Combining instance and feature neighbors for efficient multi-label classification

**Reference:**
Feremans Len, Cule Boris, Vens Celine, Goethals Bart.- Combining instance and feature neighbors for efficient multi-label classification
2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA), 19-21 Oct. 2017, Tokyo, Japan - ISBN 978-1-5090-5004-8 - Tokyo, IEEE, 2017, p. 109-118
Full text (Publisher's DOI): https://doi.org/10.1109/DSAA.2017.70
To cite this reference: https://hdl.handle.net/10067/1556380151162165141

# Combining Instance and Feature neighbors for Efficient Multi-label Classification

Len Feremans*, Boris Cule*, Celine Vens† and Bart Goethals*‡
*Department of Mathematics and Computer Science, Universiteit Antwerpen, Belgium
{len.feremans, boris.cule, bart.goethals}@uantwerpen.be
†Faculty of Medicine, Katholieke Universiteit Leuven, Belgium
celine.vens@kuleuven-kulak.be
‡Monash University, Melbourne, Australia

***Abstract*—Multi-label classification problems occur naturally in different domains. For example, within text categorization the goal is to predict a set of topics for a document, and within image scene classification the goal is to assign labels to different objects in an image. In this work we propose a combination of two variations of k nearest neighborhoods (kNN) where the first neighborhood is computed instance (or row) based and the second neighborhood is feature (or column) based. Instance based kNN is inspired by user-based collaborative filtering, while feature kNN is inspired by item-based collaborative filtering. Finally we apply a linear combination of instance and feature neighbors scores and apply a single threshold to predict the set of labels. Experiments on various multi-label datasets show that our algorithm outperforms other state-of-the-art methods such as ML-kNN, IBLR and Binary Relevance with SVM, on different evaluation metrics. Finally our algorithm uses an inverted index during neighborhood search and scales to extreme datasets that have millions of instances, features and labels.**

## I. Introduction

Multi-label classification problems occur naturally in a large variety of domains. Within text categorization the learning task is defined as assigning a set of topics or labels to each test document within a corpus. Within scene classification the goal is to assign a set of labels to each image in the test set. For instance, an image of a playground might contain a child, a swing and a tree, and the task is to predict this set of labels. In bioinformatics, in many applications the target variable is a set of labels, such as the prediction of all functions of a gene.

Two major multi-label learning tasks are multi-label ranking and classification [1]. A ranking model outputs a confidence score for each label, while a classification algorithm produces a bipartition of the set of labels into relevant and irrelevant. A ranking model can be turned into a classifier, for example, by specifying a threshold. Several techniques have been investigated for thresholding [2]. Furthermore there are two main strategies for solving the multi-label problem: transform the multi-label problem into a combination of single-label problems or adapt existing single-label classifiers to output multiple labels. The first strategy *transforms* the multi-label problem, so it can be solved using an ensemble of single-label classification algorithms. For example, the most used transformation method, *Binary Relevance* [1], ignores label dependencies and trains a separate model to predict each label independently of other labels. Here, a model is trained for each label where we consider training examples that contain the label as positive, and without the label as negative. The *Label Powerset* [1] transformation method considers each possible subset of labels as a distinct class, and then trains a multi-class classifier on this representation. More recently *Classifier Chains* [3] approximate label dependencies while also requiring the training of a separate model for each label. The second main strategy is to *adapt* an existing single-label algorithm so it can compute multiple labels directly. Many adaptations exist, covering each type of classifier. Well-known adaptations for the multi-label case have been made to adaBoost [4], decision trees [5], SVM [6], k nearest neighbor [7] and others.

Trending challenges in multi-label classification research [8] include methods that account for possible *dependencies* between labels (for example swing and child are probably more correlated than swing and adult in scene classification). A second important challenge is the *computational cost* of generating a model especially in the presence of a large dataset. Finally, learning algorithms have to deal with *label skew*, where there is a small percentage of labels that together cover most instances, and a large percentage of labels that cover only a small portion of instances.

Within the field of recommender systems, there has been increased work in the development of collaborative filtering algorithms for recommendation problems, which has seen an increased interest since the Netflix prize [9]. Here, the task is to predict preferences for items for each user. For example estimating an ordered list of movies (items) a user might like to watch. Two well-known collaborative filtering algorithms are user-based [10] [11] and item-based [12] collaborative filtering. Both are memory-based techniques that compute preferences for items based on the preferences of others users. An extra advantage of these two approaches is that the results are possible to explain. For example, feedback of an item-based recommender is used in Netflix, producing a ranked list of movies motivated by ”Because you watched Orange is the new black”.

When trying to apply collaborative filtering to a multi-label problem, we have to deal with the distinction between features and labels, which is not present in recommendation problems where every column contains item preference. In this work we adapt definitions of similarity and predictions to handle these differences. Our main contributions are that we first implement *instance based k nearest neighbors*, an adaptation of user-based collaborative filtering for multi-label classification. This algorithm searches for the $k$ nearest training instances and then produces a score based on the similarity-weighted average of the neighboring labels. We also discuss a variation which takes dependencies between labels into account by considering second order neighborhoods. As a second step, we implement *feature based k nearest neighbor* method, an adaptation of item-based collaborative filtering. This algorithm computes the $k$ nearest labels for each feature column-wise, and then computes an average score. Finally, we implement a *linear combination of both instance and feature predictions*. Concerning computational cost we remark that if the set of labels $L$ is large, training $|L|$ different models using Binary Relevance or Classifier Chains, or having $2^{|L|}$ possible classes using Label Powerset, is *not efficient*. The neighborhood computation for our method is performed once, and is thus independent from the number of labels. We also use an *inverted index*, a common data structure from information retrieval. We conduct experiments and analysis to validate that our implementation is efficient and scalable to *extreme* datasets, and compare with state-of-the-art methods implemented in `Mulan` [13]. Finally, we conduct experiments to validate the performance of our method on various benchmark datasets, using a variety of evaluation metrics, and compare with other state-of-the-art classification methods such as `ML-kNN` [7], `IBLR` [14], and Binary Relevance with SVM as base learner.

The remainder of this paper is organized as follows. Section II defines the tasks of multi-label classification and label ranking. Section III describes our proposed multi-label classification algorithm. In Section IV we present the multi-label datasets we use in our experiments, discuss existing state-of-the-art methods, present evaluation measures, describe the experimental setup and, finally, present evaluation and runtime performance results. In Section V we discuss related and future work. Finally, our conclusions are presented in Section VI.

## II. Problem Setting

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ denote the set of instances and let $\mathcal{Y} = \{y_1, y_2, \ldots, y_l\}$ denote the set of all possible labels. The training set is defined as $T = \{(x_1, Y_1), (x_2, Y_2), \ldots, (x_n, Y_n)\}$ where $x_i \in \mathcal{X}$ is an instance and $Y_i \subseteq \mathcal{Y}$ is a set of labels corresponding to an instance. The main difference with single-label classification is that $|Y_i|$ is not restricted to 1.

Let $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ be the set of all features, where $f_i \in \mathbb{R}$ and $x_i \in \mathbb{R}^m$. In this work we consider sparse datasets that occur often in real-world multi-label classification domains such as text categorization and scene classification. Here features correspond to patterns, i.e word occurrences in a document, and we assume that the number of non-zero feature values in $x_i$ is small compared to number of features $m$.

The task for a multi-label model is to predict a subset of labels for each test instance. Formally we have to learn a function $h : \mathcal{X} \to 2^{\mathcal{Y}}$ that optimizes a selected evaluation metric. Often the function $h$ is implemented as $h(x) = t(f(x))$ where $f$ produces a confidence score (or probability score) for each label and $t$ is a threshold function that produces a bipartition of relevant and not relevant labels based on the confidence scores. A single threshold function uses a single threshold $t_{min}$ for all labels, meaning that if the score for a single label $y_i$ produced by $f$, that is $s_{y_i}$, is higher than the threshold, that is $s_{y_i} > t_{min}$, then $y_i$ is predicted. A multiple threshold algorithm can apply a different threshold $t_{y_i}$ for each label score $s_{y_i}$ separately or apply a different threshold $t_i$ based on the rank in the ordered list of confidence scores [2]. Many other variations have been proposed. The parameters for the thresholding algorithms can be considered to be hyper parameters that are either set by the end-user, optimized on a validation dataset, or based on other statistics such as label cardinality [3]. We say that function $h$ (or $f$) ignores label dependencies when for each label $y_i$ predictions $h_{y_i}$ are computed without regarding other labels. The Binary Relevance transformation technique assumes labels to be independent, thus $h(x) = h_{y_1}(x) \cup h_{y_2}(x) \ldots \cup h_{y_l}(x)$ where $h_{y_i}(x)$ decides if label $y_i$ is predicted, i.e., $h_{y_i}(x)$ is either $\{y_i\}$ or $\emptyset$.

## III. Method

Our method consists of three separate steps: first we discuss instance based predictions, then we discuss feature based predictions, and finally we compute a label set for each unseen instance based on a linear combination of these two predictions.

### A. Instance based kNN

The first step of our method is to compute confidence scores for labels, based on the labels of the nearest neighbors. First we present details of how similarity and predictions are computed. Then we present and analyse our efficient implementation. Finally a variation of instance based kNN is presented that uses second order neighbor search to account for label dependencies. Note that instance based kNN is inspired by user-based collaborative filtering.

*1) Computing predictions:* The algorithm begins by searching for the top $k$ nearest neighbors $x_i$ of each test instance (or query instance) $x_q$ in the training data using cosine similarity. The cosine similarity is defined as

$$sim_{\text{INS}}(x_q, x_i) = \frac{x_q \cdot x_i}{\|x_q\|_2 \cdot \|x_i\|_2} \tag{1}$$

To compute the confidence score for instance $x_q$ for label $y_i$ we define the following function:

$$conf_{\text{INS}}(x_q, y_i) = \frac{\sum\limits_{x' \,\in\, \text{kNN}(x_q)} \gamma(x', y_i) \;\cdot\; sim_{\text{INS}}(x_q, x')}{\sum\limits_{x' \,\in\, \text{kNN}(x_q)} sim_{\text{INS}}(x_q, x')} \tag{2}$$

Here $\gamma(x', y_i)$ is 1 if label $y_i$ occurs in $x'$, and 0 otherwise.

Finally we apply a *single threshold* to obtain the bipartition as different authors have experimentally verified this is as efficient as the more complex methods [3] [15]. We determine the threshold $t_{min}$ automatically by selecting the value of

$t_{min}$ that minimizes the difference in *label cardinality* between the actual and predicted label set over all training instances. Formally, label cardinality is defined as

$$\mathrm{Lcard}(T) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{l}\gamma(x_i, y_j).$$

The single threshold is optimised using

$$\underset{t_k}{\mathrm{argmin}}\left|\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{l}\delta(conf_{\mathrm{INS}}(x_i, y_j) > t_k) - \mathrm{Lcard}(T)\right|$$

Where $\delta(conf_{\mathrm{INS}}(x_i, y_j) > t_k)$ returns 1 if the confidence value computed using $conf_{\mathrm{INS}}$ is higher than $t_k$.

An exception to this thresholding scheme, is based on an extension of BRkNN [16]. When the *highest scoring label is below the threshold* for a test instance, we still predict this label. We adopted this strategy because in practice most multi-label benchmark datasets *very rarely* (or even never) have an *empty set of labels* for any given instance.

*2) Efficient implementation:* Finding the $k$ nearest neighbors for each of $t$ test instances, given $n$ training instances, requires $\mathcal{O}(t \times n)$ similarity computations, each requiring $\mathcal{O}(m)$ time, where $m$ is the number of features. However, the real-world datasets we are interested in, such as text categorization, are by nature very sparse, with a long tail of infrequent features. We can greatly optimize the kNN search by deploying *sparse* data structures, such as an *inverted index*, and adapted computations that only calculate non-zero terms as discussed in this section.

The inverted index maps each non-zero feature to a set of training instances that contain a non-zero value for this feature. We first *look up* the set of training instances that have a non-zero value for feature $f_i \in x_q$. We then take the union over all (non-zero) features to get the set of candidates $\mathcal{X}_{cand}$, which consists of all training instances that have at least 1 feature in common with $x_q$, and is often much smaller than $\mathcal{X}$.

The remainder of our task is to compute the cosine similarity shown in Eq. 1 between each test instance $x_q$ and all $x_i \in \mathcal{X}_{cand}$ and then keep the top $k$ instances. For the denominator it is trivial to precompute the norms for all test and training instances using dynamic programming. To compute the *dot product* efficiently we do not want to compute each term $f_k \in \mathcal{F} : x_i[f_k] \cdot x_q[f_k]$, as for large datasets with possibly millions of features, and only a small fraction of non-zero values on average, doing $m$ steps is not efficient, since most of these terms will be zero. We can, however, compute the dot products incrementally, thereby only computing the non-zero parts of the dot product for each feature $f_k$ in each candidate training instance $x_i$. We loop over each non-zero feature $f_k$ of $x_q$, then fetch all candidates $x_i$ that have $f_k$ and then increment the partial dot product $x_i \cdot x_q$ for term $f_k$. A special case occurs when all features are binary. Then we can do a sparse set intersection between feature vectors, instead of computing the partial non-zero products for each feature. Our current implementation is limited to this case. Finally we mention the use of a Priority Queue data structure to maintain the top k highest instances.

For the computation of the scores shown in Eq. 2 we take a similar approach. First we compute a set of candidate labels that is present in any of the $k$ nearest neighbors. Next we also organize the computation so that only non-zero increments to the label score are computed. This avoids that when there are many (possibly millions) labels only non-zero increments to the confidence scores of each candidate label are computed.

An important advantage of our method is that it can trivially be adapted so that the expensive neighborhood computation is performed once, *independent* of the number of labels $|\mathcal{Y}|$ as remarked in [16].

Another practical performance tweak is that we cache the nearest neighbor for all values up to a maximal value of $k$ during parameter optimization. First we search for the nearest neighbors once with a maximal value of $k$ and we sort the nearest neighbors on similarity. When the top-k neighbors are required, for smaller values of $k$, we take a subset of the cached neighbors.

*3) Implementation Analysis:* A naive implementation of instance based kNN would compute the cosine similarity based on all features for each training instance, and repeat this process for each label. In this section we discuss the runtime performance of our more efficient implementation illustrated by dataset statistics from the *extreme* dataset *Wiki-LSHTC* (see Section IV).

Firstly we compute kNN once *independent* from $|\mathcal{Y}|$. In the Wiki-LSHTC dataset $|\mathcal{Y}| = 325\,060$. Secondly, when computing the cosine similarity between $x_q$ and a training instance, we only compute the *non-zero partial terms* for the *dot product*. In Wiki-LSHTC there are $|\mathcal{F}| = 1.6$ million different features, however, on average, an instance has only 42 non-zero features. Thirdly, we compute only cosine similarity for *candidate* instances having at least 1 feature in common. In Wiki-LSHTC a non-zero feature, on average, occurs only in 54 rows. In the best case the average number of features and average number of occurrences are independent, this would result in about $42 \cdot 54 = 2\,268$ candidates on average, which is drastically less than $|\mathcal{X}| = 1.7$ million training instances. However, we must be careful, since a small number of features could occur in many instances, which would result in the number of candidates in the worst case close to $|\mathcal{X}|$. Finally, we compute scores only for *candidate* labels, that is the union of all neighbor labels. This is important since, on average, a neighbor only has 3 labels in Wiki-LSHTC, while $|\mathcal{Y}| = 325\,060$.

For the feature based kNN, and the second order variance, we can make an analogous analysis of the implementation. We continue this analysis when discussing runtime performance experiments in Section IV.

*4) Second order instance variation:* A possible disadvantage of our instance based method is that *inter-label dependencies* are ignored: the prediction for a given label is obtained independently of the values of other labels. Also, in user-based collaborative filtering, there is no distinction between features and labels, and cosine similarity is defined using all items. This is not possible in the multi-label classification setting since for any test instance $x_q$, by problem definition, there is no label information. Now we propose a variant of instance based kNN that does regard other labels in the prediction process and uses cosine similarity defined on both features and labels.

Second order instance kNN does a neighbors of neighbors search where similarity between both features and labels for training instances is used. Formally, we define similarity between two training instances as

$$sim_{\text{ALL}}(x_i, x_j) = \frac{x_i \cdot x_j + Y_i \cdot Y_j}{\sqrt{\|x_i\|_1 + \|Y_i\|_1} \cdot \sqrt{\|x_j\|_1 + \|Y_j\|_1}}$$

Where $Y_i$ and $Y_j$ are both vectors of size $l$ containing the labels corresponding to instance $x_i$ and $x_j$ in our training dataset $T$. Given a training instance $x$ we define the re-weighted score for a label $y$ as

$$rw(x,y) = \frac{\sum\limits_{x' \in \text{ kNN}(x)} \gamma(x', y) \ \cdot \ sim_{\text{ALL}}(x, x')}{\sum\limits_{x' \in \text{ kNN}(x)} sim_{\text{ALL}}(x, x')}$$

Here the kNN function uses $sim_{\text{ALL}}$. Note that by definition the first neighbor is the instance itself. To compute the confidence score for test instance $x_q$ for label $y_i$ we define

$$conf_{\text{INS\_2}}(x_q, y_i) = \frac{\sum\limits_{x \in \text{kNN}(x_q)} rw(x, y_i) \ \cdot \ sim_{\text{INS}}(x_q, x)}{\sum\limits_{x \in \text{kNN}(x_q)} sim_{\text{INS}}(x_q, x)}$$

The motivation behind this re-weighted score is that we can use the similarity function $sim_{\text{ALL}}$. By using a measure of similarity in the space of both features and labels, we expect local clusters of co-occurring labels to re-weight the original label score to reflect these local clusters. It is somewhat counter-intuitive to change the given binary labels, but this approach has the advantage that we do account for local neighbors of each training instance (like a micro-cluster around each training instance).

A limitation of this variation is that in many benchmark multi-label datasets the *label cardinality is very small*, with on average fewer than 3 labels for each instance. In this case it does not make sense to apply this variation since the effect on the second order neighborhood computed using $sim_{\text{ALL}}$ versus $sim_{\text{INS}}$ will be neglectable. As a *rule of thumb* we apply it on datasets where the average number of labels is comparable to the average number of features.

From an implementation perspective the re-weighted scores can be computed during *training time*. The main computational issue is that we have to perform a kNN search over all training instances requiring $\mathcal{O}(n^2 \times (m + l))$ steps. We compute this neighborhood efficiently using an inverted index (built using both features and labels) independently of the number of labels.

### *B. Feature based kNN*

We now describe feature based kNN, an adaptation of item-based collaborative filtering for multi-label classification. Item-based collaborative filtering [12] is an extremely popular technique within recommendation algorithms. For example, item-based collaborative filtering algorithms compute the ranking of items in webshops such as Amazon with ”Customers who bought $X$ also bought $Y$” sections, or in Netflix video-on-demand service with ”Because you like movie $X$ you might also like movie $Y$” recommendations. However, there has been no attempt to apply this technique to multi-label problems. We first discuss how predictions are computed and then we discuss the implementation details.

*1) Computing predictions:* Our algorithm first computes the similarity between each feature and the k nearest neighboring labels based on cosine similarity. We compute the cosine similarity between a feature $f_i$ and a label $y_i$ using the column-wise vector of all values. Given a feature or label, we define $\vec{f_i} = \{x_1[f_i], x_2[f_i], \ldots, x_n[f_i]\}$ and $\vec{y_i} = \{Y_1[y_i], Y_2[y_i], \ldots, Y_n[y_i]\}$ as the vector of all values of that feature or label over all training instances. Furthermore we assume that all feature values $f_i$ are normalized between 0.0 and 1.0. We define cosine similarity between a feature and label vector as:

$$sim_{\text{FL}}(\vec{f_i}, \vec{y_i}) = \frac{\vec{f_i} \cdot \vec{y_i}}{\|\vec{f_i}\|_2 \cdot \|\vec{y_i}\|_2}$$

We compute the confidence score for each test instance $x_q$ and each *candidate* label $y_i$ using:

$$conf_{\text{FL}}(x_q, y_i) = \frac{\sum_{j=1}^{m} f_j \cdot sim_{\text{FL}}(\vec{f_j}, \vec{y_i})}{\sum_{j=1}^{m} f_j}$$

Note that we only consider candidate labels that are in the neighborhood, that is

$$y_i \in \bigcup_{f_i \in x_q \wedge f_i \neq 0} \text{kNN}(f_i) \tag{3}$$

If $y_i$ is not in the neighborhood kNN$(f_j)$ for any non-zero feature of $x_q$ the score is 0. Note that the original item-based collaborative filtering method does not take the top $k$ labels (or items), but computes the full similarity matrix [12]. Using our definition we can simulate this technique by setting $k$ equal to $m$. However given a large number of labels our approach is more scalable.

Finally, in order to obtain a set of labels we apply a single threshold $t_{min}$ as discussed in the previous section.

*2) Efficient implementation:* At *training time* we loop over all features and search for the top $k$ similar labels. The main computational cost is this kNN search. For each of the $m$ features, we fetch the set of all training rows, i.e. $\vec{f_i}$ from the inverted index. Here we use sparse data structures as we assume most values in $\vec{f_i}$ and $\vec{y_i}$ will be 0. Next we generate a candidate set of labels that share at least 1 value with any non-zero feature, as shown in Equation 3. Finally we compute the top $k$ most similar labels by computing $sim_{\text{FL}}$ between the current feature and each candidate label. For computation of the cosine similarity we use a technique similar to the one discussed in Section III-A2 for computing the dot product incrementally, thereby only adding partial terms that are non-zero. A limitation of our current implementation is that we only allow binary features. For binary features, the principle is the same, but we can make use of *set intersection* using a (memory-efficient) data structure for performing intersection on large sparse sets of bits[1].

Note that a big advantage of feature-based kNN search for multi-label classification is that the kNN search is only required at *training time*. Furthermore kNN search between features and labels is performed once for all labels.

### C. Linear combination instance and feature based kNN

Now we introduce a straightforward ensemble method based on the Linear Combination of the confidence scores of the Instance and Feature based k nearest neighbors (`LCIF`). Combinations of both techniques have been studied in collaborative filtering research [17] [18], but not in multi-label classification. We compute the confidence score for test instance $x_q$ for label $y_i$ using

$$conf_{\text{LCIF}}(x_q, y_i) = \lambda \; conf_{\text{INS}}(x_q, y_i) + (1-\lambda) \; conf_{\text{FL}}(x_q, y_i) \tag{4}$$

where $\lambda \in [0, 1]$ is a hyper-parameter that can be estimated on a validation sample by optimizing a certain evaluation metric. For datasets with many labels, we only compute this score only for candidate labels, that is labels that have a non-zero score for either $conf_{\text{INS}}$ or $conf_{\text{FL}}$.

We also propose a variation `LCIF2` where we re-use Equation 4 but include the second order variation discussed in Section III-A4 using $conf_{\text{INS\_2}}$ instead of $conf_{\text{INS}}$.

## IV. Experiments

In this section, we discuss our experimental design and results. First we present a set of multi-label problems, as well as existing state-of-the-art methods we compare to, a set of evaluation metrics, and finally experimental setup details for parameter optimization and thresholding. Then we discuss results in terms of evaluation and runtime performance on both large and extreme datasets.

### A. Datasets

We selected 5 *large* datasets that have been used in previous multi-label research [3] as well as a 2 *extreme* datasets. Table I shows the most important statistics of each dataset. The datasets were downloaded from the dataset repository of the multi-label library `Mulan`, `Meka` and the extreme classification repository[2].

The *Medical* dataset consists of nearly 1000 documents containing clinical free text, originally collected at a childrens hospital medical centers department of radiology. The problem is to assign one or more medical diagnoses or procedures (coded using ICD-9-CM) based on the clinical text. The documents are represented using a (sparse) *bag-of-words* encoding.

The *Corel5k* dataset is a scene classification dataset. Labels represent familiar concepts such as sea, sky, cat or forest. The images are represented using 499 (sparse) binary features. A feature value of 1 indicates that a certain segment in the image belongs to a certain cluster.

TABLE I
A SELECTION OF 5 *large* AND 2 *extreme* MULTI-LABEL DATASETS.

| | $\lvert\mathcal{X}_{train}\rvert$ | $\lvert\mathcal{X}_{test}\rvert$ | $\lvert\mathcal{F}\rvert$ | $\lvert\mathcal{Y}\rvert$ | $Lcard(T)$ |
|---|---:|---:|---:|---:|---:|
| *Large* | | | | | |
| Medical | 333 | 645 | 1 449 | 45 | 1.25 |
| Corel5k | 5 000 | 500 | 499 | 374 | 3.53 |
| Bibtex | 4 880 | 2 515 | 1 836 | 159 | 2.38 |
| Delicious | 12 920 | 3 185 | 500 | 983 | 19.04 |
| IMDB-F | 72 551 | 48 368 | 1 001 | 28 | 1.97 |
| *Extreme* | | | | | |
| Wiki-10 | 14 147 | 6 617 | 101 890 | 30 940 | 18.64 |
| Wiki-LSHTC | 1 778 352 | 587 085 | 1 617 899 | 325 060 | 3.19 |

The *Bibtex* dataset represents a tag assignment problem. The original dataset was collected from bibsonomy[3]. This website allows end users to assign tags, representing scientific topics, based on the title and abstract of a scientific publication. The *Delicious* dataset is similar to Bibtex. Here the source is the social bookmarking site del.icio.us[4]. Note that the label cardinality with social tagging is much higher.

For *IMDB-F* the task is to assign one or more of the 28 movie genres, based on movie summary texts from IMDB. This dataset is larger, containing more then 100 000 summaries. However there are only 28 movie genres and the total dictionary of terms is limited.

*Wiki-10* [19] is an extreme dataset, having a much larger number of labels than the large datasets. This datasets corresponds to 20 000 Wikipedia articles. The labels were assigned by end users of del.icio.us.

Finally, *Wiki-LSHTC* consists of more than 2 million instances, a million features and 325 000 categories. The source is Wikipedia. The large number of features is due to the large corpus size. For this dataset there is also a hierarchy between the labels available, which we ignore. The dataset originates from the large-scale hierarchical text classification challenge [20] and is also available on kaggle.

We remark that although these extreme datasets contain an order of magnitude more features and labels, they are also extremely *sparse* and the average number of labels and features remains comparable when taking advantage of this. Analogous to label cardinality we define *feature cardinality* as the average number of non-zero features over all training instances:

$$\text{Fcard}(T) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} \gamma(x_i, f_j)$$

Here $\gamma(x_i, f_j)$ is 1 if instance $x_i$ has a non-zero value for feature $f_j$. For Wiki-10 $\text{Fcard}(T)$ is 674.4 and for Wiki-LSHTC $\text{Fcard}(T)$ is 42.2.

### B. Methods

For the first 5 datasets we compare the performance of our instance and feature kNN methods, as well as their linear combination `LCIF` and the linear combination using

[1] https://github.com/brettwooldridge/SparseBitSet

[2] https://manikvarma.github.io/downloads/XC/XMLRepository.html

[3] https://www.bibsonomy.org/

[4] https://del.icio.us

the second order variant LCIF2. We selected the following state-of-the-art algorithms to compare with: ML-KNN [7], IBLR-ML [14] and Binary Relevance with SMO [21] as base-learner. SMO is Weka's implementation of Support Vector Machines. For optimizing the threshold for the other methods we use OneTreshold [22] which optimizes a single threshold on a selected evaluation metric. We use the implementations provided by Mulan, and implemented LCIF independently. Our implementation (and experiments) are publicly available from our *repository*[5].

For the extreme datasets, due to the extreme dimensions, we cannot use Mulan or Meka. Firstly, both implementations do not optimize for sparse datasets, so merely loading these datasets would require more resources than are available on our test server. Secondly, methods like Binary Relevance combined with SMO do not scale with this extreme number of dimensions.

### C. Evaluation

We compare our algorithms based on a variety of evaluation metrics for multi-label classification. This is a common practice since different methods have different biases towards each metric [8]. Multi-label evaluation metrics can be organized in several different dimensions. *Example-based* evaluation metrics are averaged over all instances. *Label-based* evaluation metrics look at the different ratio's between true-positive, false-positive, and false-negative predictions for each label. Within the label-based category *micro* scores give each instance the same weight, while *macro* scores are averaged over all labels, giving equal weight to frequent and in-frequent labels. Within the Example-based category, we also make a distinction between metrics based on the *bipartition* between relevant and non-relevant labels, metrics based on *ranking* of confidence scores, and finally metrics based on the individual *score* for each label.

We report the following example-based multi-label evaluation metrics:

$$\text{Example-based Accuracy} = \frac{1}{t}\sum_{i=1}^{t}\frac{|Y_i \cap Z_i|}{|Y_i \cup Z_i|}$$

$$\text{Example-based F1} = \frac{1}{t}\sum_{i=1}^{t}\frac{2|Y_i \cap Z_i|}{|Y_i| + |Z_i|}$$

$$\text{Hamming loss} = \frac{1}{t}\sum_{i=1}^{t}\frac{1}{l}|Y_i \triangle Z_i|$$

Here $t$ is the number of test instances, $Y_i$ is the set of actual labels and $Z_i$ is the predicted set of labels for test instance $i$. $Y_i \triangle Z_i$ is the symmetric difference (or XOR) of both sets.

For label-based metrics we define for each label $y_k$ the number of true positives, false negatives, false positives and corresponding metrics as

$$tp = \sum_{i=1}^{t}\delta(y_k \in Y_i \wedge y_k \in Z_i) \quad fn = \sum_{i=1}^{t}\delta(y_k \in Y_i \wedge y_k \notin Z_i)$$

$$fp = \sum_{i=1}^{t}\delta(y_k \notin Y_i \wedge y_k \in Z_i)$$

$$\text{precision} = \frac{tp}{tp + fp} \qquad \text{recall} = \frac{tp}{tp + fn}$$

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

For multi-label evaluation metrics we report both micro and macro F1 metrics. *Micro F1* is based on the previous definitions but based on *totals* of true positives, false negatives and false positives over all labels $l$. The definition of micro precision is (analogous for recall and F1):

$$\text{precision}_{micro} = \frac{\sum_{j=1}^{l} tp_j}{\sum_{j=1}^{l} tp_j + \sum_{j=1}^{l} fp_j}$$

For *Macro F1* we first compute precision and recall for each label separately and then compute the average. The definition of macro precision is (also analogous for recall and F1):

$$\text{precision}_{macro} = \frac{1}{l}\sum_{j=1}^{l}\frac{tp_j}{tp_j + fp_j}$$

### D. Setup

*1) Hyper-parameter optimalisation:* For our methods we have to optimize a number of *hyper-parameters*. The parameter $k$ is often set to a fixed value in other research, or only iterated over a small set of possible values (e.g. 5, 10, 15). However, optimizing $k$ can have a significant effect on reported evaluation metric values. Therefore, we vary $k$ for all kNN methods under consideration between 1 and 30 in steps of 2, and report for each evaluation metric the maximum value. For some datasets we search for $k$ between 1 and 60 if during inner validation we notice that the best performance is for the highest value of $k$.

In order to have a stable estimate for hyper-parameters, we perform *grid search* to estimate optimal hyper-parameters for each evaluation metric using 10-fold *cross validation* for both instance and feature based kNN methods and LCIF. After the 10-fold grid search is completed, we select the parameter combination that optimizes each evaluation metric on the training data, and compute the reported evaluation metrics on the publicly available train-test splits. Note that we re-use the nearest neighbor matrix and only compute it once for the maximal value of $k$ speeding up grid search considerably. For LCIF we vary $\lambda$ between 0.0 and 1.0 in steps of 0.1. We also vary $k_{ins}$ and $k_{feat}$ independent from each other, that is we search for $k_{ins}$ between 1 and 30 and then for $k_{feat}$ between 1 and 30.

For ML-kNN and IBLR-ML we report the optimal evaluation metric values using the best value of $k$. The best value of $k$ is determined directly on the test set thereby assuming an *Oracle* that knows how to select the best value for each $k$. Since many real-world datasets are sparse by nature and have many labels, the performance of these algorithms is an order of magnitude slower, making a full 10-fold grid search too slow to be practical. For BR-SMO we keep default parameters

[5] https://bitbucket.org/len_feremans/lcif_knn_pub

for the (linear) kernel function so no hyper-parameter search is necessary.

For the extreme dataset Wiki-LSHTC we forgo the grid search, and also assume an *Oracle* that knows the best possible hyper-parameters for $k_{ins}$, $k_{feat}$ and $\lambda$ for each evaluation metric.

*2) Thresholding:* After computing scores for each possible hyper-parameter combination we select the optimal value for $t_{min}$ for thresholding. We do this in two *passes*: first we take a step size of $0.1$ and go from $0.0$ until $1.0$ and compute the best threshold $t_{pass1}$. Then we take a step size of $0.01$ and go from $t_{pass1} - 0.05$ until $t_{pass1} + 0.05$. This straightforward optimalisation procedure is implemented by OneTreshold in `Mulan`. Our methods uses the same procedure, but minimize the difference between predicted and actual label cardinality (see Section III-A1) instead of maximizing Micro F1.

### *E. Results*

First we discuss evaluation performance on the large datasets. Next we discuss runtime performance on the large dataset, and finally we discuss evaluation and runtime performance on the extreme datasets.

*1) Evaluation results on large datasets:* Table II shows the results for each different evaluation metric on the *large* datasets for LCIF and the selected state-of-the-art multi-label classifiers. Results in bold are best performers (lowest value for hamming loss, otherwise highest value). Missing values for IBLR for the Delicious and IMDB-F datasets are due to time-out on our test server. Runtime performance is discussed in Section IV-E2.

Feature based kNN by itself performs worse than instance based kNN, but is by far the most efficient method, requiring no nearest neighbor computation at test time. Instance based kNN by itself seems to outperform ML-kNN and IBLR overall. Finally, the linear combination of both neighborhoods in LCIF produces the *best results* overall, ranking first for most datasets on all evaluation metrics, except hamming loss, where BR-SMO produces the best results.

As discussed in Section III-A4 we only apply the second order instance variation, and its combination with feature based kNN, LCIF2, on datasets where the average number of labels is comparable to the average number of features. The Delicious dataset has a label cardinality of $19.04$ and a feature cardinality of $18.54$, making it the only selected dataset where we apply this variation. The results for the second order instance variation on the Delicious dataset are shown in Table III. We see that the second order instance variation further improves the result of the regular instance based kNN. We remark that the effect of feature based kNN in LCIF2 is negligible in this case (having a value of $\lambda$ close to 1.0), except for Hamming Loss. The second order variation performs better than ML-kNN, IBLR and BR-SMO on Delicious on all metrics except Hamming Loss. Based on these results we conclude that second order instance kNN is a promising variation on datasets where the average number of labels per instance is relatively high. On other datasets there is no improvement.

TABLE II
COMPARING LCIF WITH STATE-OF-THE-ART METHODS ON LARGE DATASETS.

| | Ins. kNN | Feat. kNN | LCIF | ML-kNN | IBLR | BR-SMO |
|---|---|---|---|---|---|---|
| Example Accuracy | | | | | | |
| Medical | 0.597 | 0.515 | **0.709** | 0.421 | 0.442 | 0.699 |
| Corel5k | 0.199 | 0.089 | 0.196 | 0.147 | 0.105 | 0.098 |
| Bibtex | 0.351 | 0.202 | **0.362** | 0.208 | 0.174 | 0.321 |
| Delicious | **0.227** | 0.101 | **0.227** | 0.193 | *N/A* | 0.130 |
| IMDB-F | 0.247 | 0.234 | **0.252** | 0.244 | *N/A* | 0.005 |
| Example F1 | | | | | | |
| Medical | 0.641 | 0.562 | **0.751** | 0.493 | 0.699 | 0.727 |
| Corel5k | 0.285 | 0.144 | **0.289** | 0.237 | 0.178 | 0.139 |
| Bibtex | 0.419 | 0.274 | **0.444** | 0.270 | 0.224 | 0.392 |
| Delicious | **0.346** | 0.169 | **0.346** | 0.302 | *N/A* | 0.200 |
| IMDB-F | 0.329 | 0.311 | 0.335 | **0.351** | *N/A* | 0.006 |
| Hamming loss | | | | | | |
| Medical | 0.020 | 0.026 | 0.014 | 0.024 | 0.029 | **0.012** |
| Corel5k | 0.013 | 0.015 | 0.013 | 0.022 | 0.027 | **0.012** |
| Bibtex | 0.017 | 0.022 | **0.016** | 0.020 | 0.021 | **0.016** |
| Delicious | 0.024 | 0.021 | 0.021 | 0.021 | *N/A* | **0.018** |
| IMDB-F | 0.095 | 0.095 | 0.092 | 0.101 | *N/A* | **0.072** |
| Label Micro F1 | | | | | | |
| Medical | 0.643 | 0.557 | 0.750 | 0.505 | 0.506 | **0.773** |
| Corel5k | 0.302 | 0.148 | **0.306** | 0.245 | 0.163 | 0.166 |
| Bibtex | 0.432 | 0.289 | **0.457** | 0.315 | 0.250 | 0.416 |
| Delicious | 0.362 | 0.189 | **0.363** | 0.322 | *N/A* | 0.224 |
| IMDB-F | 0.337 | 0.307 | 0.344 | **0.358** | *N/A* | 0.014 |
| Label Macro F1 | | | | | | |
| Medical | 0.348 | 0.398 | 0.429 | 0.245 | 0.247 | **0.457** |
| Corel5k | 0.346 | 0.248 | **0.348** | 0.326 | 0.161 | 0.317 |
| Bibtex | 0.332 | 0.146 | **0.341** | 0.170 | 0.147 | 0.315 |
| Delicious | 0.182 | 0.050 | **0.184** | 0.088 | *N/A* | 0.098 |
| IMDB-F | 0.089 | 0.076 | **0.095** | 0.056 | *N/A* | 0.011 |

TABLE III
COMPARING INSTANCE WITH SECOND ORDER INSTANCE VARIATION ON DELICIOUS.

| Metric | Ins. kNN | Second Ins. kNN | Feat. kNN | LCIF | LCIF2 |
|---|---|---|---|---|---|
| Example Accuracy | 0.227 | **0.237** | 0.101 | 0.227 | **0.237** |
| Example F1 | 0.346 | **0.354** | 0.169 | 0.346 | **0.354** |
| Hamming Loss | 0.024 | 0.023 | 0.021 | 0.021 | 0.021 |
| Label micro F1 | 0.362 | **0.372** | 0.189 | 0.363 | **0.372** |
| Label macro F1 | 0.182 | **0.184** | 0.050 | **0.184** | **0.184** |

*2) Performance on large datasets:* Table IV shows the runtime on all large datasets for each method on our test server which has 32Gb RAM and $8 \times 2.9$ GHz processors. For kNN based methods we use a fixed value of $k = 15$. Due to order of magnitude differences in runtime, we report runtime in seconds, minutes and hours.

Medical is the only dataset where it makes sense to report in the same time unit. However for all datasets the difference in runtime is quite large and LCIF takes seconds or minutes where other methods take minutes or hours to complete. For Corel5k the relatively large runtime of 34.3 minutes for IBLR is due to learning the optimal weights using logistic regression for each label: training the optimal weights takes about 10 seconds per label but must be repeated for 374 labels. Due to this reason IBLR was not able to complete on Delicious nor IMDB-F. BR-SMO does complete for Delicious and IMDB-F

TABLE IV
COMPARING RUNTIME LCIF WITH STATE-OF-THE-ART METHODS ON LARGE DATASETS.

| | LCIF | LCIF2 | ML-KNN | IBLR | BR-SMO |
|---|---|---|---|---|---|
| Medical | **0.5 s** | 0.4 s | 0.5 s | 1.5 s | 6.5 s |
| Corel5k | **0.7 s** | 1.8 s | 15.5 s | 34.3 m | 6.5 m |
| Bibtex | **3.6 s** | 12.5 s | 1.7 s | 8.2 m | 10.1 m |
| Delicious | **4.3 s** | 32.1 s | 3.3 m | *N/A* | 14.0 h |
| IMDB-F | **1.3 m** | 11.0 m | 2.1 h | *N/A* | 29.4 h |

TABLE V
RUNTIME ON LARGE DATASETS OF DIFFERENT NEIGHBORHOOD METHODS.

| | Feat. kNN | Ins. kNN | Second Ins. kNN |
|---|---|---|---|
| Medical | **0.3 s** | 0.3 s | 0.3 s |
| Corel5k | **0.4 s** | 0.4 s | 1.6 s |
| Bibtex | **1.6 s** | 2.6 s | 12.4 s |
| Delicious | **1.5 s** | 3.1 s | 30.1 s |
| IMDB-F | **5.7 s** | 58.0 s | 658.7 s |

TABLE VI
RESULTS ON EXTREME DATASETS.

| | Ins. kNN | Feat. kNN | LCIF |
|---|---|---|---|
| Example Accuracy | | | |
| Wiki-10 | 0.196 | 0.017 | 0.197 |
| Wiki-LSHTC | 0.311 | 0.102 | 0.323 |
| Example F1 | | | |
| Wiki-10 | 0.318 | 0.032 | 0.321 |
| Wiki-LSHTC | 0.389 | 0.143 | 0.402 |
| Label Micro F1 | | | |
| Wiki-10 | 0.376 | 0.032 | 0.329 |
| Wiki-LSHTC | 0.378 | 0.127 | 0.392 |
| Label Macro F1 | | | |
| Wiki-10 | 0.304 | 0.205 | 0.304 |
| Wiki-LSHTC | 0.278 | 0.131 | 0.288 |

but runs for a full day where LCIF only takes 1 minute.

We can conclude that LCIF (and LCIF2) are *orders of magnitude* faster than ML-kNN, BR-SMO and IBLR. The current implementation of ML-kNN in Mulan could also be adjusted to adopt an inverted index, and sparse computation optimalisations, which could in theory result in a runtime closer to LCIF.

Table V shows the runtime of feature kNN, instance kNN and second order instance variation in seconds. The runtime of LCIF is approximately equal to the total runtime of its feature and instance component. For instance kNN in the worst-case k nearest neighbor computation takes $\mathcal{O}(t \times n \times m)$, but in practice we observe, without formal proof, that the average runtime is closer to $\Theta(t \times \tilde{n} \times \tilde{m})$. Here $\tilde{n}$ is proportional to the average number of candidate instances, that is the number of training instances fetched from the inverted index sharing at least one 1 feature with each test instance and $\tilde{m}$ is proportional to feature cardinality. For feature kNN the nearest neighbor search between features and labels is $\mathcal{O}(m \times l \times n)$ where $m$ is the number of features, and $l$ is the number of labels. However in practice runtime for this search is closer to $\Theta(m \times \tilde{l} \times \tilde{n})$ where $\tilde{l}$ is proportional to the average number of candidate labels, that is the number of labels sharing at least one instance with each feature, and $\tilde{n}$ is proportional to the average number of non-zero feature (or label) values column-wise. If we make a distinction between *training time* and *test time* we observe that the additional time required for second order instance variation, compared with instance kNN, is only necessary at training. Also for feature kNN, the kNN search can be done at training time, while at test time only a dot product is required between each test instance and the precomputed label-feature similarity matrix, making each prediction of a test instance instantaneous at test time.

*3) Evaluation and performance results on extreme datasets:* Table VI shows the results on the extreme datasets for the selected evaluation metrics. We have omitted hamming loss which was close to $0.0$ given the extreme number of labels and is a less suitable metric for extreme datasets [23]. As mentioned in Section IV-B the current implementation of the selected state-of-the-art methods in `Mulan` do not scale to extreme datasets. We do not compare with other scalable algorithms applicable to all domains such as `Fastxml` [24], or restricted to text categorization such as `EkNN` [25] (discussed further in Section V). Wiki-LSHTC, however, was also hosted on *kaggle*. We remark that our reported value for label-based macro F1 is $0.288$ is lower than the highest reported value in the kaggle leaderboard[6] of $0.339$, but would still rank in 5th place. However, it is unclear if our test sample, downloaded from the extreme repository, is exactly the same as the private validation sample kaggle uses. Moreover in this type of competition often a large ensemble of different algorithms, features presentations and hyper-parameters is combined and comparison with a single algorithm is arguably less relevant. We conclude that our method seems to produce promising results on extreme datasets.

Table VII shows the runtime performance. We used a single node at our university high performance computing infrastructure with 128Gb RAM and $20 \times 2.8$GHz cores. On Wiki-LSHTC the instance based kNN took $3.4$ hours to complete, averaged over all test instances, this means about $0.02$ seconds on average per test instance. For feature based kNN the total time required for search is $3.0$ hours, however for feature based kNN, the similarity matrix can be computed at training time, while at test time only $3.1$ minutes are needed to make the predictions for all test instances. Our current implementation in Java for instance based kNN search is multi threaded, using $50$ threads to compute neighbors of test instances in different batches. For feature based kNN our current search is implemented single threaded, so at least a 10-fold speedup could be expected for feature based kNN search. For LCIF the total time is the sum of both methods and about 1 minute to combine the predictions. If we include the time needed on grid search for parameter optimization, the total runtime was $11.8$ hours. We conclude that our implementation has a very reasonable time to complete on such an extreme dataset.

[6] https://www.kaggle.com/c/lshtc/leaderboard

TABLE VII
RUNTIME ON EXTREME DATASETS.

| | Ins. kNN search | Ins. kNN predictions | Feat. kNN search | Feat. kNN predictions |
|---|---|---|---|---|
| Wiki-10 | 10 s | 1 s | 2.3 m | 3.5 s |
| Wiki-LSHTC | 3.4 h | 50.7 s | 3.0 h | 3.1 m |

## V. RELATED WORK

In this section we discuss related work and future research.

There has been several adaptations to instance based learning for multi-label classification. ML-kNN [7] was one of the first methods. In ML-kNN the authors first apply traditional kNN, using Euclidean distance. Next they count the number of times each label occurs in these neighboring instances. Then they apply *maximum a posteriori* principle for each label independently to determine if a label is relevant or not. To compute the maximum a posteriori they estimate prior and posterior probabilities by computing kNN for each training instance, and then compute these probabilities for each label. It is not entirely clear if an inverted index could be used to make this part more efficient, but the neighborhood searches are comparable to LCIF2 resulting in about $\mathcal{O}(t \times n \times m)$ steps for the main search and $\mathcal{O}(n \times n \times m)$ steps for computing posterior probabilities. Experiments showed that ML-kNN outperformed other techniques such as Rank-SVM on different example-based evaluation metrics. A possible disadvantage of ML-kNN is that it does not take label dependencies into account, which was addressed by subsequent research in *dependent* multi-label k nearest neighbour (DML-kNN) [26].

In combining instance-based learning and logistic regression for multilabel classification (IBLR) [14], first the k nearest neighbors are computed, using Euclidean distance, to count the labels of the nearest neighbors for each test instance. Then the authors use these label counts as features, and apply logistic regression to learn the optimal hyper-plane for each label. This approach comes down to stacking [27] and does account with dependencies between labels, since all label counts are used as input for the logistic regression. However, applying logistic regression for each label independently does take considerable resources, as shown in our experiments, where the implementation from Mulan was unable to finish on all large datasets. Together with ML-kNN, IBLR can be considered a state-of-the art method [8].

Spyromitros et al. propose BRkNN [16], where kNN is combined with the Binary Relevance ensemble. The main contribution of BRkNN is also adopted by us. We also compute the k nearest neighbors once, independently of the number of labels. The authors also implement two extensions: BRkNN-a and BRkNN-b. BRkNN-b minimizes the label cardinality between predicted and actual label sets, while BRkNN-a returns the label with highest score as relevant, even if this label is below the threshold, since for most benchmark multi-label datasets an empty set of labels is rare. We have implemented both extensions. Compared to instance based kNN, BRkNN uses a different scoring function, which is the fraction of labels found in the $k$ nearest neighbors. Also, BRkNN does not make use of an inverted index, however it is not difficult to adapt our current implementation to support the scoring function of BRkNN. In future work it would be interesting to experimentally validate the effect on performance of trying different scoring variations and feature representations.

Wang et al. propose an Enhanced kNN algorithm (EkNN) [25], which uses a method similar to instance kNN, but based on *BM25* similarity and a more elaborate thresholding scheme, which scored first in the second Pascal challenge on Large Scale Hierarchical Text Classification (LSTHC2), that includes the Wiki-LSHTC dataset, on example-based accuracy and example-based F1, with reported values comparable to the values we report for LCIF in section IV-E3. EkNN, however, has a larger range of hyper-parameters (both for BM25 and thresholding) to tune and is only applicable to text categorization. Like EkNN we adopted an implementation based on an inverted index that can scale to extreme datasets. Note that we did not experiment with more elaborate feature representation techniques, for example selecting only the top-k features based on information-gain, or computing tf-idf values instead of binary bag-of-words representation for text categorization.

Recently there is an increased research interest in extreme multi-label classification algorithms such as FastXML [24]. Here, most methods try to reduce the dimensionality of the label space, or build a hierarchical ensemble of tree-based models where the number of models to train is logarithmic in the number of labels. The problem setting is, however, different and different ranking based evaluation metrics are optimized directly, such as nDCG for FastXML. Interestingly, this active research domain also fuses ideas from both recommendation and classification. Our approach does not make any assumptions about labels being organized hierarchically. We also try to optimize different evaluation metrics as discussed in Section IV-C.

Finally, it is worth mentioning that a lot of effort has been invested in research on exact *all-pairs similarity search* (or similarity join) based on cosine similarity (and other similarity functions), as well as *approximations* with formal bounds to the all-pairs similarity problem. It would be interesting to verify the impact on performance on the current implementation especially on extreme datasets. Our current implementation is based on a straightforward inverted index. More advanced exact techniques discussed by Bayardo et al. [28] and Awekar and Samatova [29] and/or approximations [30] could, in theory, be integrated with our methods, and result in further performance improvements.

## VI. CONCLUSION

In this paper we propose LCIF, a novel multi-label learning algorithm. Our predictions are based on the label sets of nearest neighbor instances in the training dataset. We use two different methods to locate nearest neighbors. The instance based method finds the top k instances that are most similar to the features of the current test instance. The feature based method searches for the top k labels most similar to each feature, where similarity is defined column-wise. In a final step we add the (cosine similarity) weighted scores for both methods using a straightforward linear combination, and apply a single threshold to predict the label set for each unseen test instance.

Experiments on real-world sparse multi-label datasets from the text categorization and scene classification domain, show that our method improves classification results for example-based accuracy and F1, and both label-based micro- and macro-averaged F1, compared to various state-of-the-art algorithms such as ML-kNN, IBLR and Binary Relevance with SVM.

Furthermore, we discuss the design of an efficient implementation, based on an inverted index and efficient sparse computation of cosine similarity. We analyzed the runtime behavior and experimentally validated that our implementation is extremely fast, requiring only seconds or minutes to complete on large datasets, where other methods take minutes or hours. Our method scales to extreme datasets having millions of instances, features and labels, producing promising results. In contrast to many multi-label algorithms we do not train multiple models, and compute both the instance and feature based neighborhoods only once, independent from the number of labels.

Finally, our prediction functions were inspired by related work in recommender research, such as user-based and item-based collaborative filtering algorithms and combinations of these two methods. Within the field of multi-label classification, however, the ideas from item-based or feature-based approaches have not been adapted before. For future work we see potential to further improve multi-label learning algorithms by adapting research from collaborative filtering. We also see potential to further improve the performance of the neighborhood search, integrating algorithms from exact or approximate all-pair similarity computation in our approach.

## REFERENCES

[1] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, 2006.

[2] Y. Yang, "A study of thresholding strategies for text categorization," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2001, pp. 137–145.

[3] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," *Machine learning*, vol. 85, no. 3, pp. 333–359, 2011.

[4] R. E. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Machine learning*, vol. 37, no. 3, pp. 297–336, 1999.

[5] C. Vens, J. Struyf, L. Schietgat, S. Džeroski, and H. Blockeel, "Decision trees for hierarchical multi-label classification," *Machine Learning*, vol. 73, no. 2, pp. 185–214, 2008.

[6] A. Elisseeff, J. Weston *et al.*, "A kernel method for multi-labelled classification," in *NIPS*, vol. 14, 2001, pp. 681–687.

[7] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern recognition*, vol. 40, no. 7, pp. 2038–2048, 2007.

[8] E. Gibaja and S. Ventura, "Multi-label learning: a review of the state of the art and ongoing research," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 6, pp. 411–444, 2014.

[9] J. Bennett, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007. New York, NY, USA, 2007, p. 35.

[10] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "Grouplens: an open architecture for collaborative filtering of netnews," in *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. ACM, 1994, pp. 175–186.

[11] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1998, pp. 43–52.

[12] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 285–295.

[13] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A java library for multi-label learning," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2411–2414, 2011.

[14] W. Cheng and E. Hüllermeier, "Combining instance-based learning and logistic regression for multilabel classification," *Machine Learning*, vol. 76, no. 2-3, pp. 211–225, 2009.

[15] I. Triguero and C. Vens, "Labelling strategies for hierarchical multi-label classification techniques," *Pattern Recognition*, vol. 56, pp. 170–183, 2016.

[16] E. Spyromitros, G. Tsoumakas, and I. Vlahavas, "An empirical study of lazy multilabel classification algorithms," in *Hellenic conference on Artificial Intelligence*. Springer, 2008, pp. 401–406.

[17] K. Verstrepen and B. Goethals, "Unifying nearest neighbors collaborative filtering," in *Proceedings of the 8th ACM Conference on Recommender systems*. ACM, 2014, pp. 177–184.

[18] J. Wang, A. P. De Vries, and M. J. Reinders, "Unifying user-based and item-based collaborative filtering approaches by similarity fusion," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2006, pp. 501–508.

[19] A. Zubiaga, "Enhancing navigation on wikipedia with social tags," *arXiv preprint arXiv:1202.5469*, 2012.

[20] 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.

[21] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.

[22] J. Read, B. Pfahringer, and G. Holmes, "Multi-label classification using ensembles of pruned sets," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE, 2008, pp. 995–1000.

[23] H. Jain, Y. Prabhu, and M. 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*. ACM, 2016, pp. 935–944.

[24] Y. Prabhu and M. Varma, "Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 263–272.

[25] X.-l. Wang, H. Zhao, and B. Lu, "Enhanced k-nearest neighbour algorithm for largescale hierarchical multi-label classification," in *Proceedings of the Joint ECML/PKDD PASCAL Workshop on Large-Scale Hierarchical Classification, Athens, Greece*, vol. 5, 2011.

[26] Z. Younes, F. Abdallah, and T. Denœux, "Multi-label classification algorithm derived from k-nearest neighbor rule with label dependencies," in *Signal Processing Conference, 2008 16th European*. IEEE, 2008, pp. 1–5.

[27] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[28] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 131–140.

[29] A. Awekar and N. F. Samatova, "Fast matching for all pairs similarity search," in *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT'09. IEEE/WIC/ACM International Joint Conferences on*, vol. 1. IEEE, 2009, pp. 295–300.

[30] R. B. Zadeh and A. Goel, "Dimension independent similarity computation." *Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1605–1626, 2013.