross-checking the results of an implementation with the reference implementation was crucial. The reference implementation [1] in the caffe machine learning framework [12] by the authors turned out to be quite slow in prediction and therefore feature extraction time compared to an implementation in the TensorFlow framework [13].

For this project we used a TensorFlow implementation from GitHub [14]. We had to modify the implementation to extract the features and have bigger batches of images processed at once. The fork with these changes can be found on GitHub [14]. We cross-checked the results of this implementation with the reference implementation.

We uploaded an easy to use reference implementation of VGG16 in the caffe framework here [15] - this implementation uses the original pre-trained weights and model file from the authors [1]. It is also possible to extract the features with these implementations, but note that it might be slower than the Tensorflow implementation.

3.3.2 FastXML

For FastXML we tried out two different implementations: one from the authors FastXML (implemented in C++) and another one from GitHub Referer/fastxml (implemented in a mix of Python and C++).

We decided to use the C++ implementation [16] from the authors of FastXML because with the other implementation we would have had to cross-check whether the results are consistent with the reference implementation. One major caveat of using the C++ version is that it's more difficult to debug. Retrieving information about the model created by this implementation is more difficult, too. The documentation of the implementation and comments in the code were quite sparse. The code and implementation can be downloaded from [16].

An overview of the hyperparameters the C++ implementation of FastXML can be found in Table 3.

Parameter	Description
num_tree	Number of trees to be grown
bias	Feature bias value, extra feature value to be appended
log_loss_coeff	SVM weight co-efficient
lbl_per_leaf	Number of label-probability pairs to retain in a leaf
max_leaf	Maximum allowed instances in a leaf node. Larger nodes are attempted to be split, and on failure converted to leaves

Table 3: Hyperparameters of the C++ FastXML Implementation. Directly taken from `https://manikvarma.github.io/code/FastXML/download.html`

3.3.3 Custom scripts

To handle the raw dataset we wrote scripts to pre-process the data. The scripts can be found on GitHub [7].

Because the implementations of the neural net and FastXML have different input formats we wrote scripts to convert from the format of the neural net to that of the FastXML implementation. We also wrote a script to convert the extracted features from the neural net format so it can be used with the MULAN [17] multi-label learning library.

4 Results and Discussion

4.1 Results

Following are the results on the trained FastXML classifier. In Table 4 you can see the metrics on the performance of the classifier with a set of parameters tested.

Table 5 shows the results of training the FastXML classifiers on the output of the different VGG16 layers (fc6, fc7, fc8). For these results we used 3-fold stratified cross-validation.

Metrics

The metric we used to evaluate the performance of the classifier was Precision@k [18] (where k is a number, in our case k=5): for a given datapoint only the top-k predicted labels (= the k labels with the highest predicted probability) are compared with the correct labels. This ratio of correct top-k labels gets averaged over all datapoints.

We also evaluated the Hamming Loss but due to the great number of labels and the way the Hamming Loss is calculated [19], this metric does not have a great significance for our task.

We also wanted to calculate other metrics, like the F1 micro/macro score or coverage, but the computation with such a big number of labels was quite time expensive when using off-the-shelf implementations like *sklearn*.

Findings

The results in Table 4 suggest that two parameters have a great impact on the performance of the classifier, namely the number of trees trained in the ensemble and the number of labels per leaf.

The number of trained trees in the ensemble is especially important and training a bigger number of trees is a key component of the FastXML algorithm because a singular tree is only a relatively weak classifier and has to be used in ensembles to obtain a good performance [10]. The training time increases with the number of trees and decreases with the number of trees that are trained in parallel.

The results in Table 5 suggest that training the classifier with the features from different VGG16 layers makes a difference. It has to be noted that the fc6 and fc7 layer have 4096 neurons/outputs each and the fc8 layer has only 1000 neurons/outputs. We though that using the fc6 or fc7 layers would outperform the fc8 layer, because the number of features and therefore the available data is higher in the fc6/7 layers. But the Precision@5 metric of the fc8 features is not significantly lower than the fc7 score and even higher than the fc6 score.

One explanation that the fc8 feature performance is nearly as good as the one from fc7 is that the task, on which the VGG16 network has been trained on, is similar to our classification task. Meaning that the features from fc8 are not too specific for our classification task. For more information on transfer learning and interpretations on the performance of different extraction layers, please see [9].

On thing to note is that the FastXML training time was approximately the same for the fc6/7 layers and the fc8 layer despite the lower number of features in the fc8 layers and therefore lower size per datapoint.

			Precision@5	
num_tree	max_leaf	lbl_per_leaf	Test	Train
1	1	1	0.108	0.201
1	5	1	0.123	0.196
5	5	1	0.349	0.471
5	5	5	0.646	0.891
5	10	1	0.394	0.519
10	5	1	0.438	0.604
10	5	10	0.699	0.902
10	5	20	0.701	0.904
10	5	30	0.701	0.904
10	5	40	0.701	0.904
10	5	50	0.701	0.904
10	10	10	0.696	0.882
10	10	30	0.698	0.889
10	20	10	0.693	0.855
10	20	30	0.694	0.862
10	30	30	0.690	0.844
10	40	30	0.685	0.831
10	50	30	0.683	0.821
50	50	10	0.698	0.827

Table 4: The *bias* and *log_loss_coeff* were both fixed to *1.0* (default). For an overview of the hyperparameters, see Table 3. Because the training/test split was done randomly and the test set consisted of less than 5% of all data points, these metrics are just for orientation.

		Precision@5			
		Hold Out			
VGG16 Layer	Set	2	1	0	Avg
6	test	0.6634	0.6612	0.6612	0.6620
	train	0.9959	0.9957	0.9957	0.9958
7	test	0.6848	0.6823	0.6823	0.6832
	train	0.9956	0.9954	0.9954	0.9955
8	test	0.6806	0.6798	0.6798	0.6801
	train	0.9947	0.9946	0.9946	0.9946

Table 5: Results of the trained FastXML classifier. Precision@5 (higher is better). 3-fold stratified cross-validation was used. The number in the Hold-Out column is the set in the 3 sets that was used as the test set (index from 0). FastXML parameters: start_tree = 0, num_tree = 5, bias = 1.0, log_loss_coeff = 1.0, max_leaf = 5, lbl_per_leaf = 20. Best on test/train set in bold.

5 Conclusions

5.1 Summary

We trained a multilabel classifier on the OpenImages dataset consisting of annotated images. To reduce the number of input parameters and also reduce the training time, we used a pre-trained network called VGG16 to extract so called features from different layers of the network. VGG16 was also trained on images but on a different image dataset.

We wanted to evaluate whether the performance of the new classifier depends on the layer from which the features have been extracted. Interestingly the last fully connected layer (fc8) before the softmax layer in the VGG16 network performed nearly as well or even slightly better than the previous two layers. One way to interpret this is, that the dataset the VGG16 network was trained on dataset is very similar to the OpenImages dataset.

Another key take-away is the importance of training an ensemble of FastXML trees instead of a single one. Because of the ability to massively parallelize the training/prediction of the separate trees, FastXML ensures that the computation/prediction time does not increase too much when training multiple trees.

5.2 Further Work

The OpenImages dataset is special in the way the annotations have been partially validated by humans and because of the sheer number of images in this dataset. These two properties could be further explored by using a neural network and training it on the noisy images and annotations of the training set. Afterwards one could fine-tune the network with the validation set. While this approach is not new, normally the pre-training is done in a unsupervised way (eg. by stacking restricted boltzmann machines) and not with noisy annotations.

Another further work would be to train other classifiers and compare their prediction performance.

Also, a more detailed analysis of all hyperparameters for the FastXML classifier would be interesting.

Another approach would be to explore the kind of noisiness of the machine annotations. For example by training a classifier and checking whether the classifier predicts the same labels that the humans rejected. This would maybe also question the quality of the manual validation by the humans or whether certain labels are more difficult to predict with machine learning. When we explored the dataset and inspected the labels that the humans rejected the most, we also found some instances of mis-rejected annotations. For example one of the most mis-predicted labels was “bloom”. Some of the images where the “bloom” label has been rejected by the humans, clearly had flower blossoms in them. Also some of the labels seemed to be either too specific (eg. "fairchild republic a-10 thunderbolt ii", "mikoyanaurevich mig-15" or "caller id") or too abstract (eg. "beauty", "relief" or "love").

References

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[2] Ivan Krasin, Tom Duerig, Neil Alldrin, Andreas Veit, Sami Abu-El-Haija, Serge Belongie, David Cai, Zheyun Feng, Vittorio Ferrari, Victor Gomes, Abhinav Gupta, Dhyanesh Narayanan, Chen Sun, Gal Chechik, and Kevin Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2016.

[3] Jinseok Nam, Jungi Kim, Eneldo Loza Mencía, Iryna Gurevych, and Johannes Fürnkranz. Large-scale multi-label text classification - revisiting neural networks. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD-14), Part 2*, volume 8725 of *Lecture Notes in Computer Science*, pages 437–452. Springer Berlin Heidelberg, September 2014.

[4] Maria Carolina Monard and Gustavo EAPA Batista. Learning with skewed class distributions. *Advances in Logic, Artificial Intelligence, and Robotics: LAPTEC*, 85(2002):173, 2002.

[5] N01Z3. Openimages: Complete file as torrent. `https://github.com/openimages/dataset/issues/11#issuecomment-257250800`, 2016.

[6] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. *On the Stratification of Multi-label Data*, pages 145–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] David Gengenbach. davidgengenbach/ml-praktikum. `https://github.com/davidgengenbach/ml-praktikum`, 2017.

[8] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct 2010.

[9] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.

[10] Yashoteja Prabhu and Manik Varma. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. ACM - Association for Computing Machinery, August 2014.

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[12] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[13] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[14] davidgengenbach and machrisaa. davidgengenbach/tensorflow-vgg (forked from machrisaa). `https://github.com/davidgengenbach/tensorflow-vgg`, 2017.

[15] davidgengenbach. davidgengenbach/vgg-caffe. `https://github.com/davidgengenbach/vgg-caffe`, 2017.

[16] Yashoteja Prabhu and Manik Varma. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning, c++ implementation. `https://manikvarma.github.io/code/FastXML/download.html`, 2014.

[17] Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research*, 12:2411–2414, 2011.

[18] Himanshu Jain, Yashoteja Prabhu, and Manik Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 935–944, New York, NY, USA, 2016. ACM.

[19] M. L. Zhang and Z. H. Zhou. A review on multi-label learning algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1819–1837, Aug 2014.