Extreme Multilabel Classification

Aidan Claffey (adc501), Kevin Wilson (ksw366), and Daniel Turkel (dgt238)
NYU Center for Data Science

May 19, 2020

1 Introduction

In this paper, we evaluate and tune several models for the task of predicting multiple labels for documents in the EUR-Lex data set. The problem falls under the umbrella of "extreme" multilabel classification because the high number of candidate labels—4,000—introduces challenges for performance and efficiency that don't arise in classification problems with smaller label spaces.

2 Approach

We began with an exploratory analysis of the dataset, described in Section 3. Our goal was to gain an understanding of how sparse the feature space and label space were, since these would help us predict potential obstacles and give us an idea of the types of models that would be more likely to perform well on the data.

After studying literature on extreme classification, we ran experiments on the data using six different algorithms. For algorithms with promising initial results, we delved further, testing grids of hyperparameters to optimize performance. We also implemented a baseline classifier that uses class proportions in the training data as class probabilities regardless of input. These experiments are described in Section 4. We chose the algorithm which performed best in our experiments on the dev set as the algorithm we would apply to the test set.

Our model evaluation metric (LRAP) operates on class probabilities, but in many deployment scenarios we would need "hard" binary classifications, so in Section 5 we examined the precision, recall, and Jaccard similarity curves for different probability thresholds in order to determine an appropriate cutoff for binary classification.

3 Exploratory Analysis

3.1 Description of Dataset

Our data is taken from the EUR-Lex dataset,[1] a collection of European legal documents and associated textual tags. Each example consists of 5,000 TF-IDF scores as features and a number of labels (transformed into integers) as targets. We do not have access to the original text of the documents or the tag text.

[1]http://www.ke.tu-darmstadt.de/resources/eurlex

3.2 Features

The training dataset has 15,539 examples, the dev dataset has 1,316 examples, and the test dataset has 2,493 examples. Within the training set, there is an average of 237 non-zero features (TF-IDF scores) per row and a median of 144 non-zero features per row, meaning that there is a long tail of outliers which can be seen clearly in Figure 1. We verified that the distributions of non-zero features per row are similar in train, dev, and test sets, so there will be no sampling bias during model tuning and evaluation.

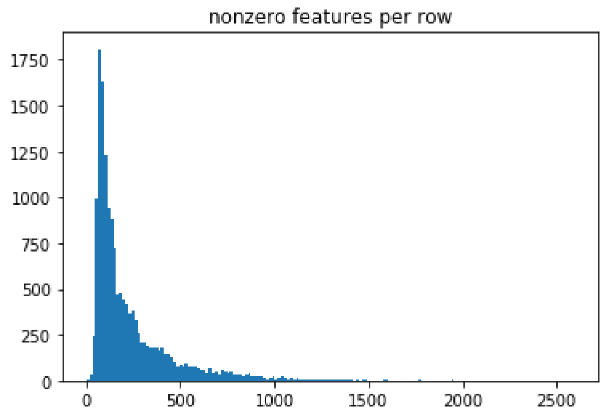

Figure 1: Non-zero features per row, training data

Features are roughly sorted (by feature number) by frequency, with lower-numbered features occurring more often in the dataset. There were at minimum 56 training examples with non-zero values for each feature. Notably, however, there are 1,172 features (over 23% of all features) for which there are just 100 or fewer examples in the training data with non-zero feature values.

3.3 Labels

The training set has an average of 5.3 labels per example, with a minimum of 1 label and a maximum of 24 labels per example. This distribution is shown in Figure 2. Knowing this range allows us to determine at a glance if a given model is over- or under-labeling examples.

Each label has on average about 25 examples.

We also examined the tendency for pairs of labels to co-occur. For all pairs of co-occurring labels in the training set, the average number of times a pair appeared in examples was 2.6 times, while the maximum was 370. This distribution, shown in Figure 3, is skewed heavily to the right, with 98.2% of label-pairs occurring 15 or fewer times. This relieved our concern that the label space might actually be substantially lower-dimensional than it appears, with certain sets of labels being effectively synonymous.

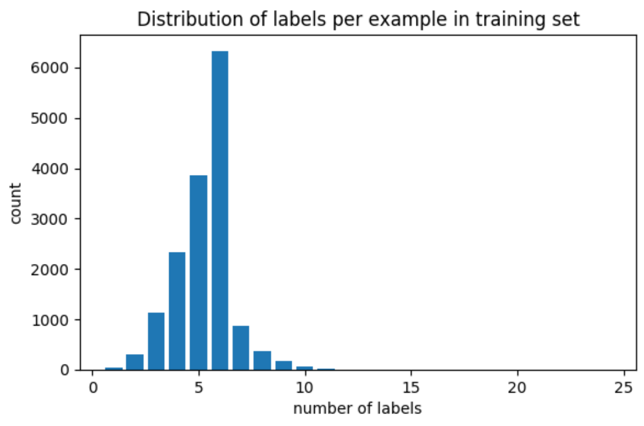

Figure 2: Labels per example in the training data.

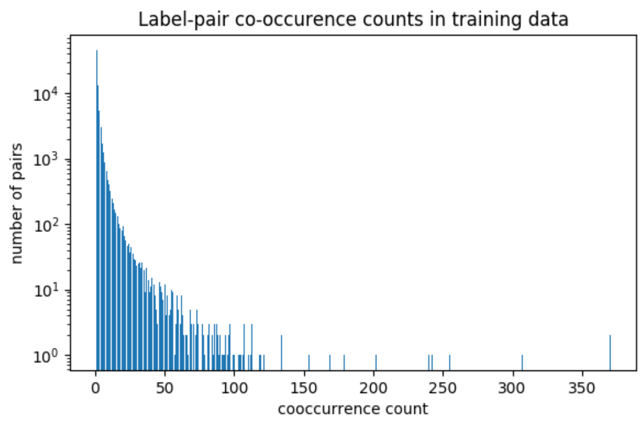

Figure 3: Label co-occurrence counts in the training data.

We further examined class label sparsity. Of the 4,000 labels in the data, 213 did not have a single training example and as such we don't expect our model to predict those labels. A further 722 labels have just 1 training example. Notably, 2,146 class labels or 54% of all classes have 5 or fewer examples in the training set.

4 Experiments

To tune each model on the training data for optimal performance, we will use a holdout set of the training data and perform k-fold cross-validation and grid-search on appropriate parameters, using LRAP (Label Ranking Average Precision) as our validation metric. For cross-validation, we used $k = 5$ folds, since the training set is large enough that 80% of the data (around 12,000 rows) is representative of the entire training set.

After each model's parameters are tuned to achieve optimal validation performance, we will train each model on the entire training set.[2] Then, we will use the provided "development set" to evaluate and compare performance between models via their LRAP scores.

4.1 Baseline Model

We wanted to see what kind of LRAP score a very simple model would give. For this baseline "model," we used as label frequency in the training set as propensity scores. Therefore, every instance in the dev set had the same prediction. Using this score and comparing it to the true values for the dev set, the baseline model received an LRAP score of 0.06. When propensity scores were chosen at random, the LRAP was around 0.005. All of the experiments we ran that extended well to extreme multi-label classification significantly outperformed the baseline.

4.2 Experimental Results

4.2.1 Multi-Layer Perceptron

This model is a multi-layer Perceptron-based neural network. It optimizes log-loss using stochastic gradient descent or a quasi-Newton method called limited memory BFGS. We used the implementation offered in sklearn.

The Multi-Layer Perceptron (MLP) classifier performed modestly well using the out-of-the-box configuration in sklearn. We chose a logistic sigmoid activation function for the hidden layer. To select the best model we used 5-fold cross validation to evaluate model hyperparemeters including `alpha` (regularization parameter) in $[10^{-7}, 10^{-5}, 10^{-3}, 0.01, 0.1, 1]$, `solver` (method used for weight optimization) in ['`adam`', '`sgd`', '`lbfgs`'], and `learning_rate_init` (initial gradient descent learning rate) in $[0.0001, 0.001, 0.01]$. We found the choice of `solver` to be most important in affecting model performance, with the '`adam`' solver, a stochastic gradient descent-based optimizer. Using this solver along with a logistic activation function, `alpha` of 0.01 and `learning_rate_init` of 0.001 produced the best model performance among the MLP models we evaluated. This configuration produced an LRAP score on the dev data of 0.4932, which was better than many other models but not the best among all models we explored.

4.2.2 Classifier Chains

Classifier chains build a sequence of classifiers where each classifier takes as input the upstream classifier's output. Classifier chains are commonly used in multilabel setting, but because they require one classifier per label (per chain), they can be highly inefficient in the extreme classification setting.

[2]For some models where cross-validation was prohibitively time of CPU-intensive, we simply used the entire training set.

As we suspected, our implementation of classifier chains was unfortunately not successful. We attempted to run the sklearn implementation of this model, however because the model builds a one-vs-rest logistic regression classifier for each label and then multiple individual "classifier chain" models, this turned out to be computationally intractable for our case where we have 4,000 labels.

4.2.3 K-Nearest Neighbors

K-nearest-neighbors in the multilabel classification setting (ML-KNN) works similarly to KNN in the traditional classification setting. Instead of picking the majority class from the nearest neighbors, we use the class proportions to assign probabilities [1].

This algorithm did not scale well to the dimensions of our dataset. With an initial attempt of $k = 5$ neighbors using the sklearn implementation, the model was not able to finish predicting features of the dev set after several hours of running.

4.2.4 Radius Neighbors

The radius neighbors classifier operates under the same principle as KNN, but rather than picking a fixed number of nearest neighbors, it looks at all neighbors within a fixed distance measured in the feature space.

Like KNN, the radius neighbors classifier performed very poorly, as we suspected it might. We used the sklearn implementation of radius neighbors, with radii 10, 100, and 1000. With all three radii, we achieved an LRAP of 0.0013 on the dev set. It's possible that results would improve if the radius was increased even more, but prediction time had already ballooned to unreasonable levels at radius 1000.

Clearly neighbors-based methods are not well-suited when the feature space is so large, as the curse of dimensionality leads to all points sitting at a great distance from each other. The algorithms might perform better with dimensionality reduction applied to the feature space, but we did not explore this avenue as other algorithms were already outperforming the neighbors-based approaches in accuracy and speed.

4.2.5 FastXML

FastXML learns a forest of trees where splits are formed with the goal of locally optimizing normalized discounted cumulative gain (NDCG) [2]. The Python implementation [3] exposes a number of hyperparameters, and we performed a grid search across several of them to find an optimal tuning. In particular, we trained 54 models using parameters max leaf size in $[10, 50, 100]$, max labels per leaf in $[20, 50]$, α in $[10^{-5}, 10^{-3}, 10^{-1}]$, and number of trees in $[1, 50, 100]$. FastXML being a forest-based method, the max leaf size and number of trees parameters carry their standard interpretations. The max labels per leaf parameter is the number of labels in a leaf node for which probability scores will be returned—this parameter allows us to cap the theoretical maximum number of labels a single example can be assigned. The α parameter is a learning rate for stochastic gradient descent, which this implementation of FastXML uses in training if the shape of the data fits certain conditions.

Because the FastXML Python implementation is not a fully-conforming sklearn estimator, we were unable to use sklearn's built-in cross-validation tools and instead simply trained each configuration on the full training set and evaluated against the dev set.

By far the most impactful hyperparameter was the number of trees. Going from 1 to 50 trees typically doubled the LRAP score, though increasing from 50 to 100 added only a modest improvement over that. Unfortunately, increasing the number of trees led to much larger models that had to be serialized and much slower training and prediction. The best configuration was $alpha = 0.001$, max of 50 labels per leaf, max leaf size of 10, and 100 trees, and it achieved an LRAP score of 0.4553.

While the model has promise (and several hyperparameters we did not experiment with), we ultimately did not pursue it due to speed and model size concerns, especially considering its performance was not the best of the models we tried.

4.2.6 CRAFTML

CRAFTML [4], like FastXML, is an algorithm specifically designed for extreme multilabel learning that takes a random-forest approach. Instead of learning a hierarchical structure of the labels as FastXML does, CraftML reduces the dimensionality of the feature and label spaces using a series of random projections.

To train each node, the algorithm projects each feature matrix X and label matrix Y using random projections P_x and P_y, where the dimensions of P_x and P_y are parameters of the model. Then, it performs a k-means clustering based on the label projection YP_y, where k is another parameter, using cosine distance as the similarity measure.

After the label space has been partitioned into k clusters, the centroid of each cluster is calculated from the feature space XP_x, and each centroid is saved for the prediction phase. Then, each of the k clusters form their own sub-tree unless the instances meet one of the traditional random-forest stopping criteria, including *minimum leaf size*, *minimum split size*, or uniform labels and/or uniform features in a node. If a node is found to be a leaf based on the above stopping criteria, an average label vector is calculated that will be used during prediction time. As an example, if 5 instances are in a leaf node and they have the labels $(0, 1), (0, 1, 2), (0, 2), (0), (0, 1, 2)$, then the average label vector will be $(1.0, 0.6, 0.6, 0.4)$

The prediction phase is similar to the training phase. For each tree, in each node, each feature instance x is projected onto the random projection space used for training, P_x. Then, it finds its closest cluster centroid using cosine-distance and goes into that node. If that node is not a leaf node, then it is split again using the same logic. If it is a leaf node, then average label vector for that leaf node is given to the instance. Therefore, for n_f trees in the forest, each instance will have n_f label vector predictions, and the final prediction is an average over these predictions.

We used a Rust implementation [5] of CRAFTML to build the models, since there was no Python implementation available. We trained different models using a 5-fold cross-validation grid search on the hyper-parameters listed above, including max-leaf size (n_{leaf}), the number of clusters (k) and the number of trees (n_f). We also varied two other hyper-parameters, the sample size of each tree (n_s) and the minimum number of samples to preserve when pruning the tree(n_{prune}).

Altering n_{prune} did not have an affect on the validation score, so we chose to use the default of 10. As for the other hyperparameters, Figure 4 shows the change in LRAP on the dev set for different hyper-parameter settings. Naturally, increasing n_f will increase the LRAP with diminishing returns, plateuing at around 200. We found that the best parameter configuration for CRAFTML is $n_{leaf} = 10$, $k = 10$, $n_f = 200$, $n_s = 500$, and $n_{prune} = 500$. This gave us a score of 0.589532 for LRAP on the dev set.

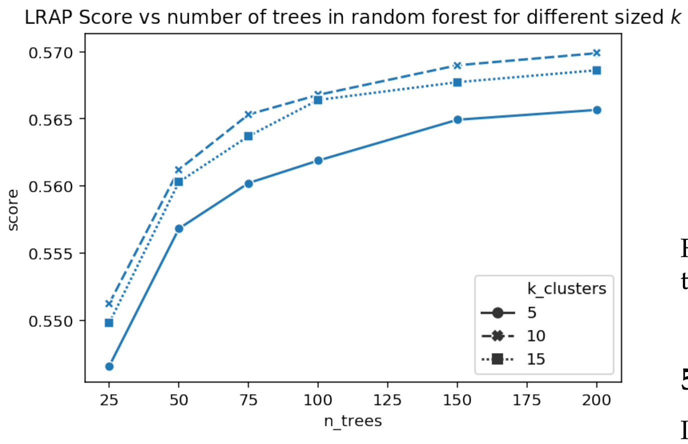

Figure 4: Comparing LRAP validation scores for different numbers of trees and different numbers of clusters

We chose to use CRAFTML for prediction of the test set not only because it performed best out of all of the models in validation, but also because of its relatively quick training time. To train the full training set ($\sim$15,000 rows) and predict on the dev set ($\sim$1,300 rows), our implementation took around 3 minutes to run. This could partially be due to the fast low-level Rust implementation, but in the original CRAFTML paper, it claims to be around twice as fast as other forest-based extreme classification classifiers like FastXML.

5 Discussion

5.1 Leveraging Prediction Probabilities

Since our evaluation metric, LRAP, takes a probability score for each label for each instance in prediction, we can evaluate our model without actually making real predictions. To make predictions, we have to map the probability score outputs from the chosen model to labels by choosing a threshold for the output label scores. If a label score falls above the threshold for a given instance, then we predict the label for that instance. This is a hyper-parameter of our decision function. Figure 5 shows the average Jaccard similarity, precision, and recall for different thresholds using predictions and true labels from the dev set on the best hyper-paremters for the CRAFTML model. The best threshold, 0.2, is at the argmax of the Jaccard similarity and the intersection of precision and recall. Although this is out of the scope of the project, we would choose 0.2 as our threshold on the test set if we had to make "hard" label predictions.

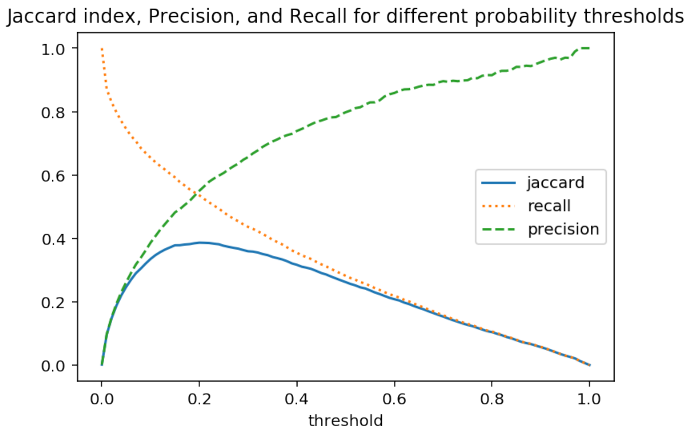

Figure 5: Precision, Recall, and Jaccard Similarity for thresholds

5.2 Possible Next Steps

In the future, having access to the original text data could enable enhanced data preprocessing and feature engineering. Methods could be implemented to, for example, remove labels that are less discriminative or less useful to the use-case of the model.

Additionally, there are several models that were computationally intractable for our purposes (including classifier chains, which we explored using). Recent advances in natural language processing and deep learning likely also offer new models to explore. To train and evaluate such models, methods for dealing with larger-scale machine learning (for instance, using Spark on a high-performance computing cluster) could be implemented.

References

[1] Min-Ling Zhang and Zhi-Hua Zhou. "A k-nearest neighbor based algorithm for multi-label classification". In: *2005 IEEE International Conference on Granular Computing*. Vol. 2. 2005, 718–721 Vol. 2.

[2] Yashoteja Prabhu and Manik Varma. "FastXML: A Fast, Accurate and Stable Tree-classifier for eXtreme Multi-label Learning". In: *ACM – Association for Computing Machinery*. Aug. 2014. URL: `https://www.microsoft.com/en-us/research/publication/fastxml-a-fast-accurate-and-stable-tree-classifier-for-extreme-multi-label-learning/`.

[3] Andrew Stanton. *fastxml*. `https://github.com/Refefer/fastxml`. 2019.

[4] Wissam Siblini, Pascale Kuntz, and Frank Meyer. "CRAFTML, an Efficient Clustering-based Random Forest for Extreme Multi-label Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, July 2018, pp. 4664–4673. URL: `http://proceedings.mlr.press/v80/siblini18a.html`.

[5] Tom Dong. *craftml-rs*. `https://github.com/tomtung/craftml-rs`. 2019.

[6] K. Bhatia et al. *The extreme classification repository: Multi-label datasets and code*. 2016. URL: `http://manikvarma.org/downloads/XC/XMLRepository.html`.