An Unsupervised Algorithm for Learning Blocking Schemes

Mayank Kejriwal
University of Texas at Austin
kejriwal@cs.utexas.edu

Daniel P. Miranker
University of Texas at Austin
miranker@cs.utexas.edu

***Abstract*—A pairwise comparison of data objects is a requisite step in many data mining applications, but has quadratic complexity. In applications such as record linkage, blocking methods may be applied to reduce the cost. That is, the data is first partitioned into a set of blocks, and pairwise comparisons computed for pairs within each block. To date, blocking methods have required the blocking scheme be given, or the provision of training data enabling supervised learning algorithms to determine a blocking scheme. In either case, a domain expert is required. This paper develops an unsupervised method for learning a blocking scheme for tabular data sets. The method is divided into two phases. First, a weakly labeled training set is generated automatically in time linear in the number of records of the entire dataset. The second phase casts blocking key discovery as a Fisher feature selection problem. The approach is compared to a state-of-the-art supervised blocking key discovery algorithm on three real-world databases and achieves favorable results.**

***Index Terms*—Blocking, Record Linkage**

I. Introduction

Record Linkage, or the identification of entities within a database that are coreferent, is a long standing problem with no less than eight separate terms referring to the same problem [1]. Wikipedia [1] lists at least fifteen different names, and despite much research the problem does not have an automated solution. Ad hoc and domain dependent solutions are still common, with human intervention required.

Record Linkage typically requires two primary steps [1]. The first step is referred to as *blocking*. Blocking methods mitigate full pairwise comparisons by selecting a small subset of pairs from the database that are considered to be good candidates for pairwise comparison, while discarding the vast majority of pairs that are clearly non-coreferent. Without blocking, each entity must be compared with every other entity to determine whether the two corefer. This naive approach grows quadratically with the input, and is impractical for large databases; hence, the need for blocking. The blocking step is comprehensively surveyed by Christen [2].

The pairs generated by blocking are then used as input for a second step, which typically involves machine-learning techniques, among others, to isolate duplicates according to some similarity measure. The second step is comprehensively surveyed by Elmagarmid et al. [1].

The blocking phase of this two-step procedure has thus far required a human in the loop. This is because blocking methods require a blocking *scheme*, a function assumed to be provided by a domain expert [2]. Although multiple methods have investigated the *use* of a given scheme in a variety of ways [2], there has been negligible research on learning the scheme itself. Two papers sought to address this gap by learning schemes given training data [3], [4]. However, labeling duplicates in large databases is troublesome, particularly if duplicates are sparse or the data is confidential. As the size and diversity of datasets continues to grow in the current era of Big Data, the need for an automated procedure is pressing. In this paper, an unsupervised method is presented for learning blocking schemes. The algorithm runs in two separate phases. In the first phase, the algorithm efficiently generates a weakly labeled training set. In the second phase, the problem of learning blocking schemes from this weakly labeled set is cast as a *feature selection* problem. The validity of both phases of the algorithm is demonstrated on three real-world datasets.

The outline of this paper is as follows. Section II lays out the formalism of blocking schemes and describes the blocking step in detail. Section III proposes an algorithm to generate a weakly labeled training set, and presents a worst-case analysis. Section IV describes the feature selection procedures that take the weakly labeled set as input and output a blocking scheme. Section V describes the experiments conducted, including methodology and datasets. Section VI presents the results and a discussion. Section VII presents related work. Finally, Section VIII details future work and concludes the paper.

II. Blocking Schemes

The formalism for the rest of the paper is introduced, along with illustrative examples. For consistency, terminology proposed by Bilenko et al. is used [4], although many terms below were not formally defined in that work.

A. Definitions and Examples

The most basic elements of a blocking scheme are *indexing functions* $h_i(x_t)$ [4]. An indexing function accepts a field value from a tuple as input and returns a set Y that contains 0 or more *blocking key values* (BKVs). A BKV identifies a block in which the tuple is placed. Intuitively, one may think of a block as a hash bucket, but more often the function is used to sort the records [5]. If the set Y contains multiple BKVs, then the tuple is assigned to multiple blocks.

[1]https://en.wikipedia.org/wiki/Record_linkage

TABLE I
TWO SAMPLE TUPLES FROM THE RESTAURANT DATASET

Name	Address	City	Cuisine
arnie morton's of chicago	435 s. la cienega blv.	los angeles	american
campanile	624 s. la brea ave.	los angeles	american

Definition 1. An *indexing function* $h_i : Dom(h_i) \rightarrow U^*$ takes as input a field value x_t from some tuple t and returns a set Y that may contain 0 or more *Blocking Key Values* (BKVs) from the set of all possible BKVs U^*.

For clarity, the domain of an indexing function is always a string, although technically it can be any primitive data type. The range is a set of Blocking Key Values that the tuple is assigned to. Each BKV is represented by a string identifier. Note that all tuples are converted to lower-case as a first preprocessing step.

Example 1. An example of an indexing function is *Tokens*. *Tokens* takes a field value of a tuple and parses it into a set of tokens using a set of common delimiters (such as whitespace and comma). This set is then returned as the BKV set that identifies the blocks in which the tuple is placed. For example, consider the first sample tuple in Table I taken from the *Restaurant* dataset, which has four fields. If *Tokens* is applied to the *name* field, the set {arnie, morton's, of, chicago} is obtained.

This leads to the notion of a *general blocking predicate*. Intuitively, a general blocking predicate $p_i(x_{t_1}, x_{t_2})$ takes as input field values from two different tuples, t_1 and t_2, and uses the i^{th} indexing function to obtain sets of keys Y_1 and Y_2 for the two arguments. The predicate is satisfied if and only if the two sets intersect i. e. if both tuples share at least one common block.

Definition 2. A *general blocking predicate* $p_i : Dom(h_i) \times Dom(h_i) \rightarrow \{True, False\}$ takes as input field values x_{t_1} and x_{t_2} from two different tuples, t_1 and t_2, and returns *True* if $h_i(x_{t_1}) \cap h_i(x_{t_2}) \neq \Phi$, and returns *False* otherwise.

Note that each general blocking predicate is always associated with an indexing function. Also note the distinction between a *field* and a *field value* (an instantiation of a field). For example, in Table I *cuisine* would be a field, but *american* would be a field value.

Example 2. Consider the general blocking predicate *Contains Common Token*, associated with the indexing function *Tokens* in Example 1. Suppose this predicate is applied to the *address* field values of the two tuples shown in Table I. The predicate first applies *Tokens* to the *address* field values of both these tuples and then intersects their resulting BKV sets. Since the intersection yields two common tokens {s., la}, the two tuples will share at least two common blocks. Hence, *Contains Common Token* will return *True* for this pair, when applied to *address*.

Such predicates are called *general* because their indexing functions are not associated, a priori, with specific fields. For example, *Contains Common Token* above could have been applied just as easily to some other field (like *name*) than to *address*.
On the other hand, a *specific blocking predicate* explicitly pairs a general blocking predicate to a specific field.

Definition 3. A *specific blocking predicate* is a pair (p_i, f) where p_i is a general blocking predicate and f is a field. A specific blocking predicate takes two tuples t_1 and t_2 as arguments and applies p_i to the appropriate field values f_1 and f_2 from both tuples. Furthermore, a tuple pair for which the specific blocking predicate is *True* is said to be *covered* by that specific blocking predicate.

Example 3. In the *Tokens* example, *(Contains Common Token, address)* and *(Contains Common Token, name)* would both be specific blocking predicates.

In this paper, it is assumed that all available general blocking predicates can be applied to all fields of the relation. Hence, given a relation R with m fields, and s general blocking predicates, the number of specific blocking predicates will be exactly ms.
A *blocking scheme* is defined as:

Definition 4. A *DNF blocking scheme* f_P is a function constructed in Disjunctive Normal Form (disjunction of terms), using a given set P of specific blocking predicates as literals, and with the constraint that negated literals may not be used in the construction. Furthermore, if the additional constraint is employed that each term comprise at most 1 literal, the blocking scheme is referred to as a *Disjunctive Blocking Scheme*.

Note that a Disjunctive Blocking Scheme is merely a disjunction of specific blocking predicates while a general DNF Blocking Scheme is a disjunction of terms, where each term is a conjunction of specific blocking predicates. A blocking scheme may be applied to a tuple pair, which is said to be *covered* if the scheme returns *True* for the pair, just like with specific blocking predicates.

B. The Blocking Step

Given a blocking scheme f_P on a relation R, the *blocking step* of the record linkage process takes each tuple t of R and applies the indexing functions corresponding with the literals (that is, the specific blocking predicates) in f_P to t. For a term C of specific blocking predicates in f_P, a cross product of the BKV sets of the individual literals is computed. Each element of this cross product set now references a block to which the tuple is assigned.

Example 4. Consider the (rather simple) blocking scheme *(Contains Common Token, address)* $\wedge$ *(Exact Match, city)* . Given this scheme, the BKV sets returned (by applying the indexing functions) for the first tuple are $\{$435, s., la, cienaga, blv.$\}$ and $\{$'los angeles'$\}$ while for the second tuple the sets $\{$624, s., la, brea, ave.$\}$ and $\{$'los angeles'$\}$ are returned. The cross product set of the first tuple is thus $\{$435 'los angeles', s. 'los angeles', la 'los angeles', cienaga 'los angeles', blv. 'los angeles'$\}$ and of the second tuple is $\{$624 'los angeles', s. 'los angeles', la 'los angeles', brea 'los angeles', ave. 'los angeles'$\}$. Whitespace is used as a delimiter within each cross product element. The first tuple is assigned to 5 blocks (the cardinality of its cross product set); similarly for the second tuple. Because the values *s. 'los angeles'* and *la 'los angeles'* are common to both sets, the tuples will share exactly two common blocks.

Because the blocking scheme is framed in DNF, each term can be thought of as a single *pass* over the relation R. A disjunction of terms in the scheme can be used to simulate the multi-pass approach of Hernandez and Stolfo [5]. Essentially, a DNF blocking scheme $C_1 \vee C_2 \ldots \vee C_k$ with k terms is the same as k blocking schemes applied to the relation R, each in its own independent pass. The union of the k total sets of blocks produced in the k passes results in a single set of blocks Π.

Within each block in Π, all tuples are paired with each other to produce a set of pairs for that block. The union of all such sets of pairs is then used to generate the final *candidate set of pairs*, as given by the formula:

$$\Gamma = \bigcup_{B \in \Pi} \{\{t_i, t_j\}\}, \forall \{t_i, t_j\} \subseteq B | i \neq j \tag{1}$$

where Γ is the candidate set of pairs, B refers to a block in Π and t_i, t_j are tuples within that block. The constraint $i \neq j$ is imposed to prevent a tuple pairing with itself. As the candidate set is formed by a union, a pair can only occur once, even though it could potentially have been generated more than once across multiple blocks. Moreover, since each pair is modeled in the above equation as a set of cardinality 2, the order of the pair elements do not matter. Note that a block with fewer than two tuples is discarded, since no pairs can be generated for that block.

This final candidate set of pairs becomes the input to the second step of the record linkage process, which essentially involves classifying each pair as a match, non-match or possible match [1]. If the blocking scheme was 'good', this set would satisfy some desirable properties. This notion is explored further in the next section.

C. Metrics

Not all blocking schemes are equally good or useful. As an example, consider a simple blocking scheme *(Exact Match, Gender)* on a personnel database. Assume that the domain of *Gender* is the set $\{'M', 'F'\}$ and that half of the personnel are men and half are women. This blocking scheme would produce two blocks with half the tuples in one block and half in the other. Obviously, such a scheme would not be of much help in reducing the cardinality of the final candidate set by more than a small constant factor. On the other hand, a blocking scheme on the last four digits of the Social Security Numbers of personnel would result in a huge reduction in the candidate set, but might accidentally exclude true duplicates from the set if typographical or other errors are present in those digits. Thus, there is a tradeoff between ensuring that true positives are included in the candidate set and cutting down the size of the candidate set. Two metrics, *Reduction Ratio* and *Pairs Completeness* are employed to express these goals [2], [6].

The *Reduction Ratio* (RR) quantifies the extent to which the blocking scheme minimizes the number of candidate pairs. It can be expressed by the formula:

$$RR = 1 - \frac{|\Gamma|}{|\Omega|} \tag{2}$$

where Γ is the final candidate set as given by (1) and Ω is the set of candidate pairs that would be generated in the absence of blocking. As an example, for a single relation with n tuples, the number of candidate pairs in the absence of blocking would be $n(n-1)/2$, since a tuple is not allowed to pair with itself. An RR close to 1.0 indicates that few candidate pairs have been generated, while an RR close to 0.0 indicates that the reduction achieved by the blocking scheme was small.

Pairs Completeness (PC) is the ratio of the true positives or duplicates in the candidate set to those in the full set of pairs Ω. PC can be expressed using the formula:

$$PC = \frac{|\Gamma_m|}{|\Omega_m|} \tag{3}$$

where Γ_m is the set of true positives in the candidate set generated by the blocking step and Ω_m is the set of true positives in the entire dataset (equivalently, the number of true positives in the full set of pairs Ω). The PC metric, therefore, captures the notion of *recall* in the framework of blocking.

Ideally, PC should be exactly 1.0, indicating perfect recall, while RR should be close to 1.0 (but not exactly 1.0 unless the dataset contains no duplicates at all), indicating the blocking scheme did a good job of reducing the candidate set. There is a clear tradeoff in achieving both goals.

In order to succinctly express this trade-off, the harmonic mean of the two quantities, called the *F-Measure* or *FM*, is computed. It can be expressed by the formula:

$$FM = \frac{2 * RR * PC}{RR + PC} \tag{4}$$

Another metric *Pairs Quality PQ* is similar to PC except that it computes the *percentage* of true positives in the candidate set of pairs $\frac{|\Gamma_m|}{|\Gamma|}$. Although this metric has been used to evaluate different blocking *methods*, given the blocking scheme, it has been unemployed by researchers seeking to *learn* blocking schemes [3], [4].

D. Learning Blocking Schemes

1) Framing As An Optimization Problem: Learning a blocking schemes can be stated as an optimization problem, *given* a training set of positive and negative examples. Bilenko et al. [4] stated this objective as:

$$f_P^* = argmin_{f_P} \sum_{(t_i,t_j)\in N} f_P(t_i,t_j) \tag{5}$$

s.t.

$$|D| - \sum_{(t_i,t_j)\in D} f_P(t_i,t_j) < \epsilon \tag{6}$$

where N is the set of negative examples (non-duplicate pairs), D is the set of positive examples (duplicate pairs), t_i, t_j are tuples, f_P is a DNF Blocking Scheme, and ϵ is a parameter that allows up to ϵ pairs to be uncovered by the learned scheme. Intuitively, an 'optimal' blocking scheme f_P^* should maximize the duplicate training set covered, while minimizing the non-duplicates covered, thereby achieving the tradeoff mentioned earlier. The parameter ϵ is a practical means of loosening the rigid constraint that *all* duplicate pairs be covered.

2) Baseline: Two independent works developed blocking schemes given a set of manually labeled examples [3], [4]. Both works employed a greedy approach. Michelson and Knoblock [3] employed a variant of a sequential set covering algorithm, *SequentialCovering*, to learn a DNF Blocking Scheme. Bilenko et al. [4] framed the optimization problem in (5) and (6) as a variant of the *Red-Blue Set Cover* problem [7].

Despite being shown to be NP-hard [7], several approximation algorithms have been proposed for this problem [7], [8]. Bilenko et al. [4] adopted the algorithm by Chvatal [9] and titled it *ApproxRBSetCover* [4]. Blocking Schemes are learned by *ApproxRBSetCover* with inputs comprising a given set of t specific blocking predicates, a training set of duplicates D and non duplicates N, and parameters ϵ and η, with η the maximum number of negative examples a predicate is allowed to cover. Full pseudocode and details can be found in the baseline paper [4].

Like *ApproxRBSetCover*, the algorithms in this paper handle the DNF case separately from the more restrictive Disjunctive case, and accept similar parameters. To enable fair comparison, therefore, *ApproxRBSetCover* was chosen as the baseline.

III. Generating Weakly Labeled Training Set

A. Algorithm

1) Pseudocode and Details: To the best of our knowledge, current proposals for learning blocking schemes mandate a manually labeled set. For several reasons, generating even a small, perfectly labeled set is non-trivial. First, many databases are sparse in duplicate pairs. For example, on the three benchmark datasets used in this paper, fewer than 1.5 percent of the full set of pairs were duplicates. Secondly, a domain expert is required to have access to the database and carefully locate duplicates.

To address this problem, a scheme for generating a *weakly* labeled training set is proposed. The pseudocode of the algorithm is shown in Algorithm 1. The user may specify

Algorithm 1 WeakTrainingSet(R,ut,lt,c,d,nd)

```
Input :
  • Relation $R$
  • Upper Threshold $ut$
  • Lower Threshold $lt$
  • Blocking Window Size $c$
  • Maximum Duplicate Pairs Requested $d$
  • Maximum Non-Duplicate Pairs requested $nd$
Output :
  • A set of positive examples $P$
  • A set of negative examples $N$
Method :
  1) Initialize set $P$ of duplicates to be empty
  2) Initialize set $N$ of non-duplicates to be empty
  3) Initialize set of tuple pairs $C$ to be empty
  4) Generate TFIDF statistics of R
  5) for all fields $f$ in the schema of $R$ do
       for all tuples $t \in R$ do
         Tokenize $t$'s field value in $f$
         Block $t$ on generated tokens
       end for
  6) end for
  7) for all blocks $B$ generated in previous step do
       Slide a window of size $c$ over tuples in $B$
       Generate all possible pairs within window and add
       to $C$
  8) end for
  9) for all pairs $(t_1, t_2) \in C$ do
       Compute TFIDF similarity $sim$ of $(t_1, t_2)$
       if $sim \geq ut$ then
         if $|P| < d$ then
           add $(t_1, t_2)$ to $P$
           continue
         end if
         if $sim >$ lowest $sim$ in $P$ then
           Replace pair with lowest $sim$ in $P$ with $(t_1, t_2)$
         end if
       end if
       if $sim < lt$ then
         if $|N| < nd$ then
           add $(t_1, t_2)$ to $N$
           continue
         end if
         if $sim >$ lowest $sim$ in $N$ then
           Replace pair with lowest $sim$ in $N$ with $(t_1, t_2)$
         end if
       end if
 10) end for
 11) Return $P$ and $N$
```

a maximum of d duplicates and nd non duplicates to be returned. By default, these are simply set to the maximum possible number of pairs. Three additional parameters, the upper threshold, ut, lower threshold, lt, and window size c may be optionally specified with $0.0 < lt \leq ut \leq 1.0$ and $c > 1$. Intuitively, ut and lt control the strictness of the algorithm in generating the weakly labeled set. The role of c is detailed subsequently.

The default values of ut, lt and c are $0.05, 0.01$ and 20 respectively. These values were found after some preliminary experimentation on one of the three benchmarks, and did not require any consequent tuning (see the *Experiments* section). Moreover, as demonstrated in *Results and Discussion*, these values are stable. That is, variation of these parameters over a non-trivial range did not significantly degrade performance on any of the three datasets, all of which have differing characteristics.

The key step of the algorithm comprises a simple Disjunctive Blocking Scheme *(Contains Common Token, f_1)* $\vee$ *(Contains Common Token, f_2)* $\ldots \vee$ *(Contains Common Token, f_m)* where the relation R is assumed to have m fields $f_1, f_2 ... f_m$. Thus, the blocking scheme merely makes m passes over R, once for each field, and tokenizes the corresponding field value of each tuple in each pass. The tuple is then placed in (possibly multiple) blocks, each indexed by a single token. Note that a token is local to its field; the same token generated from another field will reference a separate block.

After all the blocks have been generated, for all tuples and fields, a sliding window passes over c tuples at a time and all pairs within this window are generated. The similarity measure (sim in the pseudocode) of each pair (t_1, t_2) is then computed using the log TFIDF measure [10]:

$$sim(t_1, t_2) = \sum_{q \in t_1 \cap t_2} w(t_1, q).w(t_2, q) \tag{7}$$

where

$$w'(t, q) = log(tf_{t,q} + 1).log(\frac{|R|}{df_q} + 1) \tag{8}$$

and

$$w(t, q) = \frac{w'(t, q)}{\sqrt{\sum_{q \in t} w'(t, q)^2}} \tag{9}$$

where $w(t, q)$ is the normalized TFIDF weight of a term q in a tuple t, $tf_{t,q}$ is the term frequency of q in t, $|R|$ is the total number of tuples in the relation R and df_q is the number of tuples in which the term q appears.

Hence, given a pair of tuples, sim is computed, and its value is checked against ut and lt. If sim falls strictly between ut and lt, it is considered ambiguous and ignored. Otherwise, if $sim \geq ut$, the algorithm checks its current list, D, of the *best* (i. e. with highest sim values greater than ut) duplicates found so far. If D contains fewer than d elements, the current pair gets added to the list. Otherwise, the current pair replaces the lowest ranking pair in D if the current pair's sim value is higher. An analogous procedure ensues if $sim < lt$. However, the algorithm picks the nd $worst$ non-duplicates below lt, where one non-duplicate pair is worse than another if it has higher sim value. The rationale for this choice is that trivially best non-duplicates ($sim \approx 0.0$) usually do not have good *discriminative* power. For example, consider the phone number field of a personnel database of a company that has offices only in Illinois and Texas. The first three digits of the phone number for all employees will always be '512' or '217' and irrelevant for discriminating between duplicates and non-duplicates. This irrelevance should be reflected in the set of non-duplicates but would not be if only the *best* non-duplicates ($sim \approx 0.0$) had been selected.

2) Complexity Analysis: Algorithm 1 is shown to run in worst case linear time (in the number of records n) theoretically. In practice, however, multiple optimizations make it much faster.

Consider the pseudocode of Algorithm 1. First, TFIDF statistics must be collected for each tuple, which requires a single pass over R and hence, takes $O(n)$ time (line 4). Next, for each field, R must be blocked, which takes total of $O(mn)$ time where m is the number of fields in R (lines 5-6). This step presents an opportunity for parallelism: since each field is processed independent of the others, up to m processors can be used to speed up the process further, with 'broad' databases, (m at least in the tens) witnessing significant savings.

Once the blocks are generated, a *window* of size c is slid over the block and all pairs within this window generated (lines 7-8). Specifically, for a block with b tuples, the total number of pairs generated is $b(b-1)/2$ iff $b \leq c$ and $(b-c+1)(c-1) + (c-2)(c-1)/2 = O(b)$ otherwise. Note that the former achieves its maximum (and the latter its minimum) at $b = c$. Since c is fixed, the pair generation step can take at most $\sum_b O(b) = O(n)$ time/field, with total time $O(mn)$ over all fields. Given the TFIDF statistics collected, the complexity of computing sim between generated pairs is $O(1)$ time/pair, with a mere lookup required when the pair is first generated, so that lines 7-8 and lines 9-10 are essentially parallelized.

Adding the above, a worst case complexity of $O(n + mn + mn) = O((2m + 1)n)$ is obtained. Note that the sliding window c scheme was crucial for obtaining this guarantee. If instead, *all* pairs within all blocks were generated, a worst case complexity of $O(n^2)$ would have been obtained, if a single large block happened to span the entire dataset ($b \approx n$ for some block). c, therefore, mitigates the adverse effects of large blocks at a price of potentially missing some pairs. In the *Experiments* section, c is varied and we show that small values ($c = 20$) suffice in practice.

IV. Feature Selection

Given a weakly labeled set, *feature vectors* are now extracted by applying all the given specific blocking predicates on each duplicate (or non-duplicate) and storing the boolean result of each predicate as an element in a *feature vector*. Thus, a set of boolean feature vectors is obtained for the duplicate

Algorithm 2 FisherScore(P_f,N_f,i)

```
Input :
  • Set of positively labeled feature vectors $P_f$
  • Set of negatively labeled feature vectors $N_f$
  • Conjunction of one or more feature indices $i$
Output :
  • Fisher Score $\rho_i$ of feature $i$ with respect to $P_f$ and $N_f$
Method :
  1) Compute mean $\mu_{p,i}$ of feature $i$ in $P_f$
  2) Compute variance $\sigma_{p,i}$ of feature $i$ in $P_f$
  3) Compute mean $\mu_{n,i}$ of feature $i$ in $N_f$
  4) Compute variance $\sigma_{n,i}$ of feature $i$ in $N_f$
  5) Compute mean $\mu_i$ of feature $i$ in $P_f \cup N_f$
  6) Compute Fisher Score of $\rho_i$ of $i$
  7) where $\rho_i := \frac{|P_f|(\mu_{p,i}-\mu_i)^2+|N_f|(\mu_{n,i}-\mu_i)^2}{|P_f|\sigma^2_{p,i}+|N_f|\sigma^2_{n,i}}$
  8) Return $\rho_i$ computed above
```

set, and another set for the non-duplicate set.

The Fisher discrimination criterion [11] is then used to determine the best *features* using Algorithm 2. The pseudocode of the algorithm, *FisherDisjunctive*, is given in Algorithm 3. Fisher scores are used in the algorithm to impose an *ordering* in which *eligible* feature elements are considered. A feature is *ineligible* if it covers more than η negative examples. In order of descending scores of eligible features, each feature's *contribution* is evaluated. If the feature covers positive examples that the current disjunction does not cover, it is added to the current disjunction. This step of adding features continues till at most ϵ positive examples remain uncovered. When the while loop in line 11 finally terminates, the algorithm is guaranteed to return a disjunctive scheme satisfying this constraint, since otherwise, line 5 would have returned an error.

For learning DNF blocking schemes, a similar approach to the baseline is adopted. Specifically, instead of designing a completely new algorithm, the original feature vectors are supplemented with *new* features, and the expanded feature vectors input to *FisherDisjunctive*. Each new feature corresponds to a *term* or conjunction of the original features. The disjunction returned by *FisherDisjunctive* then is not necessarily a single clause but a disjunction of terms, corresponding to a DNF blocking scheme as defined earlier.

Naively supplementing the feature set with all possible conjunctions of arbitrary length would lead to an exponential blowup. Bilenko et al. [4] addressed this by taking an extra input parameter k and greedily composing terms with at most k literals. Algorithm 4 also requires a parameter k but adopts a slightly different (although still greedy) heuristic, to compose terms with (at most) k literals. The output, a set of supplemented vectors, may then be used to learn a DNF blocking scheme.

V. Experiments

This section details the methodology of the experiments and the benchmarks used.

Algorithm 3 FisherDisjunctive(P_f,N_f,ϵ,η)

```
Input :
  • Set of positively labeled feature vectors $P_f$
  • Set of negatively labeled feature vectors $N_f$
  • Maximum positive vectors that may be left uncovered $\epsilon$
  • Maximum negative vectors any feature is allowed to cover $\eta$
Output :
  • Disjunctive Blocking Scheme $f_P^{Disj.}$
Method :
  1) Initialize set $f_P^{Disj.}$ to be empty
  2) Initialize list of valid features $K$ to be empty
  3) for all features $i$ do
       if feature $i$ covers fewer than $\eta$ examples in $N_f$ then
         Add $i$ to $K$
       end if
  4) end for
  5) if Disjunction of all features in $K$ covers fewer than $\epsilon$ vectors in $P_f$ then
       Terminate with message Cannot find blocking scheme under specified parameters
  6) end if
  7) for all $i$ in $K$ do
       Call FisherScore($P_f$,$N_f$,$i$)
  8) end for
  9) Sort features in $K$ in descending order of their scores as computed above
 10) while more than $\epsilon$ vectors in $P_f$ uncovered do
       Let $i$ be feature in $K$ with highest Fisher Score
       Remove $i$ from $K$
       if $i$ covers at least 1 new vector in $P_f$ compared to $f_P^{Disj.}$ then
         Add $i$ to $f_P^{Disj.}$
       end if
 11) end while
 12) Return disjunction of elements in $f_P^{Disj.}$
```

A. Benchmarks

Three benchmark datasets were used to validate the algorithms proposed in this paper. All three sets are commonly employed in the record linkage and blocking key literature [2], and can be found in the *SecondString*[1] toolkit. The first dataset, *Restaurant*, comprises tuples from the Fodor and Zagat restaurant guides. The second dataset, *Cora* contains bibliographic citations of machine learning publications extracted from the 'Cora' search engine. Finally, *Census* contains records generated by the US Census Bureau based on real census data. Details about the datasets are provided in Table II. Note that *Restaurant* and *Cora* fall under the 'Deduplication' category because they only contain *one* relation each. *Census* falls under the 'Linkage' category because it contains two relations, albeit with the same schema (with 449 and 392 tuples each; see Table II). The linkage task involves finding duplicate pairs with one

[1]http://secondstring.sourceforge.net/

Algorithm 4 Terms(P_f, N_f, k)

```
Input :
  • Set of positively labeled feature vectors $P_f$
  • Set of negatively labeled feature vectors $N_f$
  • Maximum predicates in a term $k$
Output :
  • Set of positively labeled feature vectors $P'_f$
  • Set of negatively labeled feature vectors $N'_f$
Method :
  1) Initialize $count := 0$
  2) Initialize forbidden feature set $forbidden = \phi$
  3) Initialize $P'_f$ and $N'_f$ to be $P_f$ and $N_f$ respectively
  4) while $count < k$ do
       for all features $i$ not in $forbidden$ do
         Compute Fisher Scores for $i$ by calling FisherScore($P'_f$,$N'_f$,$i$)
       end for
       Compute average score $avg$ by averaging all feature scores in the previous step
       for all features $i \notin forbidden$ in decreasing order of scores do
         for all original features $j$ such that $j \neq i$ do
           Form new feature $l$ that is conjunction of $j$ and $i$
           Evaluate score of $l$ by calling FisherScore($P'_f$,$N'_f$,$l$)
           if score of $l$ is not less than $avg$ then
             Add $l$ as new feature to each vector in $P'_f$ and $N'_f$
           end if
         end for
         Add $i$ to $forbidden$
       end for
       Increment $count$
  5) end while
  6) Return $P'_f$ and $N'_f$
```

TABLE II
BENCHMARK DATASETS USED IN EXPERIMENTS

Dataset Name	Task	Tuples	True duplicate pairs
Restaurant	Deduplication	864	112
Cora	Deduplication	1295	17184
Census	Linkage	449+392	327

tuple coming from the first relation, and one from the second relation. The procedures and algorithms described earlier work identically but with the explicit constraint that no generated pair should contain both tuples from the same relation.

B. Methodology

1) WeakTrainingSet: To evaluate *WeakTrainingSet*, the algorithm was run on all three datasets. In a first set of experiments, the maximum number of required duplicates d and non-duplicates nd were varied and the resulting precisions against true duplicates and non-duplicates reported. The other parameters ut, lt, c were set at default values of 0.05, 0.01 and 20 respectively. These were determined by some initial tuning on the Restaurant dataset, and used, unchanged, for the other sets.
However, in practice, such parameter tuning may not be feasible. Therefore, a set of experiments was conducted to show the robustness of this algorithm to reasonable changes in these parameter values. ut, lt and c were varied individually across a non-trivial range and the resulting duplicate and non-duplicate precisions reported.

2) Feature Selection: Supervised *ApproxRBSetCover* was adopted as the baseline. The weakly labeled duplicates and non-duplicates from *WeakTrainingSet* are used as input to the feature selection algorithms. Since the baseline is supervised, an equal number of manually labeled duplicates and non-duplicates are provided to the baseline algorithms. However, while *WeakTrainingSet* can generate both false positives and negatives, the supervised baseline is given a training set that is perfectly labeled. For Restaurant and Census, 56 and 163 duplicates were input (50 percent of the true positives). For Cora, a different approach was required because under the given parameters, *WeakTrainingSet* would only return a maximum of 3063 duplicates for the default value of the upper threshold parameter, $ut = 0.05$. This is just 18 percent of the training set but is still fairly large in the absolute number of duplicates. The number of non-duplicates input was fixed at 1000 for all three datasets. A larger number of non-duplicates offered little extra discriminative power and, experimentally, showed no improvement for either algorithm.
This leads to the choice of η and ϵ for both baseline and feature selection. For convenience, ϵ and η are henceforth expressed as percentages of the duplicate and non-duplicate training set respectively, to ensure uniform comparisons across all three benchmarks.
Cast as percentage, ϵ is varied from 0.0 to 1.0 in increments of 0.1 while four values of $\eta = 0.0, 0.1, 0.5, 1.0$ are considered. Average and best achieved results across these 44 parameter settings are reported on all three datasets for FM, together with corresponding RR and PC numbers. Note that the average result indicates the *expected* value of the metrics over the parameter space. We chose to report this result because often, in practice, it will not be possible to determine the best values of η and ϵ. Moreover, only one run may be allowed if the dataset is too large. Such an expected value, therefore, has relevance in this problem context.
Separate results are shown for learning Disjunctive and DNF Blocking Schemes. For the baseline, training data was randomly sampled; all results are reported as averages of ten independent runs. For learning DNF Blocking Schemes, the parameter k was set to 2 for both baseline and *Terms*. Increasing k beyond 2 showed no experimental improvements for either method.
We use the same set of 25 general blocking predicates as the baseline to enable fair comparison. These predicates are fairly intuitive and include blocking on exact match of field values,

token based match, and blocking on common integer values, among others. Details about these predicates are provided by Bilenko et al. [4]. Recall that for a given relation R with m fields and s general blocking predicates, the number of *specific* blocking predicates was ms with $s = 25$ here.

A final point is that none of the results were amenable to statistical significance tests. The only non-deterministic component in the procedures described was the random sampling (over ten runs) of the labeled set on which the baseline was trained. An analysis showed that the same blocking scheme was discovered over the ten runs for a given set of parameters. The coarse granularity of blocking schemes compensated for this random sampling, therefore, making the baseline experimentally deterministic.

VI. Results and Discussion

A. *WeakTrainingSet*

Figure 1 shows the precision of duplicates obtained on Restaurant and Census, against the number of duplicates retrieved by the algorithm. Because the number of duplicates in Cora is an order of magnitude larger than those in Restaurant and Census, it is shown separately in Figure 2. For no dataset does the precision dip below 90 percent, even as the number of duplicates requested/retrieved keeps increasing.

The precisions of non-duplicates retrieved for Cora are shown in Figure 3. For Census and Restaurant, perfect precision (100 percent) was achieved over a range of 1000 to 20000 non-duplicates required/retrieved. The figures for these sets are therefore omitted for lack of space.

In the next experiment, the upper threshold parameter ut was varied from 0.0 to 0.1 in increments of 0.005. Setting ut above 0.1 or below 0.04 is not typically recommended, otherwise too few duplicates or too many false duplicates will be retrieved respectively. However, the algorithm performs quite well over this range, with accuracy almost always above 90 percent with a non-trivial number of duplicates requested; see Figure 4. Note that for Restaurant, the total number of true duplicate pairs is 112, although 200 are requested. Even at low levels of ut, the algorithm did not return more than 100 pairs. The algorithm, hence, is robust to requests that overestimate.

The same experiment is run again with varying lt and the precision of non-duplicates is reported. The results are shown in Figure 5. Again, the algorithm performs well as lt is varied.

A final set of experiments were run varying the window parameter c from 20 to 50 in increments of 10. The same experiments described above were run. There was no difference at all in any of the results. Hence, setting c to a low constant such as 20 is justified.

B. Feature Selection

The results for the best and average F-Measure, RR and PC for learning both Disjunctive and DNF Blocking Schemes are shown in Tables III and IV. Usually, Fisher Discrimination manages to at least equal the supervised baseline, with improvements most apparent on Cora, the largest (in both fields and tuples) and most complicated of our benchmarks. Also, the improvements usually manifest in the average, rather than the best, case. This indicates that tuning either η and ϵ is often not necessary for Fisher discrimination. A closer analysis of the results confirms their stability across the parameter space. For the baseline, on the other hand, many values of the parameters resulted in a failure of the algorithm, particularly for high levels of recall. Parameter tuning appears to be a necessity as the dataset gets noisier.

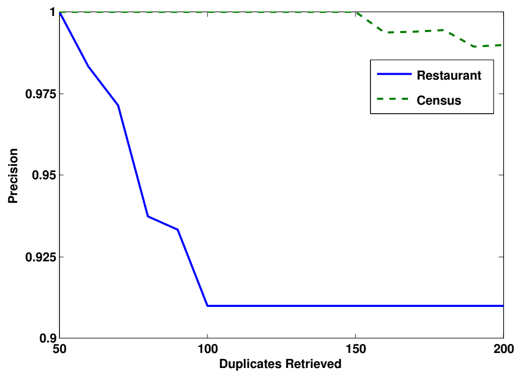

Fig. 1. The precision of duplicates retrieved by *WeakTrainingSet* on Restaurant and Census. $ut = 0.05$, $lt = 0.01$ and $c = 20$

More importantly, small percentage improvements are significant in this context. For example, Cora contains 1295 tuples and 837,865 pairs. A single percent RR improvement means not sending 8300 pairs through expensive (subsequent) record linkage. Industry scale and web databases can be orders of magnitude larger. Table IV shows a nine percent RR improvement on Cora, when learning DNF schemes.

Conversely, *recall* (or PC) can be a bigger concern in other applications, especially with sparse duplicates. In this case also, Fisher performs as well, and sometimes much better ($> 25\%$; see Table III for Cora).

Additionally, Fisher was compared against an *unsupervised* baseline by providing the same weak samples to both. The improvements remain; however, the best baseline results are not much worse compared to the supervised version. Thus, *WeakTrainingSet* may potentially be employed (standalone) for building or supplementing a training set in record linkage.

Moreover, once the noisy (or even perfectly labeled) training set is cast as a set of feature vectors, it becomes amenable to many different feature selection and clustering methods. We chose a relatively simple Fisher Discrimination criterion for this work, but more powerful criteria could potentially offer better performance. Casting blocking scheme learning as a generic feature selection problem, therefore, allows one to utilize an existing body of rich research for solving a hard problem [12].

VII. Related Work

To the best of our knowledge, there are only two works that explicitly address the learning of blocking schemes [3], [4]. The method developed by Bilenko et al. [4] was used as our baseline and has been described in detail; Michelson and

TABLE III
RESULTS: LEARNING DISJUNCTIVE BLOCKING SCHEMES

Dataset		Fisher			Baseline		
		RR	PC	FM	RR	PC	FM
Restaurant	Average	0.9971	0.9306	0.9627	0.9894	0.9513	**0.9698**
	Best	0.9800	0.9554	0.9675	0.9990	1.0000	**0.9995**
Cora	Average	0.9108	0.9113	**0.9110**	0.9029	0.6225	0.7369
	Best	0.9330	0.9443	**0.9386**	0.9330	0.9443	**0.9386**
Census	Average	0.9910	0.9887	**0.9899**	0.9985	0.9689	0.9724
	Best	0.9916	1.0000	**0.9958**	0.9916	1.0000	**0.9958**

TABLE IV
RESULTS: LEARNING DNF BLOCKING SCHEMES

Dataset		Fisher			Baseline		
		RR	PC	FM	RR	PC	FM
Restaurant	Average	0.9993	0.9554	**0.9768**	0.9987	0.9432	0.9701
	Best	0.9993	0.9554	0.9768	0.9990	1.0000	**0.9995**
Cora	Average	0.8909	0.9248	**0.9075**	0.8057	0.8854	0.8437
	Best	0.9330	0.9443	**0.9386**	0.9199	0.9565	0.9378
Census	Average	0.9910	0.9887	**0.9899**	0.9985	0.9689	0.9724
	Best	0.9916	1.0000	**0.9958**	0.9916	1.0000	**0.9958**

TABLE V
RESULTS: LEARNING DNF BLOCKING SCHEMES II

Dataset		Fisher			Unsupervised Baseline		
		RR	PC	FM	RR	PC	FM
Restaurant	Average	0.9993	0.9554	**0.9768**	0.9991	0.9464	0.9721
	Best	0.9993	0.9554	**0.9768**	0.9991	0.9464	0.9721
Cora	Average	0.8909	0.9248	**0.9075**	0.8057	0.8854	0.8437
	Best	0.9330	0.9443	**0.9386**	0.9199	0.9565	0.9378
Census	Average	0.9910	0.9887	**0.9899**	0.9916	0.9878	0.9897
	Best	0.9916	1.0000	**0.9958**	0.9916	.9878	0.9897

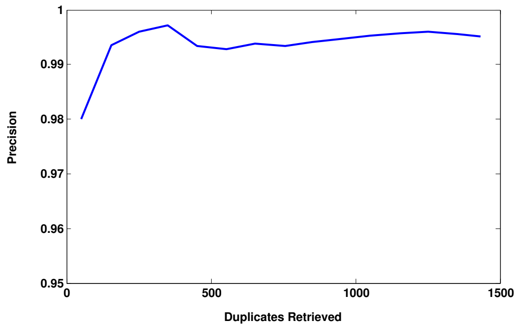

Fig. 2. The precision of duplicates retrieved by *WeakTrainingSet* on Cora. $ut = 0.05$, $lt = 0.01$ and $c = 20$

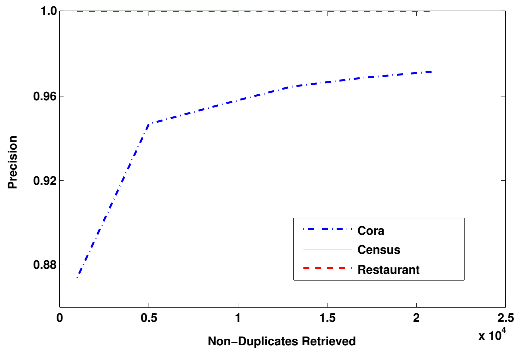

Fig. 3. The precision of non-duplicates retrieved by *WeakTrainingSet* on Cora. $ut = 0.05$, $lt = 0.01$ and $c = 20$. Census and Restaurant achieved perfect accuracy

Knoblock [3] develop a sequential set covering algorithm that adopts a greedy heuristic similar to the baseline. However, given a blocking scheme, there has been much research on refining the blocking step itself. Examples include the bi-gram indexing method [13] and the sorted neighborhood method [5],

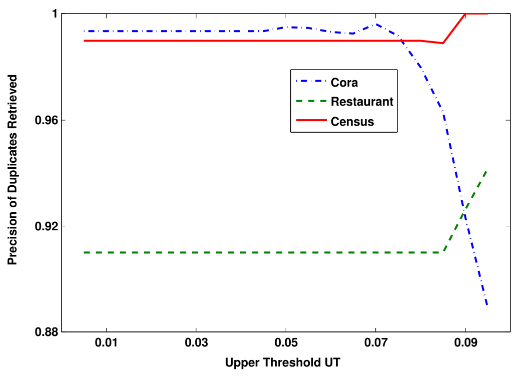

Fig. 4. Precision of duplicates retrieved as ut is varied. 200 duplicates were requested from Restaurant and Census, and 5000 from Cora. The number actually retrieved depended inversely on the magnitude of ut

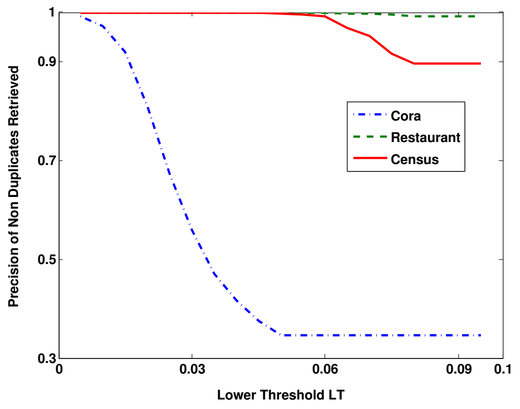

Fig. 5. Precision of non-duplicates retrieved as lt is varied. 5000 non-duplicates were requested for all datasets

with a good survey by Christen [2]. The concept of generating weakly labeled training data has also been investigated in prior work, although to the best of our knowledge, only one work has attempted to incorporate it into record linkage [14]. Moreover, this work only developed an unsupervised method for generating weak non-duplicates. Otherwise, generating artificial training data has been explored in machine learning to generate diverse learners, for instance [15]. The algorithms presented in this paper also draw heavily on feature selection. To the best of our knowledge, this work is the only one to cast blocking scheme discovery as a feature selection problem. A good survey of feature selection is provided by Guyon and Elisseeff [12].

VIII. Future Work and Conclusion

In this paper, an unsupervised framework for learning good blocking schemes was presented and experimentally validated on real-world benchmarks. The algorithms presented were found to present favorable results compared to a supervised state-of-the-art algorithm. Future research includes adapting *WeakTrainingSet* for completely automated record linkage, since an unsupervised record linkage system may be transparently integrated with many cloud workflows. This suggests, in turn, implementations inside a MapReduce framework. The MapReduce framework has garnered much attention since the paper by Google researchers [16]. Adapting our algorithms to a single MapReduce job presents challenges left for future work. Finally, much research has been done in the machine learning community on feature selection. Incorporating more sophisticated feature selection in our algorithms is an interesting opportunity for future investigation.

References

[1] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 1–16, 2007.

[2] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, no. 9, pp. 1537–1555, 2012.

[3] M. Michelson and C. A. Knoblock, "Learning blocking schemes for record linkage," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 21, no. 1. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, p. 440.

[4] M. Bilenko, B. Kamath, and R. J. Mooney, "Adaptive blocking: Learning to scale up record linkage," in *Data Mining, 2006. ICDM'06. Sixth International Conference on*. IEEE, 2006, pp. 87–96.

[5] M. A. Hernández and S. J. Stolfo, "The merge/purge problem for large databases," in *ACM SIGMOD Record*, vol. 24, no. 2. ACM, 1995, pp. 127–138.

[6] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid, "Tailor: A record linkage toolbox," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 17–28.

[7] R. D. Carr, S. Doddi, G. Konjevod, and M. Marathe, "On the red-blue set cover problem," in *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2000, pp. 345–353.

[8] D. Peleg, "Approximation algorithms for the label-cover max and red-blue set cover problems," in *Algorithm Theory-SWAT 2000*. Springer, 2000, pp. 220–231.

[9] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[10] A. Bilke and F. Naumann, "Schema matching using duplicates," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 69–80.

[11] P. E. Hart, R. O. Duda, and D. G. Stork, *Pattern classification*, 2nd ed. Wiley Chichester, 2001, ch. 3, pp. 117–121.

[12] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[13] L. Gu and R. Baxter, "Adaptive filtering for efficient record linkage," in *Proceedings of the Fourth SIAM International Conference on Data Mining*. Society for Industrial Mathematics, 2004, pp. 477–481.

[14] M. Bilenko and R. J. Mooney, "On evaluation and training-set construction for duplicate detection," in *Proceedings of the KDD-2003 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003, pp. 7–12.

[15] P. Melville and R. J. Mooney, "Constructing diverse classifier ensembles using artificial training examples," in *International Joint Conference on Artificial Intelligence*, vol. 18. Citeseer, 2003, pp. 505–512.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.