International Journal of Data Science and Analytics manuscript No.
(will be inserted by the editor)

Combining instance and feature neighbours for extreme multi-label classification

Len Feremans[1] · Boris Cule[2] · Celine Vens[3,4] · Bart Goethals[1,5]

Accepted: 26 February 2020

Abstract Extreme multi-label classification problems occur in different applications such as prediction of tags or advertisements. We propose a new algorithm that predicts labels using a linear ensemble of labels from instance- and feature-based nearest neighbours. In the feature-based nearest neighbours method, we precompute a matrix containing the similarities between each feature and label. For the instance-based nearest neighbourhood, we create an algorithm that uses an inverted index to compute cosine similarity on sparse datasets efficiently. We extend this baseline with a new top-k query algorithm that combines term-at-a-time and document-at-a-time traversal with tighter pruning based on a partition of the dataset. On ten real-world datasets, we find that our method outperforms state-of-the-art methods such as multi-label k-nearest neighbours, instance-based logistic regression, binary relevance with support vector machines and FastXML on different evaluation metrics. We also find that our algorithm is orders of magnitude faster than these baseline algorithms on sparse datasets, and requires less than 20 ms per instance to predict labels for extreme datasets without the need for expensive hardware.

Keywords Extreme multi-label classification · Item-based collaborative filtering · k-nearest neighbours · Top-k queries · Information retrieval

✉ len.feremans@uantwerpen.be
✉ boris.cule@uantwerpen.be
✉ celine.vens@kuleuven.be
✉ bart.goethals@uantwerpen.be
[1] Department of Computer Science, Universiteit Antwerpen, Belgium
[2] Department of Accountancy and Finance, Universiteit Antwerpen, Belgium
[3] Faculty of Medicine, Katholieke Universiteit Leuven, Belgium
[4] ITEC, imec research group at Katholieke Universiteit Leuven, Belgium
[5] Monash University, Melbourne, Australia

A preliminary version of this paper was published as "Combining instance and feature neighbors for efficient multi-label classification" at DSAA 2017 [1].

1 Introduction

Multi-label classification problems occur in a large variety of domains, such as text categorization, where a document has multiple categories, scene classification, where various regions of an image have a label, and bioinformatics, where we are interested in predicting numerous functions for a gene. In this work, we consider *sparse* datasets that occur naturally in these domains, i.e., where features correspond to patterns, such as term frequency-inverse document scores for word occurrences in texts or clusters in images.

Two main strategies exist for solving the multi-label task. The first strategy is to reduce the multi-label problem into a combination of single-label problems. The *binary relevance* method ignores label dependencies and trains a separate model to predict each label independently of other labels using one-versus-all sampling [2]. *Classifier chains* approximate label dependencies, but also require to train a separate model for each label [3]. If the set of labels L is large, training $|L|$ different models using binary relevance or classifier chains is *not scalable*. A second strategy is to *adapt* existing single-label classifiers to output multiple labels. Well-known adaptations of single-label classification techniques have been made to adaBoost, decision trees, support vector machines, k-nearest neighbour and others [4–7].

Trending challenges in multi-label classification research include methods that account for possible *dependencies between labels*, deal with *label skew* (where most labels are only covered by a few instances), and consider the *computational cost* of building a model [8]. *Extreme multi-label classification* is an active research topic that considers the computational cost of generating a model when the number of labels is very high. Recently several methods have been proposed that try to address these challenges. These methods reduce the *dimensionality* of the label space or build a hierarchical *ensemble of tree-based models*, such as FastXml, where the number of models to train is logarithmic in the number of labels [9–11]. While these approaches are accurate and fast at *testing* time, they require significant resources at *training* time. Moreover, each of these methods needs to tune many *hyperparameters* for optimal performance.

Related to multi-label classification is the field of *recommender systems*. Here, the task is to rank items a user might click, often based on past preferences. Two well-known recommender systems are user-based and item-based collaborative filtering [12–14]. An advantage of both approaches is that the results can be *explained*, i.e., using "people who liked this item also liked" type of explanations. Applying these techniques for multi-label classification is a major goal of this work.

User-based collaborative filtering is a memory-based learning algorithm where we need to compute the *nearest neighbours*. The problem of finding the *exact* set of k-nearest neighbours is studied under different names: all pairs similarity search, top-k set similarity joins, k-nearest neighbour graph construction and top-k queries [15–21]. We propose a new algorithm to compute the exact k-nearest neighbours using pruning. Similar to search problems in information retrieval, we want to find a set of instances in our training dataset that is the most similar to a test instance for which we wish to predict labels. A key difference with information retrieval is that our test instances, or queries, typically have many more non-zero feature values than is usual in search, which has a severe impact on performance. Therefore, we adapt research from information retrieval and create a new *top-k query* algorithm. Technically, we combine *term-at-a-time* and *document-at-a-time* traversal using *Weak-AND* pruning [22]. Different from the previous work in information retrieval, our primary reason for first traversing the instances using term-at-a-time, is based on finding proper constraints such that more candidate instances get pruned using a tighter upper-bound [21].

In this work, we make the following contributions. First, we implement *instance-based k-nearest neighbours*, an adaptation of user-based collaborative filtering, for multi-label classification. Next, we implement the *feature-based k-nearest neighbours* method, an adaptation of item-based collaborative filtering, that computes the nearest labels for each feature in a column-wise manner. Finally, we combine both instance- and feature-based neighbourhood predictions using a linear ensemble. Second, we make the k-nearest neighbours search scalable for sparse datasets with an extremely high number of labels, features and instances. The baseline method uses an *inverted index* and organises computation so that we only perform non-zero similarity term computations, which we improve with a top-k query algorithm.

We validate the accuracy of our method on 10 real-world datasets and compare with multi-label classification methods such as multi-label k-nearest neighbours, instance-based logistic regression, binary relevance with support vector machines as a base learner and FastXml on different evaluation metrics [7, 11, 23]. We also compare the pruning ability and runtime performance of our k-nearest neighbours algorithm with state-of-the-art top-k query retrieval methods, such as term-at-a-time traversal, in-memory document-at-a-time traversal with Weak-AND pruning and Fagin's threshold algorithm [19, 21, 22]. Compared to the original version of this paper [1], we improve our algorithms for making predictions in different ways and propose a new algorithm to compute the nearest neighbours more efficiently based on top-k queries.

The remainder of this paper is organised as follows. In Section 2 we define the problem setting. In Section 3 we describe our algorithm for multi-label classification. In Section 4 we describe our method for finding the k-nearest neighbours more efficiently. We experimentally validate our method and compare it with existing state-of-the-art methods in Section 5 and discuss related and future work in Section 6. Finally, we conclude in Section 7.

2 Problem Setting

Definition 1 (Multi-label Dataset) *Let $X \in \mathbb{R}^{N \times M}$ denote the set of training points and let $Y \in \{0,1\}^{N \times L}$ denote the set of labels. The training dataset $\mathcal{D}$ consists of N instances $\mathcal{D} = \{(x_1, \mathbf{y_1}), \ldots, (x_N, \mathbf{y_N})\}$ where each M-dimensional feature vector $x_i \in \mathbb{R}^M$ is associated with an L-dimensional label vector $\mathbf{y_i} \in \{0,1\}^L$.*

We assume that feature values are transformed into real numbers higher than or equal to zero.

Definition 2 (Cardinality) *We define feature cardinality as the average number of non-zero features for*

each instance. Analogously, we define label cardinality as the average number of labels for each instance, that is:

$$fcard(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\delta(x_{i,j}),$$

$$lcard(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{L} y_{i,j},$$

where $\delta(x_{i,j})$ is 1 if $x_{i,j} > 0$ and 0 otherwise and $y_{i,j}$ is the binary value of label j for instance i.

Definition 3 (Column) *We use $f_j = \{x_{1,j}, \ldots, x_{N,j}\} \in \mathbb{R}^N$ to denote the column of values for feature j. Likewise we use $l_j = \{y_{1,j}, \ldots, y_{N,j}\} \in \{0,1\}^N$ to denote the column of values for label j.*

Definition 4 (Density) *We define feature and label density as*

$$fdens(f_j) = \frac{|\{x_{i,j} \mid x_i \in X \wedge x_{i,j} \neq 0\}|}{N},$$

$$ldens(l_j) = \frac{|\{y_{i,j} \mid y_i \in Y \wedge y_{i,j} \neq 0\}|}{N}.$$

For sparse datasets, we observe that feature and label cardinality are small compared to M and L, and that there is a skewed distribution where only a few features (or labels) have a high density, and most features (or labels) have a density close to 0.

Problem 1 *The task for multi-label classification is to predict a subset of labels for each test instance $x_q \in X_{test}$ for which the set of labels $\mathbf{y_q}$ is unknown. Formally we have to learn a function $h : X \to \{0,1\}^L$ that optimises a selected evaluation metric.*

The function h can be implemented as $h(x) = t(f(x))$ where f produces a confidence (or probability) score for each label and t is a threshold function.

3 Linear Combination of Instance- and Feature-based kNN

Our classification method consists of instance-based k-nearest neighbours (kNN), feature-based kNN and the linear combination of both predictions.

3.1 Instance-based kNN

The algorithm begins by searching for the k-nearest neighbours x_i in the training data for each test (or query) instance x_q using *cosine similarity*.

Definition 5 (Instance-based Cosine Similarity) *The cosine similarity is defined as*

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

where we make sure that all instances are normalised to unit length during preprocessing.

Definition 6 (Instance-based Confidence Score) *To compute the confidence score for instance x_q for (a single) label y_j we define the following function:*

$$\hat{y}^{\text{INS}}_{q,j} = \frac{\sum_{x_i \in \text{KNN}(x_q)} y_{i,j} \cdot sim_{\text{INS}}(x_q, x_i)^\alpha}{\sum_{x_i \in \text{KNN}(x_q)} sim_{\text{INS}}(x_q, x_i)^\alpha}.$$

This function is an adaption of user-based collaborative filtering, where the similarity in feature values replaces the similarity between user preferences, and we do not recommend an item but a label [24]. Also, we apply a power transformation to the similarities. For example, if we apply the power $\alpha = 2$, we give similarities closer to 1 more weight compared to similarities closer to 0.01. Vice versa, $\alpha = 0.5$ has the reverse effect.

Algorithm We retrieve the k-nearest neighbours by using an *inverted index* (IID) and compute only non-zero terms for similarity. CreateIndex is shown in Algorithm 1. We associate each feature with a set of (training) instances and their non-zero feature value.

In InstanceKnnSearch, shown in Algorithm 2, we compute the cosine similarity between x_q and all instances incrementally thereby only computing non-zero terms of each dot product. We first loop over each non-zero feature $x_{q,j}$, then fetch all candidates x_i that have a (non-zero) $x_{i,j}$ value from the IID and then increment the partial dot product $x_{q,j} \cdot x_{i,j}$. Finally, we use *partial sort*, i.e., using heap sort, to maintain the top k instances with the highest cosine similarity.

We compute the prediction scores for each label using InstanceKnnPredict, shown in Algorithm 3. We only compute predictions for labels that are present in

Algorithm 1: CreateIndex($\mathcal{D}$) Creates an inverted index for instance-based kNN baseline

Input: A dataset $\mathcal{D}$
Result: An inverted index (IID) of the dataset

```
IID ← EMPTY_HASH_MAP();
/* For each instance */
for x_i in X do
    /* For each non-zero feature value */
    for x_{i,j} ≠ 0 in x_i do
        IID[j] ← IID[j] ∪ ⟨x_i, x_{i,j}⟩;
return IID;
```

Algorithm 2: InstanceKnnSearch($x_q, k,$ IID) Finds the k-nearest neighbours for x_q in $\mathcal{D}$

Input: A query instance x_q, number of neighbours k, an inverted index IID
Result: k-nearest neighbours and their similarities

```
S ← EMPTY_HASH_MAP();
/* For each non-zero feature value in x_q */
for x_{q,j} ≠ 0 in x_q do
    /* For each non-zero feature value x_{i,j} */
    for ⟨x_i, x_{i,j}⟩ in IID[j] do
        /* Compute partial dot product */
        S_{q,i} ← S_{q,i} + x_{q,j} · x_{i,j};
kNN ← HEAP_SORT_TOP_K(S, k);
return kNN;
```

Algorithm 3: InstanceKnnPredict(x_q, kNN, α) Computes instance-based confidence scores for labels

Input: A query instance x_q, kNN contains the k-nearest neighbours and their similarities, α for the power transform
Result: Prediction scores for labels

```
ŷ ← EMPTY_HASH_MAP();
/* For each instance in kNN */
for x_i in kNN do
    /* For each non-zero label j */
    for y_{i,j} ≠ 0 ∈ y_i do
        /* Compute partial similarity-weighted
           confidence score */
        ŷ_j ← ŷ_j + S^α_{q,i};
normalise ŷ with Σ_{x_i ∈ kNN} S^α_{q,i};
return ŷ;
```

any of the k-nearest neighbours. As in InstanceKnnSearch, we organise the computation so that we only compute non-zero increments to each label score. Remark that in our implementation, we compute similarities and predictions in *parallel*. We initialise shared hash tables statically, so subsequent updates to partial scores (or similarities) from different threads can occur in a lock-free manner [25]. This results in performance gains almost *linear* with the number of processors.

Complexity For instance-based kNN search the complexity is $\mathcal{O}(N \times M)$, but in practice, we observe that the average runtime is closer to $\mathcal{O}(\tilde{n} \times \tilde{m})$ for sparse datasets. Here $\tilde{n}$ is proportional to the average number of *candidate* instances, i.e., instances fetched from the inverted index, and $\tilde{m}$ is proportional to the feature cardinality. We will analyse the runtime of this algorithm in Section 5. We remark that the expensive neighbourhood search is performed once and is *independent* of the number of labels L [26].

Hyperparameter Optimalisation An essential advantage of our method is that for optimising k using *grid search* we need to compute the nearest neighbours only *once*. First, we search for the nearest neighbours with a maximal value of k_{max}. For smaller values of k, we just take the first k values of the cached k_{max}-nearest neighbours. We argue that this makes k a *virtual* hyperparameter, meaning that we optimise it efficiently on a validation set.

Second-order Instance Variation A possible disadvantage of the instance-based kNN method is that we ignore *inter-label dependencies*: the prediction for a given label is obtained independently of the values of other labels. We propose an extension that uses the second-order neighbourhood to handle this situation. For details, we refer to our original paper [1].

3.2 Feature-based kNN

Feature-based kNN is an adaptation of item-based collaborative filtering for multi-label classification [14].

Definition 7 (Feature-based Cosine Similarity) *For feature-based predictions, we compute the cosine similarity between each feature column f_i and each label column l_j using,*

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

where we make sure that all feature and label vectors are normalised to unit length during preprocessing.

Definition 8 (Feature-based Confidence Score) *We compute the confidence score for a test instance x_q and a label y_j using:*

$$\hat{y}^{\text{FL}}_{q,j} = \frac{\sum_{i=1}^{M} x_{q,i} \cdot sim_{\text{FL}}(f_i, l_j)^{\beta}}{\sum_{i=1}^{M} x_{q,i}},$$

where we apply the power β to the similarities.

We compute the full similarity matrix between all pairs of feature and label columns at training time. When L is extremely large, we consider a variation to feature-based kNN, that only computes a (non-zero) prediction for labels that occur at least once in the neighbourhood, i.e.,

$$y_j \in \bigcup_{x_{q,i} \in x_q \wedge x_{q,i} \neq 0} \text{kNN}(f_i),$$

which is more scalable given an extreme number of labels.

Algorithm For feature-based kNN we first compute a matrix containing the similarities between all features and labels in $\mathcal{D}$. We use sparse data structures

Algorithm 4: CreateSimilMatrix($\mathcal{D}$) Computes similarities between all features and labels for feature-based kNN

Input: A dataset $\mathcal{D}$
Result: Similarity matrix S

```
/* Create inverted index for labels */
IID ← EMPTY_HASH_MAP();
/* For each instance */
for ⟨x_i, y_i⟩ in D do
    /* For each non-zero label value */
    for y_{i,j} ≠ 0 in y_i do
        IID[j] ← IID[j] ∪ {x_i};
/* Compute similarities */
S ← 0.0^{M×L};
/* For each label j get instances from IID */
for y_j ≠ 0 in IID do
    for x_i in IID[j] do
        /* For each non-zero feature k */
        for x_{i,k} ≠ 0 in x_i do
            /* Compute partial dot product */
            S_{j,k} ← S_{j,k} + x_{i,k} · y_{i,j};
return S;
```

Algorithm 5: FeatureKnnPredict(x_q, S, β) Computes feature-based confidence scores for labels

Input: A query instance x_q, a similarity matrix S, β for the power transform
Result: Prediction scores for labels

```
ŷ ← EMPTY_HASH_MAP();
/* For each non-zero feature i */
for x_{q,i} in x_q do
    /* For each label j with non-zero similarity
       with feature i */
    for S_{j,i} ≠ 0 ∈ S_{*,i} do
        /* Compute partial confidence score */
        ŷ_j ← ŷ_j + x_{q,i} · S^β_{j,i};
normalise ŷ with Σ x_{q,i};
return ŷ;
```

as we assume most values for label column y_j and feature column f_k will be 0. We use a technique similar to InstanceKnnSearch and compute the feature-based cosine similarity incrementally. The function CreateSimilMatrix is shown in Algorithm 4. First, we create an index that associates each of the L labels with a set of positive instances. After indexing, we fetch positive instances x_i for each label. For each instance x_i, we traverse over each non-zero feature $x_{i,k}$ and compute a non-zero term of the dot product between label y_j and feature f_k.

FeatureKnnPredict is shown in Algorithm 5. We follow a similar approach as InstanceKnnPredict to only compute non-zero terms of each confidence score. We remark that we also experimented with an alternative confidence score that considers the k nearest features for each label, but in preliminary experiments, this did not increase average results while requiring an extra hyperparameter.

Complexity Computing the similarity matrix has a complexity of $\mathcal{O}(\frac{1}{2}M \times L \times N)$. However, in practice runtime is closer to $\mathcal{O}(\frac{1}{2}M \times \tilde{l} \times \tilde{n})$ for sparse datasets. Here $\tilde{l}$ 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 values (or labels) column-wise. We observe that the similarity matrix can be computed once at *training time* for all test instances, while at *test time* only prediction scores have to be computed. This makes feature-based kNN very efficient and is arguably one of the reasons why item-based collaborative filtering is so popular in real-world web applications.

3.3 Linear Combination

We introduce a straightforward ensemble method based on the Linear Combination of the confidence scores of the Instance- and Feature-based k-nearest neighbours (Lcif). Combinations of the two techniques have been studied in collaborative filtering research, but not in multi-label classification [24, 27].

Definition 9 (LCIF Confidence Score) *We compute the confidence score for test instance x_q for label y_i using*

$$\hat{y}_{q,i} = \lambda\ \hat{y}^{\text{Ins}}_{q,i} + (1 - \lambda)\ \hat{y}^{\text{Fl}}_{q,i},$$

where $\lambda \in [0, 1]$ is a hyperparameter that is optimised on a validation sample for each evaluation metric.

For datasets with many labels we compute this score only for candidate labels, that is, labels i that have a non-zero score for either $\hat{y}^{\text{Ins}}_{q,i}$ or $\hat{y}^{\text{Fl}}_{q,i}$. The main Lcif algorithm is shown in Algorithm 6.

3.4 Thresholding

To obtain a set of predicted labels, we apply a *single* threshold [3, 28].

Definition 10 (Single Threshold) *Given confidence scores $\hat{y}_{q,j}$ for instance x_q and each label y_j, we predict a set of labels using a single threshold t:*

$$h(x_q) = \{y_j \mid \hat{y}_{q,j} >= t\},\ \forall y_j \in L.$$

Algorithm 6: LCIF($\mathcal{D}, X_{test}, k, \alpha, \beta, \lambda, t$) Predicts labels based on a linear combination of instance- and feature-based weighted similarities

Input: A training dataset $\mathcal{D}$, one or more test instances in X_{test}, number of neighbours k, parameters α and β for the power transform, λ for the linear combination and t the single threshold
Result: Predicted labels for each instance in X_{test}

```
/* Train: create index and feature-based
   similarity matrix */
IID ← CreateIndex(D);
S ← CreateSimilMatrix(D);
/* Predict: compute kNN and predictions for each
   test instance */
Ŷ ← ∅;
for x_q in X_test do
    kNN_q ← InstanceKnnSearch(x_q, k, IID);
    ŷ_j^Ins ← InstanceKnnPredict(x_q, kNN_q, α);
    ŷ_q^Fl ← FeatureKnnPredict(x_q, S, β);
    ŷ_q^Lcif ← λ ŷ_q^Ins + (1 − λ) ŷ_q^Fl;
    ŷ_q ← {ŷ_{q,j} | ŷ_{q,j} ∈ ŷ_q^Lcif : ŷ_{q,j} ≥ t};
    Ŷ ← Ŷ ∪ ŷ_q;
return Ŷ;
```

We determine t automatically by selecting the value of t that minimises the difference in *label cardinality* between the actual and predicted label set. That is,

$$\operatorname*{argmin}_t \left| \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{L} \delta(\hat{y}_{i,j} > t) - lcard(\mathcal{D}) \right|,$$

where $\delta(\hat{y}_{i,j} > t)$ returns 1 if the confidence score is higher than t and 0 otherwise. Alternatively, we make use of a *label-specific threshold* [29, 30]. This allows us to lower the threshold for *minority* labels in imbalanced datasets.

Definition 11 (Label-specific threshold) *We predict a set of labels using a separate threshold t_{y_j} for each label y_j:*

$$h(x_q) = \{y_j \mid \hat{y}_{q,j} >= t_{y_j}\},\ \forall y_j \in L$$

We remark that for many multi-label datasets, there is at least one label for every instance. Therefore, if the highest-scoring label is below the threshold, we ignore the threshold value [26].

4 Fast kNN Search

A problem with the instance-based kNN search is that because of its inherent $O(N)$ complexity to search for the k-nearest neighbours, it does not scale to extreme datasets. In the previous section, we created a baseline algorithm optimised for sparse datasets. In this section, we improve on this baseline. This is important in interactive applications since we must perform the search at *test time* and every millisecond is important.

4.1 Indexing

The problem of instance-based kNN is similar to top-k queries algorithms in information retrieval. A key difference, however, is that search queries are typically much *shorter*, having less than 10 terms. Real-world search engines often limit search queries to 50 terms. In our case, each query is a test instance, that consists of many more non-zero dimensions on average. We will show that existing state-of-the-art top-k query algorithms are less efficient in this setting. Therefore, we propose a new top-k query algorithm for computing the exact set of k-nearest neighbours ranked on cosine similarity.

Top-k Query Algorithm Our method is an extension of the work of Fontoura et al. that combines two ways to traverse the most relevant instances (or *documents*) given a certain test instance (or *query*): Document-at-a-time (DAAT) and Term-at-a-time (TAAT) [21]. Using this framework, Algorithm 2 computes cosine similarity following a TAAT strategy, that is, for every non-zero feature, or *term*, of a test instance we fetch all instances, or *documents*, from the inverted index and increment the partial similarity of each document with the non-zero weight in that dimension. In DAAT traversal we keep all documents in the inverted index sorted on document order and traverse through all *posting lists* (the inverted index for each term) simultaneously similar to a merge join. Using this document-per-document manner, we can compute the complete similarity score for each document in turn. We propose to first traverse using TAAT and next using DAAT. We focus on *memory-resident* indexes, thereby assuming that memory in present-day is often large enough to maintain the complete index [21].

Partitioning We observe that in real-world datasets feature values in most dimensions have a high standard deviation. For example, we can encode text documents using a bag-of-words encoding with term frequency-inverse document frequency and get a significant difference in values between terms that frequently occur in one document but seldom occur in others, and words that are infrequent in one document and frequent overall. This variation is a useful property that we exploit.

Algorithm 7: CreateIndexPartition($\mathcal{D}, m$)
Partitions data and builds indexes for both Taat and Daat traversal

Input: A dataset $\mathcal{D}$, a parameter m that controls the partition
Result: Index structures for Taat en Daat traversal

```
IID ← CreateIndex(D);
/* Compute partition */
I_taat ← ∅;
for f_j in D do
    {⟨x'_1, x'_{1,j}⟩, ..., ⟨x'_m, x'_{m,j}⟩} ← heap_sort_top_k(IID[f_j], m);
    I_taat ← I_taat ∪ {x'_1, ..., x'_m};
I_daat ← D \ I_taat;
/* Create indexes */
IID_taat ← CreateIndex(I_taat);
IID_daat ← CreateIndex(I_daat);
sort IID_taat descending on feature value;
sort IID_daat ascending on document id;
Max_daat ← max feature value for each f_j in I_daat;
Φ ← {I_taat, I_daat, IID_taat, IID_daat, Max_daat};
return Φ;
```

Definition 12 (Partition) *First, we partition all instances into two disjoint sets:*

$$I_{taat} = \{x_i \mid x_i \in \mathcal{D} \wedge \exists\ x_{i,j} \in x_i : rank(x_{i,j}, f_j) \leq m\},$$

$$I_{daat} = \mathcal{D} \setminus I_{taat},$$

where we use $rank(x_{i,j}, f_j) \leq m$ to denote that feature value $x_{i,j}$ is ranked before place m in the (descending) ordered posting list of feature j.

The partition strategy has three useful consequences. Firstly, by using the partitioned and sorted inverted index, we encounter high feature values first during Taat traversal and are more likely to find instances with a high cosine similarity early on. This is important since we prune instances during Daat if the similarity cannot be higher than the k^{th} candidate after Taat traversal. Secondly, we use Weak-AND (Wand) for pruning during Daat traversal [22]. The Wand upper bound depends on the maximum feature value in each dimension. Because of the partitioning, we guarantee that this maximum is smaller than the first m values. Thirdly, by having two disjoint partitions, additional overhead for pruning and index creation is minimal.

Algorithm The algorithm CreateIndexPartition for both Taat and Daat index creation is shown in Algorithm 7 and consists of two phases. In the first phase, we create an inverted index for all documents and compute the partition of all instances. For each feature or term, we find the instances with the m highest feature values and add these instances to I_{taat}. We can compute this efficiently using heap sort on the posting list for each term. In the second phase, we make a complete Taat index by adding all feature values for all instances in I_{taat}. Remark that this index also includes feature values that are *not* in the top-m. The rationale for making the index complete is that we can compute the full similarity for each instance in I_{taat} without resorting to less efficient random access operations. Finally, we add all remaining documents to the Daat index sorted on document ID. We also maintain the maximal feature value in each posting list. In our implementation, we keep the feature values local to the index as we want to avoid cache misses. We remark that in practice we set the parameter m to a small value, e.g., between 1 and 25 for large datasets and closer to 100 for extreme datasets. Alternatively, m could be defined relatively as the percentage of all documents in I_{taat}.

4.2 Fast k-nearest Neighbour Search using Term- and Document-at-a-time with Weak-And Pruning

Pruning In essence Daat is a merge join over the different posting lists sorted on document ID. We can, however, skip instances based on an upper bound using the Weak-And iterator [22].

Definition 13 (Upper Bound Cosine Similarity) *Given a query x_q we compute an upper bound on each cosine similarity term:*

$$UB_{q,j} = max(\{x_{i,j} \mid x_i \in I_{daat}\}) \cdot x_{q,j}.$$

For any $x_i \in I_{daat}$ it holds that

$$sim_{\text{Ins}}(x_q, x_i) \leq \sum_{x_{q,j} \in x_q} UB_{q,j}.$$

Therefore, we prune instances without computing the full similarity if

$$\sum_{x_{q,j} \in x_q \wedge x_{i,j} \neq 0} UB_{q,j} \leq \sum_{x_{q,j} \in x_q} UB_{q.j} \leq \theta,$$

where θ is the k^{th} largest similarity of instances already visited during Taat and the ongoing Daat traversal. Remark that for most instances x_i only a small subset of the features of x_q will also be non-zero in x_i.

Algorithm The main algorithm for finding the exact k-nearest neighbours is shown in Algorithm 8. InstanceKnnFast consists of two phases. We start by computing the k nearest neighbours of all instances in I_{taat} using Taat by calling InstanceKnnSearch. We then add these instances to a *heap*. A heap is more efficient for managing the current instances as we want to access the current k^{th} maximal similarity efficiently. In

the second phase, we traverse candidate instances using DAAT (line 3-20) and our extension to WAND pruning. We start by initialising the upper bound for x_q by computing the dot product between every non-zero feature value $x_{q,j}$ and the precomputed value $\text{MAX}_{daat}[j]$. Next, we iterate over instances in I_{daat} that have at least one feature shared with x_q (as determined by IID_{daat}). We start our while loop with the first document ($offsets_j = 0$) in each posting list and order these instances on ascending ID (line 8-11).

If x_{k_1} is the document with the smallest ID for any posting list then for any other document x_{k_2}, with $k_2 > k_1$, we know that x_{k_1} has a zero value in that posting list. Therefore, we prune x_{k_1} if UB_{q,k_1} is smaller than θ. Likewise, we prune x_{k_2} if $UB_{q,k_1} + UB_{q,k_2}$ is smaller than θ, etc. We increment the upper bound UB_{cur} in a feature-by-feature manner and prune any documents that are below this accumulated value (line 12-17). We stop when a *pivot* document is identified, meaning the first document that is higher than the upper bound. We then compute the full similarity for the pivot instance and add it to the heap, where it will replace a candidate if the similarity is higher (line 19). Finally, we advance each posting list to the next document. If the next document identifier is larger than the pivot, we do not update the offset. Otherwise, we advance the posting list to point to a document with an identifier after the current pivot. In the worst case, the pivot document is always the first candidate, and we advance by one document at a time. In the best case, however, there are $|x_q|$ documents and the pivot document is the last document. Then we can advance by $|x_q| - 1$ documents, thereby pruning these documents without computing the full similarity.

Algorithm 8: INSTANCEKNNFAST(x_q, k, Φ)
Finds the *exact* k-nearest neighbours from $\mathcal{D}$ based on two traversal strategies and pruning

Input: A query instance x_q, number of neighbours k, partitioned inverted index structures Φ
Result: k-nearest neighbours
`/* (i) TAAT traversal */`
1 $\text{kNN}_{taat} \leftarrow$ INSTANCEKNNSEARCH($x_q, k, \text{IID}_{taat}$);
2 $heap \leftarrow$ CREATE_HEAP(kNN_{taat});
`/* (ii) DAAT traversal and pruning with WAND */`
3 $UB_q \leftarrow 0.0^{|x_q|}$;
4 **for** $x_{q,j} \neq 0$ ***in*** x_q **do** `// Computer upper bound`
5 $UB_{q,j} \leftarrow x_{q,j} \cdot \text{MAX}_{daat}[j]$;
6 $offsets \leftarrow 0^{|x_q|}$;
7 **while** $offsets \neq [-1, \ldots, -1]$ **do**
8 $next \leftarrow \{\}$; `// Enumerate next instances`
9 **for** $x_{q,j} \neq 0$ ***in*** x_q ***and*** $offsets_j \neq -1$ **do**
10 $next \leftarrow next \cup IID_{daat}[j][offsets_j]$;
11 $next \leftarrow$ SORT ASCENDING on document ID;
12 $pivot \leftarrow \emptyset$; `// Find the first candidate`
13 $UB_{cur} \leftarrow 0$;
14 **for** $\langle x_i, x_{i,j} \rangle$ ***in*** $next$ **do**
15 $UB_{cur} \leftarrow UB_{cur} + UB_{q,j}$;
16 **if** $UB_{cur} >$ MIN_HEAP($heap$) **then**
17 $pivot \leftarrow x_i$; **break**;
18 $sim_{q,p} \leftarrow x_q \cdot pivot$; `// Compute similarity`
19 $heap \leftarrow$ PUSH_POP($heap, pivot, sim_{q,p}$);
20 Advance *offsets* so next instances are after *pivot*;
21 **return** *heap*;

5 Experiments

We now study the accuracy and efficiency of LCIF and INSTANCEKNNFAST.

5.1 Experimental Setup

Datasets We have selected five *large* and five *extreme* datasets. Table 1 shows the most important characteristics of each dataset. The datasets are available in well known multi-label repositories [31–33]. The *Medical* dataset consists of nearly 1 000 documents containing free clinical text, originally collected at a children's hospital medical centre's department of radiology. The problem is to assign one or more medical diagnoses or procedures coded using ICD-9-CM based on free 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 binary features. A feature value of 1 indicates that a certain segment in the image belongs to a certain cluster. The *Bibtex* dataset represents a tag assignment problem. The *Delicious* dataset is similar to *Bibtex*. *Wiki10* corresponds to 20 000 Wikipedia articles. For these three datasets, the labels (or tags) were assigned using the social tagging sites Bibsonomy and Del.icio.us. Note that the label cardinality with social tagging is higher. In the *IMDB-F* dataset, 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 than 100 000 summaries. However, there are only 28 movie genres, and the total dictionary of terms is limited.

The *extreme Eurlex* dataset is a collection of documents about European law and has close to 4 000 categories. Reuters Corpus Volume I (*RCV1*) is a benchmark dataset for text categorisation containing more than 700 000 labelled news articles made available by the press agency Reuters. *AmazonCat* contains over a million instances of Amazon products with labels and reviews. We selected the version that has more than 13 000 labels. Finally, *WikiLSHTC* consists of more than

Table 1 Characteristics of five large and five extreme multi-label datasets

Dataset	Train N	Test N_{test}	Features M	Labels L	*lcard*	*fcard*	Avg. *ldens*	Avg. *fdens*
Medical	333	645	1 449	45	1.2	13.4	0.0277	0.0092
Corel5k	5 000	500	499	374	3.5	8.3	0.0094	0.0166
Bibtex	4 880	2 515	1 836	159	2.4	68.7	0.0151	0.0374
Delicious	12 920	3 185	500	983	19.1	18.3	0.0193	0.0366
IMDB-F	72 551	48 368	1 001	28	2.0	19.4	0.0714	0.0194
Eurlex	15 539	3 809	5 000	3 956	5.3	237.0	0.0013	0.0474
Wiki10	14 147	6 617	101 890	30 940	18.6	669.0	0.0006	0.0065
RCV1	623 847	155 962	46 672	2 456	4.8	74.0	0.0019	0.0016
AmazonCat	1 186 239	306 782	203 873	13 330	5.1	71.1	0.0004	0.0003
WikiLSHTC	1 778 352	587 085	1 617 899	325 056	3.3	42.5	0.0001	0.0001

a million instances and features and 325 000 categories. The source is Wikipedia. A 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 [34]. *Eurlex*, *Wiki10*, *RCV1*, *AmazonCat* and *WikiLSHTC* are extreme datasets since they have thousands of labels [33]. We remark that although the extreme datasets contain more labels and features, they are also extremely *sparse* and the cardinality of labels and features remains comparable to the large datasets. A key advantage of our method is that it is optimised for sparse datasets.

State-of-the-art Methods For the large datasets, we compare the performance of the instance- and feature-based kNN methods and their linear combination LCIF with the following state-of-the-art algorithms. Multi-label k-nearest neighbours (ML-KNN) [7] and instance--based logistic regression (IBLR) [23] are two seminal instance-based multi-label algorithms. Binary relevance with support vector machines as a binary classifier (BR-SMO) is one of the best-performing algorithms [2, 35]. For optimising the threshold for the other methods, we use OneThreshold, which optimises a single threshold on the selected evaluation metric [36]. For the extreme datasets, we compare the results of LCIF with the published results of FASTXML [11], a fast tree-based method for extreme multi-label classification.

For the state-of-the-art methods, we use the implementations available in the Mulan library [31]. We implemented LCIF in C++ and made the source code publicly available[1]. We use 64 threads to compute similarities and predictions in parallel on a test server from 2013, which has two 8-core processors (Intel E5-2690) and 64 GB RAM. Remark that we cannot use Mulan (or Meka [32]) for extreme datasets since methods like BR-SMO employ the binary relevance strategy for training L binary classifiers, which is not feasible.

[1] `https://bitbucket.org/len_feremans/lcif`

Evaluation Metrics Multi-label evaluation metrics can be organised in different ways. *Example-based* evaluation metrics are averaged over all instances. *Label--based* evaluation metrics look at the different ratios between true-positive, false-positive and false-negative predictions for each label. Label-based *micro* scores give each instance the same weight, while *macro* scores give each label the same weight, giving equal weight to frequent and infrequent labels. Within the Example-based category, we make the distinction between metrics based on the bipartition between relevant and non-relevant labels, metrics based on the *ranking* of confidence scores, and metrics based on the individual score for each label. We report the following example-based multi-label evaluation metrics:

Definition 14 (Example-based Metrics)

$$\textit{Example-based Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\frac{|\mathbf{y_i} \cap \hat{\mathbf{y}}_\mathbf{i}|}{|\mathbf{y_i} \cup \hat{\mathbf{y}}_\mathbf{i}|},$$

$$\textit{Hamming loss} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{L}|\mathbf{y_i} \triangle \hat{\mathbf{y}}_\mathbf{i}|,$$

where N is the number of test instances, $\hat{\mathbf{y}}_\mathbf{i}$ is the predicted set of labels and $\mathbf{y_i}\triangle\hat{\mathbf{y}}_\mathbf{i}$ is the symmetric difference (or XOR) of actual and predicted labels.

Definition 15 (Label-based Metrics) *We define for each label y_k the number of true positives, false negatives, false positives and corresponding metrics as*

$$tp = \sum_{i=1}^{N}\delta(y_k \in \mathbf{y_i} \wedge y_k \in \hat{\mathbf{y}}_\mathbf{i}),$$

$$fn = \sum_{i=1}^{N}\delta(y_k \in \mathbf{y_i} \wedge y_k \notin \hat{\mathbf{y}}_\mathbf{i}),$$

$$fp = \sum_{i=1}^{N}\delta(y_k \notin \mathbf{y_i} \wedge y_k \in \hat{\mathbf{y}}_\mathbf{i}),$$

$$precision = \frac{tp}{tp + fp}, \qquad recall = \frac{tp}{tp + fn},$$
$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

For label-based 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. For *macro F1*, we first compute precision and recall for each label separately and then compute the average.

Definition 16 (Micro and Macro Precision) *We define micro and macro precision (analogous for recall and F1) as:*

$$precision_{micro} = \frac{\sum_{j=1}^{L} tp_j}{\sum_{j=1}^{L} tp_j + \sum_{j=1}^{L} fp_j},$$
$$precision_{macro} = \frac{1}{L} \sum_{j=1}^{L} \frac{tp_j}{tp_j + fp_j}.$$

For the extreme datasets, we omit hamming loss which was close to 0 given the extreme number of labels and is a less suitable metric in such settings [37]. Instead, we report precision@k.

Definition 17 (Precision@k) *We compute precision@k based on the k predictions with the highest confidence score, defined as:*

$$precision@k = \frac{1}{N \cdot k} \sum_{i=1}^{N} \sum_{y_j \in \mathbf{y_i}} \delta(rank(y_j, \hat{\mathbf{y}}_\mathbf{i}) \leq k),$$

where $rank(y_j, \hat{\mathbf{y}}_\mathbf{i})$ returns the rank for label y_j in the list of predictions sorted on descending confidence score.

Hyperparameter Tuning For each method, we have to optimise several *hyperparameters*. 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, optimising k can have a significant effect on reported evaluation metric values. Therefore, we vary k for Ml-knn and Iblr between 1 and 59 in steps of 2. For instance-based kNN, we vary k in steps of 50, that is $k \in \{1, 5, 50, 100, \ldots, 350\}$. We remark that the large values of k are due to the similarity weighted scores and common within user-based collaborative filtering. We vary α en β for the power transform of instance- and feature-based kNN in $\{0.5, 1.0, 1.5, 2.0\}$. For Lcif, we vary λ between 0.0 and 1.0 in steps of 0.1. Remark that we re-use the neighbour (and similarity) matrix and only compute it once for the maximal value of k speeding up grid search considerably. For optimising the single threshold t, we perform two passes: first, we vary t between 0.0 and 1.0 in steps of 0.1 to obtain a temporary optimum t_{pass1}, and then we take steps of 0.01 and vary between $t_{pass1} - 0.05$ and $t_{pass1} + 0.05$ to obtain the final value. For the state-of-the-art methods, this is implemented by OneThreshold in Mulan. We use the same procedure for Lcif but minimise the difference between predicted and actual label cardinality (see Section 3.4) instead of maximising a selected evaluation metric. Finally, for the extreme datasets, we employ *feature selection* and select the top s features using *entropy*. We search for the optimal value of $s \in M \times \{0.01, 0.1, 0.25, 0.5, 0.75, 0.99, 1.0\}$. Note that we do not perform a full grid search, but instead first find the optimal value of s, assuming default parameters for other values (i.e., $k = 100, \lambda = 0.5, \alpha = \beta = 1.0$). Next, we find the optimal value of k assuming $\alpha = 1.0$ and the value of s previously found, then for α using the optimal k and s, then for β, λ and finally for t using the previously computed parameters.

To have a stable estimate for hyperparameters selected using *grid search*, we perform 10-fold *cross-validation* for Lcif on the training set for the large datasets. After the 10-fold cross-validation search finishes, we select the average parameter combination that optimises the selected evaluation metric on the training data. For Ml-knn and Iblr, we report the optimal hyperparameters optimised directly on the test set, thereby making the *Oracle* assumption for performance reasons. For Br-Smo, we keep default parameters for the (linear) kernel function. For the extreme datasets, we skip 10-fold cross-validation for performance reasons and instead perform grid search on a sample consisting of the first 10 000 instances (1 000 instances for *Eurlex* and *Wiki10*). We report results computed on the publicly available train-test splits.

5.2 Classification Performance Lcif

Here we discuss accuracy on different evaluation metrics for large and extreme datasets.

Large Datasets We compare our algorithms based on a variety of evaluation metrics for multi-label classification. This is common practice since different methods have different biases towards each metric [8]. Table 2 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 highlighted in bold perform best on the selected metric and dataset. Missing values for Iblr for the *Delicious* and *IMDB-F* datasets are due to time-out on our test server.

If we compare the ranking of all algorithms, we see that LCIF performs better than ML-KNN, IBLR and BR-SMO for both accuracy, micro F1 and macro F1. On hamming loss, BR-SMO performs best, but the difference with LCIF is small. We also see that instance-based kNN ranks second for both accuracy, micro F1 and macro F1 outperforming both ML-KNN and BR-SMO. The feature-based kNN, by itself, does not perform great, but is comparable with ML-KNN, while requiring no k-nearest neighbour search at test time. Note that the current version of feature-based kNN is significantly better than the preliminary version described in [1]. Compared to the original version, we now compute the full similarity matrix and not only the top-k highest similarities and scale similarities using the parameter β.

Extreme Datasets For extreme multi-label classification (XML) we compare with the published results of FASTXML [11, 33]. Table 3 shows the results on the extreme datasets. Results of micro or macro F1 are generally not available for other XML methods and are thus provided for information. LCIF performs better than FASTXML on *Eurlex*, *Wiki10* and *WikiLSHTC*, but not on *AmazonCat*. We remark that many other recent, rank-based optimised, XML methods exist [33]. However, in classification benchmarks, the best results are reported by ensemble methods that combine many models, possibly using different algorithms and feature representations. Comparing a single simple model with a complex ensemble-based method would not be informative. We conclude that our method produces excellent results on extreme datasets.

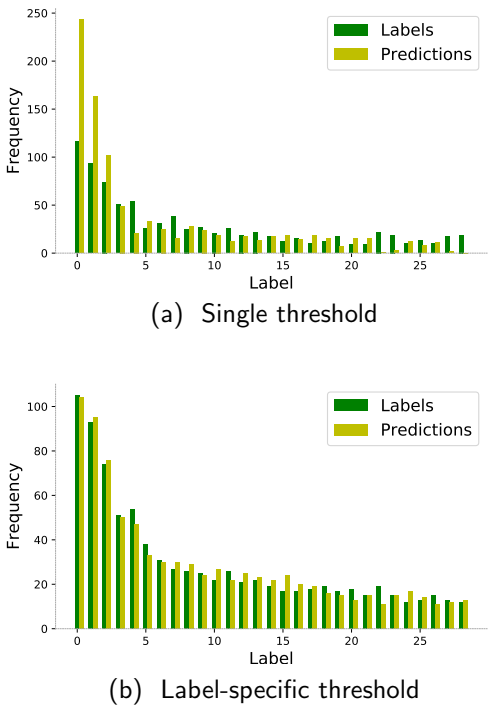

Fig. 1 Effect of thresholding on the distribution of actual versus predicted labels on the imbalanced dataset *Corel5k*

Table 2 Comparing accuracy of LCIF with ML-KNN, IBLR and BR-SMO on the large datasets

	INS. KNN	FEAT. KNN	LCIF	ML-KNN	IBLR	BR-SMO
Accuracy ↑						
Medical	0.563	0.601	0.636	0.421	0.442	**0.699**
Corel5k	0.163	**0.174**	0.170	0.147	0.105	0.098
Bibtex	**0.347**	0.218	0.341	0.208	0.174	0.321
Delicious	**0.230**	0.122	**0.230**	0.193	*n/a*	0.130
IMDB-F	**0.250**	0.235	**0.250**	0.244	*n/a*	0.005
Avg. rank	*2.2*	*3.4*	***1.8***	*4.2*	*5.6*	*3.8*
Micro F1 ↑						
Medical	0.622	0.645	0.690	0.505	0.506	**0.773**
Corel5k	0.266	0.263	**0.274**	0.245	0.163	0.166
Bibtex	0.426	0.276	**0.427**	0.315	0.250	0.416
Delicious	0.368	0.214	**0.369**	0.322	*n/a*	0.224
IMDB-F	0.341	0.337	0.346	**0.358**	*n/a*	0.014
Avg. rank	*2.6*	*4.0*	***1.4***	*3.6*	*5.8*	*3.6*
Macro F1 ↑						
Medical	0.339	0.447	**0.492**	0.245	0.247	0.457
Corel5k	0.315	0.308	0.315	**0.326**	0.161	0.317
Bibtex	0.321	0.129	**0.328**	0.170	0.147	0.315
Delicious	0.180	0.067	**0.181**	0.088	*n/a*	0.098
IMDB-F	0.090	0.055	**0.084**	0.056	*n/a*	0.011
Avg. rank	*2.5*	*4.6*	***1.7***	*3.6*	*5.6*	*3.0*
Hamming loss ↓						
Medical	0.025	0.026	0.021	0.024	0.029	**0.012**
Corel5k	0.014	**0.010**	**0.010**	0.022	0.027	0.012
Bibtex	0.017	0.017	**0.014**	0.020	0.021	0.016
Delicious	0.025	0.030	0.024	0.021	*n/a*	**0.018**
IMDB-F	0.094	0.083	0.082	0.101	*n/a*	**0.072**
Avg. rank	*3.9*	*3.6*	*1.9*	*4.0*	*6.0*	***1.6***

Imbalanced Datasets LCIF improves the accuracy on minority labels in imbalanced datasets. Firstly, we remark that feature-based cosine similarity is corrected for label imbalance since we normalise all label vectors to unit length during preprocessing. As such, weights for infrequent labels (and features) will be much higher. In Figure 1 we show the frequency of the top 30 most frequent labels (and predicted labels) on the *Corel5k* dataset where hyperparameters are optimised on the macro F1 metric during grid search. By using a single threshold, our method overestimates majority labels and underestimates minority labels. However, by using label-specific thresholds, we get a better match in the distribution of actual versus predicted labels. Here, we base the class-specific thresholds on the prior distribution of classes in the training dataset. We report an average gain of 1.1% on macro F1 on the large datasets.

Table 3 Comparing accuracy of LCIF with FASTXML on the extreme datasets

	INS. KNN	FEAT. KNN	LCIF	FASTXML
Micro F1 ↑				
Eurlex	0.506	0.222	**0.517**	*n/a*
Wiki10	**0.359**	0.274	0.358	*n/a*
RCV1	**0.637**	0.450	0.632	*n/a*
AmazonCat	0.611	0.418	**0.625**	*n/a*
WikiLSHTC	0.329	0.144	**0.342**	*n/a*
Macro F1 ↑				
Eurlex	0.509	0.381	**0.512**	*n/a*
Wiki10	**0.324**	0.261	**0.324**	*n/a*
RCV1	**0.152**	0.071	**0.152**	*n/a*
AmazonCat	**0.463**	0.285	**0.463**	*n/a*
WikiLSHTC	**0.130**	**0.130**	**0.130**	*n/a*
Precision@1 ↑				
Eurlex	0.763	0.407	**0.776**	0.713
Wiki10	**0.832**	0.722	**0.832**	0.830
RCV1	0.830	0.762	**0.840**	*n/a*
AmazonCat	0.775	0.657	0.811	**0.931**
WikiLSHTC	0.491	0.258	**0.518**	0.497
Precision@3 ↑				
Eurlex	0.611	0.302	**0.622**	0.599
Wiki10	**0.724**	0.531	**0.724**	0.675
RCV1	0.663	0.615	**0.672**	*n/a*
AmazonCat	0.665	0.589	0.698	**0.782**
WikiLSHTC	0.317	0.178	**0.336**	0.331
Precision@5 ↑				
Eurlex	0.504	0.244	**0.514**	0.504
Wiki10	**0.637**	0.469	**0.637**	0.578
RCV1	0.480	0.449	**0.487**	*n/a*
AmazonCat	0.547	0.504	0.577	**0.634**
WikiLSHTC	0.238	0.141	**0.251**	0.244

5.3 Runtime Performance INSTANCEKNNFAST

We now compare INSTANCEKNNFAST with the following state-of-the-art methods in top-k query retrieval: TAAT without pruning, Fagin's Threshold Algorithm (FAGIN TA) and the in-memory variant of DAAT with Weak-AND pruning (M-WAND) [19, 21].

Analysis Term-at-a-time First, we analyse the runtime behaviour of baseline kNN algorithm INSTANCEKNNSEARCH shown in Algorithm 2. This algorithm has three properties that make it efficient. We will use dataset characteristics from the extreme dataset *WikiLSHTC* for illustration (see Table 1). Firstly, we compute a *sparse dot product* between the query instance and each instance from the training dataset. In this dataset, there are $M \approx 1.6 \times 10^6$ features, however on average an instance has only 42 non-zero features, i.e., *fcard* is 42. Clearly, computing a million of zero multiplications for the naive full dot product is wasteful. Using the TAAT traversal strategy, we only compute $x_{q,j} \cdot x_{i,j}$ terms that have a non-zero value for feature value $x_{i,j}$. Secondly, the inverted index causes a form of *rudimentary pruning* by only considering candidate instances x_i having a non-zero feature in common with the query instance. We experimented on *WikiLSHTC* and computed the average number of candidates for 500 random test instances. We found that on average, only for 42% of instances the similarity is computed, thereby pruning about a million of instances from the training dataset (see Table 4). Thirdly, we also compute *confidence scores in a sparse manner*, thereby only computing non-zero terms for the confidence score for candidate labels, i.e., labels that occur for at least one neighbour. This is important since on average, each training instance has only 3 labels, i.e. *lcard* is 3.3, while $L \approx 0.3 \times 10^6$.

Pruning Performance The runtime performance is dependent on pruning, i.e., the number of candidate instances for which we compute the cosine similarity. We compare the average *number of candidate instances* for each state-of-the-art method. We set k to 100 and use the first 1 000 test instances to compute this average. We remark that INSTANCEKNNFAST is identical to M-WAND if $m = 0$, and identical to INSTANCEKNN when m is set high (such that $\mathcal{I}_{daat} = \emptyset$). For INSTANCEKNNFAST, we vary the hyperparameter $m \in \{1, 5, 10, 15, 20, 25, 100\}$ (not $m = 0$) and assume an Oracle that selects the best parameter.

The results are shown in Table 4 where we report both the absolute value and relative percentage of the average number of candidates. For the large datasets, all features are binary, however, after normalisation to unit length there is more variation in feature values, which is beneficial for pruning with our method. On the large datasets we see this effect, where INSTANCEKNNFAST evaluates fewer candidate instances compared to other techniques. For example, on the *IMDB-F* dataset, we compute cosine similarity for 46% of the training instances for INSTANCEKNNSEARCH, 39% for FAGIN TA, 28% for M-WAND and only 22% for INSTANCEKNNFAST. If we look at the extreme datasets, we find that M-WAND outperforms both FAGIN TA and INSTANCEKNNFAST on the majority of datasets. For small values of m, the number of instances in I_{taat} is large, leading to a higher number of full evaluations. By considering only instances with a maximum feature value for low-density features, this could be resolved, but we will see in the next subsection, where we compare runtimes, that this is less important. Overall we conclude that INSTANCEKNNFAST for pruning is theoretically always better than or equal to M-WAND without partitioning and performs better than FAGIN TA and INSTANCEKNNSEARCH, under the assumptions of sparse datasets and high-dimensional queries.

Table 4 Pruning of InstanceKnnFast and state-of-the-art top-k query retrieval methods

Dataset	Fagin Ta	M-Wand	InstanceKnnFast	InstanceKnnSearch
Avg number of candidate instances ↓				
Medical	**197** (59%)	214 (64%)	214 (64%)	224 (67%)
Corel5k	681 (14%)	641 (14%)	**455** (10%)	719 (16%)
Bibtex	**4 404** (90%)	4 553 (93%)	4 762 (97%)	4 860 (99%)
Delicious	3 412 (26%)	3 120 (24%)	**3 115** (24%)	4 141 (32%)
IMDB-F	28 389 (39%)	20 716 (28%)	**16 170** (22%)	33 509 (46%)
Eurlex	14 091 (91%)	**11 785** (76%)	12 355 (79%)	15 528 (99%)
Wiki10	13 727 (97%)	**12 592** (89%)	14 110 (99%)	14 118 (99%)
RCV1	305 002 (48%)	**176 038** (28%)	196 453 (31%)	569 297 (91%)
AmazonCat	**195 488** (11%)	227 326 (19%)	290 450 (24%)	741 380 (62%)
WikiLSHTC	462 939 (26%)	**367 645** (20%)	594 107 (33%)	899 071 (50%)

Table 5 Runtime of InstanceKnnFast and state-of-the-art top-k query retrieval methods

Dataset	M-Wand	InstanceKnnFast				Inst. Knn Search
		m=1	m=20	m=100	m=500	m=1000	
Avg time (ms) to retrieving top kNN ↓							
Eurlex	17.4	12.8	1.4	**1.3**	1.4	1.4	1.5
Wiki10	62.4	1.8	1.6	**1.5**	1.6	1.7	1.7
RCV1	103.7	97.4	43.8	**32.6**	32.6	33.9	37.5
AmazonCat	215.4	174.1	45.1	**37.7**	38.7	45.1	45.2
WikiLSHTC	171.1	94.3	35.7	**27.7**	27.9	27.6	28.6

Runtime Performance We now compare our method with state-of-the-art-methods and report elapsed wall time. We do not report results for Fagin Ta since in our experiments we found that the random access cost and associated zero computations for computing full cosine similarity at each iteration caused much worse performance than the Taat baseline without pruning. We report the number of milliseconds required for InstanceKnnSearch, M-Wand and InstanceKnnFast with m in $\{1, 20, 100, 500, 1000\}$. For each dataset, we take the first 1 000 test instances and report the average time it takes to retrieve the *exact* set of 100 nearest neighbours using each algorithm. From the timings (averaged over 10 runs) we excluded time needed to load the data and create the inverted indexes since this took less than 1 minute on *WikiLSHTC*.

The results are shown in Table 5. We omitted the results for the large datasets since the differences in milliseconds are too small. First, we remark that there is no clear one-to-one correspondence between pruning and runtime performance. Because of its simplicity and sparse optimisations, our current implementation of the InstanceKnnSearch method is far more efficient than M-Wand. M-Wand has overhead because of the computation and bookkeeping required for computing and comparing with the upper bound, such as the sort on document ID required to find the pivot. Note that both M-Wand and InstanceKnnFast use the same code. We see that InstanceKnnFast, when m is set appropriately large, outperforms both state-of-the-art methods by a considerable margin on all extreme datasets. This is especially so for *AmazonCat*, where it is 6 times faster than M-Wand and 25% faster than InstanceKnnSearch. We find that, on the one hand, when only a few instances in the inverted index match the current query, the overhead of computing and verifying the upper bound is probably not justified. On the other hand, when only low similarity instances remain, and there is a high likelihood for pruning, pruning becomes the faster alternative. We see this in *AmazonCat*, where for $m = 100$, 98.6% of instances are indexed for Taat traversal and 1.4% for Daat traversal. However, during Taat only 39% of instances are pruned (because of the inverted index), while during Daat more than 99% of instances (in the remaining sample of about 16.000 instances) are pruned. We conclude that InstanceKnnFast finds a *natural balance* between fast unpruned Taat traversal and fast Daat traversal with Wand pruning.

5.4 Runtime Performance Lcif

Large Datasets We now compare the total runtime required for both training and applying the model of Lcif and each of the state-of-the-art algorithms for large datasets. The results are shown in Table 6. For the large datasets, the difference in wall time is quite large as Lcif takes seconds or minutes where other methods take minutes or hours to complete. For *Corel5k* the relatively long 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* and *IMDB-F*. BR-SMO does finish for *Delicious* and *IMDB-F* but runs for a full day when LCIF takes less than 1 minute. Table 6 also shows the runtime of instance- and feature-based kNN. The runtime of LCIF is approximately equal to the total runtime of the two components. We conclude that LCIF is *orders of magnitude* faster than ML-KNN, BR-SMO and IBLR.

Extreme Datasets In Table 7 we show the total time required by LCIF on the extreme datasets. We report subtotals for running instance-based k-nearest neighbours search, feature-based similarity matrix computation and feature-based predictions. For LCIF the total time is the sum of these steps. The remaining time needed for data loading, indexing, making instance-based predictions, combining predictions and computing and applying a single threshold is relatively small. Additionally, we report time to perform grid search for tuning hyperparameters. All experiments were run on a single test server with very moderate hardware specifications as described previously.

On *WikiLSHTC*, LCIF took less than 3 hours to finish on more than 500 000 test instances, including grid search using a validation set of 10 000 instances. Averaged over the number of test instances this means 22 ms per instance on average, of which the bulk is required for running the instance-based k-nearest neighbours search. Feature-based kNN predictions require less than 10 minutes in total and less than 1 ms per instance on average. In FASTXML the authors report wall times of 1.5 hours on *WikiLSHTC* for training only, depending on the hyperparameters [11]. However, some hyperparameters, such as the parameter that controls the number of iterations have a serious influence on both runtime and precision, and it is unclear how to optimise this and the 7 other hyperparameters efficiently. We conclude that our algorithm is very efficient and does hyperparameter tuning, training and predictions in a few hours on commodity hardware for extreme datasets, taking less than 22 ms per instance to predict labels.

Table 6 Runtime results of LCIF and the state-of-the-art multi-label classifications methods on the large datasets

Dataset	FEAT. KNN	INS. KNN	LCIF	ML-KNN	IBLR	BR-SMO
Total train and test time for classifier ↓						
Medical	0.1s	0.0s	0.1s	0.5s	1.5s	6.5s
Corel5k	0.0s	0.1s	0.1s	15.5s	34.3m	6.5m
Bibtex	1.9s	0.3s	2.6s	1.7s	8.2m	10.1m
Delicious	0.6s	0.2s	1.3s	3.3m	*n/a*	14.0h
IMDB-F	13.5s	33.8s	49.2s	2.1h	*n/a*	29.4h

Table 7 Runtime results of LCIF on the extreme datasets

Dataset	Grid search	INS. KNN search	FEAT. KNN simil	FEAT. KNN predict	LCIF total
Total time for classifier ↓					
Eurlex	51.4s	4.6s	0.8s	12.7s	24.4s
Wiki10	9.9m	9.8s	26.4s	3.6m	5.1m
RCV1	8.2m	58.9m	6.0s	2.9m	63.0m
AmazonCat	11.6m	136.3m	28.8s	5.7m	145.2m
WikiLSHTC	11.8m	154.8m	31.1s	8.3m	167.9m

6 Related Work

We have examined the most important related work in Section 1 and experimentally compared our method with existing state-of-the-art methods in Section 5. We now place our work into the wider context of multi-label classification.

Instance-based Learning Several instance-based learning methods for multi-label classification have been developed. ML-KNN was one of the first methods [7]. In ML-KNN the authors first apply traditional kNN, using Euclidean distance. Next, they count the number of times each label occurs in the neighbourhood. Then they apply the *maximum a posteriori* principle for each label independently to determine if a label is relevant or not. They estimate prior probabilities by computing kNN for each training instance and then compute these probabilities for each label. In theory, ML-KNN could also adopt an inverted index and sparse computation of similarities, probabilities and predictions. However, the complexity is worse than that of LCIF, which is $\mathcal{O}(N \times M)$ for the kNN search and $\mathcal{O}(N \times N \times M)$ steps for computing probabilities. The authors experimentally show 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 into *dependent* multi-label k-nearest neighbours [38].

In combining instance-based learning and logistic regression for multi-label classification (IBLR) the k-nearest neighbours are computed using Euclidean distance [23]. Then the authors use label counts from the nearest neighbours as a feature vector and apply logistic regression to learn the optimal hyper-plane for each label. This approach comes down to *stacking* and does account with dependencies between labels since all label counts are used as input for the logistic re-

gression. 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 some large datasets. Both ML-KNN and IBLR are considered state-of-the-art methods [8].

Spyromitros et al. propose BRKNN, where kNN is combined with the binary relevance method [26]. Like BRKNN, we compute the k-nearest neighbours once, independently of the number of labels. The authors also implement two extensions: BRKNN-a and BRKNN-b. The BRKNN-b variant minimises the label cardinality between predicted and actual label sets, while BRKNN-a returns the highest-scoring label as relevant, even if this label is below the threshold since for most benchmark multi-label datasets an empty set of labels is rare. Both ideas are implemented by our method. Compared to instance-based kNN, BRKNN uses a different scoring function, which is the fraction of labels found in the k-nearest neighbours. Also, BRKNN does not make use of an inverted index. In future work, it could be interesting to experimentally validate different variations of instance-based prediction functions.

Wang et al. propose the Enhanced kNN algorithm (EKNN), that uses a weighted prediction function similar to instance-based kNN, but based on BM25 similarity and a more elaborate thresholding scheme. EKNN scored first in the challenge on large scale hierarchical text classification on example-based accuracy and F1 [39,40]. However, EKNN has a larger range of hyperparameters (both for BM25 and thresholding) to tune and is only applicable for text categorisation. The implementation of EKNN is based on an inverted index, similar to INSTANCEKNNSEARCH.

Imbalanced Datasets Different authors have studied how to improve the accuracy of imbalanced datasets. SMOTE is an algorithm for synthetic oversampling of multi-class instances with minority labels [41]. In preliminary experiments, we tried to adopt SMOTE for oversampling instances with minority labels together with downsampling of majority labels to generate balanced datasets. However, this did not significantly improve results on macro F1. Moreover, SMOTE has additional parameters for each minority label, making adoption challenging.

Tan proposes NWKNN, a neighbour-weighted kNN algorithm that achieves a significant performance improvement for text categorisation on imbalanced datasets [42]. Like NWKNN, our instance-based kNN method performs distance weighting and a power transform. Unlike NWKNN, we do not take the size of the membership of labels into account in the instance-based confidence score.

Liu et al. present a hybrid coupled k-nearest neighbour classification algorithm (HC-KNN) for mixed-type data [43]. They employ feature weighting proportional to the number of feature-label co-occurrences and inversely proportional to the global label frequency. This is related to feature-based cosine similarity since we measure the feature-label co-occurrences by computing the cosine similarity between each feature and label column-wise and by normalising label vectors to unit length, we are dividing by the global label frequency (assuming binary positive data). The instance-based cosine similarity is related to the inter-coupled similarity measure if we employ one-hot-encoding of categorical features during preprocessing.

There is no related work on instance-based multi-label classification optimised for imbalanced extreme datasets. For example, Liu et al. propose an optimisation procedure to learn the correspondence between each feature value and label, but this has a complexity of $O(M^3 \cdot L)$ [43]. Therefore, we propose to adopt a label-specific threshold to improve results on macro F1 [30]. However, we acknowledge that given a long tail of minority labels (e.g. occurring less than 5 times), high precision and recall remain challenging.

Fast Nearest Neighbours We did not consider combining TAAT traversal with pruning [21]. For example, we could prune entire dimensions using max_score pruning. While this technique is useful for pruning more instances, it remains uncertain if this would decrease the overall wall time for high-dimensional sparse datasets as is the case with INSTANCEKNNFAST. We also did not consider *approximate* kNN strategies used by other authors in extreme multi-label classification [10,44]. Since our algorithm can compute the *exact* set of k-nearest neighbours efficiently on extreme datasets, approximations are of less interest.

7 Conclusion

Inspired by recent work in recommender systems research, i.e., user-based and item-based collaborative filtering, we propose LCIF, a new algorithm for multi-label classification. Our predictions are based on the labels of the nearest neighbours in the training dataset. The instance-based method finds the top k instances that are most similar using the features of the current test instance. The feature-based method gives higher weight to labels that are the most similar to each feature, where similarity is defined column-wise over all instances. A linear combination of the similarity weighted instance-

and feature-based neighbourhood is computed to make the final prediction.

We created an efficient algorithm for finding the k-nearest neighbours using an inverted index and efficient sparse computation of cosine similarities and predictions. We extend this algorithm and create an even faster k-nearest neighbours search algorithm, by partitioning instances and combining term-at-a-time and document-at-a-time traversal with a tighter upper bound for Weak-AND pruning. We validated that this method can be 25% faster than the baseline method, and up to 6 times faster than existing top-k query retrieval algorithms, assuming high-dimensional sparse datasets. LCIF requires only seconds to complete on large datasets, where classic methods take minutes or hours. For extreme datasets, we require less than 20 milliseconds per instance to predict labels on commodity hardware.

Experiments on ten real-world multi-label datasets from different domains, i.e., text categorisation, scene classification and social tagging domain, show that LCIF outperforms state-of-the-art algorithms such as multi-label kNN, instance-based logistic regression and binary relevance with support vector machines on accuracy, micro F1 and macro F1. LCIF also produces excellent results on extreme datasets compared to FASTXML. Because of its efficiency at both train and test time, the possibility to generate explainable results, and excellent evaluation accuracy, LCIF is interesting for any extreme multi-label application, especially when making a trade-off between model/computational complexity and performance improvement. The source code of LCIF is publicly available and enables end-users to perform accurate extreme multi-label classification without the need for expensive clusters.

In future work, we see potential to improve further extreme multi-label learning algorithms inspired by advances in the related field of collaborative filtering. We also see potential to boost the prediction accuracy of LCIF, for example, by creating ensembles using boosting or stacking and adopting embedding algorithms, such as word or sentence embeddings learned using neural networks [45, 46].

References

1. L. Feremans, B. Cule, C. Vens, and B. Goethals, "Combining instance and feature neighbors for efficient multi-label classification," in *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2017, pp. 109–118.
2. G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, 2006.
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. K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain, "Sparse local embeddings for extreme multi-label classification," in *Advances in Neural Information Processing Systems*, 2015, pp. 730–738.
10. Y. Tagami, "Annexml: Approximate nearest neighbor search for extreme multi-label classification," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 2017, pp. 455–464.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 916–927.
18. D. C. Anastasiu and G. Karypis, "Fast parallel cosine k-nearest neighbor graph construction," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2016, pp. 50–53.
19. R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of computer and system sciences*, vol. 66, no. 4, pp. 614–656, 2003.

20. S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, 2011, pp. 993–1002.
21. M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, pp. 1213–1224, 2011.
22. A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proceedings of the twelfth international conference on Information and knowledge management*. ACM, 2003, pp. 426–434.
23. 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.
24. 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.
25. Y. Liu, "Crafting concurrent data structures," PhD dissertation, Lehigh University, 2015.
26. 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.
27. 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.
28. I. Triguero and C. Vens, "Labelling strategies for hierarchical multi-label classification techniques," *Pattern Recognition*, vol. 56, pp. 170–183, 2016.
29. 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.
30. K. Draszawka and J. Szymański, "Thresholding strategies for large scale multi-label text classifier," in *2013 6th International Conference on Human System Interactions (HSI)*. IEEE, 2013, pp. 350–355.
31. 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.
32. J. Read, P. Reutemann, B. Pfahringer, and G. Holmes, "Meka: a multi-label/multi-target extension to weka," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 667–671, 2016.
33. K. Bhatia, K. Dahiya, H. Jain, Y. Prabhu, and M. Varma, "The extreme classification repository: multi-label datasets & code," *URL http://manikvarma.org/downloads/XC/XMLRepository. html*, 2016.
34. 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.
35. J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods*, pp. 185–208, 1999.
36. 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.
37. 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.
38. 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.
39. 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.
40. I. Partalas, A. Kosmopoulos, N. Baskiotis, T. Artières, G. Paliouras, É. Gaussier, I. Androutsopoulos, M. Amini, and P. Gallinari, "LSHTC: A benchmark for large-scale text classification," *CoRR*, vol. abs/1503.08581, 2015.
41. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority oversampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.
42. S. Tan, "Neighbor-weighted k-nearest neighbor for unbalanced text corpus," *Expert Systems with Applications*, vol. 28, no. 4, pp. 667–671, 2005.
43. C. Liu, L. Cao, and S. Y. Philip, "A hybrid coupled k-nearest neighbor algorithm on imbalance data," in *2014 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2014, pp. 2011–2018.
44. R. B. Zadeh and A. Goel, "Dimension independent similarity computation." *Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1605–1626, 2013.
45. T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
46. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.