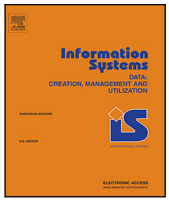
Information Systems 120 (2024) 102307

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



GSM: A generalized approach to Supervised Meta-blocking for scalable entity resolution

Luca Gagliardelli [a,*], George Papadakis [b], Giovanni Simonini [a], Sonia Bergamaschi [a], Themis Palpanas [c]

[a] University of Modena and Reggio Emilia, Modena, Italy
[b] University of Athens, Athens, Greece
[c] Université Paris Cité & IUF, Paris, France

ARTICLE INFO

Dataset link: https://github.com/Gaglia88/sparker

Keywords:
Entity resolution
Data integration
Progressive entity resolution
Meta-blocking

ABSTRACT

Entity Resolution (ER) constitutes a core data integration task that relies on Blocking in order to tame its quadratic time complexity. Schema-agnostic blocking achieves very high recall, requires no domain knowledge and applies to data of any structuredness and schema heterogeneity. This comes at the cost of many irrelevant candidate pairs (i.e., comparisons), which can be significantly reduced through Meta-blocking techniques, i.e., techniques that leverage the co-occurrence patterns of entities inside the blocks: first, a weighting scheme assigns a score to every pair of candidate entities in proportion to the likelihood that they are matching and then, a pruning algorithm discards the pairs with the lowest scores. Supervised Meta-blocking goes beyond this approach by combining multiple scores per comparison into a feature vector that is fed to a binary classifier. By using probabilistic classifiers, Generalized Supervised Meta-blocking associates every pair of candidates with a score that can be used: *(i)* by any pruning algorithm for retaining the set of candidate comparisons; and *(ii)* by state-of-the-art progressive ER methods to identify the most promising candidates as early as possible (when time is a critical component for the downstream applications that consume the data). For higher effectiveness, new weighting schemes are examined as features. Through an extensive experimental analysis, we identify the best pruning algorithms, their optimal sets of features as well as the minimum possible size of the training set. The resulting approaches achieve excellent performance across several established benchmark datasets.

1. Introduction

Entity Resolution (ER) is the task of identifying entities that describe the same real-world object among different datasets [1]. ER constitutes a core data integration task with many applications that range from Data Cleaning in databases to Link Discovery in Semantic Web data [2,3]. Despite the bulk of works on ER, it remains a challenging task [1]. One of the main reasons is its quadratic time complexity: in the worst case, every entity has to be compared with all others, thus scaling poorly to large volumes of data.

To tame its high complexity, *Blocking* is typically used [4,5]. Instead of considering all possible pairs of entities, it restricts the computational cost to entities that are similar. This is efficiently carried out by associating every entity with one or more signatures and clustering together entities that have identical or similar signatures. Extensive experimental analyses have demonstrated that the *schema-agnostic* signatures outperform the schema-based ones, without requiring domain or schema knowledge [6,7]. As a result, parts of any attribute value in each entity can be used as signatures.

Example 1 (*Schema-Agnostic Blocking*)**.** An example of schema-agnostic blocking is illustrated in Fig. 1. The profiles in Fig. 1a contain three duplicate pairs $\langle e_1, e_3\rangle$, $\langle e_2, e_4\rangle$ and $\langle e_6, e_7\rangle$. The profiles are clustered together by using Token Blocking, i.e., a block is created for every token that appears in the values of each profile. ER examines all pairs inside each block and, thus, can detect all duplicate pairs, as they co-occur in at least one block.

On the downside, the resulting blocks involve high levels of redundancy: every entity is associated with multiple blocks, thus yielding numerous *redundant* and *superfluous comparisons* [8,9]. The former are pairs of entities that are repeated across different blocks, while the latter involve non-matching entities. For example, the pair $\langle e_1, e_3\rangle$ is

* Corresponding author.
E-mail addresses: luca.gagliardelli@unimore.it (L. Gagliardelli), gpapadis@di.uoa.gr (G. Papadakis), giovanni.simonini@unimore.it (G. Simonini), sonia.bergamaschi@unimore.it (S. Bergamaschi), themis@mi.parisdescartes.fr (T. Palpanas).

https://doi.org/10.1016/j.is.2023.102307
Received 20 November 2022; Received in revised form 15 July 2023; Accepted 14 October 2023
Available online 17 October 2023

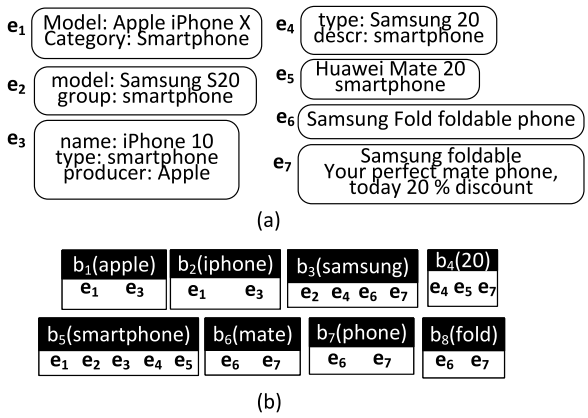


Fig. 1. (a) The input entities (smartphone models), and (b) the redundancy-positive blocks produced by token blocking.

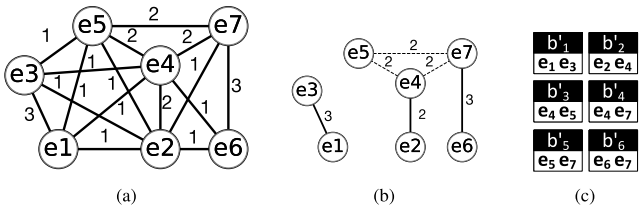


Fig. 2. Unsupervised Meta-blocking example: (a) The blocking graph of the blocks in Fig. 1b, using the number of common blocks as edge weights, (b) a possible pruned blocking graph, and (c) the new blocks.

redundant in b_2, as it is already examined in b_1, while the pair $\langle e_2, e_6\rangle \in b_3$ is superfluous, as the two entities are not duplicates. Both types of comparisons can be skipped, reducing the computational cost of ER without any impact on recall [10,11].

A core approach to this end is *Meta-blocking* [12], which discards all redundant comparisons, while reducing significantly the portion of superfluous ones. It relies on two components to achieve this goal:

1. A *weighting scheme* is a function that receives as input a pair of entities along with their associated blocks and returns a score proportional to their matching likelihood. The score is based on the co-occurrence patterns of the entities into the original set of blocks: the more blocks they share and the more distinctive (i.e., infrequent) the corresponding signatures are, the more likely they are to match and the higher is their score.
2. A *pruning algorithm* receives as input all weighted pairs and retains the ones that are more likely to be matching.

Example 2 (*Unsupervised Meta-Blocking*)**.** Unsupervised Meta-blocking builds a blocking graph (Fig. 2(a)) from the blocks in Fig. 1b as follows: each entity profile is represented as a node; two nodes are connected by an edge if the corresponding profiles co-occur in at least one block; each edge is weighted according to a weighting scheme — in our example, the number of blocks shared by the adjacent profiles. Finally, the blocking graph is pruned according to a pruning algorithm — in our example, for each node, we discard the edges with a weight lower than the average of its edges. The pruned blocking graph appears in Fig. 2(b), with the dashed lines representing the superfluous comparisons. A new block is then created for every retained edge. Fig. 1c presents the final blocks, which involve significantly fewer pairs without missing the matching ones. This is a schema-agnostic process, just like the original blocking method.

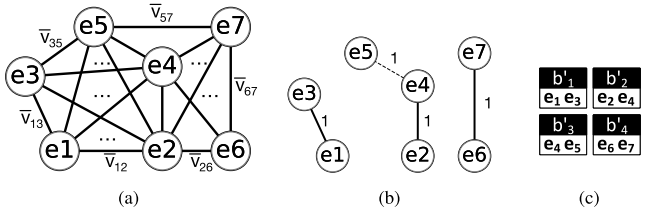


Fig. 3. Supervised Meta-blocking example: (a) a graph where each edge is associated with a feature vector; (b) the graph pruned by employing a binary classifier (c) the output contains a new block per remaining edge.

1.1. Supervised meta-blocking

Supervised Meta-blocking models the restructuring of a set of blocks as a binary classification task [13]. Its goal is to train a model that learns to classify every comparison as *positive* (i.e., likely to be matching) and *negative* (i.e., unlikely to be matching). To this end, every pair is associated with a feature vector that comprises a combination of the most distinctive weighting schemes that are used by unsupervised meta-blocking. Thus, Supervised Meta-blocking considers more comprehensive evidence, outperforming the unsupervised approaches to a significant extent.

In more detail, the thorough experimental analysis in [13] performed an analytical feature selection that considered all combinations of 7 features to determine the one achieving the best balance between effectiveness and efficiency. The resulting vector comprises four features, yielding high time efficiency and classification accuracy.

Example 3 (*Supervised Meta-Blocking*)**.** Fig. 3(a) shows the feature vectors generated for every distinct comparison (i.e., edge in the blocking graph) in the blocks of Fig. 1b. For instance, each pair of entities $\langle e_i, e_j\rangle$ can be represented by a feature vector $v_{i,j} = \{CB(e_i, e_j), JS(e_i, e_j)\}$, where $CB(e_i, e_j)$ is the number of their common blocks and $JS(e_i, e_j)$ is the Jaccard coefficient of blocks associated with e_i and e_j. Then, a *binary classifier* is trained with a sample of labelled vectors and is used to predict whether a pair $\langle e_i, e_j\rangle$ is a match ($l_{i,j} = 1$) or not ($l_{i,j} = 0$). The results appear in Fig. 3(b), where the dashed lines represent superfluous comparisons. The comparisons classified as positive are retained, yielding the new blocks in Fig. 3(c).

Note, though, that ER suffers from intense class imbalance, since the vast majority of comparisons belongs to the negative class. To address it, *undersampling* is used to create a training set that is equally split between the two classes. Through extensive experiments, the number of labelled instances per class in the training set was set to 5% of the minority (positive) class in the ground-truth [13]. This size combined high effectiveness with high efficiency, through the learning of generic classification models. Additionally, the learned classification models were empirically verified to be robust with respect to the configuration of the classification algorithm: minor changes in the internal parameters of the algorithm yield minor changes in its overall performance.

Even though Supervised Meta-blocking outperforms its unsupervised counterpart to a significant extent, there is plenty of room for improving its classification accuracy, for reducing its overhead, i.e., running time, and for minimizing the size of the training set it requires, as we explain below.

Supervised Meta-blocking requires the effort to provide labelled edges, but by representing each edge with multiple features, it is more accurate in discriminating matching and non-matching pairs than Unsupervised Meta-blocking, which employs a single weight per edge. Indeed, Supervised Meta-blocking consistently yields better precision and recall than the unsupervised approach [13]. Yet, the binary classifier that lies at its core acts as a learned, unique, *global* threshold that is

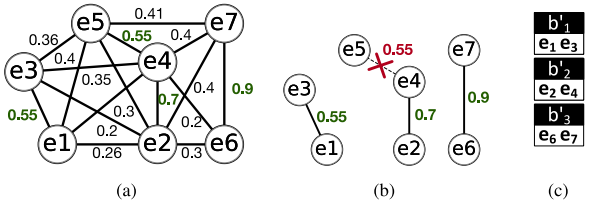


Fig. 4. Generalized Supervised Meta-blocking example: (a) a graph weighted with a probabilistic classifier; (b) the pruned graph; and (c) the new blocks.

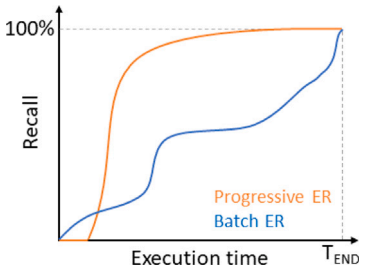


Fig. 5. Typical batch execution compared with progressive one.

then employed to prune the edges. Defining a *local* threshold for each node would allow a finer control on which edges to prune. This is the intuition behind Generalized Supervised Meta-blocking, as illustrated in the following example.

Example 4 (*Generalized Supervised Meta-Blocking*)**.** Our new approach builds a graph where every edge is associated with a feature vector (as Supervised Meta-blocking does in Fig. 3(a)) and trains a *probabilistic classifier*, which assigns a weight (the matching probability) to each edge (Fig. 4(a)). Then, a wide range of weight- and cardinality-based algorithms can be applied. For example, Supervised WNP prunes the graph as follows: for each node, all its adjacent edges with a weight lower that 0.5 are discarded; for the remaining edges, only those with a weight greater than the average one are kept. Fig. 4(b) shows the result of this step: two edges may be assigned the same weight by the probabilistic classifier, e.g., $\langle e_1, e3\rangle$ and $\langle e_4, e5\rangle$, but they may be kept (e.g., the matching pair $\langle e_1, e3\rangle$) or discarded (e.g., the non-matching pair $\langle e_4, e5\rangle$) depending on their context, i.e., the weights in their neighbourhood. Note that $\langle e4, e5\rangle$ is not discarded by Supervised Meta-blocking in Fig. 3(b), which thus underperforms Supervised WNP in terms of precision (for the same recall).

1.2. Progressive ER

For applications that require timely results, performing ER on the entire data might not be the best choice: if the data is changing fast, the output of the ER process is outdated as it is produced if ER takes too long to execute w.r.t. the update frequency of the underlying data. For instance, in a stock market trading scenario [14], there might be very limited time to match companies' records from a news feed: it is more useful to produce partial-but-timely-resolved entities than completely-resolved-but-outdated ones. In these settings, a more suitable approach is to return the result of ER as soon as two pairs of records are found to be match. Yet, without a proper strategy, just executing the candidate comparisons of a *batch ER* method in a random order is extremely inefficient [14]. For this reason, *Progressive ER* methods [14,15] aim to execute candidate comparisons in an order that maximizes the recall over the execution time, as shown in Fig. 5. In this work, we use the probabilities generated by Generalized Supervised Meta-blocking to perform progressive ER with the algorithms proposed in [16], demonstrating that generated probabilities outperform the heuristics of individual weighting schemes.

1.3. Our contributions

Our goal is to facilitate real-world ER applications by minimizing the set of candidate pairs, while restricting human involvement for the generation of the labelled instances. At the same time, we want to maximize the throughput of ER applications with limited temporal and/or computational resources. To this end, we go beyond Supervised Meta-blocking and in the following ways:

1. We generalize Supervised Meta-blocking from a binary classification task to a binary *probabilistic* classification process. The resulting probabilities are used as comparison weights, on top of which we apply new pruning algorithms that are incompatible with the binary classification approach.
2. Using the original features, we demonstrate that the new pruning algorithms significantly outperform the existing ones through an extensive experimental study that involves 9 real-world datasets. We also specify the top performers among the weight- and cardinality-based algorithms.
3. To further improve their performance, we use four new weighting schemes as features for Generalized Supervised Meta-blocking. We examine the performance of all 255 feature combinations over all nine datasets for the top-performing algorithms. We identify the top-10 feature sets per algorithm in terms of effectiveness and then, we select the optimal one by considering their time efficiency, too, significantly reducing the overall run-time.
4. For each algorithm, we examine how the size of the training set affects its performance. We experimentally demonstrate that it suffices to train our approaches over just 50 labelled instances, 25 from each class.
5. We perform a scalability analysis over 5 synthetic datasets with up to 300,000 entities, proving that our approaches scale well both with respect to effectiveness and time-efficiency under versatile settings.
6. We experimentally demonstrate that the probabilities generated by the probabilistic classifier when using the optimal selected feature set outperform the existing schema-agnostic progressive ER [16], which uses a single weighting scheme.
7. We compared the results obtained by our proposed solution with the state-of-the-art blocking solutions based on pre-trained language models [17–19] and a recently published one based on TF/IDF [20] to demonstrate the efficiency and efficacy of our work.
8. We have publicly released our data as well as an implementation (in Java) of all pruning algorithms and weighting schemes.[1]

This work is an extension of our previous work [21]. Here, we significantly extend the original conference version by showing how our method can be adapted to support *pay-as-you-go* execution, i.e., Progressive Entity Resolution, which was not investigated in the original paper. To this end, we combined our unsupervised progressive algorithm [16] with the probabilities generated by the probabilistic classifier. Moreover, we further improve the description of the proposed method by adding Algorithms 1–5 in Section 3 and providing a deeper and more formal definition of the new and existing weighting scheme. Finally, in the experimental evaluation, we explain the behaviour of the algorithm when the training set size increases, we add considerations regarding the obtained recall in each employed dataset, and we compare our work with the state-of-the-art pretrained language model based blocking solutions [17–19].

[1] https://sourceforge.net/p/erframework.

The rest of the paper is organized as follows: Section 2 provides background knowledge on the task of Supervised Meta-blocking, Section 3 introduces the new pruning algorithms, and Section 4 discusses the weighing schemes that are used as features. The experimental analysis is presented in Section 5, the main works in the field are discussed in Section 6 and the paper concludes in Section 7 along with directions for future work.

2. Preliminaries

An entity profile e_i is defined as a set of name-value pairs, i.e., $e_i = \{\langle n_j, v_j \rangle\}$, where both the attribute names and the attribute values are textual. This simple model is flexible and generic enough to seamlessly accommodate a broad range of established data formats — from the structured records in relational databases to the semi-structured entity descriptions in RDF data [6]. Two entities, e_i and e_j, that describe the same real-world object are called *duplicates* or *matches*, denoted by $e_i \equiv e_j$. A set of entities is called *entity collection* and is denoted by E_l. An entity collection E_l is *clean* if it is duplicate-free, i.e., $\nexists\, e_i, e_j \in E_l : e_i \equiv e_j$.

In this context, we distinguish Entity Resolution into two tasks [4,7]: (i) *Clean-Clean ER* or *Record Linkage* receives as input two clean entity collections, $E_1 \& E_2$, and detects the set of duplicates D between their entities, $D = \{(e_i, e_j) \subseteq E_1 \times E_2 : e_i \equiv e_j\}$; (ii) *Dirty ER* or *Deduplication* receives as input a dirty entity collection and detects the duplicates it contains, $D = \{(e_i, e_j) \subseteq E \times E : i \neq j \wedge e_i \equiv e_j\}$.

In both cases, the time complexity is quadratic with respect to the input, i.e., $O(|E_1| \times |E_2|)$ and $O(|E|^2)$, respectively, as every entity profile has to be compared with all possible matches. To reduce this high computational cost, Blocking restricts the search space to similar entities [5].

Meta-blocking operates on top of Blocking, refining an existing set of blocks, B, a.k.a. *block collection*, as long as it is *redundancy-positive*. This means that every entity e_i participates into multiple blocks, i.e., $|B_i| \geq 1$, where $B_i = \{b \in B : e_i \in b\}$ denotes the set of blocks containing e_i, and the more blocks two entities share, the more likely they are to be matching, because they share a larger portion of their content. Blocks of this type are generated by methods like Token Blocking, Suffix Arrays and Q-Grams Blocking and their variants [10,11].

The redundancy-positive block collections involve a large portion of *redundant comparisons*, as the same pairs of entities are repeated across different blocks [13]. These can be easily removed by aggregating for every entity $e_i \in E_1$ the *set* of all entities from E_2 that share at least one block with it [22]. The union of these individual sets yields the distinct set of comparisons, which is called *candidate pairs* and is denoted by C. Every non-redundant comparison between e_i and e_j, $c_{i,j} \in C$, belongs to one of the following types:

- *Positive pair* if e_i and e_j are matching: $e_i \equiv e_j$.
- *Negative pair* if e_i and e_j are not matching: $e_i \not\equiv e_j$.

The sets of all positive and negative pairs in a block collection B are denoted by P_B and N_B, respectively. The goal of Meta-blocking is to transform a given block collection B into a new one B' such that the negative pairs are drastically reduced without any significant impact on the positive ones, i.e., $|P_{B'}| \approx |P_B|$ and $|N_{B'}| \ll |N_B|$.

2.1. Problem definition

Supervised Meta-blocking models every pair $c_{i,j} \in C$ as a feature vector $f_{i,j} = [s_1(c_{i,j}), s_2(c_{i,j}), \ldots, s_n(c_{i,j})]$, where each s_i is a weighting scheme that returns a score proportional to the matching likelihood of $c_{i,j}$. The feature vectors for all pairs in C are fed to a *binary classifier*, which labels them as `positive` or `negative`, if their constituent entities are highly likely to match or not. We assess the performance of this process based on the following measures:

1. $TP(C)$, the true positive pairs, involve matching entities and are correctly classified as `positive`.
2. $FP(C)$, the false positive pairs, entail non-matching entities, but are classified as `positive`.
3. $TN(C)$, the true negative pairs, entail non-matching entities and are correctly classified as `negative`.
4. $FN(C)$, the false negative pairs, comprise matching entities, but are classified as `negative`.

Supervised Meta-blocking discards all candidate pairs labelled as `negative`, i.e., $TN(C) \cup FN(C)$, retaining those belonging to $TP(C) \cup FP(C)$. A new block is created for every `positive` pair, yielding the new block collection B'. Thus, the effectiveness of Supervised Meta-blocking is assessed with respect to the following measures:

- *Recall*, a.k.a. *Pairs Completeness*, expresses the portion of existing duplicates that are retained, i.e., $Re = |TP(C)|/|D| = (|D| - FN(C))/|D|$.
- *Precision*, a.k.a. *Pairs Quality*, expresses the portion of positive candidate pairs that are indeed matching, i.e., $Pr = |TP(C)|/(|TP(C)| + |FP(C)|)$.
- *F-Measure*[2] is the harmonic mean of recall and precision, i.e., $F1 = 2 \cdot Re \cdot Pr/(Re + Pr)$.

All measures are defined in $[0, 1]$, with higher values indicating higher effectiveness.

In this context, the task of Supervised Meta-blocking is formally defined as follows [13]:

Definition 1. Given the candidate pairs C of block collection B, the labels $L = \{\texttt{positive}, \texttt{negative}\}$ and a training set $T = \{\langle c_{i,j}, l_k \rangle : c_{i,j} \in C \wedge l_k \in L\}$, Supervised Meta-blocking aims to learn a classification model M that minimizes the cardinality of $FN(C) \cup FP(C)$ so that the new block collection B' achieves much higher precision than B, $Pr(B') \gg Pr(B)$, but maintains the original recall, $Re(B') \approx Re(B)$.

The time efficiency of Supervised Meta-blocking is assessed through its running time, RT. This includes the time required to: (i) generate the feature vectors for all candidate pairs in C, (ii) train the classification model M, and (iii) apply the trained classification model M to C.

2.1.1. Generalized supervised meta-blocking

This new task differs from Supervised Meta-blocking in two ways: (i) instead of a *binary* classifier that assigns class labels, it trains a *probabilistic* classifier that assigns a weight $w_{i,j} \in [0, 1]$ to every candidate pair $c_{i,j}$. This weight expresses how likely it is to belong to the positive class. (ii) The candidate pairs with a probability lower than 0.5 are discarded, but the rest, which are called *valid pairs*, are further processed by a pruning algorithm. The ones retained after pruning give raise to the new block collection B', i.e., B' contains a new block per retained valid pair.

Hence, the performance evaluation of Generalized Supervised Meta-blocking relies on the following measures:

1. $TP'(C)$, the probabilistic true positive pairs, involve matching entities that are assigned a probability ≥0.5 and are retained by the pruning algorithm.
2. $FP'(C)$, the probabilistic false positive pairs, entail non-matching entities, that are assigned a probability ≥0.5 and are retained by the pruning algorithm.
3. $TN'(C)$, the probabilistic true negative pairs, entail non-matching entities that are assigned a probability <0.5 and are discarded by the pruning algorithm.

[2] Hand et al. [23] have discussed how F1-score may be an unreliable measure for comparing different ER algorithms. We report F1-score for the sake of completeness — it has been used in many related works [13,22,24] – yet we draw conclusions on the basis of precision and recall only.

4. $FN'(C)$, the probabilistic false negative pairs, comprise matching entities, that are assigned a probability <0.5 and are discarded by the pruning algorithm.

The measures of recall, precision and F-Measure are redefined accordingly. In this context, the task of Generalized Supervised Meta-blocking is formally defined as follows:

Definition 2. Given the candidate pairs C of block collection B, the labels $L = \{\texttt{positive}, \texttt{negative}\}$, and a training set $T = \{\langle c_{i,j}, l_k\rangle : c_{i,j} \in C \wedge l_k \in L\}$, the goal of Generalized Supervised Meta-blocking is to train a probabilistic classification model M that assigns a weight $w_{i,j} \in [0,1]$ to every candidate pair $c_{i,j} \in C$; these weights are then processed by a pruning algorithm so as to minimize the cardinality of $FN'(C) \cup FP'(C)$, yielding a new block collection B' that achieves much higher precision than B, $Pr(B') \gg Pr(B)$, while maintaining the original recall, $Re(B') \approx Re(B)$.

The time efficiency of Generalized Supervised Meta-blocking is assessed through its run-time, RT, which adds to that of Supervised Meta-blocking the time required to process the assigned probabilities by a pruning algorithm.

2.2. Progressive ER

Given a block collection B, batch ER executes the comparisons in random order, while progressive ER aims to execute them in a specific order that maximizes throughput, i.e., maximizing recall over the execution time by processing the most likely match comparisons first [16]. More formally, Progressive ER is defined as follows [14]:

Definition 3. Given the candidate pairs C of block collection B, let T_{END} the overall time required to perform batch ER on C. At time $t \ll T_{END}$, the recall obtained by progressive ER should be much higher than those obtained by batch ER, while at the time T_{END}, both progressive and batch ER should produce the same result.

Progressive methods are composed of two phases:

1. The **initialization phase**, which computes the data structures needed to process the comparisons in the best order.
2. The **emission phase**, which retrieves the next best comparison from a list of candidate pairs sorted in decreasing order of their matching likelihood.

The initialization phase is computed only once, while the emission phase is repeated whenever a new comparison is requested.

The performance of progressive methods is estimated through the progressive recall, which is the fraction of recall reached after emitting ec comparisons. Following [16], we measure the performance by using the **normalized Area Under the Curve**, $AUC^*@ec^*$, which varies in the interval $[0,1]$, with 1 denoting the best progressiveness. ec^* is the normalized number of emitted comparisons $ec \ast= ec/|D|$, which allows for comparing different datasets on an equal basis. For example, $AUC^*@10$ is the normalized AUC after emitting $ec = 10 \cdot |D|$ comparisons.

Among the progressive methods, we decided to employ *Progressive Profile Scheduling* (**PPS**), which was experimentally shown in [16] to be the top performer in terms of progressiveness and time efficiency. In essence, it associates every input entity with a duplication likelihood, i.e., the likelihood that it matches with at least one of its candidate pairs. This likelihood is determined as the average weight of the edges connecting every entity with its candidate pair. The weights are determined by the same schemes as Meta-blocking (see Section 4). Next, PPS sorts and processes all entities in decreasing duplication likelihood. For each entity, it emits its top-k weighted candidates, in decreasing weight. In this way, PPS increases the likelihood that the matching candidate pairs are processed before the non-matching ones.

Algorithm 1: Supervised Weighted Edge Pruning

```
Input: Learned Model M, Candidate Pairs C
Output: New Candidate Pairs C'
1  p̄ ← 0
2  counter = 0
3  foreach c_{i,j} ∈ C do
4      p_{i,j} ← M.getProbability(c_{i,j})
5      if 0.5 ≤ p_{i,j} then
6          p̄ += p_{i,j}
7          counter += 1
8  p̄ ← p̄ / counter
9  C' ← {}
10 foreach c_{i,j} ∈ C do
11     p_{i,j} ← M.getProbability(c_{i,j})
12     if p̄ ≤ p_{i,j} then
13         C' ← C' ∪ {c_{i,j}}
14 return C'
```

To further increase this likelihood, PPS identifies the top-weighted candidate per entity during the initialization phase, i.e., while estimating the duplication likelihood per entity. The resulting pairs are placed in a priority queue that sorts them in decreasing weight. These are the first comparisons to be emitted by PPS, which then proceeds with the entity-centric processing described above.

3. Pruning algorithms

A supervised pruning algorithm operates as follows: given a specific set of features, it trains a probabilistic classifier on the available labelled instances. Then, it applies the trained classification model M to each candidate pair, estimating its classification probability. It exceeds 0.5, it is compared with a threshold in order to determine whether the corresponding pair of entities will be retained or not.

Depending on the type of threshold, the pruning algorithms are categorized into two types:

1. The *weight-based algorithms* determine the weight(s), above which a comparison is retained.
2. The *cardinality-based algorithms* determine the number k of top-weighted comparisons to be retained.

In both cases, the determined threshold is applied either *globally*, on all candidate pairs, or *locally*, on the candidate pairs associated with every individual entity. Below, we delve into the supervised algorithms of each category.

3.1. Weight-based pruning algorithms

This category includes the following four algorithms. None of them was considered in [13] - only WEP was approximated through the binary classification task in Definition 1.

Weigted Edge Pruning (WEP). Algorithm 1 iterates over the set of candidate pairs C twice: first, it applies the trained classifier to each pair in order to estimate the average probability $\bar{p}$ of the valid ones (Lines 1–8). Then, it applies again the trained classifier to each pair and retains only those pairs with a probability higher than $\bar{p}$ (Lines 9–13).

Weighted Node Pruning (WNP). Algorithm 2 iterates twice over C, too. Yet, instead of a global average probability, it estimates a local average probability per entity. To this end, it keeps in memory two arrays: $\bar{p}[]$ with the sum of valid probabilities per entity (Line 1) and $counter[]$ with the number of valid candidates per entity (Line 2). These

Algorithm 2: Supervised Weighted Node Pruning

```
Input: Learned Model M, Candidate Pairs C
Output: New Candidate Pairs C'
1  p̄[] ← {}
2  counter[] ← {}
3  foreach c_{i,j} ∈ C do
4      p_{i,j} ← M.getProbability(c_{i,j})
5      if 0.5 ≤ p_{i,j} then
6          p̄[i] += p_{i,j}
7          counter[i] += 1
8          p̄[j] += p_{i,j}
9          counter[j] += 1
10 foreach i ∈ |p̄| do
11     p̄[i] ← p̄[i] / counter[i]
12 C' ← {}
13 foreach c_{i,j} ∈ C do
14     p_{i,j} ← M.getProbability(c_{i,j})
15     if p̄[i] ≤ p_{i,j} ∨ p̄[j] ≤ p_{i,j} then
16         C' ← C' ∪ {c_{i,j}}
17 return C'
```

Algorithm 3: Supervised BLAST

```
Input: Learned Model M, Candidate Pairs C, Pruning Ratio
       r ∈ (0, 1]
Output: New Candidate Pairs C'
1  max[] ← {}
2  foreach c_{i,j} ∈ C do
3      p_{i,j} ← M.getProbability(c_{i,j})
4      if 0.5 ≤ p_{i,j} then
5          if max[i] < p_{i,j} then
6              max[i] = p_{i,j}
7          if max[j] < p_{i,j} then
8              max[j] = p_{i,j}
9  C' ← {}
10 foreach c_{i,j} ∈ C do
11     p_{i,j} ← M.getProbability(c_{i,j})
12     if 0.5 ≤ p_{i,j} ∧ r · (max[i] + max[j]) ≤ p_{i,j} then
13         C' ← C' ∪ {c_{i,j}}
14 return C'
```

arrays are populated during the first iteration over C (Lines 3–9). The average probability per entity is then computed in Lines 10–11. Finally, WNP iterates over C and retains every comparison $c_{i,j}$ only if its estimated probability $p_{i,j}$ exceeds either of the related average probabilities (Line 15).

Reciprocal Weighted Node Pruning (RWNP). The only difference from WNP is that a comparison is retained if its classification probability exceeds both related average probabilities, i.e., $\bar{p}[i] \leq p_{i,j} \wedge \bar{p}[j] \leq p_{i,j}$. This way, it applies a consistently deeper pruning than WNP.

BLAST. This algorithm is similar to WNP, but uses a fundamentally different pruning criterion. Instead of the average probability per entity, it relies on the maximum probability per entity e_i. Algorithm 3 stores these probabilities in the array $max[]$ (Line 1), which is populated during the first iteration over C (Lines 2–8). The second iteration over C retains a valid pair $c_{i,j}$ if it exceeds a certain portion r of the sum of the related maximum probabilities (Line 12).

3.2. Cardinality-based pruning algorithms

This category includes the three algorithms described below. Only the first two were considered in [13].

Cardinality Edge Pruning (CEP). This algorithm retains the top-K weighted comparisons among the candidate pairs, where K is set to half the sum of block sizes in the original block collection B, i.e., $K = \sum_{b_i \in B} |b|/2$, where $|b|$ stands for the number of entities in block b [12]. Algorithm 4 essentially maintains a priority queue Q (Line 1), which sorts the comparisons in decreasing probability. Q is populated through a single iteration over C (Lines 3–9). Every valid candidate pair that exceeds the minimum probability min_p (Line 5), is pushed to the queue (Line 6). Whenever the size of the queue exceeds K, the lowest-weighted comparison is removed from the queue and min_p is updated accordingly (Lines 7–9). At the end of the iteration, the contents of Q correspond to the new set of candidates C'.

Algorithm 4: Supervised Cardinality Edge Pruning

```
Input: Learned Model M, Candidate Pairs C, K
Output: New Candidate Pairs C'
1  Q ← {}
2  min_p ← 0
3  foreach c_{i,j} ∈ C do
4      p_{i,j} ← M.getProbability(c_{i,j})
5      if 0.5 ≤ p_{i,j} ∧ min_p < p_{i,j} then
6          Q.push(c_{i,j})
7          if K < |Q| then
8              c_{k,l} ← Q.pop()
9              min_p ← p_{k,l}
10 return Q
```

Cardinality Node Pruning (CNP). Algorithm 5 adapts CEP to a local operation, maintaining an array $Q[]$ with a separate priority queue per entity (Line 1). The maximum size of each queue depends on the characteristics of the original block collection, as it amounts to the average number of blocks per entity: $k = max(1, \sum_{b \in B} |b|/(|E_1| + |E_2|))$ [12]. During the first iteration over C, CNP populates the priority queue of every entity following the same procedure as CEP (Lines 3–15); if the probability of the current candidate pair exceeds the minimum probability of one of the relevant queues (Lines 6 and 11), the pair is pushed into the queue (Lines 7 and 12). Whenever the size of a queue exceeds k (Lines 8 and 13), the least-weighted comparison is removed (Lines 9 and 14) and the corresponding threshold is updated accordingly (Lines 10 and 15). CNP involves a second iteration over C (Lines 17–21), which retains a candidate pair $c_{i,j}$ if its contained in the priority queue of e_i or e_j (Line 20).

Reciprocal Cardinality Node Pruning (RCNP). This algorithm adapts CNP so that it performs a consistently deeper pruning, requiring that every retained comparison is contained in the priority queue of both constituent entities. That is, the condition of Line 20 in Algorithm 5 changes into a conjunction: $Q[i]$.contains($c_{i,j}$) $\wedge$ $Q[j]$.contains($c_{i,j}$).

4. Weighting schemes

The goal of *weighting schemes* is to infer the matching likelihood of candidate pairs from their co-occurrence patterns in the input blocks [12]. All schemes are schema-agnostic, being generic enough to apply to any redundancy-positive block collection. In [13], four weighting schemes formed the optimal feature vector in the sense that it achieves the best balance between effectiveness and time efficiency:

Algorithm 5: Supervised Cardinality Node Pruning

Input: Learned Model M, Candidate Pairs C, k
Output: New Candidate Pairs C'

```
Q[] ← {}
min_p[] ← {}
foreach c_{i,j} ∈ C do
    p_{i,j} ← M.getProbability(c_{i,j})
    if 0.5 ≤ p_{i,j} then
        if min_p[i] < p_{i,j} then
            Q[i].push(c_{i,j})
            if k < |Q[i]| then
                c_{l,m} ← Q[i].pop()
                min_p[i] ← p_{l,m}
        if min_p[j] < p_{i,j} then
            Q[j].push(c_{i,j})
            if k < |Q[j]| then
                c_{l,m} ← Q[j].pop()
                min_p[j] ← p_{l,m}
C' ← {}
foreach c_{i,j} ∈ C do
    p_{i,j} ← M.getProbability(c_{i,j})
    if 0.5 ≤ p_{i,j} then
        if Q[i].contains(c_{i,j}) ∨ Q[j].contains(c_{i,j}) then
            C' ← C' ∪ {c_{i,j}}
return C'
```

1. *Co-occurrence Frequency-Inverse Block Frequency* (CF-IBF). Inspired from Information Retrieval's TF-IDF, it assigns high scores to entities that participate in few blocks, but co-occur in many of them. More formally:

$$CF-IBF(c_{i,j}) = |B_i \cap B_j| \cdot \log \frac{|B|}{|B_i|} \cdot \log \frac{|B|}{|B_j|}.$$

2. *Reciprocal Aggregate Cardinality of Common Blocks* (RACCB). The smaller the blocks shared by a pair of candidates, the more distinctive information they have in common and, thus, the more likely they are to be matching. This idea is captured by the following sum:

$$RACCB(c_{i,j}) = \sum_{b \in B_i \cap B_j} \frac{1}{\|b\|},$$

where $\|b\|$ denotes the total number of candidate pairs in block b (including the redundant ones).

3. *Jaccard Scheme* (JS). It expresses the portion of blocks shared by a pair of candidates:

$$JS(c_{i,j}) = \frac{|B_i \cap B_j|}{|B_i| + |B_j| - |B_i \cap B_j|}$$

This captures the core characteristic of redundancy-positive block collections that the more blocks two entities share, the more likely they are to match.

4. *Local Candidate Pairs* (LCP). It measures the number of candidates for a particular entity. More formally:

$$LCP(e_i) = |\{e_j : i \neq j \wedge |B_i \cap B_j| > 0\}|.$$

The rationale is that the less candidate matches correspond to an entity, the more likely it is to match with one of them. Entities with many candidates convey no distinctive information, being unlikely for any match.

The last feature applies to an individual entity. Thus, the feature vector of c_{ij} includes both LCP(e_i) and LCP(e_j) [13].

In this work, we aim to enhance the effectiveness of the resulting feature vector. To this end, we additionally consider the following new weighting schemes [25]:

1. *Enhanced Jaccard Scheme* (EJS). Similar to TF-IDF, it enhances JS with the inverse frequency of an entity's candidates in the set of all candidate pairs:

$$EJS(c_{i,j}) = JS(c_{i,j}) \cdot \log \frac{\|B\|}{\|e_i\|} \cdot \log \frac{\|B\|}{\|e_j\|},$$

where $\|B\| = \sum_{b \in B} \|b\|$ and $\|e_i\| = \sum_{b \in B_i} \|b\|$.

2. *Weighted Jaccard Scheme* (WJS). Its goal is to alter JS so that it considers the size of the blocks containing every entity, promoting the smallest (and most distinctive) ones in terms of the total number of candidates. Thus, it multiplies every block in the Jaccard coefficient with its inverse size:

$$WJS(c_{i,j}) = \frac{\sum_{b \in B_i \cap B_j} \frac{1}{\|b\|}}{\sum_{b \in B_i} \frac{1}{\|b\|} + \sum_{b \in B_j} \frac{1}{\|b\|} - \sum_{b \in B_i \cap B_j} \frac{1}{\|b\|}}.$$

WJS can be seen as normalizing RACCB.

3. *Reciprocal Sizes Scheme* (RS). This scheme is based on the premise that the more entities the common blocks between $c_{i,j}$ contain, the less likely they are to match because the information forming these blocks is not distinctive enough to group highly similar entities:

$$RS(c_{i,j}) = \sum_{b \in B_i \cap B_j} \frac{1}{|b|}.$$

4. *Normalized Reciprocal Sizes Scheme* (NRS). It normalizes RS, multiplying every block in the Jaccard coefficient with its inverse size:

$$NRS(c_{i,j}) = \frac{\sum_{b \in B_i \cap B_j} \frac{1}{|b|}}{\sum_{b \in B_i} \frac{1}{|b|} + \sum_{b \in B_j} \frac{1}{|b|} - \sum_{b \in B_i \cap B_j} \frac{1}{|b|}}.$$

5. Experimental evaluation

5.1. Experimental setup

Hardware and Software—All the experiments were performed on a machine equipped with four Intel Xeon E5-2697 2.40 GHz (72 cores), 216 GB of RAM, running Ubuntu 18.04. We employed the *SparkER* library [26] to perform blocking and features generation. Moreover, we integrated Generalized Supervised Meta-blocking in the *SparkER* library; the code is available on the GitHub page of the project.[3] Unless stated otherwise, we perform machine learning analysis using Python 3.7 and the Support Vector Classification (SVC) model of scikit-learn [27], in particular. We used the default configuration parameters, enabling the generation of probabilities and fixing the random state so as to reproduce the probabilities over several runs. We performed all experiments with logistic regression, too, obtaining almost identical results, but we omit them for brevity.

Datasets— Table 1a lists the 9 real-world datasets employed in our experiments. They have different characteristics and cover a variety of domains. Each dataset involves two different, but overlapping data sources, where the ground truth of the real matches is known. `AbtBuy` matches products extracted from Abt.com and Buy.com [28]. `DblpAcm` matches scientific articles extracted from dblp.org and dl.acm.org [28]. `ScholarDblp` matches scientific articles extracted from scholar.google.com and dblp.org [28]. `ImdbTmdb`, `ImdbTvdb` and `TmdbTvdb` match movies and TV series extracted from IMDB, TheMovieDB and TheTVDB [29], as suggested by their names. `Movies` matches information about films that are extracted from imdb.com and dbpedia.org [6]. `WMAmazon` matches products from Walmart.com and Amazon.com [30].

[3] https://github.com/Gaglia88/sparker.

Table 1
Technical characteristics of the (a) real and (b) synthetic datasets used in the experimental study. $|E_x|$ stands for the number of entities in a constituent dataset, $|D|$ for the number of duplicate pairs, and $|C|$ for the number of distinct candidate pairs generated in the corresponding block collection. The datasets are ordered in increasing $|C|$. The rightmost part reports the performance of the original blocks that are given as input to the supervised meta-blocking methods.

	Dataset	$\|E_1\|$	$\|E_2\|$	$\|D\|$	$\|C\|$	Recall	Precision	$F1$
(a)	AbtBuy	1.1k	1.1k	1.1k	36.7k	0.948	$2.78 \cdot 10^{-2}$	$5.40 \cdot 10^{-2}$
	DblpAcm	2.6k	2.3k	2.2k	46.2k	0.999	$4.81 \cdot 10^{-2}$	$9.18 \cdot 10^{-2}$
	ScholarDblp	2.5k	61.3k	2.3k	832.7k	0.998	$2.80 \cdot 10^{-3}$	$5.58 \cdot 10^{-3}$
	AmazonGP	1.4k	3.3k	1.3k	84.4k	0.840	$1.29 \cdot 10^{-2}$	$2.54 \cdot 10^{-2}$
	ImdbTmdb	5.1k	6.0k	1.9k	109.4k	0.988	$1.78 \cdot 10^{-2}$	$3.50 \cdot 10^{-2}$
	ImdbTvdb	5.1k	7.8k	1.1k	119.1k	0.985	$8.90 \cdot 10^{-3}$	$1.76 \cdot 10^{-2}$
	TmdbTvdb	6.0k	7.8k	1.1k	198.6k	0.989	$5.50 \cdot 10^{-3}$	$1.09 \cdot 10^{-2}$
	Movies	27.6k	23.1k	22.8k	26.0M	0.976	$8.59 \cdot 10^{-4}$	$1.72 \cdot 10^{-3}$
	WalmartAmazon	2.5k	22.1k	1.1k	27.4M	1.000	$4.22 \cdot 10^{-5}$	$8.44 \cdot 10^{-5}$
(b)	D_{10k}	10k		8.7k	$2.69 \cdot 10^{7}$	0.999	$3.23 \cdot 10^{-4}$	$6.47 \cdot 10^{-4}$
	D_{50k}	50k		43.1k	$6.73 \cdot 10^{8}$	0.999	$6.40 \cdot 10^{-5}$	$1.28 \cdot 10^{-4}$
	D_{100k}	100k		85.5k	$2.69 \cdot 10^{9}$	0.999	$3.17 \cdot 10^{-5}$	$6.34 \cdot 10^{-5}$
	D_{200k}	200k		172.4k	$1.08 \cdot 10^{10}$	1.000	$1.60 \cdot 10^{-5}$	$3.19 \cdot 10^{-5}$
	D_{300k}	300k		257.0k	$2.43 \cdot 10^{10}$	0.999	$1.06 \cdot 10^{-5}$	$2.12 \cdot 10^{-5}$

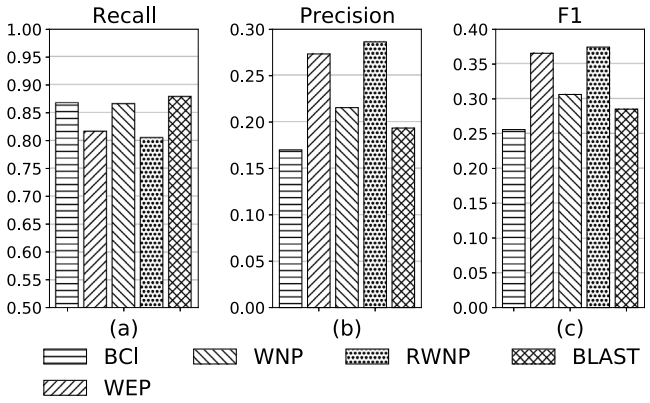


Fig. 6. The average performance of all weight-based pruning algorithms over the block collections of Table 1a.

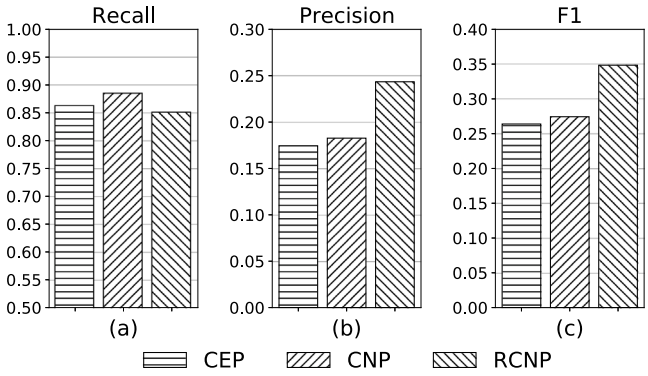


Fig. 7. The average performance of all cardinality-based pruning algorithms over the block collections of Table 1a.

Blocking—To each dataset, we apply Token Blocking, the only parameter-free redundancy-positive blocking method [11]. The original blocks are then processed by Block Purging [6], which discards all the blocks that contain more than half of all entity profiles in a parameter-free way. These blocks correspond to highly frequent signatures (e.g., stop-words) that provide no distinguishing information. Finally, we apply Block Filtering [22], removing each entity e_i from the largest 20% blocks in which it appears.

The performance of the resulting block collections is reported in the rightmost part of Table 1a. We observe that in most cases, the block collections achieve an almost perfect recall that significantly exceeds 90%. The only exception is `AmazonGP`, where some duplicate entities share no *infrequent* attribute value token — the recall, though, remains quite satisfactory, even in this case. Yet, the precision is consistently quite low, as its highest value is lower than 0.003. As a result, F1 is also quite low, far below 0.1 across all datasets. *These settings undoubtedly call for Supervised Meta-blocking.*

To apply Generalized Supervised Meta-blocking to these block collections, we performed 10 runs and averaged the values of precision, recall, and F1. In each run, a different seed is used to sample the pairs that compose the training set. Using undersampling, we formed a balanced training set per dataset that comprises 500 labelled instances. We choose to use a small fixed training set instead of splitting the labelled instances 80/20, because in a real use case scenario manually labelling 80% of the pairs is not feasible in large datasets. Due to space limitations, we mostly report the average performance of every approach over the 9 block collections.

5.2. Pruning algorithm selection

We now investigate which are the best-performing weight- and cardinality-based pruning algorithms for Generalized Supervised Meta-blocking among those discussed in Section 3. As baseline methods, we employ the pruning algorithms proposed in [13]: the binary classifier **BCl** for weight-based algorithms as well as CEP and CNP for the cardinality-based ones. We fixed the training set size to 500 pairs and used the feature vector proposed in [13] as optimal; every candidate pair $c_{i,j}$ is represented by the vector: $\{CF-IBF(c_{i,j}), RACCB(c_{i,j}), JS(c_{i,j}), LCP(e_i), LCP(e_j)\}$. Based on preliminary experiments, we set the pruning ratio of BLAST to $r = 0.35$. The average effectiveness measures of the weight- and cardinality based algorithms across the 9 block collections of Table 1a are reported in Figs. 6 and 7, respectively.

Among the weight-based algorithms, we observe that the new pruning algorithms trade slightly lower recall for significantly higher precision and F1. Comparing BCl with WEP, recall drops by −5.9%, while precision raises by 60.8% and F1 by 42.9%. This pattern is more intense in the case of RWNP, which reduces recall by −7.2%, increasing precision by 68.5% and F1 by 46.3%. These two algorithms actually monopolize the highest F1 scores in every case: for `ImdbTmdb`, `ImdbTvdb` and `TmdbTvdb`, WEP ranks first with RWNP second and vice versa for the rest of the datasets. Their aggressive pruning, though, results in very low recall ($\ll$0.8) in four datasets. E.g., in the case of `AbtBuy`, BCl's recall is 0.852, but WEP and RWNP reduce it to 0.755 and 0.699, respectively.

The remaining algorithms are more robust with respect to recall. Compared to BCl, WNP reduces recall by just −0.2%, while increasing precision by 26.8% and F1 by 19.7%. Yet, BLAST outperforms WEP

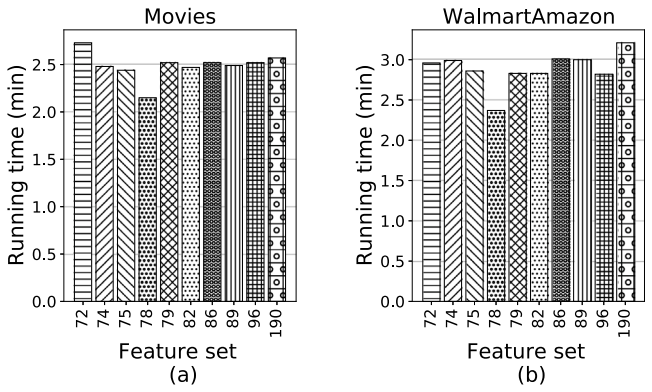


Fig. 8. Running time of top-10 features sets when applied to BLAST.

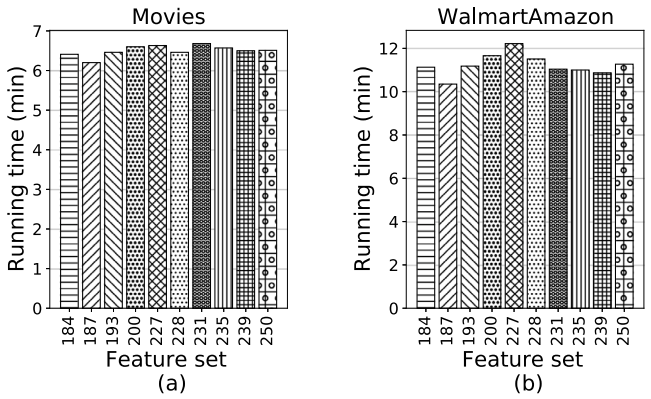


Fig. 9. Running time of top-10 features sets when applied to RCNP.

Table 2
The 10 feature sets that achieve the highest F1 with BLAST.

ID	Feature set	Recall	Precision	$F1$
72	{CF-IBF, RACCB, JS, RS}	.8816	.1932	.2892
74	{CF-IBF, RACCB, JS, NRS}	.8816	.1932	.2892
75	{CF-IBF, RACCB, JS, WJS}	.8816	.1932	.2892
78	{CF-IBF, RACCB, RS, NRS}	.8816	.1932	.2892
79	{CF-IBF, RACCB, RS, WJS}	.8816	.1932	.2892
82	{CF-IBF, RACCB, NRS, WJS}	.8816	.1932	.2892
86	{CF-IBF, JS, RS, WJS}	.8816	.1932	.2892
89	{CF-IBF, JS, NRS, WJS}	.8816	.1932	.2892
96	{CF-IBF, RS, NRS, WJS}	.8816	.1932	.2892
190	{CF-IBF, RACCB, JS, RS, NRS, WJS}	.8816	.1932	.2892

Table 3
The 10 feature sets that achieve the highest F1 when applied to RCNP.

ID	Feature set	Recall	Precision	$F1$
184	{CF-IBF, RACCB, JS, LCP, RS}	.8489	.2463	.3527
187	{CF-IBF, RACCB, JS, LCP, WJS}	.8490	.2464	.3526
193	{CF-IBF, RACCB, LCP, RS, NRS}	.8490	.2463	.3526
200	{CF-IBF, JS, LCP, RS, NRS}	.8488	.2474	.3526
227	{CF-IBF, RACCB, JS, LCP, RS, NRS}	.8493	.2473	.3537
228	{CF-IBF, RACCB, JS, LCP, RS, WJS}	.8494	.2473	.3537
231	{CF-IBF, RACCB, JS, LCP, NRS, WJS}	.8496	.2473	.3537
235	{CF-IBF, RACCB, LCP, RS, NRS, WJS}	.8496	.2473	.3536
239	{CF-IBF, JS, LCP, RS, NRS, WJS}	.8494	.2473	.3534
250	{CF-IBF, RACCB, JS, LCP, RS, NRS, WJS}	.8502	.2479	.3542

with respect to all effectiveness measures: recall, precision and F1 raise by 1.3%, 13.8% and 11.5%, respectively. This means that BLAST is able to discard much more non-matching pairs, while retaining a few more matching ones, too.

Among the cardinality-based algorithms, we observe that RCNP is a clear winner, outperforming both CEP and CNP. Compared to the former, it reduces recall by −1.1%, while increasing precision by 44% and F1 by 34.4%; compared to the latter, recall drops by −3.5%, but precision and F1 raise by 37.5% and 29.3%, respectively.

Overall, RCNP constitutes the best choice for cardinality-based pruning algorithms, which are crafted for applications that promote precision at the cost of slightly lower recall [12,22]. BLAST is the best among the weight-based pruning algorithms, which are crafted for applications that promote recall at the cost of slightly lower precision [12,22].

Note that their F1 is significantly higher than the original ones in Table 1a, but still far from perfect. The reason is that (Supervised) Meta-blocking merely produces a new block collection, not the end result of ER. This block collection is then processed by a Matching algorithm, whose goal is to raise F1 close to 1.

5.3. Feature selection

We now fine-tune the selected algorithms, BLAST and RCNP, by identifying the feature sets that optimize their performance in terms of effectiveness and time-efficiency. We adopted a brute force approach, trying all the possible combinations of the eight features presented in Sections 2 and 4. Fixing again the training set size to a random sample of 500 balanced instances, the top-10 feature vectors with respect to F1 for BLAST and RCNP are reported in Tables 2 and 3, respectively.

We observe that both algorithms are robust with respect to the top-10 feature sets, as they all achieve practically identical performance, on average. For BLAST, we obtain recall = 0.882, precision = 0.193 and F1 = 0.289 when combining $CF-IBF$ and $RACCB$ with any two features from $f = \{JS, RS, NRS, WJS\}$; even $RACCB$ can be replaced with a third feature from f without any noticeable impact. For RCNP, we obtain recall = 0.850, precision = 0.248 and F1 = 0.353 when combining $CF-IBF$, $RACCB$ and LCP with any pair of features from $\{JS, RS, NRS, WJS\}$. In this context, we select the best feature set for each algorithm based on *time efficiency*.

In more detail, we compare the top-10 feature sets per algorithm in terms of their running times. This includes the time required for calculating the features per candidate pair and for retrieving the corresponding classification probability (we exclude the time required for producing the new block collections, because this is a fixed overhead common to all feature sets of the same algorithm). Due to space limitations, we consider only the two datasets with the most candidate pairs, as reported in Table 1a: `Movies` and `WMAmazon`. We repeated every experiment 10 times and took the mean time.

In Fig. 8, we observe that the feature set **78** is consistently the fastest one for BLAST, exhibiting a clear lead. Compared to the second fastest feature sets over `movies` (75) and `WMAmazon` (96), it reduces the average run-time by 11.9% and 16.0%, respectively. For RCNP, the differences are much smaller in Fig. 9, yet the same feature set (**187**) achieves the lowest run-time over both datasets. Compared to the second fastest feature sets over `movies` (184) and `WMAmazon` (239), it reduces the average run-time by 3.3% and 4.8%, respectively.

Overall, BLAST models each candidate pair as the 4-dimensional feature vector (ID 78 in Table 2): $\{CF-IBF, RACCB, RS, NRS\}$. Compared to the feature set of [13], recall raises by ∼0.5% and F1 by ∼1.5%, while the run-time is reduced to a significant extent (>50% as explained below), due to the absence of the time-consuming LCP feature. RCNP represents every candidate pair with the 5-dimensional feature vector (ID 187 in Table 3): $\{CF-IBF, RACCB, JS, LCP, WJS\}$. This reduces recall by <0.3%, but raises precision and F-Measure by 1.2%, which is in-line with the desiderata of cardinality-based algorithms.

Comparison with Supervised Meta-blocking—We now compare BLAST and RCNP in combination with the features selected above with BCl and CNP, which use the feature set proposed in [13], {$CF-$

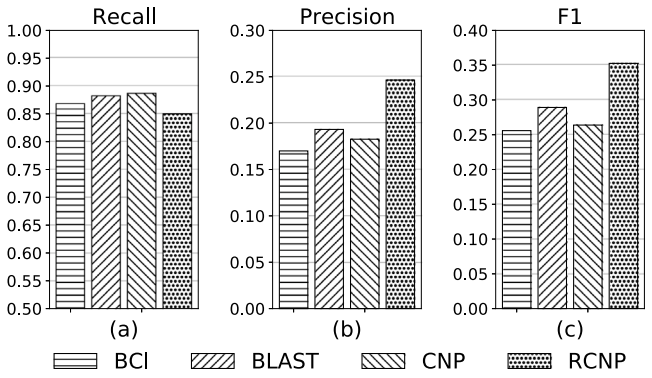


Fig. 10. Comparison of the best algorithms for Supervised (BCl, CNP) and Generalized Supervised Meta-blocking (BLAST, RCNP).

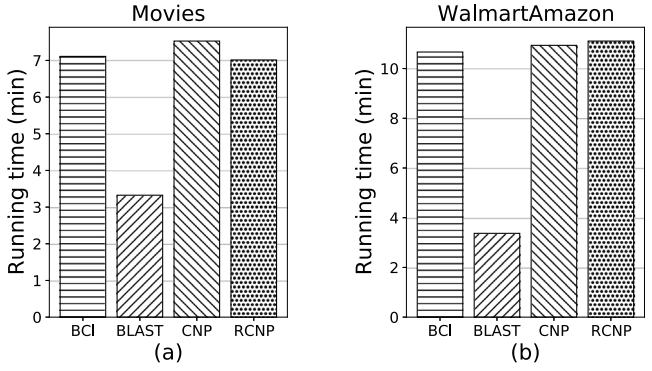


Fig. 11. Run-time comparison of the best algorithms for Supervised (BCl, CNP) and Generalized Supervised Meta-blocking (BLAST, RCNP).

$IBF, RACCB, JS, LCP\}$. All algorithms were trained over the same randomly selected set of 500 labelled instances, 250 from each class, and were applied to all datasets in Table 1a. Their average performance is presented in Fig. 10.

We observe that BLAST outperforms BCl with respect to all effectiveness measures: its recall, precision and F1 are higher by 1.6%, 13.6% and 13%, respectively, on average. Thus, *BLAST is much more accurate in the classification of the candidate pairs and more suitable than BCl for recall-intensive applications.* Among the cardinality-based algorithms, RCNP trades slightly lower recall than CNP for significantly higher precision and F1: on average, across all datasets, its recall is lower by −4.1%, while its precision and F1 are higher by 34.9% and by 33.6%, respectively. As a result, *RCNP is more suitable than CNP for precision-intensive applications.*

Regarding time efficiency, Fig. 11 reports the running times of these algorithms on the largest datasets, i.e., `Movies` and `WalmartAmazon`. We observe that BCl, CNP and RCNP exhibit similar RT in both cases, since they all employ more complex feature sets that include the time-consuming feature LCP. BLAST is substantially faster than these algorithms, reducing RT by more than 50%. In particular, comparing it with its weight-based competitor, we observe that BLAST is faster than BCl by 2.1 times over `Movies` and by 3.2 times over `WalmartAmazon`.

We can conclude, therefore, that Generalized Supervised Meta-blocking conveys significant improvements with respect to Supervised Meta-blocking.

5.4. The effect of training set size

We now explore how the performance of BLAST and RCNP changes when varying the training set size. We used the features sets selected above (ID 78 and 187 in Tables 2 and 3, respectively) and varied the number of labelled instances starting from 20, then from 50 to 500 with a step of 50. Figs. 12 and 13 report the results in terms of recall, precision and F1, on average across all datasets, for BLAST and RCNP, respectively.

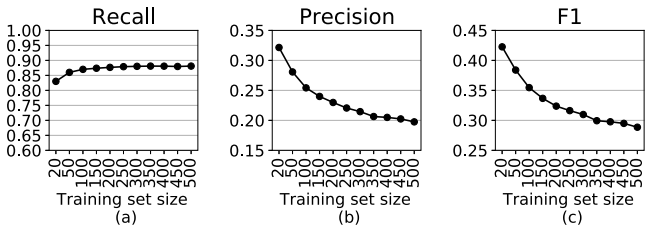


Fig. 12. The effect of the training set size on BLAST.

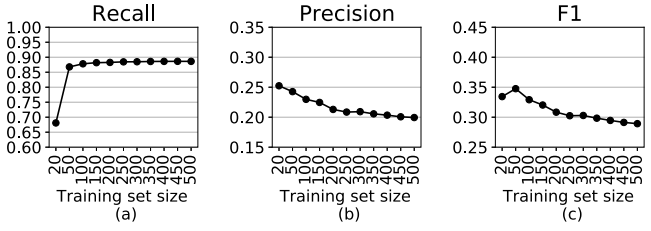


Fig. 13. The effect of the training set size on RCNP.

Notice that both algorithms exhibit the same behaviour: as the training set size increases, recall gets higher at the expense of lower precision and F1. However, the increase in recall is much lower than the decrease in the other two measures. More specifically, comparing the largest training set size with the smallest one, the average recall of BLAST raises by 2.4%, while its average precision drops by 29.7% and its average F1 by 24.8%. Similar patterns apply to RCNP: recall raises by 2.1%, but precision and F1 drop by 17.8% and 16.8%, respectively, when increasing the labelled instances from 50 to 500.

This might seem counter-intuitive, as larger training sets are expected to improve performance by increasing both recall and precision. In our cases, only recall raises with higher training sets, unlike precision. This behaviour should be attributed to the distribution of the matching probabilities that are produced by the trained probabilistic classifier. In more detail, with larger training sets, the matching probabilities of duplicate pairs are pushed up, thus raising recall. The same applies to the probabilities of non-matching pairs, though, thus decreasing precision.

This is illustrated in Fig. 14, which depicts the density of matching probabilities of candidate pairs over the AbtBuy dataset for various training set sizes, when using the Logistic Regression as the classification algorithm. The duplicate pairs are shown in red and the non-matching ones in blue. The upper line corresponds to the maximum pruning threshold and the lower line to the average one, across all nodes/entities. We observe that for the smallest training set, the matching probabilities of both types of candidate pairs fluctuate in $[0.5, 0.95]$, with the density of the matching and the non-matching pairs being higher in the upper and the lower part, respectively. As we move to larger training sets, the matching pairs are pushed up, fluctuating in $[0.75, 0.95]$. This indicates the higher discriminatory power of the probabilistic classifier, which increases the number of true positives, raising recall. However, the probabilities of the non-matching pair continue to fluctuate in $[0.5, 0.85]$, but are now concentrated around 0.7, while the most pruning thresholds are confined in $[0.63, 0.65]$. As a result, the number of false positive increases, too, dropping precision to lower levels. Similar patterns appear in the rest of the datasets in combination with other classification algorithms, such as SVC.

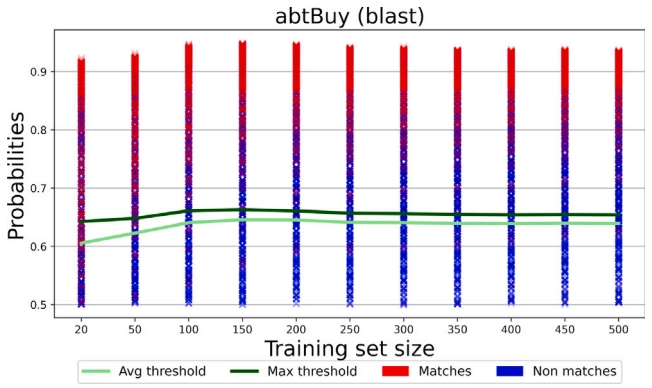


Fig. 14. The matching probabilities for both types of candidate pairs, the duplicate and the non-matching ones, as the size of the training set increases. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

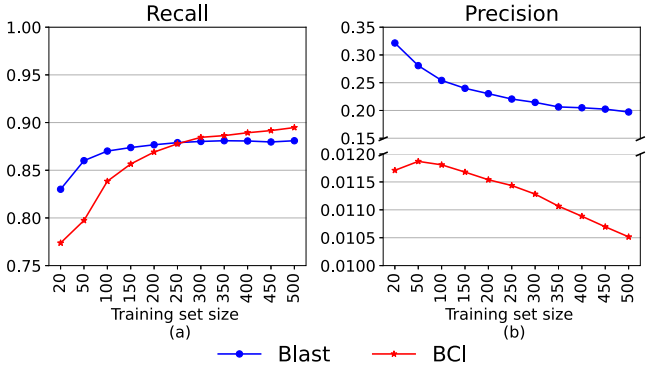


Fig. 15. Recall and precision of BCl and BLAST as the size of the training set increases.

It is worth noting at this point that the same behaviour as BLAST is exhibited by BCl, the original binary classifier presented in [13], which simply retains all candidate pairs with a matching probability above 0.5. As shown in Fig. 15, both algorithms increase their recall and decrease their precision as more labelled instances as used during their training.

Given that 20 labelled instances yield very low recall, especially for RCNP, but with 50 instances, the recall becomes quite satisfactory for both algorithms ($\gg$0.85, on average, across all datasets), we can conclude that *the optimal training set involves just 50 labelled instances, equally split among positive and negative ones*.

Comparison with Supervised Meta-blocking—Based on the above experimental results, we now compare the final algorithms of Generalized Supervised Meta-blocking with their baseline counterparts from [13].

Table 4a–c reports a full comparison between the main weight-based algorithms, i.e., BLAST and BCl (note that BCl_2 uses the training set specified in [13], i.e., a random sample involving 5% of the positive instances in the ground-truth along with an equal number of randomly selected negative instances).

We observe that on average, BLAST outperforms BCl_2 with respect to all effectiveness measures, increasing the average recall, precision and F1 by 7.1%, 5.0% and 9.9%, respectively. Compared to BCl_1, BLAST increases the average recall by 3.95%, but lowers the precision by 5.9% and the F1 by 2.2%. Recall drops below 0.8 in four datasets for BCl_1 (and BCl_2), whereas BLAST violates this limit in just two datasets. This should be attributed to duplicate pairs that share just one block in the original block collection, due to missing or erroneous values, as explained in detail in Section 5.4.1. BCl_1 outperforms BCl_2 in all respects, demonstrating the effectiveness of the new feature set.

In terms of run-time, BLAST is slower than BCl_1 by 8.2%, on average, because it iterates once more over all candidate pairs. Compared to BCl_2, BLAST is 6.7 times faster, on average across all datasets, because of LCP and of the large training sets, which learn complex binary classifiers with a time-consuming processing.

Similar patterns are observed in the case of cardinality-based algorithms in Table 4d-f, where RCNP with its best features and 50 labelled instances is compared with CNP, which uses the best feature set from [13] and two different training sets: the same 50 labelled instances as RCNP (CNP_1) and the training set specified in [13] (CNP_2).

In more detail, RCNP typically outperforms both baseline methods with respect to all effectiveness measures. Compared to CNP_1 (CNP_2), RCNP raises the average recall by 5.3% (9.2%), while achieving the highest precision and F1 across all datasets, except for `AbtBuy` and `ImdbTmdb` (and `ScholarDblp` in the case of CNP_1). The relative increase in precision in comparison to CNP_1 ranges from 7.5% over `TmdbTvdb` to 10 and 24 times over `Movies` and `WalmartAmazon`, respectively. Compared to CNP_2, precision raises from 5.0% over `DblpAcm` to 49 times over `WalmartAmazon`. In all cases, F1 increases to a similar extent. These patterns suggest that RCNP is typically more accurate in classifying the positive candidate pairs.

In terms of run-time, RCNP is slower than CNP_1 by 6.2%, on average, as it retains the candidate pairs that are among the top-k weighted ones for both constituent entities (i.e., it searches for pairs in two lists), whereas CNP_1 simply merges the lists of all entities. CNP_2 employs a much larger training set, yielding more complicated and time-consuming classifiers than RCNP, which is 3 times faster, on average, across all datasets.

Overall, Generalized Supervised Meta-blocking outperforms Supervised Meta-blocking to a significant extent, despite using a balanced training set of just 50 labelled instances.

5.4.1. Considerations regarding the obtained recall

Blocking recall typically sets the upper bound on matching recall, i.e., on the overall recall of ER. The problem of low recall appears in cases where $PC < 0.9$ in Table 4a–c, which reports analytically the performance over all datasets for the weight-based algorithms that emphasize recall. The rest of the tables report the average recall across all datasets, which is very close to 0.9 for BLAST in most cases, while Table 4d–f examines cardinality-based algorithms, which emphasize precision (e.g., progressive ER applications that by default operate with lower recall).

Studying Table 4a–c, we observe that the recall of BLAST is lower than 0.9 in five datasets: AbtBuy, AmazonGP, ImdbTmdb, ImdbTvdb and TmdbTvdb. Comparing these datasets with the remaining four, where recall is well above 0.9, we observe that they involve high levels of noise, which causes a large part of the duplicate entities to share just one block. Inevitably, these matching pairs lack significant co-occurrence patterns and receive low probabilities by all weighting schemes used by our approach. As a result, most of them are pruned at the cost of lower recall. In other words, BLAST exhibits low recall in block collections with very low levels of redundancy (due to noisy and missing values), where the matching pairs have just one block in common. Note, though, that Generalized Supervised Meta-blocking outperforms the original Supervised Meta-blocking approach, whose PC is lower by ~5% in these cases; this means that more duplicates pairs with a single common block are retained by our approach, while increasing precision, too. This should be attributed to the new features we have proposed in this work, which normalize the co-occurrence patterns by considering all blocks containing every entity. In block collections with high levels of redundancy, where the vast majority of duplicates co-occurs in multiple blocks, the recall remains high even after Supervised Meta-blocking.

This is verified in Figs. 17 and 18, which plot the portion of matching pairs (on the vertical axis) with respect to the number of common blocks (on the horizontal axis). In every dataset, the bar that

Table 4
Performance of the main weight- and cardinality-based algorithms across all datasets in a–c and d–f, respectively. *RT* is the mean run-time (in seconds) over 10 repetitions. The μ column reports the average performance across all the datasets.

	AbtBuy	DblpAcm	ScholarDblp	AmazonGP	ImdbTmdb	ImdbTvdb	TmdbTvdb	Movies	WalmartAmazon	μ
(a) BLAST with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, RS, NRS\}$										
Re	0.8345	0.9511	0.9638	0.7001	0.8223	0.7483	0.8466	0.9151	0.9587	0.8601
Pr	0.2037	0.6509	0.3418	0.1441	0.5756	0.2304	0.2477	0.1300	0.0025	0.2807
F1	0.3265	0.7690	0.4988	0.2385	0.6726	0.3456	0.3770	0.2221	0.0050	0.3839
RT	6.58	5.62	11.90	6.83	6.46	6.36	7.51	96.01	107.82	28.34
(b) BCl_1 with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, RS, NRS\}$										
Re	0.8345	0.9521	0.9588	0.6265	0.7889	0.6966	0.6972	0.9039	0.9500	0.8232
Pr	0.1821	0.5971	0.3595	0.1607	0.6445	0.2616	0.3737	0.0972	0.0020	0.2976
F1	0.2981	0.7303	0.5195	0.2572	0.7086	0.3785	0.4613	0.1735	0.0041	0.3923
RT	5.40	5.66	10.51	6.02	5.79	5.49	6.69	82.71	107.51	26.19
(c) BCl_2 with the training set and the features of [13], i.e., $\{CF\text{-}IBF, RACCB, JS, LCP\}$										
Re	0.8183	0.9513	0.9303	0.7316	0.7872	0.7074	0.8172	0.9100	0.5757	0.8032
Pr	0.2039	0.6130	0.3921	0.1131	0.5969	0.2323	0.2312	0.0239	0.0001	0.2674
F1	0.3261	0.7425	0.5401	0.1908	0.6604	0.3395	0.2991	0.0465	0.0001	0.3495
RT	15.07	9.37	27.73	13.22	11.04	9.68	10.86	1328.81	276.19	46.65
(d) RCNP with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, JS, LCP, WJS\}$										
Re	0.8405	0.9759	0.9623	0.7358	0.8395	0.7465	0.8696	0.9275	0.9122	0.8678
Pr	0.1764	0.6463	0.3591	0.1264	0.3540	0.2325	0.1848	0.0992	0.0050	0.2426
F1	0.2914	0.7747	0.5190	0.2148	0.4971	0.3498	0.2954	0.1758	0.0100	0.3476
RT	6.20	5.67	11.73	6.83	6.55	6.77	8.32	126.13	107.56	31.75
(e) CNP_1 with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, JS, LCP, WJS\}$										
Re	0.8294	0.9613	0.9218	0.7462	0.8045	0.7615	0.8641	0.8200	0.7087	0.8242
Pr	0.1797	0.5984	0.3745	0.1031	0.5471	0.1867	0.1720	0.0090	0.0002	0.2412
F1	0.2939	0.7355	0.5095	0.1748	0.6394	0.2847	0.2487	0.0177	0.0004	0.3227
RT	5.95	5.80	11.33	6.40	5.91	6.19	6.89	122.72	107.62	30.98
(f) CNP_2 with the training set and the features of [13], i.e., $\{CF\text{-}IBF, RACCB, JS, LCP\}$										
Re	0.8347	0.9539	0.9581	0.7742	0.8345	0.7641	0.8677	0.9347	0.2332	0.7950
Pr	0.1895	0.6158	0.2184	0.0848	0.4132	0.1764	0.1484	0.0291	0.0001	0.2084
F1	0.3081	0.7457	0.3453	0.1514	0.5247	0.2754	0.2363	0.0564	0.0002	0.2937
RT	15.61	9.64	28.51	13.63	11.37	9.99	11.41	1351.54	365.03	58.15

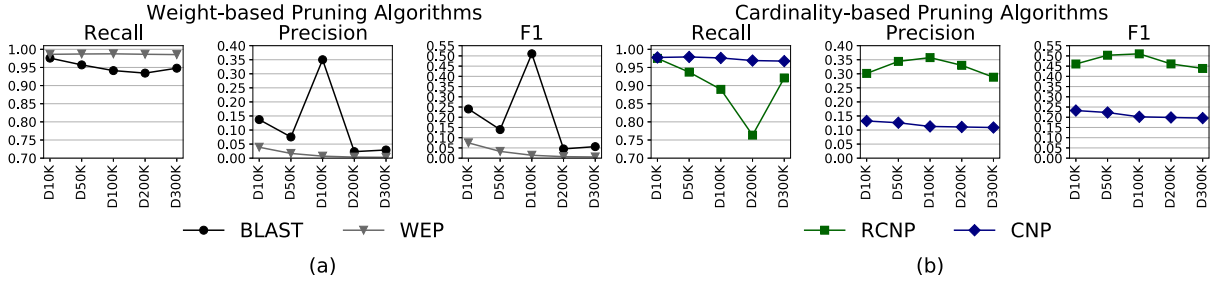


Fig. 16. Scalability over the datasets in Table 1b: (a) the weight-based pruning algorithms, (b) the cardinality-based ones, and (c) speedup.

corresponds to $x = 0$ indicates the portion of duplicates that are missed by the original block collection. The bar that corresponds to $x = 1$ indicates the portion of duplicates that are missed by (Generalized) Supervised Meta-blocking. We observe that for all datasets in Fig. 17 both bars are below 5%, while for all datasets in Fig. 18, more than 10% corresponds to $x = 1$. As a result, $PC > 0.9$ for the former datasets and vice versa for the latter ones.

5.5. Scalability analysis

We assess the scalability of our approaches as the number of candidate pairs $|C|$ increases, verifying their robustness under versatile settings: instead of real-world Clean-Clean ER datasets, we now consider the synthetic Dirty ER datasets, and instead of SVC, we train our models using Weka's default implementation of Logistic Regression [31].

The characteristics of the datasets, which are widely used in the literature [4,11], appear in Table 1b. To extract a large block collection from every dataset, we apply Token Blocking. In all cases, the recall is almost perfect, but precision and F1 are extremely low, as shown in the rightmost part of Table 1b.

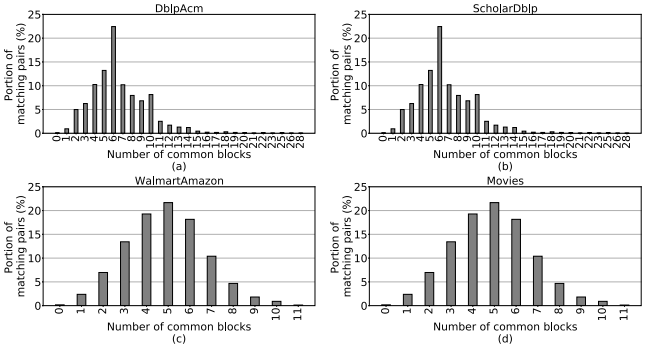


Fig. 17. The distribution of common blocks (horizontal axis) per portion of duplicate pairs (vertical axis) in datasets with $PC > 0.9$ for Supervised BLAST.

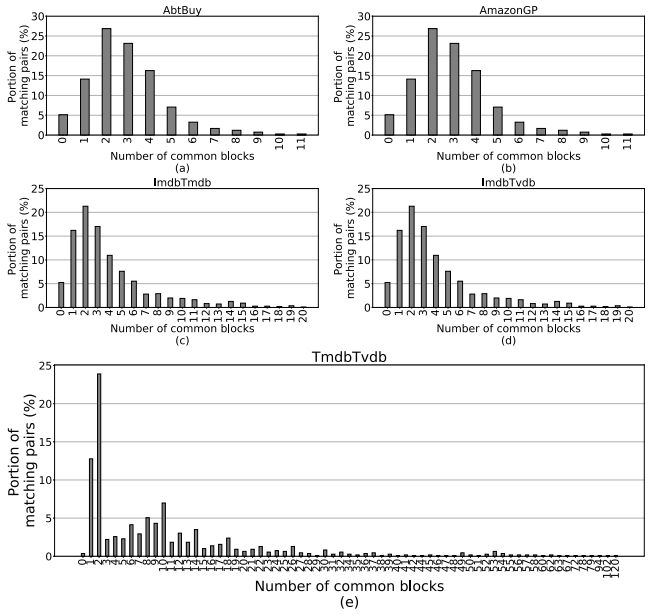


Fig. 18. The distribution of common blocks (horizontal axis) per portion of duplicate pairs (vertical axis) in datasets with $PC < 0.9$ for Supervised BLAST.

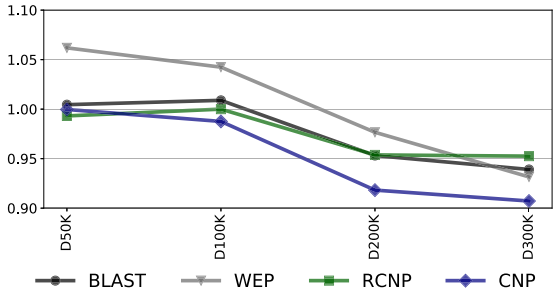


Fig. 19. The speedup of the algorithms used in the scalability analysis of Fig. 16.

We consider four methods: BCl and CNP with the features and the training set size specified in [13] as well as BLAST and RCNP with the features in Table 4a and d, respectively, trained over 50 labelled instances (25 per class). In each dataset, we performed three repetitions per algorithm and considered the average performance.

The effectiveness of the weight- and cardinality-based algorithms over all datasets appear in Fig. 16a and b, respectively. BLAST significantly outperforms BCl in all cases: on average, it reduces recall by 3.5%, but consistently maintains it above 0.93, while increasing precision and F1 by a whole order of magnitude. Note that BLAST's precision is much higher than expected over D_{100K}, due to the effect of random sampling: a different training set is used in every one of the three iterations, with two of them performing a very deep pruning that decrease recall to a minor extent.

RCNP outperforms CNP to a significant extent: on average, it reduces recall by 7.9%, but maintains it to very high levels — except for D_{200K}, where it drops to 0.77, due to the effect of random sampling; yet, precision raises by 2.8 times and F1 by 2.3 times. These results verify the strength of our approaches, even though they require orders of magnitude less labelled instances than [13].

Most importantly, our approaches scale better to large datasets, as demonstrated by speedup in Fig. 19. Given two sets of candidate pairs, $|C_1|$ and $|C_2|$, such that $|C_1| < |C_2|$, this measure is defined as follows:

$$speedup = \frac{|C_2|}{|C_1|} \times \frac{RT_1}{RT_2},$$

where RT_1 (RT_2) denotes the running time over $|C_1|$ ($|C_2|$) – in our case, C_1 corresponds to D_{10K} and C_2 to all other datasets. In essence, speedup extrapolates the running time of the smallest dataset to the largest one, with values close to 1 indicating linear scalability, which is the ideal case. We observe that all methods start from very high values, but BCl and CNP deteriorate to a significantly larger extent than BLAST and RCNP, respectively, achieving the lowest values for D_{300K}. This should be attributed to their lower accuracy in pruning the non-matching comparisons, which deteriorates as the number of candidate pairs increases. As a result, they end up retaining and processing a much larger number of comparisons, which slows down their functionality.

Overall, Generalized Supervised Meta-blocking scales much better to large datasets than Supervised Meta-blocking [13] for both weight- and cardinality-based algorithms. For a bit lower recall, it raises precision and F1 by ≥2 times and maintains a much higher speedup.

5.6. Progressive ER

We now assess the performance of Generalized Supervised Meta-blocking on Progressive ER (see Section 2). To this end, we integrate it with PPS, using as edge weights the probabilities generated by Logistic Regression in combination with the best feature vector selected for weight-based algorithms, i.e. $\{CF-IBF, RACCB, RS, NRS\}$. The probabilistic classifier is trained over a balanced training set of 50 samples. For brevity, we call this approach *Supervised PPS*.

As a baseline method, we consider the average performance of the original PPS algorithm in combination with each of the weighting schemes described in Section 4. In particular, the following schemes were employed: $CF-IBF(c_{i,j})$, $RACCB(c_{i,j})$, $JS(c_{i,j})$, $EJS(c_{i,j})$, $WJS(c_{i,j})$, $RS(c_{i,j})$, and $NRS(c_{i,j})$. We call this approach *Unsupervised PPS*.

Note that both versions of PPS might produce ties, i.e., edges/comparisons with identical weights. If these ties pertain to the first k edges that should be emitted, the end result is arbitrary. To break these ties, we run the experiment 10 times and in each iteration, we emit randomly the required number of edges among those with the same weight. For example, for $AUC^*_m@1$, if for a profile there are three edges corresponding to the maximum weight, only one of them is randomly emitted in every repetition of the experiment. The average performance of the 10 iterations is then taken into account.

To compare Supervised PPS with its unsupervised counterpart, we consider the normalized area under the curve (AUC^*_m) at an increasing normalized number of emitted comparisons: $ec* \in \{1, 5, 10, 20\}$. This applies to all Clean-Clean and Dirty ER datasets in Table 1. The results appear in Figs. 20 and 21, respectively.

Starting with the Clean-Clean ER datasets, we observe that Supervised PPS consistently outperforms Unsupervised PPS to a significant extent across all datasets and settings. The lowest difference between the two approaches with respect to $AUC^*_m@1$, $AUC^*_m@5$ and $AUC^*_m@10$ corresponds to ScholarDBLP, where the former outperforms the latter by 14.3%, 19.6% and 26%, respectively. For $AUC^*_m@20$, the lowest difference (25.5%) appears in Amazon-GP. Yet, there is a large variation in the difference between the two methods among the various settings. For example, the progressive recall of the supervised approach is at least twice higher than the unsupervised one in many cases: in Movies for $ec* \in \{1, 5, 10\}$, in WalmartAmazon for $ec* \in \{5, 10, 20\}$ and in Imdb-Tvdb and Tmdb-Tvdb for $ec* = 1$. On average, Supervised PPS is superior by 76%, 66.6%, 58.8% and 50.6% for $ec* = 1$, 5, 10 and 20, respectively (note that the difference decreases with the increase of computational budget, because progressive recall raises proportionally to ec, i.e., the more comparisons are executed within the allocated budget, the higher recall gets). These results indisputably verify that the probabilities learned by Generalized Supervised Meta-blocking provide more accurate and comprehensive evidence for scheduling the processing of comparisons than the weights of individual schemes.

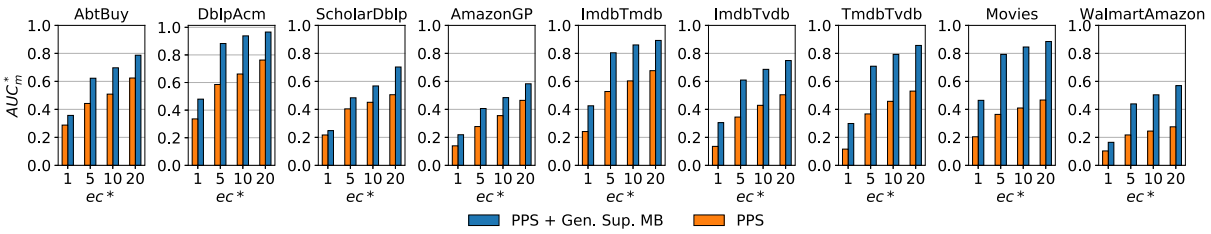


Fig. 20. Progressive recall over the Clean-Clean ER datasets as the number of emitted comparisons increases.

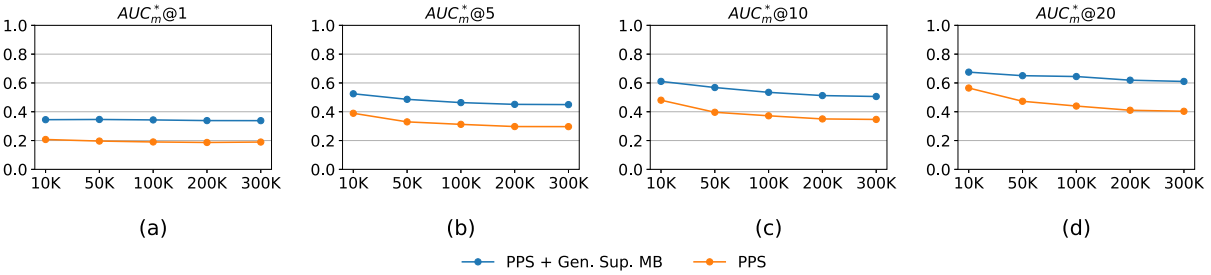


Fig. 21. Progressive recall over the Dirty ER datasets as the number of emitted comparisons increases.

The same patterns apply to the Dirty ER: as shown in Fig. 21, which depicts the evolution of Progressive Recall with the increase of the input data, Supervised PPS consistently outperforms its unsupervised counterpart to a statistically significant extent ($p = 2.7 \cdot 10^{-17}$) under all settings. We actually observe that the deviation between the two methods is low for D_{10K}, but gets higher and almost stable for the rest of the datasets. This applies to all settings of Progressive Recall, i.e., for $ec* \in \{1, 5, 10, 20\}$. In more detail, $AUC_m^*@1$ is higher for Supervised PPS than for Unsupervised PPS by 66.3% over D_{10K} and by 79.1 ± 2.2% over $D_{50K} - D_{300K}$. For $AUC_m^*@5$ ($AUC_m^*@10$), the superiority of the former approach raises from 34.8% (27.2%) over D_{10K} to 49.8 ± 2.2% (44.9 ± 1.5%) over the rest of the datasets. This is caused by the more challenging tasks that are posed by the larger datasets: the number of duplicates increases linearly with the size of the input data, while the possible comparisons increase quadratically. As a result, the portion of non-matching pairs over all comparisons is lower in D_{10K} than the rest of the datasets, thus enabling Unsupervised PPS to provide more accurate weights. In other words, the individual weights of Unsupervised PPS become less capable of distinguishing between matching and non-matching comparisons than the probabilities of Supervised PPS when the candidate pairs increase.

The only exception is $AUC_m^*@20$, because the higher computational budget yields higher absolute recall for both methods. As a result, the difference between Supervised and Unsupervised PPS increases gradually, as the task becomes more challenging, due to a higher portion on non-matching comparison. Indeed, the former approach is better than the latter by 19.5% over D_{10K}, by 37.6% over D_{50K} and by 46.6% over D_{100K}, while stabilizing to 51.1 ± 0.4% for the two largest datasets.

Overall, *Generalized Supervised Meta-blocking outperforms unsupervised PPS to a statistically significant extent, and avoids the burden of trying different weighting schemes to identify the best performer*.

5.7. Comparison with recently published blocking techniques

Comparison with pre-trained language models-based blocking techniques. Recently, several works use pre-trained language models (LMs) to perform blocking [17–19]. The techniques proposed in these works first employ the LM to transform the input entity profiles into dense embedding vectors. Then, they generate the candidate pairs by selecting the K nearest neighbours (kNN) of each profile according to a similarity measure between these embedding vectors (e.g., the cosine similarity). The use of LMs aims to produce more accurate blocking results than using the syntactic similarity of the textual descriptions (e.g., as in token blocking), since it relies on the semantic meaning and similarity of these descriptions. To experimentally verify this assumption, we compare our Generalized Supervised Meta-blocking approaches with three recent works that have publicly released their implementation: DeepBlocker [17],[4] Sudowoodo [18][5] and ContextualBlocker [19].[6]

DeepBlocker [17] proposes a self-supervised technique to learn the embeddings per entity profile, without requiring any labelled training data. As stated in [17], it is crafted for datasets with perfectly aligned attributes. However, among the datasets in Table 1(a), only DblpAcm, AbtBuy, and Scholar-Dblp satisfy this requirement. We solved this problem by concatenating all the values of the different attributes in a single string on which the blocking was performed, i.e., using schema-agnostic settings just like Generalized Supervised Meta-blocking. For the generation of tuple embeddings, we used the *AutoEncoder*, because it constitutes the most effective module under the schema-aware settings, while ranking second in the schema-agnostic ones [17]. The *Hybrid* module exhibits slightly higher effectiveness under these settings, but suffers from significantly higher space and time complexity, raising out-of-memory exceptions in most cases, while being orders of magnitude slower than the AutoEncoder. For these reasons, we exclusively combine DeepBlocker with the Autoencoder module. Following [17], the LM that lies at the core of this approach is fastText [32].

Sudowoodo [18] starts with a pretraining phase to fine-tune a pre-trained LM using positive/negative pairs of labelled data. Then, the fine-tuned model is used to generate the embeddings. It does not require aligned attributes. We used DistilBERT [33] as LM, since it is reported in [18] that is the best model for blocking. Moreover, we set all the training parameters as reported in the Sudowoodo repository. To compare it with our approach, we pretrained it with 50 labelled pairs, and then with 500, divided in 80% for training and 20% for validation. To ensure that the number of supplied labels is used, we have disabled the pseudo-labelling function that generates additional labels.

ContextualBlocker [19] is an unsupervised and schema-agnostic method: first, it generates the embedding vectors by combining SimCSE

[4] https://github.com/qcri/DeepBlocker.
[5] https://github.com/megagonlabs/sudowoodo.
[6] https://github.com/boscoj2008/ContextualBlocker-for-EM.

Table 5
Performance of our main (a) weight-based and (b) cardinality-based algorithms, compared with the recent, state-of-the-art, pre-trained language models-based blocking solutions (c–f), and with Sparkly [20] a recent solution based on TF/IDF. *RT* is the mean run-time (in seconds) over 10 repetitions. The μ column reports the average performance across all the datasets.

	AbtBuy	DblpAcm	ScholarDblp	AmazonGP	ImdbTmdb	ImdbTvdb	TmdbTvdb	Movies	WalmartAmazon	μ
(a) BLAST with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, RS, NRS\}$										
Re	0.8345	0.9511	0.9638	0.7001	0.8223	0.7483	0.8466	0.9151	0.9587	0.8601
Pr	0.2037	0.6509	0.3418	0.1441	0.5756	0.2304	0.2477	0.1300	0.0025	0.2807
F1	0.3265	0.7690	0.4988	0.2385	0.6726	0.3456	0.3770	0.2221	0.0050	0.3839
RT	7	6	12	7	6	6	8	96	108	28
(b) RCNP with 50 labelled pairs and $\{CF\text{-}IBF, RACCB, JS, LCP, WJS\}$										
Re	0.8405	0.9759	0.9623	0.7358	0.8395	0.7465	0.8696	0.9275	0.9122	0.8678
Pr	0.1764	0.6463	0.3591	0.1264	0.3540	0.2325	0.1848	0.0992	0.0050	0.2426
F1	0.2914	0.7747	0.5190	0.2148	0.4971	0.3498	0.2954	0.1758	0.0100	0.3476
RT	6	6	12	7	7	7	8	126	108	32
(c) DeepBlocker										
Re	0.8411	0.9987	0.9424	0.7215	0.7907	0.6754	0.9489	0.2754	0.8137	0.7786
Pr	0.0168	0.0170	0.0173	0.0138	0.0061	0.0028	0.0034	0.0004	0.0074	0.0094
F1	0.0330	0.0334	0.0340	0.0270	0.0121	0.0056	0.0068	0.0008	0.0146	0.0186
RT	5	7	47	8	10	11	13	103	75	31
(d) Sudowoodo with 50 labelled pairs										
Re	0.2128	0.9996	0.9788	0.4931	0.1494	0.0653	0.0210	0.0758	0.2132	0.3565
Pr	0.0043	0.0194	0.0007	0.0040	0.0010	0.0002	0.0001	0.0015	0.0002	0.0035
F1	0.0084	0.0380	0.0015	0.0079	0.0019	0.0004	0.0001	0.0029	0.0004	0.0068
RT	74	80	405	114	243	179	248	399	473	246
(e) Sudowoodo with 500 labelled pairs										
Re	0.5994	1.0000	0.9944	0.5608	0.1784	0.1390	0.0292	0.1401	0.8016	0.4936
Pr	0.0120	0.0194	0.0007	0.0045	0.0012	0.0004	0.0001	0.0028	0.0008	0.0047
F1	0.0235	0.0380	0.0015	0.0090	0.0023	0.0008	0.0002	0.0054	0.0017	0.0092
RT	74	80	406	125	242	181	249	407	493	251
(f) ContextualBlocker										
Re	0.8968	0.6515	0.6118	0.6454	0.0295	0.0224	0.5817	0.0047	0.7938	0.4708
Pr	0.0038	0.0007	$9.33 \cdot 10^{-6}$	0.0008	$1.08 \cdot 10^{-5}$	$2.94 \cdot 10^{-6}$	0.0001	$6.16 \cdot 10^{-7}$	0.0001	0.0006
F1	0.0075	0.0015	$1.87 \cdot 10^{-5}$	0.0016	$2.16 \cdot 10^{-5}$	$5.88 \cdot 10^{-6}$	0.0001	$1.23 \cdot 10^{-6}$	0.0001	0.0012
RT	5	10	266	19	26	33	40	230	107	82
(g) Sparkly with fine-tuned k										
Re	0.8894	0.9883	0.9918	0.6992	0.9050	0.7556	0.9434	0.9033	0.9558	0.8924
Pr	0.8894	0.9582	0.0373	0.0704	0.2941	0.1037	0.1323	0.0045	0.0125	0.2780
F1	0.8894	0.9730	0.0719	0.1280	0.4439	0.1824	0.2320	0.0089	0.0247	0.3282
RT	7	6	55	17	8	8	11	500	265	97
(h) Sparkly with k = 10										
Re	0.9805	0.9991	0.9996	0.7692	0.9705	0.9440	0.9763	0.7884	0.9757	0.9337
Pr	0.0980	0.0969	0.0038	0.0310	0.0315	0.0130	0.0137	0.0778	0.0051	0.0412
F1	0.1783	0.1766	0.0075	0.0596	0.0611	0.0256	0.0270	0.1415	0.0101	0.0764
RT	7	6	66	9	10	9	14	475	280	97

[34] with the BERT language model [35] and then, it computes the k nearest neighbours of each profile. The retrieved candidates construct a graph, where every node corresponds to an entity profile and edges connect the candidate pairs. Finally, it applies the Louvain community detection algorithm [36] on the resulting graph to generate the blocks.

The experimental evaluation was conducted by setting $k = 50$ for the kNN selection so as to maximize recall. All kNN experiments were performed on an NVIDIA GeForce RTX 4090 GPU with 24 GB of RAM. The obtained results across all the datasets are reported in Table 5.

DeepBlocker exhibits a slightly better recall than BLAST on four datasets (namely AbtBuy, Dblp-Acm, Amazon-GP, and Tmdb-Tvdb) and on three (i.e., AbtBuy, Dblp-Acm, and Tmdb-Tvdb) when compared to RCNP. On all of the other datasets, their recall is basically the same. The only exception is Movies, where DeepBlocker obtains a very low recall. That happens because the duplicate films are mostly identified by the names of the cast rather than the film's title; yet the embedding vectors of personal names have low accuracy (even for the fine-grained character-level LM of fastText), given that the training corpora are generic, not domain-specific.

Sudowoodo with 50 labelled pairs performs worse that our approach basically on all the datasets but Dblp-ACM and Scholar-Dblp—on these two datasets, almost all blocking methods achieve very high recall, due to the relatively clean, unambiguous bibliographic records they contain. Overall, Sudowoodo obtains a very low recall, which is improved when increasing the number of labelled training data from 50 to 500—even though it does not reach the performance of Generalized Supervised Meta-blocking. This indicates that if more labels are provided, Sudowoodo raises its effectiveness to a significant extent, at the expense of a much higher labelling effort by human experts.

ContextualBlocker outperforms our solutions only on AbtBuy, obtaining a much lower recall on all the other datasets.

Regarding precision, BLAST and RCNP outperform all the other methods on all the datasets by at least two orders of magnitude.

Finally, on average, the execution time is almost the same for BLAST, RCNP, and DeepBlocker, while ContextualBlocker and Sudowoodo are 2.9 and 8.8 times, respectively, slower than BLAST, on average. The differences are significantly higher (in favour of BLAST and CNP), when running the semantic-based blocking methods on a CPU core.

Comparison with Sparkly [20]. Sparkly is a recent approach based on TF/IDF [37] that has been shown to outperform pre-trained language model-based blocking techniques [20]. Built on top of Apache

Lucene[7] and Apache Spark,[8] it works as follows: given two tables (datasets) A and B with the same schema, it tokenizes each tuple (entity) by using n-grams, indexes the smallest table with Lucene (let us say A) and distributes table B across Spark' workers; using the resulting Lucene index, Sparkly searches for each tuple of B the top-k similar ones of A according to an appropriately adapted version of the *Okapi BM25* similarity function, which relies on the TF/IDF weights.

Sparkly is unsupervised, yet it requires the user to specify (i) the number of neighbours, k, to retrieve for each record, and (ii) the attributes that will be used in blocking. It can also work in an automatic fashion by selecting the best attributes for blocking, but this is possible only if datasets have perfectly aligned attributes. However, among the datasets in Table 1(a), only Dblp-Acm, AbtBuy, and Scholar-Dblp satisfy this requirement. Thus, we used Sparkly by manually setting the value of k, while concatenating all the attribute values per tuple in a single string on which blocking was performed, i.e., we used schema-agnostic settings, just like with Generalized Supervised Meta-blocking. We combined these schema-agnostic settings with the 3-gram tokenizer that is by default used in [20].

In this context, the performance of Sparkly heavily depends on k, i.e., the number of candidates per query. We ran preliminary experiments (not reported here) to fine-tune k, determining the right value per dataset that allows for achieving a recall that is as close as possible to the one obtained by Generalized Supervised Meta-blocking—in this way, we can compare the two methods on an equal basis. As a result, we set $k = 4$ for Amazon-GP and Walmart-Amazon, $k = 200$ for Movies, and $k = 1$ for the rest of the datasets. The results for this configuration are reported in Table 5(g).

On average, Sparkly exhibits a better recall at the expense of lower precision. However, the high precision of Sparkly stems from its exceptional performance on two datasets, namely AbtBuy and DblpAcm, where its precision exceeds 0.89 (just like its recall). This is much higher, though, than its precision in the rest of the datasets. Excluding these two datasets, Sparkly averages 0.879 for recall and 0.094 for precision, compared to 0.850 and 0.239, respectively, for Supervised BLAST. This means that on average, across the seven largest datasets, our approach trades a considerably higher precision (2.5x higher) for only a slightly lower recall (3% decrease).

Looking more carefully into the performance of Supervised BLAST and Sparkly, we distinguish the nine considered datasets into three groups:

- The first group includes AbtBuy and DblpAcm, where Sparkly dominates Supervised BLAST with respect to both recall and precision. The former is higher by 5% and the latter by 55%, on average. Note that in both datasets, $k = 1$ for Sparkly.
- The second group includes AmazonGP and Movies, where Supervised BLAST dominates Sparkly with respect to both effectiveness measures. Its recall is higher by just 0.7%, because k has been adjusted to higher values (4 and 200, respectively) in order to approach that of Supervised BLAST. In terms of precision, though, Sparkly underperforms Supervised BLAST by 73.8%. The superiority of the two methods is more intense in Movies, where BLAST's precision is 29 times higher than Sparkly. This should be attributed to the extreme levels of noise and missing values in the profiles of this dataset.
- The third groups includes the datasets where Supervised BLAST trades lower recall for much higher precision. These are ScholarDblp, ImdbTmdb, ImdbTvdb and TmdbTvdb. On average, BLAST's recall is lower by 5.8% than Sparkly, while its precision is higher by 59.9%. Note that in all four datasets, $k = 1$ for Sparkly, which means that we cannot lower its recall in order to increase precision. Note also that all datasets of this group involve much less duplicate pairs than the number of entities in each data source, i.e., $|D| \ll |E_1|$ and $|D| \ll |E_2|$, as can be seen in Table 1. This is in contrast to the datasets of the first group, where $|D| = |E_1| = |E_2$ for AbtBuy and $|D| \approx |E_1| \approx |E_2$ for DblpAcm. This means that *Sparkly works vey well in cases, where there is an almost 1-1 mapping between the entities of two data sources, but is not flexible enough to reduce the number of candidate pairs in datasets with a low portion of duplicates*.
- The last group includes only WalmartAmazon, where the two methods exhibit practically identical recall, but Sparkly achieves much higher precision for $k = 4$. The reason for the lower effectiveness of Supervised BLAST is the very large set of candidate pairs that are generated by Token Blocking, Block Purging and Block Filtering (see $|C|$ in Table 1). Inevitably, Supervised BLAST retains a larger portion of false positives in comparison to all other datasets, which yield fewer candidate pairs.

Finally, we measured the execution time of Sparkly after configuring it so that it uses a single CPU core — this way, we can perform a fair comparison with Supervised BLAST. With the exception of the two smallests datasets, AbtBuy and DblpAcm, where the two methods are equally fast, Sparkly is consistently slower than Supervised BLAST to a significant extent. Tts average run-time, across all nine datasets, is 3.5 times higher than that of Supervised BLAST.

It is worth stressing at this point that one major limitation of Sparkly is that the parameter k has to be manually set. This is rarely easy, but has a significant impact on the final performance. For example, if we set $k = 1$ on AmazonGP, Movies and WalmartAmazon, its recall drops to 0.4908, 0.6559 and 0.8536, respectively, which is much lower than that of Supervised BLAST. The opposite is true if $k = 10$, the lowest value that is used in [20], whose performance is reported in Table 5(h). We observe that on average, across all nine datasets, its recall raises by ∼4%, but its precision drops by 6.7 times, being 7 times lower than Supervised BLAST. This means that expert knowledge or perhaps a training set is required for fine-tuning k on the data at hand.

Overall, we conclude that our techniques are still competitive with LM-based blocking solutions, yielding the same level of recall for a significantly better level of precision, while exhibiting a lower execution time without requiring GPUs. Compared to Sparkly [20], a recent TF/IDF-based approach that outperforms LM-based ones, Supervised BLAST offers a much better trade-off between recall and precision in most cases, while being consistently faster.

6. Related work

The unsupervised pruning algorithms WEP, WNP, CEP, and CNP were introduced in [12]. WNP and CNP were redefined in [22] so that they do not produce block collections with redundant comparisons. Unsupervised Reciprocal WNP and Reciprocal CNP were coined in [22], while unsupervised BLAST was proposed in [38].

Over the years, more unsupervised pruning algorithms have been proposed in the literature. [39] proposes a variant of CEP that retains the top-weighted candidate pairs with a cumulative weight higher than a specific portion of the total sum of weights. Crafted for Semantic Web data, MinoanER [24] combines meta-blocking evidence from two complementary block collections: the blocks extracted from the names of entities and from the attribute values of their neighbours. BLAST2 [8] leverages loose schema information in order to boost the performance of Meta-blocking's weighting schemes. Finally, a family of pruning algorithms that focuses on the comparison weights inside individual blocks is presented in [40]; for example, Low Entity Co-occurrence Pruning removes from every block a specific portion of the entities with the lowest average weights. Our approaches can be generalized to these algorithms, too, but their analytical examination lies out of our scope.

[7] https://lucene.apache.org.
[8] https://spark.apache.org.

The above works consider Meta-blocking in a static context that ignores the outcomes of Matching. A dynamic approach that leverages Meta-blocking to make the most of the feedback of Matching is *pBlocking* [41]. After applying Matching to the smallest blocks, intersections of the initial blocks are formed and scored based on their ratio of matching and non-matching entities. Meta-blocking is then applied to produce the next set of candidate pairs that will be processed by Matching. This process is iteratively applied until convergence. *BEER* [42] is an open-source tool that implements pBlocking.

The work closest to ours is BLOSS [43]. It introduces an active learning approach that reduces significantly the size of the labelled set required by Supervised Meta-blocking. Initially, it partitions the unlabelled candidate pairs into similarity levels based on CF-IBF. Then, it applies rule-based active sampling inside every level in order to select the unlabelled pairs with the lowest commonalities with the already labelled ones so as to maximize the captured information. In the final step, BLOSS cleans the labelled sample from non-matching outliers with high Jaccard weight.

Note that we tried to use BLOSS [43] as a baseline method, but we could not reproduce its performance, since our implementation of the algorithm exclusively selected non-matching candidate pairs — instead of a balanced training set. (We contacted the authors, but they were not able to provide us with their own implementation.) Nevertheless, our experimental results demonstrate that active learning is not necessary for our approaches, given that they achieve high performance with just 50 labelled instances.

Progressive ER [14] is employed when resources are limited (e.g., cloud budget or human time to refine the ER pipeline), or the data is periodically changing and it has to be consumed within a certain time to be valuable for downstream applications. Indeed, existing progressive ER methods [14,16,41,44] try to quickly sort candidate matches by their matching likelihood (typically estimated through a proxy measure derived from a blocking strategy), so as to discover as many matches as fast as possible. In particular, PPS [16] is the state-of-the-art schema-agnostic progressive ER method; in this paper we adapted it to our proposed weighting strategy and showed the non-negligible benefit of doing so on benchmark datasets.

7. Conclusions

We have presented Generalized Supervised Meta-blocking, which casts Meta-blocking as a probabilistic binary classification task and weights all candidate pairs in a block collection with the probabilities produced by the trained classifier. These weights are processed by a pruning algorithm that can be: (i) weight-based, determining the minimum weight of retained pairs in a way that promotes recall, or (ii) cardinality-based, determining the maximum number of retained pairs in a way that promotes precision.

Through a thorough experimental study over 9 established, real-world datasets, we verified that BLAST and RCNP constitute the best weight- and cardinality-based pruning algorithms, respectively. We also demonstrated that four new weighting schemes give rise to feature sets that outperform the one determined in [13] as optimal. We showed that a very small, balanced training set with just 50 labelled instances suffices for consistently achieving high effectiveness, high time efficiency and high scalability. Finally, we demonstrated that Generalized Supervised Meta-blocking in combination with PPS can be used effectively in Progressive Entity Resolution contexts and that it is still competitive with respect to recently published blocking techniques.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Code and data are available on GitHub as reported in the paper. Link: https://github.com/Gaglia88/sparker.

Acknowledgments

This work was partially supported by the Horizon Europe project STELAR (Grant No. 101070122) and the Department of Engineering "Enzo Ferrari" within the FARD-2022 (FAR2022DIP_DIEF) and FARD-2023 (FAR_DIP_2023_DIEF-SIMONINI - Grant No. E93C23000280005) projects.

References

[1] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, K. Stefanidis, An overview of end-to-end entity resolution for big data, ACM Comput. Surv. 53 (6) (2021) 127:1–127:42.
[2] V. Christophides, V. Efthymiou, K. Stefanidis, Entity Resolution in the Web of Data, Morgan & Claypool, 2015.
[3] X.L. Dong, D. Srivastava, Big Data Integration, Morgan & Claypool Publishers, 2015.
[4] P. Christen, A survey of indexing techniques for scalable record linkage and deduplication, TKDE 24 (9) (2012) 1537–1555.
[5] G. Papadakis, D. Skoutas, E. Thanos, T. Palpanas, Blocking and filtering techniques for entity resolution: A survey, ACM Comput. Surv. 53 (2) (2020) 31:1–31:42.
[6] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, W. Nejdl, A blocking framework for entity resolution in highly heterogeneous information spaces, TKDE 25 (12) (2012) 2665–2682.
[7] G. Papadakis, E. Ioannou, E. Thanos, T. Palpanas, The Four Generations of Entity Resolution, Morgan & Claypool Publishers, 2021.
[8] D. Beneventano, S. Bergamaschi, L. Gagliardelli, G. Simonini, BLAST2: An efficient technique for loose schema information extraction from heterogeneous big data sources, ACM J. Data Inf. Qual. 12 (4) (2020) 18:1–18:22.
[9] G. Simonini, L. Gagliardelli, S. Bergamaschi, H.V. Jagadish, Scaling entity resolution: A loosely schema-aware approach, Inf. Syst. 83 (2019) 145–165, http://dx.doi.org/10.1016/j.is.2019.03.006.
[10] G. Papadakis, G. Alexiou, G. Papastefanatos, G. Koutrika, Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data, PVLDB 9 (4) (2015) 312–323.
[11] G. Papadakis, J. Svirsky, A. Gal, T. Palpanas, Comparative analysis of approximate blocking techniques for entity resolution, PVLDB 9 (9) (2016) 684–695.
[12] G. Papadakis, G. Koutrika, T. Palpanas, W. Nejdl, Meta-blocking: Taking entity resolutionto the next level, TKDE 26 (8) (2014) 1946–1960.
[13] G. Papadakis, G. Papastefanatos, G. Koutrika, Supervised meta-blocking, PVLDB 7 (14) (2014) 1929–1940.
[14] S.E. Whang, D. Marmaros, H. Garcia-Molina, Pay-as-you-go entity resolution, IEEE Trans. Knowl. Data Eng. 25 (5) (2013) 1111–1124, http://dx.doi.org/10.1109/TKDE.2012.43.
[15] G. Simonini, L. Zecchini, S. Bergamaschi, F. Naumann, Entity resolution on-demand, Proc. VLDB Endow. 15 (7) (2022) 1506–1518, URL https://www.vldb.org/pvldb/vol15/p1506-simonini.pdf.
[16] G. Simonini, G. Papadakis, T. Palpanas, S. Bergamaschi, Schema-agnostic progressive entity resolution, TKDE 31 (6) (2019) 1208–1221.
[17] S. Thirumuruganathan, H. Li, N. Tang, M. Ouzzani, Y. Govind, D. Paulsen, G. Fung, A. Doan, Deep learning for blocking in entity matching: a design space exploration, Proc. VLDB Endow. 14 (11) (2021) 2459–2472.
[18] R. Wang, Y. Li, J. Wang, Sudowoodo: Contrastive self-supervised learning for multi-purpose data integration and preparation, in: 2023 IEEE 39th International Conference on Data Engineering (ICDE), 2023.
[19] J.B. Mugeni, T. Amagasa, A graph-based blocking approach for entity matching using contrastively learned embeddings, SIGAPP Appl. Comput. Rev. 22 (4) (2023) 37–46.
[20] D. Paulsen, Y. Govind, A. Doan, Sparkly: A simple yet surprisingly strong TF/IDF blocker for entity matching, Proc. VLDB Endow. 16 (6) (2023) 1507–1519.
[21] L. Gagliardelli, G. Papadakis, G. Simonini, S. Bergamaschi, T. Palpanas, Generalized supervised meta-blocking, Proc. VLDB Endow. 15 (9) (2022) 1902–1910.
[22] G. Papadakis, G. Papastefanatos, T. Palpanas, M. Koubarakis, Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking., in: EDBT, 2016, pp. 221–232.
[23] D.J. Hand, P. Christen, A note on using the F-measure for evaluating record linkage algorithms, Stat. Comput. 28 (3) (2018) 539–547, http://dx.doi.org/10.1007/s11222-017-9746-6.

[24] V. Efthymiou, G. Papadakis, K. Stefanidis, V. Christophides, MinoanER: Schema-agnostic, non-iterative, massively parallel resolution of web entities, in: EDBT, 2019, pp. 373–384.
[25] N. Augsten, R. Kwitt, M. Lissandrini, W. Mann, T. Palpanas, G. Papadakis, New Weighting Schemes for Meta-blocking, Tech. Rep. LIPADE-TR 5, Laboratoire d'Informatique PAris DEscartes (LIPADE), 2021, Available at http://lipade.mi.parisdescartes.fr/wp-content/uploads/2021/10/LipadeTR-5.pdf.
[26] L. Gagliardelli, G. Simonini, D. Beneventano, S. Bergamaschi, SparkER: Scaling entity resolution in spark, in: EDBT, 2019, pp. 602–605.
[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[28] H. Köpcke, A. Thor, E. Rahm, Evaluation of entity resolution approaches on real-world match problems, PVLDB 3 (1–2) (2010) 484–493.
[29] D. Obraczka, J. Schuchart, E. Rahm, EAGER: Embedding-assisted entity resolution for knowledge graphs, 2021, arXiv preprint arXiv:2101.06126.
[30] S. Das, A. Doan, P.S. G. C., C. Gokhale, P. Konda, Y. Govind, D. Paulsen, The Magellan data repository, https://sites.google.com/site/anhaidgroup/projects/data.
[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.
[32] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, T. Mikolov, FastText.zip: Compressing text classification models, 2016, arXiv preprint arXiv:1612.03651.
[33] V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, 2019, CoRR, abs/1910.01108.
[34] T. Gao, X. Yao, D. Chen, SimCSE: Simple contrastive learning of sentence embeddings, in: EMNLP, 2021, pp. 6894–6910.
[35] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: NAACL-HLT, 2019, pp. 4171–4186.
[36] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech.: Theory Exp. 2008 (10) (2008).
[37] H. Schütze, C.D. Manning, P. Raghavan, Introduction to Information Retrieval, Vol. 39, Cambridge University Press Cambridge, 2008.
[38] G. Simonini, S. Bergamaschi, H.V. Jagadish, BLAST: a loosely schema-aware meta-blocking approach for entity resolution, PVLDB 9 (12) (2016) 1173–1184.
[39] F. Zhang, Z. Gao, K. Niu, A pruning algorithm for meta-blocking based on cumulative weight, in: Journal of Physics, Vol. 887, 2017.
[40] D.C. do Nascimento, C.E.S. Pires, D.G. Mestre, Exploiting block co-occurrence to control block sizes for entity resolution, Knowl. Inf. Syst. 62 (1) (2020) 359–400.
[41] S. Galhotra, D. Firmani, B. Saha, D. Srivastava, Efficient and effective ER with progressive blocking, VLDB J. 30 (4) (2021) 537–557.
[42] S. Galhotra, D. Firmani, B. Saha, D. Srivastava, BEER: Blocking for effective entity resolution, in: SIGMOD, 2021, pp. 2711–2715.
[43] G.D. Bianco, M.A. Gonçalves, D. Duarte, BLOSS: Effective meta-blocking with almost no effort, Inf. Syst. 75 (2018) 75–89.
[44] T. Papenbrock, A. Heise, F. Naumann, Progressive duplicate detection, IEEE Trans. Knowl. Data Eng. 27 (5) (2015) 1316–1329, http://dx.doi.org/10.1109/TKDE.2014.2359666.