

Toward Hands-Off Crowdsourcing: Crowdsourced Entity Matching for the Masses

Technical Report

ABSTRACT

Recent approaches to crowdsourcing entity matching (EM) are limited in that they crowdsource only parts of the EM workflow, *requiring a developer* to execute the remaining parts. Consequently, these approaches do not scale to the growing EM need at enterprises and crowdsourcing startups, and cannot handle scenarios where ordinary users (i.e., the masses) want to leverage crowdsourcing to match entities. In response, we propose the notion of *hands-off crowdsourcing (HOC)*, which crowdsources the entire workflow of a task, thus requiring no developers. We show how HOC can represent a next logical direction for crowdsourcing research, scale up EM at enterprises and crowdsourcing startups, and open up crowdsourcing for the masses. We describe *Corleone*, a HOC solution for EM, which uses the crowd in all major steps of the EM process. Finally, we discuss the implications of our work to executing crowdsourced RDBMS joins, cleaning learning models, and soliciting complex information types from crowd workers.

1. INTRODUCTION

Entity matching (EM) finds data records that refer to the same real-world entity, such as (David Smith, JHU) and (D. Smith, John Hopkins). This problem has received significant attention (e.g., [8, 2, 5, 13]). In particular, in the past few years crowdsourcing has been increasingly applied to EM. In crowdsourcing, certain parts of a problem are “farmed out” to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed several crowdsourced EM solutions have been proposed (e.g., [27, 28, 6, 30, 31]).

These pioneering solutions demonstrate the promise of crowdsourced EM, but suffer from a major limitation: they crowdsource only parts of the EM workflow, *requiring a developer* who knows how to code and match to execute the remaining parts. For example, several recent solutions require a developer to write heuristic rules to reduce the number of candidate pairs to be matched, then train and apply a matcher to the remaining pairs to predict matches (see Section 2). They use the crowd only at the end, to verify the predicted matches. The developer must know how to

code (e.g., to write heuristic rules in Perl) and match entities (e.g., to select learning models and features).

As described, current solutions do not scale to the growing EM need at enterprises and crowdsourcing startups. Many enterprises (e.g., eBay, Microsoft, Amazon, Walmart) routinely need to solve tens to hundreds of EM tasks, and this need is growing rapidly. It is not possible to crowdsource all these tasks if crowdsourcing each requires the involvement of a developer (even when sharing developers across tasks). To address this problem, enterprises often ask crowdsourcing startups (e.g., CrowdFlower) to solve the tasks on their behalf. But again, if each task requires a developer, then it is difficult for a startup, with a limited staff, to handle hundreds of EM tasks coming in from multiple enterprises. This is a bottleneck that we have experienced firsthand in our crowdsourcing work at two e-commerce enterprises and two crowdsourcing startups, and this was a major motivation for the work in this paper.

Furthermore, current solutions cannot help ordinary users (i.e., the “masses”) leverage crowdsourcing to match entities. For example, suppose a journalist wants to match two long lists of political donors, and can pay up to a modest amount, say \$500, to the crowd on Amazon’s Mechanical Turk (AMT). He or she typically does not know how to code, thus cannot act as a developer and use current solutions. He or she cannot ask a crowdsourcing startup to help either. The startup would need to engage a developer, and \$500 is not enough to offset the developer’s cost. The same problem would arise for domain scientists, small business workers, end users, and other “data enthusiasts”.

To address these problems, in this paper we introduce the notion of *hands-off crowdsourcing (HOC)*. HOC crowdsources the *entire* workflow of a task, thus requiring no developers. HOC can be a next logical direction for EM and crowdsourcing research, moving from no-, to partial-, to complete crowdsourcing for EM. By requiring no developers, HOC can scale up EM at enterprises and crowdsourcing startups.

HOC can also open up crowdsourcing for the masses. Returning to our example, the journalist wanting to match two lists of donors can just upload the lists to a HOC Web site, and specify how much he or she is willing to pay. The Web site will use the crowd to execute a HOC-based EM workflow, then return the matches. Developing crowdsourcing solutions for the masses (rather than for enterprises) has received rather little attention, despite its potential to magnify many times the impact of crowdsourcing. HOC can significantly advance this direction.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

We then describe **Corleone**, a HOC solution for EM (named after Don Corleone, the Godfather figure who managed the mob in a hands-off fashion). **Corleone** uses the crowd (no developers) in all four major steps of the EM matching process:

- Virtually any large-scale EM problem requires blocking, a step that uses heuristic rules to reduce the number of tuple pairs to be matched (e.g., “if the prices of two products differ by at least \$20, then they do not match”). Current solutions require a developer to write such rules. We show how to use the crowd instead. As far as we know, ours is the first solution that uses the crowd, thus removing developers from this important step.
- We develop a solution that uses crowdsourcing to train a learning-based matcher. We show how to use active learning to minimize crowdsourcing costs.
- Users often want to estimate the matching accuracy, e.g., as precision and recall. Surprisingly, very little work has addressed this problem, and we show that this work breaks down when the data is highly skewed by having very few matches (a common situation). We show how to use the crowd to estimate accuracy in a principled fashion. As far as we know, this is the first in-depth solution to this important problem.
- In practice developers often do EM iteratively, with each iteration focusing on the tuple pairs that earlier iterations have failed to match correctly. So far this has been done in an ad-hoc fashion. We show how to address this problem in a rigorous way, using crowdsourcing.

We present extensive experiments over three real-world data sets, showing that **Corleone** achieves comparable or significantly better accuracy (by as much as 19.8% F_1) than traditional solutions and published results, at a reasonable crowdsourcing cost. Finally, we discuss the implications of our work to crowdsourced RDBMSs, learning, and soliciting complex information types from the crowd. For example, recent work has proposed crowdsourced RDBMSs (e.g., [9, 20, 18]). Crowdsourced joins lie at the heart of such RDBMSs, and many such joins in essence do EM. Today executing such a join on a large amount of data requires developers, thus making such RDBMSs impractical. Our work can help build hands-off no-developer crowdsourced join solutions.

2. BACKGROUND & RELATED WORK

Entity matching has received extensive attention (see [8] for a recent survey). A common setting finds all tuple pairs ($a \in A, b \in B$) from two relational tables A and B that refer to the same real-world entity. In this paper we will consider this setting (leaving other EM settings as ongoing work).

Recently, crowdsourced EM has received increasing attention in academia (e.g., [27, 28, 6, 30, 31, 24]) and industry (e.g., (e.g., CrowdFlower, CrowdComputing, SamaSource, etc.)). Current works use the crowd to verify predicted matches [27, 28, 6], finds the best questions to ask the crowd [30], and finds the best UI to pose such questions [31]. These works still crowdsource only parts of the EM workflow, requiring a developer to execute the remaining parts. In contrast, **Corleone** tries to crowdsource the entire EM workflow, thus requiring no developers.

Specifically, virtually any large-scale EM workflow starts with blocking, a step that uses heuristic rules to reduce the

number of pairs to be matched. This is because the Cartesian product $A \times B$ is often very large, e.g., 10 billion tuple pairs if $|A| = |B| = 100,000$. Matching so many pairs is very expensive or highly impractical. Hence many blocking solutions have been proposed (e.g., [8, 5]). These solutions however do not employ crowdsourcing, and still require a developer (e.g., to write and apply rules, create training data, build indexes, etc.). In contrast, **Corleone** completely crowdsources this step.

After blocking, the next step builds and applies a matcher (e.g., using hand-crafted rules or learning) to match the surviving pairs [8]. Here the works closest to ours are those that use active learning [21, 2, 3, 19]. These works however either do not use crowdsourcing (requiring a developer to label training data) (e.g., [21, 2, 3]), or use crowdsourcing [19] but do not consider how to effectively handle noisy crowd input and to terminate the active learning process. In contrast, **Corleone** considers both of these problems, and uses only crowdsourcing, with no developer in the loop.

The next step, estimating the matching accuracy (e.g., as precision and recall), is vital in real-world EM (e.g., so that the user can decide whether to continue the EM process), but surprisingly has received very little attention in EM research. Here the most relevant work is [12, 22]. [12] uses a continuously refined stratified sampling strategy to estimate the accuracy of a classifier. However, it can not be used to estimate recall which is often necessary for EM. [22] considers the problem of constructing the optimal labeled set for evaluating a given classifier given the size of the sample. In contrast, we consider the different problem of constructing a minimal labeled set, given a maximum allowable error bound.

Subsequent steps in the EM process involve “zooming in” on difficult-to-match pairs, revising the matcher, then matching again. While very common in industrial EM, these steps have received little or no attention in EM research. **Corleone** shows how they can be executed rigorously, using only the crowd.

Finally, crowdsourcing in general has received significant recent attention [7]. In the database community, the work [9, 20, 18] build crowdsourced RDBMSs. Many other works crowdsource joins [17], find the maximal value [10], collect data [25], match schemas [33], and perform data mining [1] and analytics [16].

3. PROPOSED SOLUTION

We now discuss hands-off crowdsourcing and our proposed **Corleone** solution.

Hands-Off Crowdsourcing (HOC): Given a problem P supplied by a user U , we say a crowdsourced solution to P is *hands-off* if it uses no developers, only a crowd of ordinary workers (such as those on AMT). It can ask user U to do a little initial setup work, but this should require no special skills (e.g., coding) and should be doable by any ordinary workers. For example, **Corleone** only requires a user U to supply

1. two tables A and B to be matched,
2. a short textual instruction to the crowd on what it means for two tuples to match (e.g., “these records describe products sold in a department store, they should match if they represent the same product”), and

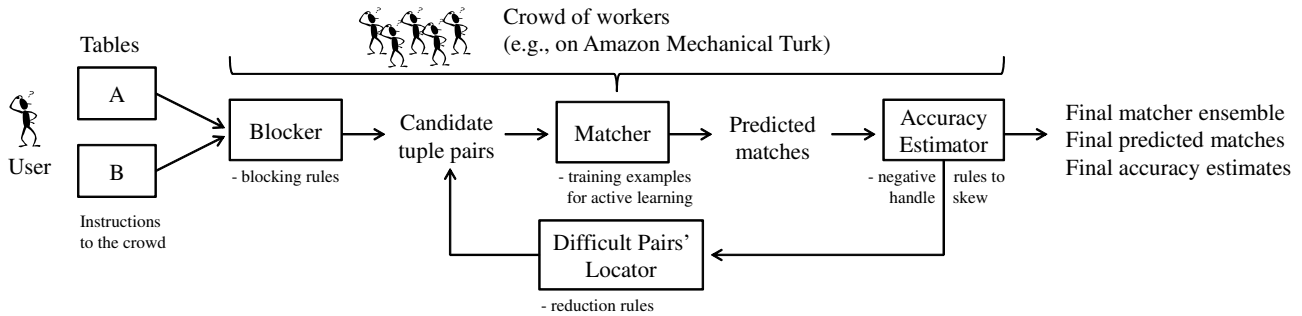


Figure 1: The Corleone architecture.

- four examples, two positive and two negative (i.e., pairs that match and do not match, respectively), to illustrate the instruction. EM tasks posted on AMT commonly come with such instruction and examples.

Corleone then uses the crowd to match A and B (sending them information in (2) and (3) to explain what user U means by a match), then returns the matches. As such, Corleone is a hands-off solution. The following real-world example illustrates Corleone and contrasts it with current EM solutions.

EXAMPLE 3.1. *Consider a retailer that must match tens of millions of products between the online division and the brick-and-mortar division (these divisions often obtain products from different sets of suppliers). The products fall into 500+ categories: toy, electronics, homes, etc. To obtain high matching accuracy, the retailer must consider matching products in each category separately, thus effectively having 500 EM problems, one per category.*

Today, solving each of these EM problems (with or without crowdsourcing) requires extensive developer’s involvement, e.g., to write blocking rules, to create training data for a learning-based matcher, to estimate the matching accuracy, and to revise the matcher, among others. Thus current solutions are not hands-off. One may argue that once created and trained, a solution to an EM problem, say for toys, is hands-off in that it can be automatically applied to match future toy products, without using a developer. But this ignores the initial non-negligible developer effort put into creating and training the solution (thus violating our definition). Furthermore, this solution cannot be transferred to other categories (e.g., electronics). As a result, extensive developer effort is still required for all 500+ categories, a highly impractical approach.

In contrast, using Corleone, per category the user only has to provide Items 1-3, as described above (i.e., the two tables to be matched; the matching instruction which is the same across categories; and the four illustrating examples which virtually any crowdsourcing solutions would have to provide for the crowd). Corleone then uses the crowd to execute all steps of the EM workflow. As such, it is hands-off in that it does not use any developer when solving an EM problem, thus potentially scaling to all 500+ categories. □

We believe HOC is a general notion that can apply to many problem types, such as entity matching, schema matching, information extraction, etc. In this paper we will focus on entity matching. Realizing HOC poses serious challenges, in large part because it has been quite hard to figure out how to make the crowd do certain things. For example, how can

the crowd write blocking rules (e.g., “if prices differ by at least \$20, then two products do not match”)? We need rules in machine-readable format (so that we can apply them). However, most ordinary crowd workers cannot write such rules, and if they write in English, we cannot reliably convert them into machine-readable ones. Finally, if we ask them to select among a set of rules, we often can only work with relatively simple rules and it is hard to construct sophisticated ones. Corleone addresses such challenges, and provides a HOC solution for entity matching.

The Corleone Solution: Figure 1 shows the Corleone architecture, which consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs’ Locator. The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs. The Matcher uses active learning to train a random forest [4], then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs’ Locator finds pairs that the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

As described, Corleone is distinguished in three important ways. (1) All four modules do not use any developers, but heavily use crowdsourcing. (2) In a sense, the modules use crowdsourcing not just to label the data, as existing work has done, but also to “create” complex rules (blocking rules for the Blocker, negative rules for the Estimator, and reduction rules for the Locator, see Sections 4-7). And (3) Corleone can be run in many different ways. The default is to run multiple iterations until the estimated accuracy no longer improves. But the user may also decide to just run until a budget (e.g., \$300) has been exhausted, or to run just one iteration, or just the Blocker and Matcher, etc.

In the rest of the paper we describe Corleone in detail. Sections 4-7 describe the Blocker, Matcher, Estimator, and Locator, respectively. We defer all discussions on how Corleone engages the crowd to Section 8.

4. BLOCKING TO REDUCE SET OF PAIRS

We now describe the Blocker, which generates and applies blocking rules. As discussed earlier, this is critical for large-scale EM. Prior work requires a developer to execute this step. Our goal however is to completely crowdsource it. To do so, we must address the challenge of using the crowd to generate machine-readable blocking rules.

To solve this challenge, Blocker takes a relatively small sample S from $A \times B$; applies crowdsourced active learning, in which the crowd labels a small set of informative pairs

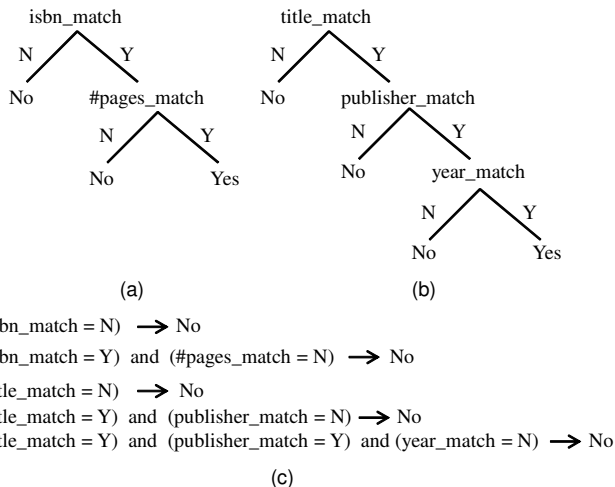


Figure 2: (a)-(b) A toy random forest consisting of two decision trees, and (b) negative rules extracted from the forest.

in S , to learn a random forest matcher; extracts potential blocking rules from the matcher; uses the crowd again to evaluate the quality of these rules; then retain only the best ones. We now describe these steps (see Algorithm 1 for the pseudo code).

4.1 Generating Candidate Blocking Rules

1. Decide Whether to Do Blocking: Let A and B be the two tables to be matched. Intuitively, we want to do blocking only if $A \times B$ is too large to be processed efficiently by subsequent steps. Currently we deem this is the case if $A \times B$ exceeds a threshold t_B , set to be the largest number such that if after blocking we have t_B tuple pairs, then we can fit the feature vectors of all these pairs in memory (we discuss feature vectors below), thus minimizing I/O costs for subsequent steps. The goal of blocking is then to generate and apply blocking rules to remove as many obviously non-matched pairs from $A \times B$ as possible.

Before we go on, a brief remark on minimizing I/O costs. One may wonder if the consideration of reducing I/O costs makes sense, given that *Corleone* already uses crowdsourcing, which can take a long time. We believe it is still important to minimize system time (including I/O time) for three reasons. First, we use active learning, so the sooner the system finishes an iteration, the faster it can go back to the crowd, thereby minimizing total time. Second, depending on the situation, the crowd time may or may not dominate the system time. For example, if we pay 1 penny for a relatively complex question, it may take hours or days before we get 3 workers to answer the question. But if we pay 8 pennies (a rate that many crowdsourcing companies pay), it may take just a few minutes, in which case the crowd time may just be a fraction of the system time. Finally, when working at the scale of millions of tuples per table, system time can be quite significant, taking hours or days (on par or more than crowd time).

2. Take a Small Sample S from $A \times B$: We want to learn a random forest F , then extract candidate blocking rules from it. Learning F directly over $A \times B$ however is impractical because this set is too large. Hence we will sample a far smaller set S from $A \times B$, then learn F over S . Naively, we can randomly sample tuples from A and B ,

then take their Cartesian product to be S . Random tuples from A and B however are unlikely to match. So we may get no or very few positive pairs in S , rendering learning ineffective.

To address this problem, we sample as follows. Let A be the smaller table. We randomly sample $t_B/|A|$ tuples from B (assuming that t_B is much larger than $|A|$, please see below), then take S to be the Cartesian product between this set of tuples and A . Note that we also add the four examples (two positive, two negative) supplied by the user to S . This way, S has roughly t_B pairs, thus having the largest possible size that still fits in memory, to ensure efficient learning. Furthermore, if B has a reasonable number of tuples that have matches in A , and if these tuples are distributed uniformly in B , then the above strategy ensures that S has a reasonable number of positive pairs.

We now discuss the assumption that t_B is much larger than $|A|$. We make this assumption because we consider the current targets of *Corleone* to be matching tables of up to 1 million tuples each, frequently less (e.g., in the range of 50K-300K tuples per table). The vast majority of EM problems that we have seen in industry fall into this range, and we are not aware of any current publication or software that successfully matches tables of 1 million tuples each, even with using Hadoop (unless they do very aggressive blocking). For this target range, t_B , set to be 3M to 5M, is much larger than $|A|$, the smaller table of the two.

That said, our eventual goal is to scale *Corleone* to tables of millions of tuples. Hence, we are exploring better sampling strategies. In Section 10.2.1 we report some preliminary results in this direction.

Our experiments show that the current naive sampling method works well on the current data sets (i.e., we successfully learned good blocking rules from the samples). Briefly, they worked because there are often *many* good negative rules (i.e., rules that find non-matched pairs) with good coverage (i.e., can remove many pairs). Even a naive sampling strategy can give the blocker enough data to find some of these good negative rules, and the blocker just needs to find *some* in order to do a good job at blocking.

3. Apply Crowdsourced Active Learning to S : In the next step, we convert each tuple pair in S into a feature vector, using features taken from a pre-supplied feature library. Example features include edit distance, Jaccard measure, Jaro-Winkler, TF/IDF, Monge-Elkan, etc. [8]. Then we apply crowdsourced active learning to S to learn a random forest F . Briefly, we use the two positive and two negative examples supplied by the user to build an initial forest F , use F to find informative examples in S , ask the crowd to label them, then use the labeled examples to improve F , and so on. A random forest is a set of decision trees [21]. We use decision trees because blocking rules can be naturally extracted from them, as we will see, and we use active learning to minimize the number of examples that the crowd must label. We defer describing this learning process in detail to Section 5.

4. Extract Candidate Blocking Rules from F : The active learning process outputs a random forest F , which is a set of decision trees, as mentioned earlier. Figures 2.a-b show a toy forest with just two trees (in our experiments each forest has 10 trees, and the trees have 8-655 leaves). Here, the first tree states that two books match only if the

ISBNs match and the numbers of pages match. Observe that the leftmost branch of this tree forms a *decision rule*, shown as the first rule in Figure 2.c. This rule states that if the ISBNs do not match, then the two books do *not* match. It is therefore a *negative rule*, and can clearly serve as a blocking rule because it identifies book pairs that do not match. In general, given a forest F , we can extract *all* tree branches that lead from a root to a “no” leaf to form negative rules. Figure 2.c show all five negative rules extracted from the forest in Figures 2.a-b. We return all negative rules as the set of candidate blocking rules.

Algorithm 1 Pseudo-code for the Blocker

Input: Tables A and B ($|A| < |B|$), Set of user-provided labeled pairs L
Output: Candidate tuple pairs C

```

1: /* 1. Decide whether to do blocking */
2: if  $|A \times B| \leq t_B$  then
3:   return  $A \times B$  // no need to block
4: end if
5: /* 2. Take sample  $S$  from  $A \times B$  */
6:  $B_s \leftarrow$  Uniform sample of  $t_B/|A|$  records from  $B$ 
7:  $S \leftarrow (A \times B_s) \cup L$ 
8: /* 3. Apply crowdsourced active learning to  $S$  */
9:  $T \leftarrow L, M \leftarrow$  Train initial random forest on  $T$ 
10: repeat
11:    $E \leftarrow$  Select  $q$  unlabeled examples from  $S$ 
12:   Label all the pairs in  $E$  using the crowd
13:    $T \leftarrow T \cup E, M \leftarrow$  Train a random forest on  $T$ 
14: until  $M$  has stopped improving
15: /* 4. Select top  $k$  blocking rules */
16:  $X_- \leftarrow$  Generate all the negative rules from  $M$ 
17: for all  $R \in X_-$  do
18:   Compute  $|cov(R, S)|, P_{ub} = |cov(R, S) - T|/|cov(R, S)|$ 
19: end for
20: Sort the rules in  $X_-$  in decreasing order of  $P_{ub}, |cov(R, S)|$ 
21:  $V =$  Top  $k$  rules in  $X_-$ 
22: /* 5. Jointly evaluate the rules in  $V$  */
23:  $V_a \leftarrow V, \forall R \in V, X(R) = \emptyset$ 
24: while  $A \neq \emptyset$  do
25:    $Cov \leftarrow \bigcup_{R \in V_a} cov(R, S)$ 
26:    $Q \leftarrow$  Sample  $b$  pairs from  $Cov$  (without replacement)
27:   Label  $Q$  using the crowd
28:   for all  $p \in Q$  do
29:      $\forall R \in V_a$ , if  $p \in cov(R, S)$ , then  $X(R) = X(R) \cup \{p\}$ 
30:   end for
31:   for all  $R \in V_a$  do
32:     Estimate precision  $P$  and error  $\epsilon$ 
33:     if  $P \geq P_{min}$  and  $\epsilon \leq \epsilon_{max}$  then
34:        $V_a \leftarrow V_a - \{R\}$ 
35:     else if  $(P + \epsilon) < P_{min}$  or  $(\epsilon \leq \epsilon_{max}$  and  $P < P_{min})$  then
36:        $V_a \leftarrow V_a - \{R\}, V \leftarrow V - \{R\}$ 
37:     end if
38:   end for
39: end while
40: /* 6. Selecting the blocking rules  $J$  */
41:  $J \leftarrow \emptyset, s_0 \leftarrow |S|, Y \leftarrow V$ 
42: while  $Y \neq \emptyset$  and  $|S| > (t_B/|A \times B|) \cdot s_0$  do
43:   Sort the rules in  $Y$  in decreasing order of  $v$  ( $v = < prec(R, S), |cov(R, S)|, cost(R) >$ )
44:   Pick the topmost rule  $R$  and remove it from  $Y$ 
45:    $Y = Y \cup \{R\}, S \leftarrow S - cov(R, S)$ 
46: end while
47: /* 7. Applying the blocking rules  $J$  */
48:  $C \leftarrow \emptyset$ 
49: for all  $(a, b) \in A \times B$  do
50:   while  $(R \leftarrow getNext(Y)) \neq null$  do
51:     if  $(a, b)$  satisfies  $R$ , continue to the next pair.
52:   end while
53:    $C \leftarrow C \cup \{(a, b)\}$  // if  $(a, b)$  survives all the rules in  $Y$ 
54: end for
55: return  $C$ 

```

4.2 Evaluating Rules using the Crowd

1. Select k Blocking Rules: The extracted blocking rules can vary widely in precision. So we must evaluate and discard the imprecise ones. Ideally, we want to evaluate *all* rules, using the crowd. This however can be very expensive money-wise (we have to pay the crowd), given the large number of rules (e.g., up to 8943 in our experiments). So we pick only k rules to be evaluated by the crowd (current $k = 20$).

Specifically, for each rule R , we compute the coverage of R over sample S , $cov(R, S)$, to be the set of examples in S for which R predicts “no”. We define the precision of R over S , $prec(R, S)$, to be the number of examples in $cov(R, S)$ that are indeed negative divided by $|cov(R, S)|$. As Figure 3 shows, $prec(R, S) = |cov(R, S) - G|/|cov(R, S)|$. Of course, we cannot compute $prec(R, S)$ because we do not know the true labels of examples in $cov(R, S)$, and hence, we do not know the set G . However, we can compute an upper bound on $prec(R, S)$. Let T be the set of examples in S that (a) were selected during the active learning process in Step 3, Section 4.1, and (b) have been labeled by the crowd as positive. Then clearly $prec(R, S) \leq |cov(R, S) - T|/|cov(R, S)|$, since $T \subseteq G$ as can be seen in Figure 3. We then select the rules in decreasing order of the upper bound on $prec(R, S)$, breaking tie using $cov(R, S)$, until we have selected k rules, or have run out of rules. Intuitively, we prefer rules with higher precision and coverage, all else being equal.

2. Evaluate the Selected Rules Using the Crowd:

Let V be the set of selected rules. We now use the crowd to estimate the precision of rules in V , then keep only highly precise rules. Specifically, for each rule $R \in V$, we execute the following loop:

1. We randomly select b examples in $cov(R, S)$, use the crowd to label each example as matched / not matched, then add the labeled examples to a set X (initially set to empty).
2. Let $|cov(R, S)| = m$, $|X| = n$, and n_- be the number of examples in X that are labeled negative (i.e., not matched) by the crowd. Then we can estimate the precision of rule R over S as $P = n_-/n$, with an error margin $\epsilon = Z_{1-\delta/2} \sqrt{\left(\frac{P(1-P)}{n}\right) \left(\frac{m-n}{m-1}\right)}$ [29]. This means that the true precision of R over S is in the range $[P - \epsilon, P + \epsilon]$ with a δ confidence (currently set to 0.95).
3. If $P \geq P_{min}$ and $\epsilon \leq \epsilon_{max}$ (which are pre-specified thresholds), then we stop and add R to the set of precise rules. If (a) $(P + \epsilon) < P_{min}$, or (b) $\epsilon \leq \epsilon_{max}$ and $P < P_{min}$, then we stop and drop R (note that in case (b) with continued evaluation P may still exceed P_{min} , but we judge the continued evaluation to be costly, and hence drop R). Otherwise return to Step 1.

This incremental sampling in batches of size b eventually stops. At this point we either have determined that R is a precise rule, or we have dropped R as it may not be good enough. Currently we set $b = 20, P_{min} = 0.95, \epsilon_{max} = 0.05$. Asking the crowd to label an example is rather involved, and will be discussed in Section 8.

It is important to emphasize that in the above evaluation we do not take just a sample of size b . Instead, we go through multiple iterations, in each we take a sample of size b . Put other ways, we start with b pairs. If we find that these pairs

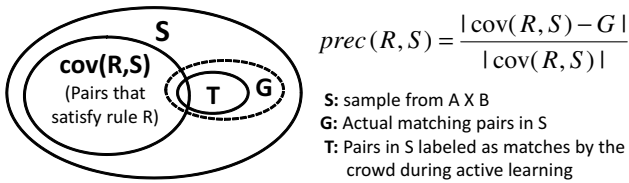


Figure 3: Coverage and precision of rule R over S .

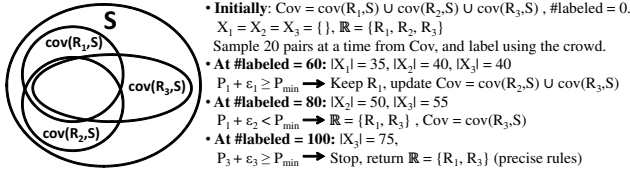


Figure 4: Example illustrating joint evaluation of rules.

do not allow us to compute rule precisions with sufficient accuracy, then we take another b pairs, and add those to the previous ones, and so on.

The above procedure evaluates each rule in V in isolation. We can do better by evaluating all rules in V jointly, to reuse examples across rules. Specifically, let R_1, \dots, R_q be the rules in V . Then we start by randomly selecting b examples from the union of the coverages of R_1, \dots, R_q , use the crowd to label them, then add them to X_1, \dots, X_q , the set of labeled examples that we maintain for the R_1, \dots, R_q , respectively. (For example, if a selected example is in the coverage of only R_1 and R_2 , then we add it to X_1 and X_2 .) Next, we use X_1, \dots, X_q to estimate the precision of the rules, as detailed in Step 2, and then to keep or drop rules, as detailed in Step 3. If we keep or drop a rule, we remove it from the union, and sample only from the union of the remaining rules. Lines 22 - 39 in Algorithm 1 show the pseudocode for joint evaluation of rules.

EXAMPLE 4.1. To illustrate this joint evaluation algorithm, suppose that we have a set of three rules $V = R_1, R_2, R_3$ that need to be evaluated. Figure 4 shows how the joint evaluation algorithm proceeds. Initially, all the rules are “active”, i.e., need to be evaluated, and thus, the set that we sample from, Cov , is set to the union of coverages of all the three rules, i.e., $\text{cov}(R_1, S) \cup \text{cov}(R_2, S) \cup \text{cov}(R_3, S)$.

At this point, we have not sampled any pairs thus, the sample X_i for each rule is empty and the number of labeled pairs ($\#labeled$) = 0. Now we start sampling 20 pairs at a time from Cov , get labels for these pairs, and update X_i , as well as the estimated P_i and ϵ_i for each rule R_i . Suppose that after sampling and labeling a total of 60 pairs, we have 35 pairs in X_1 (the sample for R_1), 40 of the pairs in X_2 , and 40 pairs in X_3 . At this point, suppose that R_1 satisfies the condition for a “precise” rule (line 33 from Algorithm 1). Clearly, we do not need to sample to evaluate R_1 anymore. Thus, we update the set Cov to only include coverages of R_2 and R_3 , i.e., $Cov = \text{cov}(R_2, S) \cup \text{cov}(R_3, S)$.

We now continue sampling from this updated set Cov . Suppose that after sampling and labeling 20 more pairs (thus, $\#labeled = 80$) we have 50 pairs in X_2 and 55 pairs in X_3 . At this point, suppose that we find that R_2 is a “bad” rule (i.e., satisfies the condition in line 35 from Algorithm 1). In that case, we drop R_2 , and only continue the evaluation of R_3 . Thus, we have $V = \{R_1, R_3\}$, and $Cov = \text{cov}(R_3, S)$. Now on sampling 20 more pairs from Cov , we will have 75

pairs in X_3 . At this point suppose R_3 satisfies the condition for a “precise” rule. Clearly we are done evaluating, and we return $\{R_1, R_3\}$ as the set of precise rules. \square

4.3 Applying Blocking Rules

Let Y be the set of rules in V that have survived crowd-based evaluation. We now consider which subset of rules \mathcal{R} in Y should be applied as blocking rules to $A \times B$.

This is highly non-trivial. Let $Z(\mathcal{R})$ be the set of pairs obtained after applying the subset of rules \mathcal{R} to $A \times B$. If $|Z(\mathcal{R})|$ falls below threshold t_B (recall that our goal is to try to reduce $A \times B$ to t_B pairs, if possible), then among all subsets of rules that satisfy this condition, we will want to select the one whose set $Z(\mathcal{R})$ is the *largest*. This is because we want to reduce the number of pairs to be matched to t_B (at which point we can fit all the pairs into main memory), but do not want to go too much below that, because then we run the risk of eliminating many true positive pairs. On the other hand, if no subset of rules from Y can reduce $A \times B$ to below t_B , then we will want to select the subset that does the most reduction, because we want to minimize the number of pairs to be matched.

One may wonder why we do not want to apply all blocking rules. For example, if a rule can reduce the Cartesian product by 80%, why not applying it? The answer is that blocking rules are often not perfect. That is, they often remove not just negative pairs, but some positive pairs too. Unfortunately, a priori there is no good way to evaluate how good a blocking rule is (our precision calculations give only estimates of the true precisions, of course). So if one’s goal is to keep as many positive pairs as one can (because recall is important), then one may choose not to apply a blocking rule even though it can filter out a large number of negative pairs.

For the above reason, we have found that in practice people typically initiate the blocking process only if the original number of pairs is too large to be processed in a reasonable amount of time, and then they do blocking only to the extent that the resulting data set can now be processed practically. They do not apply all blocking rules that they can write, for fear of accidentally removing too many positive pairs.

Returning to our current setting, we cannot execute all subsets of Y on $A \times B$, in order to select the optimal subset. So we use a greedy solution. First, we rank all rules in Y based on the precision $prec(R, S)$, coverage $cov(R, S)$, and the tuple cost. The tuple cost is the cost of applying rule R to a tuple, primarily the cost of computing the features mentioned in R . We can compute this because we know the cost of computing each feature in Step 3, Section 4.1. Next, we select the first rule, apply it to reduce S to S' , re-estimate the precision, coverage, and tuple cost of all remaining rules on S' , re-rank them, select the second rule, and so on. We repeat until the set of selected rules when applied to S has reduced it to a set of size no more than $|S| * (t_B / |A \times B|)$, or we have selected all rules. We then apply the set of selected rules to $A \times B$ (using a Hadoop cluster), to obtain a smaller set of tuple pairs to be matched. This set is passed to the Matcher, which we describe next.

5. TRAINING & APPLYING A MATCHER

Let C be the set of tuple pairs output by the Blocker. We now describe Matcher M , which applies crowdsourcing to learn to match tuple pairs in C . We want to maximize

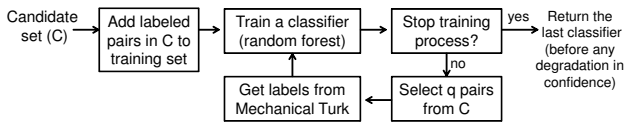


Figure 5: Crowdsourced active learning in Corleone.

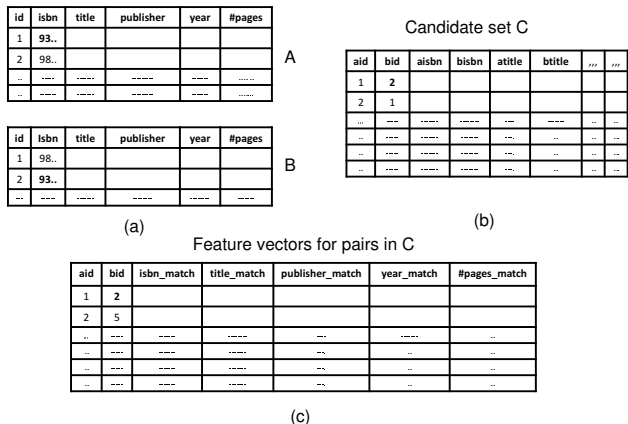


Figure 6: Example: candidate set and feature vectors.

the matching accuracy, while minimizing the crowdsourcing cost. To do this, we use active learning. Figure 5 shows the overall workflow for learning the matcher. Specifically, we train an initial matcher M , use it to select a small set of informative examples from C , ask the crowd to label the examples, use them to improve M , and so on. A key challenge is deciding when to stop training M . Excessive training wastes money, and yet surprisingly can actually *decrease*, rather than increase the matcher’s accuracy. We now describe matcher M and our solution to the above challenge.

5.1 Training the Initial Matcher

We begin by converting all examples (i.e., tuple pairs) in C into feature vectors, for learning purposes. This is done at the end of the blocking step: any surviving example is immediately converted into a feature vector, using all features that are appropriate (e.g., no TF/IDF features for numeric attributes) and available in our feature library. Figure 6 shows an illustrative example from the books domain. Note that in the rest of the paper, we use the terms example, pair, and feature vector interchangeably, when there is no ambiguity.

EXAMPLE 5.1. *Suppose we have two tables A and B to be matched, each containing book tuples. Each tuple contains 6 attributes: id , $isbn$, $title$, $publisher$, $year$, and $\#pages$ (number of pages). Suppose that blocking is triggered in this case. Figure 6.b shows the candidate set C output by the blocking step. C contains a small number of potential matching pairs of tuples from A and B . Suppose that Corleone selects 5 features to compute for each pair: $isbn_match$, $title_match$, $publisher_match$, $year_match$, and $\#pages_match$. Each of these features are binary, i.e., if the two values exactly match the feature evaluates to 1, if not then 0.¹ Next, Corleone computes all the 5 features for each pair in C , to get the feature vectors, as shown in Figure 6.c. This table of feature vectors is now used in the learning step.*

¹In practice, our feature library contains a variety of exact as well as fuzzy matching features, we only use simple features here to illustrate.

Next, we use all labeled examples available at that point (supplied by the user or labeled by the crowd) to train an initial classifier that when given an example (x, y) will predict if x matches y . Currently we use an ensemble-of-decision-trees approach called *random forest* [4]. In this approach, we train k decision trees independently, each on a random portion (typically set at 60%) of the original training data. When training a tree, at each tree node we randomly select m features from the full set of features f_1, \dots, f_n , then use the best feature among the m selected to split the remaining training examples. The values k and m are currently set to be the default 10 and $\log(n) + 1$, respectively. Once trained, applying a random forest classifier means applying the k decision trees, then taking the majority vote.

To illustrate, going back to the book examples (Example 5.1), Figure 2 shows a sample random forest that could be learned from this dataset.

5.2 Consuming the Next Batch of Examples

Once matcher M has trained a classifier, M evaluates the classifier to decide whether further training is necessary (see Section 5.3). Suppose M has decided yes, then it must select new examples for labeling.

In the simplest case, M can select just a single example (as current active learning approaches often do). A crowd however often refuses to label just one example, judging it to be too much overhead for little money. Consequently, M selects q examples (currently set to 20) for the crowd to label. Intuitively, M wants these examples to be “most informative”. A common way to measure the “informativeness” of an example e is to measure the disagreement of the component classifiers using entropy [23]:

$$entropy(e) = -[P_+(e) \cdot \ln(P_+(e)) + P_-(e) \cdot \ln(P_-(e))], \quad (1)$$

where $P_+(e)$ and $P_-(e)$ are the fractions of the decision trees in the random forest that label example e positive and negative, respectively. The higher the entropy, the stronger the disagreement, and the more informative the example is.

Thus, M selects the p examples (currently set to 100) with the highest entropy from set C (excluding those that have been selected in the previous iterations). Next, M selects q examples from these p examples, using weighted sampling, with the entropy values being the weights. This sampling step is necessary because M wants the q selected examples to be not just informative, but also diverse. The following example illustrates the weighted sampling step.

EXAMPLE 5.2. *To keep the example simple, suppose that $p = 5$ and $q = 2$. Suppose the top 5 pairs with the highest entropy are as follows: p_1 (0.6), p_2 (0.6), p_3 (0.4), p_4 (0.4), p_5 (0.4), where the number in parentheses shows the entropy for the pair. Now we randomly draw a total of 2 pairs, drawing one pair at a time from these 5 pairs. However, each pair does not have an equal chance of getting picked. The probability of picking a pair is proportional to its entropy. Thus, when drawing the first pair, the probability for picking p_1 will be $0.6 / (0.6 + 0.6 + 0.4 + 0.4 + 0.4) = 1/4$. Similarly, the probability of picking p_2 will be $1/4$. However, for p_3 , p_4 , and p_5 the probability of being picked will be $1/6$ each.*

In the next step, M sends the q selected examples to the crowd to label (described in Section 8), adds the labeled examples to the current training data, then re-trains the classifier.

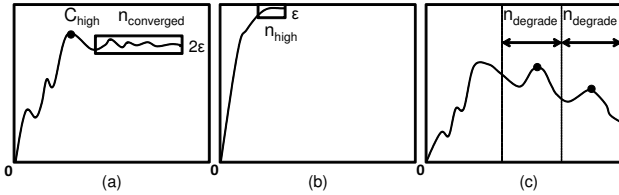


Figure 7: Typical confidence patterns that we can exploit for stopping.

At this point one may wonder how expensive entropy computation is. We note that this computation requires just a linear scan over the pairs in the candidate set (i.e., those pairs surviving the blocking step). As such, its time (per iteration) grows proportional to the candidate set size, and has stayed in the range of seconds in our experiments. For example, on the Product data set, on average the candidate set’s size is 200K, entropy computation time is about 2.4 seconds per iteration, and total entropy computation time is 2 minutes (for 50 iterations, see the experiment section). Of course, on large data sets (and thus larger candidate sets), this time will grow. Fortunately, this step is trivially parallelizable.

5.3 Deciding When to Stop

Recall that matcher M trains in iteration, in each of which it pays the crowd to label q training examples. We must decide then when to stop the training. Interestingly, more iterations of training not only cost more, as expected, but can actually *decrease* rather than increase M ’s accuracy. This happens because after M has reached peak accuracy, more training, even with perfectly labeled examples, does not supply any more informative examples, and can mislead M instead. This problem became especially acute in crowdsourcing, where crowd-supplied labels can often be incorrect, thereby misleading the matcher even more.

To address this problem, we develop a solution that tells M when to stop training. Our solution defines the “confidence” of M as the degree to which the component decision trees agree with one another when labeling. We then monitor M and stop it when its confidence has peaked, indicating that there are no or few informative examples left to learn from.

Specifically, let $conf(e) = 1 - entropy(e)$, where $entropy(e)$ is computed as in Equation 1, be the *confidence* of M over an example e . The smaller the entropy, the more decision trees of M agree with one another when labeling e , and so the more confident M is that it has correctly labeled e .

Before starting the active learning process, we set aside a small portion of C (currently set to be 3%), to be used as a monitoring set V . We monitor the confidence of M over V , defined as $conf(V) = \sum_{e \in V} conf(e) / |V|$. We expect that initially $conf(V)$ is low, reflecting the fact that M has not been trained sufficiently, so the decision trees still disagree a lot when labeling examples. As M is trained with more and more informative examples (see Section 5.2), the trees become more and more “robust”, and disagree less and less. So $conf(V)$ will rise, i.e., M is becoming more and more confident in its labeling. Eventually there are no or few informative examples left to learn from, so the disagreement of the trees levels off. This means $conf(V)$ will also level off. At this point we stop the training of matcher M .

We now describe the precise stopping conditions, which,

as it turned out, was quite tricky to establish. Ideally, once confidence $conf(V)$ has leveled off, it should stay level. In practice, additional training examples may lead the matcher astray, thus reducing or increasing $conf(V)$. This is exacerbated in crowdsourcing, where the crowd-supplied labels may be wrong, leading the matcher even more astray, thus causing drastic “peaks” and “valleys” in the confidence line. This makes it difficult to sift through the “noise” to discern when the confidence appears to have peaked. We solve this problem as follows.

First, we run a smoothing window of size w over the confidence values recorded so far (one value per iteration), using average as the smoothing function. That is, we replace each value x with the average of the w values: $(w - 1)/2$ values on the left of x , $(w - 1)/2$ values on the right, and x itself. (Currently $w = 5$.) We then stop if we observe any of the following three patterns over the smoothed confidence values:

- **Converged confidence:** In this pattern the confidence values have stabilized and stayed within a 2ϵ interval (i.e., for all values v , $|v - v^*| \leq \epsilon$ for some v^*) over $n_{converged}$ iterations. We use $\epsilon = 0.01$ and $n_{converged} = 20$ in our experiments (these parameters and those described below are set using simulated crowds). Figure 7.a illustrates this case. When this happens, the confidence is likely to have converged, and unlikely to still go up or down. So we stop the training.
- **Near-absolute confidence:** This pattern is a special case of the first pattern. In this pattern, the confidence is at least $1 - \epsilon$, for n_{high} consecutive iterations (see Figure 7.b). We currently use $n_{high} = 3$. When this pattern happens, confidence has reached a very high, near-absolute value, and has no more room to improve. So we can stop, not having to wait for the whole 20 iterations as in the case of the first pattern.
- **Degrading confidence:** This pattern captures the scenarios where the confidence has reached the peak, then degraded. In this pattern we consider two consecutive windows of size $n_{degrade}$, and find that the maximal value in the first window (i.e., the earlier one in time) is higher than that of the second window by more than ϵ (see Figure 7.c). We currently use $n_{degrade} = 15$. We have experimented with several variations of this pattern. For example, we considered comparing the average values of the two windows, or comparing the first value, average value, and the last value of a (relatively long) window. We found however that the above pattern appears to be the best at accurately detecting degrading confidence after the peak.

Afterward, M selects the last classifier before degrading to match the tuple pairs in the input set C .

6. ESTIMATING MATCHING ACCURACY

After applying matcher M , Corleone estimates M ’s accuracy. If this exceeds the best accuracy obtained so far, Corleone continues with another round of matching (see Section 7). Otherwise, it stops, returning the matches together with the estimated accuracy. This estimated accuracy is especially useful to the user, as it helps decide how good the crowdsourced matches are and how best to use them. We now describe how to estimate the matching accuracy.

6.1 Current Methods and Their Limitations

To motivate our method, we begin by describing current evaluation methods and their limitations. Suppose we have applied matcher M to a set of examples C . To estimate the accuracy of M , a common method is to take a random sample S from C , manually label S , then compute the precision $P = n_{tp}/n_{pp}$ and the recall $R = n_{tp}/n_{ap}$, where (a) n_{pp} is the number of predicted positives: those examples in S that are labeled positive (i.e., matched) by M ; (b) n_{ap} is the number of actual positives: those examples in S that are manually labeled as positive; and (c) n_{tp} is the number of true positives: those examples in S that are both predicted positive and actual positive.

Let P^* and R^* be the precision and recall on the set C (computed in an analogous fashion, but over C , not over S). Since S is a random sample of C , we can report that with δ confidence, $P^* \in [P - \epsilon_p, P + \epsilon_p]$ and $R^* \in [R - \epsilon_r, R + \epsilon_r]$, where the error margins are defined as

$$\epsilon_p = Z_{1-\delta/2} \sqrt{\left(\frac{P(1-P)}{n_{pp}}\right) \left(\frac{n_{pp}^* - n_{pp}}{n_{pp}^* - 1}\right)}, \quad (2)$$

$$\epsilon_r = Z_{1-\delta/2} \sqrt{\left(\frac{R(1-R)}{n_{ap}}\right) \left(\frac{n_{ap}^* - n_{ap}}{n_{ap}^* - 1}\right)}, \quad (3)$$

where n_{ap}^* and n_{pp}^* are the number of actual positives and predicted positives on C , respectively, and $Z_{1-\delta/2}$ is the $(1 - \delta/2)$ percentile of the standard normal distribution [29].

As described, the above method has a major limitation: it often requires a very large sample S to ensure small error margins, and thus ensuring meaningful estimation ranges for P^* and R^* . For example, assuming $R^* = 0.8$, to obtain a reasonable error margin of, say $\epsilon_r = 0.025$, using Equation 3 we can show that $n_{ap} \geq 984$ (regardless of the value for n_{ap}^*). That is, S should contain at least 984 actual positive examples.

The example universe for EM however is often quite skewed, with the number of positive examples being just a small fraction of the total number of examples (e.g., 0.06%, 2.64%, and 0.56% for the three data sets in Section 9, even after blocking). A fraction of 2.64% means that S must contain at least 37273 examples, in order to ensure at least 984 actual positive examples. Labeling 37000+ examples however is often impractical, regardless of whether we use a developer or the crowd, thus making the above method inapplicable.

When finding too few positive examples, developers often apply heuristic rules that eliminate negative examples from C , thus attempting to “reduce” C into a smaller set C_1 with a far higher “density” of positives. They then randomly sample from C_1 , in the hope of boosting n_{ap} and n_{pp} , thereby reducing the margins of error. This approach, while promising, is often carried out in an ad-hoc fashion. As far as we know, no strategy on how to do reduction systematically has been reported. In what follows, we show how to do this in a rigorous way, using crowdsourcing and negative rules extracted from the random forest.

6.2 Crowdsourced Estimation with Corleone

Our solution incrementally samples from C . If it detects data skew, i.e., too few positive examples, it performs reduction (i.e., using rules to eliminate certain negative examples from C) to increase the positive density, then samples again. This continues until it has managed to estimate P and R

within a given margin of error ϵ_{max} . Our solution does not use any developer. Rather, it uses the crowd to label examples in the samples, and to generate reduction rules, as described below.

1. Generating Candidate Reduction Rules: When applied to a set of examples (e.g., C), reduction rules eliminate negative examples, thus increasing the density of positive examples in the set. As such, they are conceptually the same as blocking rules in Section 4. Those rules cannot be used on C , however, because they are already applied to $A \times B$ to generate C .

Instead, we can generate candidate reduction rules exactly the way we generate blocking rules in Section 4, except for the followings. First, in the blocking step in Section 4 we extract the rules from a random forest trained over a relatively small sample S . Here, we extract the rules from the random forest of matcher M , trained over the entire set C . Second, in the blocking step we select top k rules, evaluate them using the crowd, then keep only the precise rules. Here, we also select top k rules, but we do not yet evaluate them using the crowd (that will come later, if necessary). We return the selected rules as candidate reduction rules.

2. Repeating a Probe-Eval-Reduce Loop: We then perform the following online search algorithm to estimate the accuracy of matcher M over C :

- Enumerating our options:* To estimate the accuracy, we may execute no reduction rule at all, or just one rule, or two rules, and so on. Let $\mathcal{R} = \{R_1, \dots, R_n\}$ be the set of candidate reduction rules. Then we have a total of 2^n possible options, each executing a subset of rules in \mathcal{R} .
- Estimating and selecting the lowest-cost option:* A priori we do not know which option is the best. Hence, we perform a limited sampling of C (the *probe* operation) to estimate the cost of each option (to be discussed below), then select the one with the lowest cost.
- Partially evaluating the selected option:* Without loss of generalization, suppose we have selected the option that executes rules $\mathcal{D} = \{R_1, \dots, R_d\}$. Fully evaluating this option means (a) using the crowd to evaluate rules R_1, \dots, R_d , exactly the way we evaluate blocking rules in Section 4.2 (the *eval* operation), (b) keeping only good, i.e., highly precise, rules, (c) executing these rules on C to reduce it, thereby increasing the positive density, then (d) sampling from the reduced C until we have managed to estimate P and R within the margin of error ϵ_{max} .

Instead of fully evaluating the selected option, we do mid-execution optimization. Specifically, after executing (a)-(c), we do not do (d). Instead we return to Step 1 to re-enumerate our options. Note that now we have a reduced set C (because we have applied the good rules in \mathcal{D}), and also a reduced set \mathcal{R} (because we have removed all rules in \mathcal{D} from \mathcal{R}).

The above strategy is akin to mid-query re-optimization in RDBMSs, where given a SQL query, we select a good execution plan, partially evaluate it, then use the newly gathered statistics to re-optimize to find a potentially better execution plan. Similarly, in our setting, once we have selected a plan, we perform a partial evaluation by executing Steps (a)-(c). At this point we may have gained more information, such as which rules are bad. So we skip Step (d), and return to Step 1 to see if we can find

Algorithm 2 Pseudo-code for the *estimate* operation

Input: Original candidate set C , Reduced candidate set C' , Matcher M

Output: $P, \epsilon_p, R, \epsilon_r$

```
1: /* We first sample from  $C'$  until the recall error is below  $\epsilon_{max}$  */
2:  $S = \emptyset, n_{ap} = 0, n_{tp} = 0, n = 0$ 
3:  $recallDone = false$ 
4: while (not recallDone) do
5:   Uniformly draw next batch  $B$  of  $b$  examples from  $(C' - S)$ 
6:   Get label  $l(t)$  for each example  $t \in B$ , from the crowd
7:    $S = S \cup B, n = n + b$ 
8:   for all  $t \in B$  do
9:     if ( $l(t) = +$ ) then
10:        $n_{ap} = n_{ap} + 1$ 
11:       if  $M(t) = +$  then
12:          $n_{tp} = n_{tp} + 1$ 
13:       end if
14:     end if
15:   end for
16:    $max_{ap} = n_{ap} + |C'| - n$ 
17:    $R = \frac{n_{tp}}{n_{ap}}, \epsilon_r = Z_{1-\delta/2} \sqrt{\left(\frac{R(1-R)}{n_{ap}}\right) \left(\frac{max_{ap}-n_{ap}}{max_{ap}-1}\right)}$ 
18:   if  $\epsilon_r \leq \epsilon_{max}$  then
19:     recallDone = true
20:   end if
21: end while
22: /* Now we check if precision error is already below  $\epsilon_{max}$ , if yes we are done */
23:  $S_{pp} = M(S), n_{pp} = |S_{pp}|, done = false$  /* here  $M(A)$  denotes  $\{t \in A : M(t) = +\}$  */
24:  $\alpha = \frac{|M(C')|}{|M(C)|}, P = \alpha \cdot \frac{n_{tp}}{n_{pp}}$ 
25:  $\epsilon_p = \alpha \cdot Z_{1-\delta/2} \sqrt{\left(\frac{P(1-P)}{n_{pp}}\right) \left(\frac{n_{pp}^* - n_{pp}}{n_{pp}^* - 1}\right)}$ 
26: if  $\epsilon_p \leq \epsilon_{max}$  then
27:   done = true
28: end if
29: /* If not done, then sample more from  $(M(C') - S_{pp})$  */
30: while (not done) do
31:   Uniformly draw next batch  $B$  of  $b$  examples from  $(M(C') - S_{pp})$ 
32:   Get label  $l(t)$  for each example  $t \in B$ , from the crowd
33:    $S_{pp} = S_{pp} \cup B, n_{pp} = n_{pp} + b$ 
34:   for all  $t \in B$  do
35:     if ( $l(t) = +$ ) then
36:        $n_{tp} = n_{tp} + 1$ 
37:     end if
38:   end for
39:   Compute  $P$  and  $\epsilon_p$  as in 24 and 25.
40:   if  $\epsilon_p \leq \epsilon_{max}$  then
41:     done = true
42:   end if
43: end while
44: return  $P, \epsilon_p, R, \epsilon_r$ 
```

a potentially better plan. Eventually we do have to execute Step (d), but only after we have concluded that we cannot find any potentially better plan.

4. *Termination:* If we have not terminated earlier (e.g., in Step 2, after sampling of C , see below), then eventually we will select the option of using no rules (in the worst-case scenario this happens when we have applied all rules). If so, we sample until we have managed to estimate P and R within a margin of error ϵ_{max} . Algorithm 2 shows the pseudo-code for this estimation step that terminates the algorithm.

All that is left is to describe how we estimate the costs of the options in Step 2. Without loss of generalization, consider an option that executes rules $\mathcal{Q} = \{R_1, \dots, R_q\}$. We estimate its cost to be (1) the cost of evaluating all rules in \mathcal{Q} , plus (2) the cost of sampling from the reduced set C after we have applied all rules in \mathcal{Q} (note that we are making an optimistic assumption here that all rules in \mathcal{Q} turn out to be good).

Currently we estimate the cost in (1) to be the sum of

Algorithm 3 Pseudo-code for *probe* operation

Input: Candidate set C , Matcher M, ϵ_{max}

Output: Density of actual positives d

```
1: Uniformly sample  $b$  examples from  $C$ , and label them using the crowd to create sample  $S$ .
2:  $n_{ap} \leftarrow$  Number of actual positives in  $S$ 
3:  $n_{tp} \leftarrow$  Number of true positives in  $S$ 
4:  $n_{pp} \leftarrow$  Number of predicted positives in  $S$ 
5: Compute  $P, R, \epsilon_p$ , and  $\epsilon_r$  (using equations 2)
6: if  $\epsilon_p \leq \epsilon_{max}$  and  $\epsilon_r \leq \epsilon_{max}$ , then stop the estimation process.
7: return  $d = \frac{n_{ap}}{|S|}$ 
```

the costs of evaluating each individual rule. In turn, the cost of evaluating a rule is the number of examples that we would need to select from its coverage for the crowd to label, in order to estimate the precision to be within ϵ_{max} (see Section 4.2). We can estimate this number using the formulas for precision P and error margin ϵ given in Section 4.2.

Suppose after applying all rules in \mathcal{Q} , C is reduced to set C' . We estimate the cost in (2) to be the number of examples we need to sample from C' to guarantee margin of error ϵ_{max} . If we know the positive density d' of C' , we estimate the above number. It is easy to prove that $d' = d * |C|/|C'|$, where d is the positive density of C (assuming that the rules are 100% precise).

To estimate d , we perform a “limited sampling”, i.e., the *probe* operation, by sampling b examples from the set C (currently $b = 50$). Algorithm 3 shows the pseudo-code for the probe operation. We use the crowd to label these examples, then estimate d to be the fraction of examples being labeled positive by the crowd. (We note that in addition, we also use these labeled b examples to estimate $P, R, \epsilon_p, \epsilon_r$, as shown in Section 6.1, and immediately exit if ϵ_p and ϵ_r are already below ϵ_{max} .) We now present an example to illustrate the algorithm.

EXAMPLE 6.1. *Suppose that we have a candidate set C containing 50000 pairs, and we want to estimate the precision and recall of a given matcher M over the set C . Figure 8 shows a step-by-step execution of the crowdsourced estimation algorithm for this example. Suppose that generating top rules from M gives us a set \mathcal{R} containing 3 rules $\mathcal{R} = \{R_1, R_2, R_3\}$. As the first step, Corleone enumerates all the options, i.e., lists all the subsets of \mathcal{R} . Next, it performs the probe operation, taking a sample S of size 50 to estimate the density of positives d . Suppose d is 0.01 (i.e. 1%).*

It then estimates the expected cost (i.e. number of examples to be labeled) for each of the options listed, starting from $\{\}$, i.e., the option of applying zero rules to reduce, and sampling all the way from the current candidate set. As explained earlier, the estimated cost of an option $\mathcal{Q} = \{R_1, \dots, R_q\}$ is $c(\mathcal{Q}) = c_r + c_s$, where c_r is the cost of evaluating all the rules in \mathcal{Q} , c_s is the cost of sampling from the reduced set after applying all the rules in \mathcal{Q} . Here $c_r = q \cdot c_1$ where q is the number of rules in \mathcal{Q} , and $c_s = c_2 \cdot (1/d')$ where c_2 is the number of positives we need in the sample to guarantee an error margin below ϵ_{max} , and d' is the expected positive density in the reduced set we will obtain on applying all the rules in \mathcal{Q} .

To illustrate how the cost is computed, let us consider $\mathcal{Q} = \{\}$. In this case $c_r = 0$ as $q = 0$, and $d' = d$, since C is not reduced at all. Suppose, $c_1 = 70$ and $c_2 = 392$. Thus, we get $c(\{\}) = 0 + 392(1/0.01)$, i.e. 39.2k.

Next, Corleone picks the option with the lowest cost, which

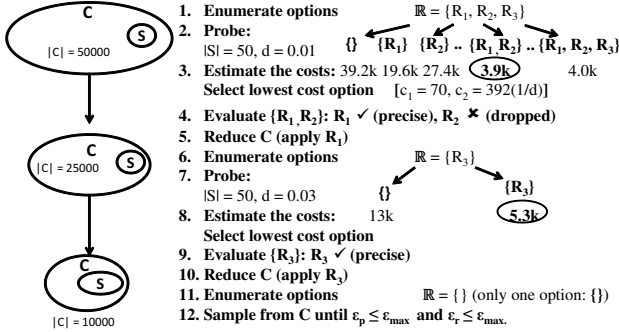


Figure 8: Example to illustrate the estimation process.

is $\{R_1, R_2\}$ in this case, as shown in Figure 8. It then evaluates the selected rules. Suppose R_1 passes the test for precision, but R_2 fails. In this case, it applies only the rule R_1 to reduce C , and removes R_1 and R_2 from the set of rules \mathcal{R} . After the reduction, C contains 25000 pairs as shown in Figure 8. At this point, $\mathcal{R} = \{R_3\}$. Now Corleone again enumerates the options and performs the probe operation to estimate the cost of each option. We only have two possible options either to use no rules ($\{\}$) and sample all the way from current C , or $\{R_3\}$, i.e., to evaluate R_3 and if it is precise, apply it to reduce C . The second option has lower expected cost (5.3k as opposed to 13k), and thus, the option $\{R_3\}$ gets picked. Next, the algorithm evaluates R_3 to find that it is precise. It then applies R_3 to reduce C further. At this stage, we are left with a candidate set C of size 10000, and no more rules to reduce. Thus, the algorithm picks the default option $\{\}$, and samples from C until it has estimated both precision and recall, within ϵ_{max} error margin. This terminates the estimation step.

Correctness of the Estimates: In the crowdsourced estimation algorithm above, the final estimates for precision and recall over the original candidate set C are computed in the sampling step at the end (lines 17, 24, and 25 from Algorithm 2). However, the sample here is drawn only from the reduced candidate set C' . Since this sampling procedure is the same as described in 6.1, estimating precision and recall over C' is straightforward. Thus, to prove the correctness of the equations from lines 17, 24, and 25, we simply need to show how to estimate the precision and recall over original candidate set C , using estimates over reduced set C' . This is exactly what we do next.

PROPOSITION 1. *Let M be a given binary classifier. Let P and R be the precision and recall values of M on the candidate set of matching pairs C . Let P' and R' be the precision and recall values of M on the reduced set C' obtained at the end of crowdsourced estimation with Corleone. With the assumption that all the rules used for reducing C are perfect, i.e., they do not eliminate any positive pairs, and the labels provided by the crowd are perfect, we have:*

$$R = R', P = \alpha \cdot P'$$

where $\alpha = \frac{|M(C')|}{|M(C)|}$, and $M(X)$ returns the set of pairs in set X predicted as positive by M .

PROOF. We first show that $Matches(C') = Matches(C)$, where $Matches(X)$ returns the set of actual matching pairs in set X . We then derive the expressions for recall and precision of f on the set C .

In the crowdsourced estimation algorithm, we begin with C as the candidate set, and then iteratively apply reduction rules on the candidate set to finally obtain C' . Since each of these reduction rules retain all the positive examples in C , C' must contain all the positive examples in C , i.e., $Matches(C') = Matches(C)$.

Let G denote this set of all the matching pairs in C . We have, $G = Matches(C') = Matches(C)$.

Given G as the set of actual positives in C (as well as C'), $M(C)$ as the set of predicted positives in C , and $M(C')$ as the set of predicted positives in C' , we can show that (a) for the set C , $|Actual\ positives| = |G|$, $|Predicted\ positives| = |M(C)|$, $|True\ positives| = |M(C) \cap G|$, and (b) for the set C' , $|Actual\ positives| = |G|$, $|Predicted\ positives| = |M(C')|$, $|True\ positives| = |M(C') \cap G|$.

Using the definitions for precision and recall, from Section 6.1, we can now write R , R' , P and P' as follows:

$$R = \frac{|M(C) \cap G|}{|G|}, R' = \frac{|M(C') \cap G|}{|G|} \quad (4)$$

$$P = \frac{|M(C) \cap G|}{|M(C)|}, P' = \frac{|M(C') \cap G|}{|M(C')|} \quad (5)$$

Let $C'' = C \setminus C'$. We can write, $M(C) = M(C') \cup M(C'')$, where $M(C'')$ is the set of predicted positives in C'' . Thus, $M(C) \cap G = (M(C') \cup M(C'')) \cap G$, i.e., $M(C) \cap G = (M(C') \cap G) \cup (M(C'') \cap G)$.

However, G is completely contained inside C' . Thus, $M(C'') \cap G = \emptyset$. Therefore,

$$M(C) \cap G = M(C') \cap G$$

Substituting this in the equations 4 and 5, we get:

$$R = R', P = \frac{|M(C')|}{|M(C)|} \cdot P'.$$

□

7. ITERATING TO IMPROVE

In practice, entity matching is not a one-shot operation. Developers often estimate the matching result, then revise and match again. A common way to revise is to find tuple pairs that have proven difficult to match, then modify the current matcher, or build a new matcher specifically for these pairs. For example, when matching e-commerce products, a developer may find that the current matcher does reasonably well across all categories, except in Clothes, and so may build a new matcher specifically for Clothes products.

Corleone operates in a similar fashion. It estimates the matching accuracy (as discussed earlier), then stops if the accuracy does not improve (compared to the previous iteration). Otherwise, it revises and matches again. Specifically, it attempts to locate difficult-to-match pairs, then build a new matcher specifically for those. The challenge is how to locate difficult-to-match pairs. Our key idea is to identify precise positive and negative rules from the learned random forest, then remove all pairs covered by these rules (they are, in a sense, easy to match, because there already exist rules that cover them). We treat the remaining examples as difficult to match, because the current forest does not contain any precise rule that covers them. We now describe this idea in detail.

1. Extract Positive & Negative Rules: Let F be the random forest learned by matcher M . In Section 4 we have discussed how to extract negative rules from F , select top rules, use the crowd to evaluate them, then keep only the highly precise ones. Here we do exactly the same thing to obtain k highly precise negative rules (or as many as F has). Note that some of these rules might have been used in estimating the matching accuracy (Section 6).

We then proceed similarly to obtain k highly precise positive rules (or as many as F has). A positive rule is similar to a negative rule, except that it is a path from a root to a “yes” leaf node in F . That is, if it applies to a pair, then it predicts that the pair match.

2. Apply Rules to Remove Easy-to-Match Pairs: Let \mathcal{E} be the set of positive and negative rules so obtained. Recall that in the current iteration we have applied matcher M to match examples in set C . We now apply all rules in \mathcal{E} to C , to remove examples covered by any of these rules. Let the set of remaining examples be C' . As mentioned earlier, we treat these examples as difficult to match, because they have not been covered by any precise (negative or positive) rule in the current matcher M .

3. Learn a New Matcher for Surviving Pairs: In the next iteration, we learn a new matcher M' over the set C' , using the same crowdsourced active learning method described in Section 5, and so on. In the end we use the so-constructed set of matchers to match examples in C . For example, if we terminate after two iterations, then we use matcher M to make prediction for any example in $C \setminus C'$ and M' for any example in C' .

Note that if the set C' is too small (e.g., having less than 200 examples), or if no significant reduction happens (e.g., $|C'| \geq 0.9 * |C|$), then we terminate without learning a new matcher M' for C' .

8. ENGAGING THE CROWD

As described so far, Corleone heavily uses crowdsourcing. In particular, it engages the crowd to label examples, to (a) supply training data for active learning (in blocking and matching), (b) supply labeled data for accuracy estimation, and (c) evaluate rule precision (in blocking, accuracy estimation, and locating difficult examples). We now describe how Corleone engages the crowd to label examples, highlighting in particular how we address the challenges of noisy crowd answers and example reuse.

1. Crowdsourcing Platforms: Currently we use Amazon’s Mechanical Turk (AMT) to label the examples. However we believe that much of what we discuss here will also carry over to other crowdsourcing platforms. To label a batch of examples, we organize them into HITs (i.e., “Human Intelligence Tasks”), which are the smallest tasks that can be sent to the crowd. Crowd often prefer many examples per HIT, to reduce their overhead (e.g., the number of clicks). Hence, we put 10 examples into a HIT. Within each HIT, we convert each example (x, y) into a question “does x match y ?”. Figure 9 shows a sample question. Currently we pay 1-2 pennies per question, a typical pay rate for EM tasks on AMT.

2. Combining Noisy Crowd Answers: Several solutions have been proposed for combining noisy answers, such as golden questions [15] and expectation maximization [11]. They often require a large number of answers to work well,



Do these products match?		
	Product 1	Product 2
Product image		
Brand	Kingston	Kingston
Name	Kingston HyperX 4GB Kit 2 x 2GB ...	Kingston HyperX 12GB Kit 3 x 4GB ...
Model no.	KHX1800C9D3K2/4G	KHX1600C9D3K3/12GX
Features	o Memory size 4 GB o 2 x 2GB 667 MHz ...	o 3 x 4 GB 1600 MHz o HyperX module with ...
<input type="button" value="Yes"/> <input type="button" value="No"/> <input type="button" value="Not sure"/>		

Figure 9: A sample question to the crowd.

and it is not yet clear when they outperform simple solutions, e.g., majority voting [26]. Hence, we started out using the 2+1 majority voting solution: for each question, solicit two answers; if they agree then return the label, otherwise solicit one more answer then take the majority vote. This solution is commonly used in industry and also by recent work [9, 32, 17].

Soon we found that this solution works well for supplying training data for active learning, but less so for accuracy estimation and rule evaluation, which are quite sensitive to incorrect labels. Thus, we need a more rigorous scheme than 2+1. We adopted a scheme of “strong majority vote”: for each question, we solicit answers until (a) the number of answers with the majority label minus that with the minority label is at least 3, or (b) we have solicited 7 answers. In both cases we return the majority label. For example, 4 positive and 1 negative answers would return a positive label, while 4 negative and 3 positive would return negative.

The strong majority scheme works well, but is too costly compared to the 2+1 scheme. So we improved it further, by analyzing the importance of different types of error, then using strong majority only for the important ones. Specifically, we found that false positive errors (labeling a true negative example as positive) are far more serious than false negative errors (labeling a true positive as negative). This is because false positive errors change n_{ap} , the number of actual positives, which is used in estimating $R = n_{tp}/n_{ap}$ and in Formula 3 for estimating ϵ_r . Since this number appears in the *denominators*, a small change can result in a big change in the error margins, as well as estimated R and hence F_1 . The same problem does not arise for false negative errors. Based on this analysis, we use strong majority voting only if the current majority vote on a question is positive (thus can potentially be a false positive error), and use 2+1 otherwise. We found empirically that this revised scheme works very well, at a minimal overhead compared to the 2+1 scheme.

3. Re-using Labeled Examples: Since Corleone engages the crowd to label at many different places (blocking, matching, estimating, locating), we cache the already labeled examples for reuse. When we get a new example, we check the cache to see if it is there and has been labeled the way we want (i.e., with the 2+1 or strong majority scheme). If yes then we can reuse without going to the crowd.

Interestingly this simple and obviously useful scheme poses

Datasets	Table A	Table B	# of Matches
Restaurants	533	331	112
Citations	2616	64263	5347
Products	2554	22074	1154

Table 1: Data sets for our experiment.

complications in how we present the questions to the crowd. Recall that at any time we typically send 20 examples, packed into two HITs (10 questions each), to the crowd. What happens if we find 15 examples out of 20 already in the cache. It turns out we cannot send the remaining 5 examples as a HIT. Turkers avoid such “small” HITs because they contain too few questions and thus incurs a high relative overhead.

To address this problem, we require that a HIT always contains 10 questions. Now suppose that k examples out of 20 have been found in the cache and $k \leq 10$, then we take 10 example from the remaining $20 - k$ examples, pack them into a HIT, ask the crowd to label, then return these 10 plus the k examples in the cache (as the result of labeling this batch). Otherwise if $k > 10$, then we simply return these k examples as the result of labeling this batch (thus ignoring the $20 - k$ remaining examples).

9. EMPIRICAL EVALUATION

We now empirically evaluate *Corleone*. Table 1 describes three real-world data sets for our experiments. Restaurants matches restaurant descriptions. Citations matches citations between DBLP and Google Scholar [14]. These two data sets have been used extensively in prior EM work (Section 9.1 compares published results on them with that of *Corleone*, when appropriate). Products, a new data set created by us, matches electronics products between Amazon and Walmart. Overall, our goal is to select a diverse set of data sets, with varying matching difficulties.

We used Mechanical Turk and ran *Corleone* on each data set three times, each in a different week. The results reported below are averaged over the three runs. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. We paid 1 cent per question for Restaurants & Citations, and 2 cents for Products (it can take longer to answer Product questions due to more attributes involved).

9.1 Overall Performance

Accuracy and Cost: We begin by examining the overall performance of *Corleone*. The first five columns of Table 2 (under “*Corleone*”) show this performance, broken down into P , R , F_1 , the total cost, and the total number of tuple pairs labeled by the crowd. The results show that *Corleone* achieves high matching accuracy, 89.3-96.5% F_1 , across the three data sets, at a reasonable total cost of \$9.2-256.8. The number of pairs being labeled, 274-3205, is low compared to the total number of pairs. For example, after blocking, Products has more than 173,000 pairs, and yet only 3205 pairs need to be labeled, thereby demonstrating the effectiveness of *Corleone* in minimizing the labeling cost.

The total number of pairs labeled is lowest for Restaurants, followed by Citations, and Products. This can be attributed to three factors:

1. Restaurants is small enough not to trigger blocking, and thus avoids the blocking cost.
2. Restaurants and Citations are both *relatively easier* to

match compared to Products, i.e., they have less diverse matching pairs. As a result, they require fewer training examples to achieve similar matching accuracy.

3. Being harder to match, Products is much harder to reduce during the estimation step, e.g., Restaurants requires only 1 reduction rule during estimation, while Products requires an average of 16 rules. It also has a lower positive density than Citations. Thus, it requires many more labeled pairs to estimate precision and recall, than the other two datasets.

Run time: The total run times for *Corleone* are 2.3 hours, 2.5 days and 2.1 days for Restaurants, Citations and Products datasets respectively. To understand the run times for each of the components of *Corleone*, let us focus on Products. Here, *Corleone* takes 2.5 hours for blocking, 1.4 days for learning, 14 hours for estimation and 1.4 hours for reduction. If we exclude the crowd time, then the runtimes for *Corleone* over the three datasets are 12 seconds, 12.4 minutes and 49 minutes respectively. The total machine time taken to compute entropy for all the examples in the candidate set is only 2 minutes, which is negligible compared to the overall run time. This clearly shows that time spent to obtain labels from the crowd dominates the run time.

Comparison to Traditional Solutions: In the next step, we compare *Corleone* to two traditional solutions: Baseline 1 and Baseline 2. Baseline 1 uses a developer to perform blocking, then trains a random forest using the same number of labeled pairs as the average number of labeled pairs used by *Corleone*. Baseline 2 is similar to Baseline 1, but uses 20% of the candidate set (obtained after blocking) for training. For example, for Products, Baseline 1 uses 3205 pairs for training (same as *Corleone*), while Baseline 2 uses 20% * 180,382 = 36,076 pairs, more than 11 times what *Corleone* uses. Baseline 2 is therefore a very strong baseline matcher.

The next six columns of Table 2 show the accuracy (P , R , and F_1) of Baseline 1 and Baseline 2. The results show that *Corleone* significantly outperforms Baseline 1 (89.3-96.5% F_1 vs. 7.6-87.1% F_1), thereby demonstrating the importance of active learning, as used in *Corleone*. *Corleone* is comparable to Baseline 2 for Restaurants and Citations (92.1-96.5% vs. 92.0-96.4%), but significantly outperforms Baseline 2 for Products (89.3% vs. 69.5%). This is despite the fact that Baseline 2 uses 11 times more training examples.

Baseline 1 uses passive learning (i.e. training examples are randomly sampled once at the beginning), while *Corleone* uses active learning, selecting the training examples iteratively, and only those judged informative are added to the training set, until it has a satisfactory matcher. This explains why Baseline 1 performs a lot worse than *Corleone*, in spite of using the same number of labeled examples. For Restaurants, Baseline 1 does especially worse due to the extremely low positive density (0.06%) in the candidate set, resulting in very few (or no) positive example in the training set.

Baseline 2 also uses passive learning, but with a significantly larger training set. This explains the improved performance of Baseline 2 over Baseline 1. On Products, however, Baseline 2 does not fare very well compared to *Corleone*. This again has to do with Products dataset being *harder to match*, i.e., requiring a larger and more diverse training set.

When comparing Baseline 1 and Baseline 2 against *Corleone*, it is important to note that *Corleone* not only returns

Datasets	Corleone					Baseline 1			Baseline 2			Published Works
	P	R	F_1	Cost	# Pairs	P	R	F_1	P	R	F_1	F_1
Restaurants	97.0	96.1	96.5	\$9.2	274	10.0	6.1	7.6	99.2	93.8	96.4	92-97 [27, 13]
Citations	89.9	94.3	92.1	\$69.5	2082	90.4	84.3	87.1	93.0	91.1	92.0	88-92 [14, 13, 3]
Products	91.5	87.4	89.3	\$256.8	3205	92.9	26.6	40.5	95.0	54.8	69.5	Not available

Table 2: Comparing the performance of Corleone against that of traditional solutions and published works.

the matched results, but also the estimated precision and recall, while Baseline 1 and Baseline 2 do not report any estimates for accuracy.

Comparison to Published Results: The last column of Table 2 shows F_1 results reported by prior EM work for Restaurants and Citations. On Restaurants, [13] reports 92-97% F_1 for several works that they compare. Furthermore, CrowdER [27], a recent crowdsourced EM work, reports 92% F_1 at a cost of \$8.4. In contrast, Corleone achieves 96.5% F_1 at a cost of \$9.2 (including the cost of estimating accuracy). On Citations, [14, 13, 3] report 88-92% F_1 , compared to 92.1% F_1 for Corleone. It is important to emphasize that due to different experimental settings, the above results are not directly comparable. However, they do suggest that Corleone has reasonable accuracy and cost, while being hands-off.

Summary: The overall result suggests that Corleone achieves comparable or in certain cases significantly better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost. The important advantage of Corleone is that it is totally hands-off, requiring no developer in the loop, and it provides accuracy estimates of the matching result.

9.2 Performance of the Components

We now “zoom in” to examine Corleone in more details.

Datasets	Cartesian Product	Umbrella Set	Recall (%)	Cost	# Pairs
Restaurants	176.4K	176.4K	100	\$0	0
Citations	168.1M	38.2K	99	\$7.2	214
Products	56.4M	173.4K	92	\$22	333

Table 3: Blocking results for Corleone.

Blocking: Table 3 shows the results for crowdsourced automatic blocking executed on the three data sets. From left to right, the columns show the size of the Cartesian product (of tables A and B), the size of the umbrella set (i.e., the set after applying the blocking rules), recall (i.e., the percentage of positive examples in the Cartesian product that are retained in the umbrella set), total cost, and total number of pairs being labeled by the crowd. Note that Restaurants is relatively small and hence does not trigger blocking.

The results show that automatic crowdsourced blocking is quite effective, reducing the total number of pairs to be matched to be just 0.02-0.3% of the original Cartesian product, for Citations and Products. This is achieved at a low cost of \$7.2-22, or just 214-333 examples having to be labeled. In all the runs, Corleone applied 1-3 blocking rules. These rules have 99.9-99.99% precision. Finally, Corleone also achieves high recall of 92-99% on Products and Citations. For comparison purposes, we asked a developer well versed in EM to write blocking rules. The developer achieved 100% recall on Citations, reducing the Cartesian product to 202.5K pairs (far higher than our result of 38.2K pair). Blocking on Products turned out to be quite difficult, and the developer achieved a recall of 90%, compared to our

result of 92%. Overall, the results suggest that Corleone can find highly precise blocking rules at a low cost, to dramatically reduce the original Cartesian products, while achieving high recall.

Performance of the Iterations: Table 4 shows Corleone’s performance per iteration on each data set. To explain, consider for example the result for Restaurants (the first row of the table). In Iteration 1 Corleone trains and applies a matcher. This step uses the crowd to label 140 examples, and achieves a true F_1 of 96.5%. Next, in Estimation 1, Corleone estimates the matching accuracy in Iteration 1. This step uses 134 examples, and produces an estimated F_1 of 96% (very close to the true F_1 of 96.5%). Next, in Reduction 1, Corleone identifies the difficult pairs and comes up with 157 such pairs. It uses no new examples, being able to re-use existing examples. At this point, since the set of difficult pairs is too small (below 200), Corleone stops, returning the matching results of Iteration 1.

The result shows that Corleone needs 1-2 iterations on the three data sets. The estimated F_1 is quite accurate, always within 0.5-5.4% of true F_1 . Note that sometimes the estimation error can be larger than our desired maximal margin of 5% (e.g., Estimation 2 for Products). This is due to the noisy labels from the crowd. Despite the crowd noise, however, the effect on estimation error is relatively insignificant. Note that the iterative process can indeed lead to improvement in F_1 , e.g., by 3.3% for Products from the first to the second iteration (see more below). Note further that the cost of reduction is just a modest fraction (3-10%) of the overall cost.

Crowd Workers: For the end-to-end solution, an average of 22 (Restaurants) to 104 (Citations) turkers worked on our HITs. The average accuracy of turkers was the highest for Restaurants (94.7%), and the lowest for matching citations (75.9%). For Products, it was again quite high (92.4%), which is understandable given the familiarity of turkers with products as opposed to citations. The accuracy of the labels inferred by Corleone (using majority voting) was higher than the average turker accuracy for all the datasets (96.3% for Restaurants, 77.3% for Citations, and 96% for Products). Note that in spite of the low labeling accuracy for Citations, Corleone still performs just as good as the traditional solutions and published works.

9.3 Additional Experimental Results

We have run a large number of additional experiments to extensively evaluate Corleone.

Estimating Matching Accuracy: Section 9.2 has shown that our method provides accurate estimation of matching accuracy, despite noisy answers from real crowds. Compared to the baseline accuracy estimation method in Section 6.1, we found that our method also used far fewer examples. We now compare the average cost (i.e., the number of pairs labeled) of the two methods. For a fair comparison, we use a simulated crowd that labels everything perfectly for both

Datasets	Iteration 1				Estimation 1				Reduction 1		Iteration 2				Estimation 2			
	# Pairs	P	R	F_1	# Pairs	P	R	F_1	# Pairs	Reduced Set	# Pairs	P	R	F_1	# Pairs	P	R	F_1
Restaurants	140	97	96.1	96.5	134	95.6	96.3	96	0	157								
Citations	973	89.4	94.2	91.7	366	92.4	93.8	93.1	213	4934	475	89.9	94.3	92.1	0	95.2	95.7	95.5
Products	1060	89.7	82.8	86	1677	90.9	86.1	88.3	94	4212	597	91.5	87.4	89.3	0	96	93.5	94.7

Table 4: Corleone’s performance per iteration on the data sets.

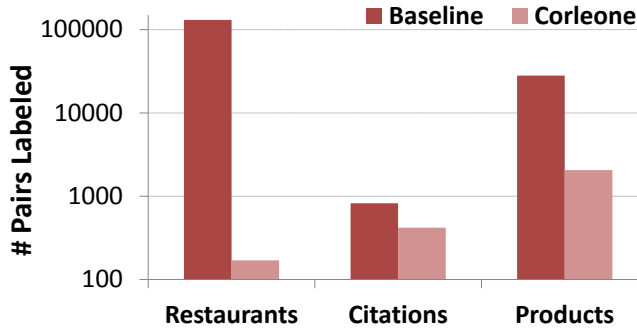


Figure 10: Comparing estimation cost of Corleone vs. Baseline.

the methods, and we start the estimation procedure without any cached labels.

Figure 10 shows the number of pairs labeled for estimation for all three datasets, for the both the methods (*Baseline*) and *Corleone*). For Restaurants, the baseline method needs 100,000+ examples to estimate both P and R within a 0.05 error margin, while ours uses just 170 examples. For Citations and Products, we use 50% and 92% fewer examples, respectively. The result here is not as striking as for Restaurants primarily because of the much higher positive density for Citations and Products.

Effectiveness of Reduction: Section 9.2 has shown that the iterative matching process can improve the overall F_1 , by 0.4-3.3% in our experiments. This improvement is actually much more pronounced over the set of difficult-to-match pairs, primarily due to increase in recall. On this set, recall improves by 3.3% and 11.8% for Citations and Products, respectively, leading to F_1 increases of 2.1% and 9.2%. These results suggest that in subsequent iterations *Corleone* succeeds in zooming in and matching correctly more pairs in the difficult-to-match set, thereby increasing recall.

Note that this increase in recall is a lot more pronounced for Products (11.8%) than for Citations (3.3%). This is mainly due to the lower positive density for Products (1.9% compared to 21.4%). The lower positive density results in fewer positives getting selected in the training set in the first iteration, and thus, a less representative training set. In the second iteration, we narrow down to a set with a much higher positive density, and thus, many of these previously unrepresented positives get added to the training set which leads to a higher recall.

Effectiveness of Rule Evaluation: Section 9.2 has shown that blocking rules found by *Corleone* are highly precise (99.9-99.99%). We have found that rules found in later steps (estimation, reduction, i.e., identifying difficult-to-match pairs) are highly precise as well, at 97.5-99.99%. For the estimation step, *Corleone* uses 1, 4.33, and 7.67 rules on average (over three runs) for Restaurants, Citations, and Products, respectively. For the reduction step, Citations uses on average 11.33 negative rules and 16.33 positive rules, and Products uses 17.33 negative rules and 9.33 positive rules.

The number of rules used during estimation depends on

the ease of matching the dataset and the positive density, e.g., for Restaurants, which is the easiest to match among the three, just one rule is sufficient for the estimation procedure, while Products, which is the most difficult to match among the three, and has lower positive density than Citations, requires more rules (7.67) for the estimation procedure.

For the rules used in reduction step, the average accuracy is still very high (97.5% and above), but a little lower than that for estimation (99.9% and above). This is because we select only the topmost precise rules for estimation (since we need near-perfect rules here). For reduction, there is no such necessity, and thus, we consider even the not-so-precise rules.

Finally, if we look at the number of rules used, then we see a similar trend as for estimation rules, except for the positive rules. For Products, only 9.33 precise positive rules are applied during reduction, whereas for Citations we apply 16.33 positive rules. This is because Citations has almost 5 times the total number of positive examples as Products (5347 vs. 1154). With more positives, we get more positive rules, and thus, higher number of precise positive rules. Overall, the crowdsourced rule evaluation works extremely well to give us almost perfect rules.

Using Corleone up to a Pre-specified Budget: *Corleone* runs until the estimated matching accuracy no longer improves. However, the user can stop it at any time. In particular, he or she can run *Corleone* only until a pre-specified budget for crowdsourcing has been exhausted, an operation mode likely to be preferred by users in the “masses”, with modest crowdsourcing budgets. We found that even with a modest budget, *Corleone* already delivers reasonable results (which improve steadily with more money). In the case of Products, for instance, a budget of \$50, \$150, and \$250.9 (the end) delivers F_1 score of 79.1%, 85.9%, and 88.5%, respectively. Table 5 shows the detailed execution status of *Corleone* for Products, from start (\$0) to finish (\$250.9).

9.4 Sensitivity Analysis

Each of the components of *Corleone* has some parameters that can be used to fine tune the performance. We have run extensive sensitivity analyses for *Corleone* to test the robustness of the system. Overall we observe that small changes in these parameters do not affect the performance of *Corleone* in any significant way. However, one can certainly tune them to extract the best performance out of the system. We report here the results for the most important factors that may affect the performance of *Corleone*. For all the sensitivity analyses, we performed experiments on the Products dataset, since that is the most difficult one to match, as can be seen from the results reported earlier. Additionally, we used a simulated crowd for all the experiments, since performing so many experiments with real crowd is prohibitively expensive and time consuming.

Blocking Threshold t_B : The effect of t_B is most pronounced in the blocking stage, so we vary t_B from 1 million

Money	\$50	\$100	\$150	\$200	End (\$250.9)
True P/R/F1	79.8/78.4/79.1	86.6/85.2/85.9	86.6/85.2/85.9	86.6/85.2/85.9	89.1/88/88.5
Execution status	Iteration#1	Estimation#1	Estimation#1	Reduction#1	Finished
Additional info.	Finished blocking at cost = \$22.44 in 3.1 hours. So far spent \$27.04 on matching, to create a training set with 415 examples. Total time = 9.4 hours.	Finished matching iteration#1 at cost = \$97.68. Estimated positive density = 4.9%. Currently evaluating rules to reduce the universe. No P/R estimates yet. Total time = 24.6 hours.	Finished reduction. So far spent \$51.48 on estimation. 535 labeled examples used for estimation. Est. P = 89.8 ± 7.5%, Est. R = 86.2 ± 8.6%. Total time = 29.6 hours.	Finished estimation #1 at total cost = \$197.03. Currently reducing the input set for matching iteration#2. Est. P = 91 ± 4.2%, Est. R = 88.7 ± 4.9%. Total time = 34.8 hours.	Finished matching iteration#2 and estimation#2. Stopped since training set contains > 25% of the input set. Est. P = 95.5 ± 3%, Est. R = 94.4 ± 3.6%. Total time = 57.2 hours.

Table 5: Execution status of Corleone at different time points during one particular run on Products.

to 20 million to see how it affects the blocking time, size of the candidate set, and the recall of the candidate set.

Effect of t_B on the blocking time: The total blocking time can be broken down into 4 main components:

- Sampling and feature vector generation time (t_1)
- Rule learning time (t_2)
- Rule evaluation time (t_3)
- Applying the rule on the Cartesian product time (t_4)

t_1 , t_2 and t_3 are directly proportional to t_B because higher the t_B , larger is the sample size, longer it takes to sample and compute features and longer it takes to learn and evaluate rules. From our plots we observe that $t_1 + t_2 + t_3$ increases linearly from 3m 43s to 1h 15m 40s as we increase t_B from 1M to 20M. t_4 , however, does not directly depend on t_B but depends on the blocking rule applied. So we do not observe any particular trend for t_4 as we vary t_B . The highest t_4 is 34m 49s, the lowest is 11m 38s and the average is 21m 2s.

Effect of t_B on the size of the candidate set: We observe that, on increasing t_B the candidate set size increases in the average case. For example, we obtain candidate sets of sizes 22.4K, 247K and 3.8M for $t_B = 1M$, 3M and 5M respectively. However, note that the size of the candidate set depends on the blocking rule that was applied. Thus, in certain cases, this trend may not be strictly followed, as we observe for $t_B = 10M$ for which we have a candidate set of size 583.6K.

Effect of t_B on the recall of the candidate set: In general, larger the candidate set higher is the recall. We observe this behavior in our experiments where we have recalls of 92.63%, 95.67% and 99.13% for candidate sets of sizes 247K, 3.8M and 5.9M respectively. For a very small t_B of 1 million, we get a recall of 82.8%, otherwise the recall is in the range 92.63% to 99.31%.

Number of Trees (k) and Features (m) in Random Forest: Section 5.1 describes the random forest ensemble model and its parameters: the number of trees in the forest (k) and the number of features (m) considered for splitting at each node in the tree. These parameters can only affect the training step, hence, to understand the effect of varying these parameters on Corleone, we only execute the training step in the workflow. In particular, we start with a candidate set returned by the blocking step, and the 4 user-provided

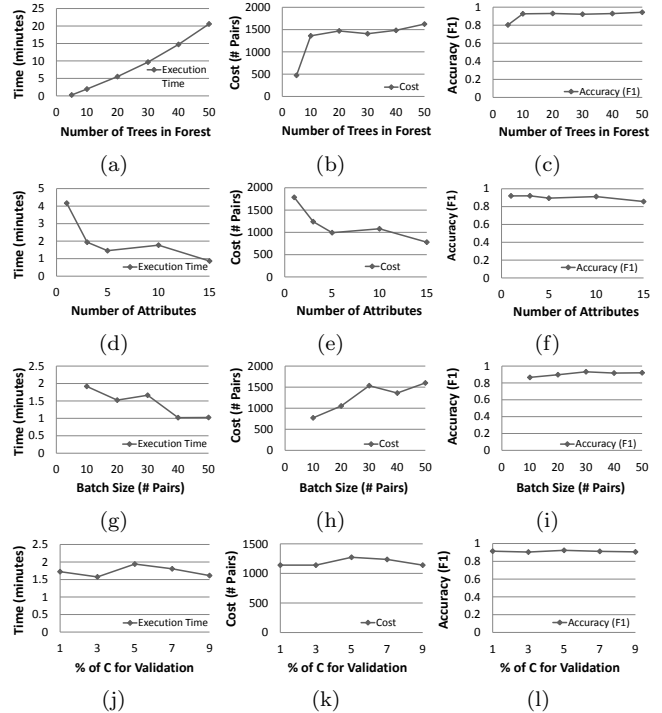


Figure 11: Sensitivity analysis for parameters in learning.

labeled pairs, and then perform active learning over the candidate set until the stopping condition kicks in.

Figures 11a, 11b, and 11c show the effect of varying the number of trees (k) in the forest from 5 to 100, on the execution time, cost, i.e., the number of pairs labeled for training, and accuracy (F_1 score). We observe that the execution time grows linearly as we increase k , from 1 minute for 5 trees to 20 minutes for 50 (Figure 11a). This is expected since for every new tree added, the forest needs to apply one more tree to classify a given pair. The increasing number of trees also lead to a better F_1 , however, the increase in F_1 is significant (12%) only as we go from 5 to 10 trees. Going from 10 to 50 trees in the forest, the F_1 increases by just 2% (Figure 11c). This marginally higher accuracy comes at a higher labeling cost as well (Figure 11b). From 5 to 10 trees, the labeling cost rises from 500 to almost 1400, but beyond this,

the cost rises at a very slow pace, going up by just 200 as we go from 10 to 50 trees. Overall, we can see that at $k = 10$, the execution time is very small (2 minutes), while the F_1 is almost as good as that for $k = 50$. Thus, the default value of $k = 10$ used in *Corleone* is highly justified.

The small rise in F_1 , and cost as we increase the number of trees can be explained. By increasing the number of trees, we get a more diverse ensemble (forest), since each tree is trained on a different portion of the training data. This results in a higher F_1 . However, more diverse trees also have more disagreement (higher entropy), and thus, the confidence of the whole forest takes longer to stabilize leading to a higher labeling cost.

Next, we describe the effect of increasing the number of features, i.e., attributes (m) considered at each node when learning the trees in the forest. Figures 11d, 11e, and ?? plot the execution time, labeling cost, and F_1 as we increase number of attributes m from 1 to 15. Note that for Products dataset, *Corleone* sets $m = 5$, since the total number of attributes n for Products is 23, and m is set to $\log(n) + 1$.

We found that on increasing m , the execution time goes down from 4 minutes for $m = 1$, to 1.4 minutes for $m = 5$, and then decreases very slowly beyond this value of m (Figure 11d). The labeling cost goes down by more than 50% from $m = 1$ to $m = 5$, and then reduces by just 20% as m goes to 15. This reduction in time and cost, comes at a cost. The F_1 drops by 7% on increasing m from 1 to 15. With a higher m , we get a less diverse ensemble, and thus, the confidence converges sooner leading to stopping active learning sooner. This results in a lower cost as well as lower F_1 . Note that at $m = 5$, which is the default in *Corleone* the F_1 is within 2% of that for $m = 1$, while both execution time and cost are much lower than for $m = 1$.

Batch Size (q) for Active Learning: In all the experiments reported so far we had set the batch size q to 20. We examine the effect of batch size on the active learning step by executing only the training step, exactly as done for varying k and m above.

Figures 11g, 11h, and 11i show the effect of varying the batch size (q) from 10 to 50, on execution time, the labeling cost (# examples) for learning, and F_1 score. On increasing q from 10 to 50, we found that the execution time reduced from 2 to 1 minutes, since the algorithm required fewer learning iterations. The number of iterations dropped by 55% from 77 to 32. This is expected since the algorithm can learn more in each iteration by selecting a bigger batch. The labeling cost, however, increased by more than 100%, from 770 for $q = 10$, to 1600 for $q = 50$, while there is a small rise in F_1 from 86% to 92%. At $q = 20$, which is the default in *Corleone* we observe that F_1 is again within 2% of the maximum, while the labeling cost is within 20% of the best attainable. Given that there is a trade-off between cost, F_1 , and time, we choose 20 as our “sweet spot” for batch size.

Size of Validation Set to Decide Stopping: The active learning algorithm sets aside part of the candidate set as validation set, to decide when to stop (as described in 5.3). *Corleone* uses 3% of the candidate set as validation set. To examine the effect of varying the size of validation set, we perform just the training step over the candidate set C , as done above for k , m , and q , and vary the validation set size from 1% of candidate set (C) to 9% of C . Figures 11j, 11k, and 11l show the effect of increasing the percentage of C set

aside for validation.

Overall, we observe that increasing the size of validation set has almost no effect on time, cost, or F_1 . The execution time stays within 1.6 and 1.9 minutes as we increase t_B . The labeling cost stays within 1140 and 1240, while F_1 stays in the range of 90% and 92%. Thus, we observe that the active learning algorithm is quite robust to a change in the size of validation set.

Parameters Used for Rule Pruning: Section 4.2 describes how we use P_{min} and maximum number of rules (k), to select at most top k rules (currently, $k = 20$). In blocking step, the number of rules that are finally applied is no more than 3 in all our experiments with the three datasets, thus, varying k from 10 to 50 had no effect on the system.

When evaluating the top rules, we use P_{min} as a threshold to remove the imprecise rules. On varying P_{min} from 90% to 99%, we did not observe any change in the rules that get picked, and thus, on the whole system. This can be explained by the fact that in all our experiments the top rules that got evaluated for precision were either highly precise (precision being higher than 99% for most), or had a much lower (less than 80%) precision. These low precision rules may get picked when we fail to get a tight upper bound on their precision. These factors could have more significant effect on other datasets. We would consider exploring this in future work.

Labeling Accuracy of the Crowd: To test the effect of labeling accuracy, we use the random worker model in [11, 10] to simulate a crowd of random workers with a fixed error rate (i.e., the probability of incorrectly labeling an example). We found that a small change in the error rate causes only a small change in *Corleone*’s performance. However, as we vary the error rate over a large range, the performance can change significantly. With a perfect crowd (0% error rate), *Corleone* performs extremely well on all three data sets. With moderate noise in labeling (10% error rate), F_1 reduces by only 2-4%, while the cost increases by up to \$20. As we move to a very noisy crowd (20% error rate), F_1 further dips by 1-10 % for Products and Citations, and 28% for Restaurants. The cost on the other hand shoots up by \$250 to \$500. Managing crowd’s error rates better therefore is an important topic for future research.

Number of Labels Per Pair: In Section 8, we mentioned that we use 2+1 labeling scheme during the training phase, to keep the cost low.

For the training step, we only get a maximum of 3 labels per pair. We now analyze the effect of using more labels per pair during training. The number of labels requested per pair is of value only in presence of a noisy crowd. Hence, to analyze the effect of per-pair labels, we used the low accuracy simulated crowd (20% error rate). This crowd performed especially worse on Restaurants. Hence, we report here the results for increasing the number of labels for Restaurants, with a crowd having 20% error rate.

On increasing the maximum requested labels from 3 to 5 to 7, we found that the F_1 improved significantly from 70% to 97%, while the cost reduced by more than \$500. Intuitively, the cost reduces drastically because the quality of inferred labels improves a lot, which in turn, leads to a quick termination of the active learning algorithm. This experiment demonstrates a little non-intuitive fact that with a noisy crowd, getting more labels could not only give better

performance, but sometimes it can also help to drastically lower the total cost.

9.5 Evaluating and Setting System Parameters

Finally, we discuss how we evaluated and set system parameters. In the blocker, t_B is set to be the maximal number of tuple pairs that can fit into memory (a heuristic used to speed up active learning during blocking), and is currently set to 3 million, based on the amount of memory available on our machine. We have experimented and found that Corleone is robust to varying t_B (e.g., as we increase t_B , the time it takes to learn blocking rules increases only linearly, due to processing larger samples). See Section 9.4 for details.

The batch size $b = 20$ is set using experimental validation with simulated crowds (of varying degrees of accuracy). The number of rules k is set to a conservative value that tries to ensure that the blocker does not miss any good blocking rules. Our experiments show that k can be set to as low as 5 without affecting accuracy. Similarly, experiments suggest we can vary P_{min} from 0.9 to 0.99 without noticeable effects, because the rules we learned appear to be either very accurate (at least .99 precision) or very inaccurate (well below 0.9). Given this, we current set P_{min} to 0.95. The confidence interval 0.95 and error margin 0.95 are set based on established conventions.

In the matcher, the parameters for the random forest learner are set to default values in the Weka package. The stopping parameters (validation set size, smoothing window w , etc.) are set using experiments with simulated crowds with varying degrees of accuracy.

For engaging the crowd, we solicit 3 labels per pair because 3 is the minimum number of labels that give us a majority vote, and it has been used extensively in crowdsourcing publications as well as in industry. When we need higher crowd accuracy for the estimator, we need to consider 5 labels or 7 labels. After extensive experiments with simulated crowds, we found that 5 gave us too wide error ranges, whereas 7 worked very well (for the estimator). Hence our decision to solicit 7 labels per pair in such cases. Finally, the estimator and the difficult pairs’ locator use many algorithms used by the blocker and the matcher, so their parameters are set as described above.

10. DISCUSSION

In this section we discuss the key design choices that went into building Corleone, the scope of our current work, and various opportunities to extend the current system.

10.1 Design Choices

Our goal behind building Corleone was to (i) have a first end-to-end HOC system for the entity matching problem that performs well on real datasets, thus, demonstrating the immense potential in HOC, and (ii) have the first starting point for further research in building HOC systems. As a result, when designing Corleone and its components, our focus was on having a practical solution that actually works with real data and real crowd, and keeping it simple unless absolutely necessary. At the same time, we wanted the overall architecture to be very general, so that each of the components can be improved and extended.

We now take a top-down look at the choices we have made while designing the Corleone system. At the very top, Corleone is aimed at solving the entity matching problem using

the crowd, starting from the two input tables, all the way to returning the matching pairs and the estimated matching accuracy. The ideal goal for such a system would be to maximize the matching accuracy (F_1 score), minimize the monetary cost of crowdsourcing, and minimize the end-to-end execution time. This is a very challenging optimization problem, since it is very difficult to model the trade-off between matching accuracy, monetary cost, and execution time. This trade-off can be highly dependent on the particular application setting, e.g., a large company may be willing to pay a large monetary price for a small gain in accuracy, while a domain scientist may care a lot more about the cost than the accuracy.

Finding the globally optimal solution for this problem is an entirely different challenge in itself. As a starting point, we pursue a more modest goal focusing on maximizing the accuracy while minimizing the monetary cost. Optimizing for both accuracy and cost for the entire EM workflow is again highly non-trivial, since it is very hard to estimate the accuracy and cost of any step in the workflow before executing that particular step. To illustrate, if we could predict the accuracy and cost for the matching step when we are executing the blocking step, then we could use that knowledge to stop blocking at an optimal point. However, before we perform the actual matching step, it is very hard to predict its performance. Hence, we break down the problem, focusing on optimizing the accuracy and cost for each individual step. Now we look at some key choices made in the various steps in EM workflow.

Why Need a Threshold for Blocking? We trigger blocking only if the size of the Cartesian product ($A \times B$) is above a threshold (t_B) (as described in Section 4.1). Having such a threshold has to do with the fundamental reason blocking is needed in the first place, which is to execute the EM workflow in an *acceptable* amount of time.

Building and applying a matching solution is often computationally expensive, e.g., if we are learning a classifier, then we need to enumerate all the possible pairs that may match, compute all the features for all the pairs, train the classifier using the labeled pairs, and then apply the classifier to predict each pair as matching or non-matching. If we have to do this for the entire $A \times B$, then for very large tables, it could take several days to even weeks to execute, even on a cluster of machines. Blocking is just a fast solution that reduces the original input to a small size on which we can apply the expensive matching solution. However, blocking comes at a cost as it is typically not as accurate as matching. Thus, intuitively one would want to perform blocking *only if* applying matching is going to be prohibitively expensive. If the Cartesian product is small enough that we can execute matching within an acceptable time, then it would certainly be better than to block first and risk losing some matching pairs. To model this notion of “small enough”, we use a parameter t_B that represents the threshold for blocking.

Setting the Blocking Threshold: The blocking threshold limits the size of the input to the matching step, and in turn, limits the execution time. Thus, intuitively, we should set the blocking threshold as small as we can to minimize the execution time for the matching step. On the other hand, the smaller the threshold we set, the more pairs we would need to eliminate during blocking, i.e., more matching pairs will be lost in blocking. To balance these conflicting goals,

we look further into the components that dominate the execution time for matching.

The execution time T for the matching step (Section 5), can be expressed as $T = n \cdot T_{iter}$, where n is the number of iterations of active learning, T_{iter} is the execution time for each iteration. Now T_{iter} is dominated by t_1 , the time to select the next batch of pairs for training, and t_2 , the time for labeling the selected pairs. Now t_2 is constrained by the batch size and the maximum number of labels we can request per pair and is independent of the threshold we set. However, t_1 is very much dependent on the threshold we set, since t_1 involves computing the entropy for every pair in the candidate set. t_1 involves only the CPU cost if the feature vectors for the candidate set fit in memory and thus, is relatively small compared to t_2 . However, if the feature set is larger, then t_1 also has an I/O component, which can get significant for large candidate sets. To balance our goals of (i) keeping t_B small to constrain the execution, while (ii) keeping it large enough to avoid loss in matching accuracy, we set t_B such that the feature vectors of the candidate set fit in memory, which avoids I/O cost, and constrains the execution time for matching.

A few brief remarks about the above strategy. First, we set the goal of trying to obtain a candidate set of size t_B , and we try to get as close to that goal as possible. But we cannot guarantee that we will have a candidate set size of t_B or less. For example, if no blocking rule is good, then we cannot perform any blocking and we would still have a candidate set size of $|A \times B|$. Second, the above strategy is just a reasonable heuristic; other strategies are possible and should be explored in future work. Finally, setting the value of t_B depends on the computational infrastructure used for matching, e.g., if we are performing active learning over a very large cluster then we could set t_B to a much larger value without affecting the execution time. A more principled solution to determine the blocking threshold can be an interesting direction for future work.

Why Sample to Learn Blocking Rules? As far as we know, ours is the first work that learns the blocking rules starting from scratch, i.e. just the input tables, without requiring a developer. To learn such rules we need some labeled examples to train on. To obtain such training examples in a cost minimizing fashion, a natural choice is to use active learning. However, if we were to use the entire $A \times B$ as the input to the active learning algorithm (which is the same as the one used in the matching step), then the learning process will be prohibitively expensive. This defeats the very purpose of blocking. Hence, we take the approach of sampling a small set of pairs from $A \times B$, and using only this sample to learn the blocking rules. This way we constrain the cost as well as execution time for blocking.

Setting the Sample Size: Given that we take the sampling approach to constrain the cost and execution time for blocking, we should minimize the sample size as much as we can. On the other hand, to learn effective rules, we need to have a representative sample with a “sufficient” number of positives in it. To illustrate, if our sample contains zero positive examples, then any rule will have 100% accuracy on the sample and we will have no way to judge which rule is more effective. Thus, we also need to have a “large enough” sample. These conflicting goals are very similar to what we faced when setting the blocking threshold. In fact, just like

the matching step, the execution time for blocking is also dominated by the crowdsourced active learning algorithm. Thus, following similar reasoning we set the sample size to t_B such that the feature vectors for the sample just fit in memory.

How Many Labels to Obtain per Pair?: In Section 8, we describe the labeling scheme used in *Corleone* in the Estimator component, which requires “strong majority” only if the majority label is positive. In this scheme, we get a minimum of 3 labels and a maximum of 7. Getting a minimum of 3 labels is quite straightforward as you need at least at least 3 labels to avoid a possibility of a tie. In fact, this is a standard value used in many works that use majority voting for combining crowd answers [9, 32, 17].

We limit the maximum number of labels that can be obtained per pair to 7. Now clearly we need some limit on the total labels for a single pair we may get as otherwise, in the worst case, the algorithm may never stop, and we would end up spending an exorbitant amount. Since we want to minimize the cost ideally we should set this limit as low as possible. The first choice for this limit would be 5. However, in our experiments with simulated noisy crowd, we found that getting 5 labels was not sufficient to estimate the matcher accuracy with low error, especially with a very noisy crowd (error rate of 20% or more). After increasing this limit to 7 labels, on the other hand, we found the total cost to increase only marginally by up to 100\$, while the accuracy of estimation improved significantly (by more than 10%), very close to what we would obtain with a perfect crowd. On increasing the limit further to 9 or 11 labels, the cost continues to increase, whereas there is very little gain in estimation accuracy. Based on these experiments with simulated crowd, we set the maximum number of labels to get to 7.

10.2 Opportunities for Extension

The *Corleone* system is just a first step toward building HOC systems. In this work, we have focused on only some of the many novel challenges involved in building a robust scalable HOC system for entity matching. As for any system, addressing all of the challenges in the first attempt is next to impossible, and thus, *Corleone* is designed with a clean separation between the various components so that each of them can be easily extended. We describe here some key opportunities to further extend the system. We have already started work on a few of these ideas.

10.2.1 Scaling Up to Very Large Datasets

While the *Corleone* system, as described in the paper, shows its promise through the extensive experiments on three datasets, our next goal is to ensure that it can handle datasets of any nature and size. For scaling up to very large datasets, we need to ensure that each of the components can scale up. Intuitively, the blocking component is the one that is most responsible to handle the scale problem. If blocking works the way it is supposed to, then in most cases the output of blocking should return a candidate set small enough for efficient execution of the matching and subsequent steps in the workflow. Hence, we now discuss how we can scale up the blocking component.

Sampling Strategy for Large Datasets: After the system has decided to block, the first step in blocking is to sample from the Cartesian product. The sampling strat-

Datasets	$ A \times B $	Positive density (%)	# positives in sample
Citations	168.1 M	3.2×10^{-3}	4123
Citations 2X	672.4 M	1.6×10^{-3}	4219
Citations 5X	4202.5 M	0.64×10^{-3}	4186
Citations 10X	16810 M	0.32×10^{-3}	4145

Table 6: Stratified sampling for blocking.

egy currently used (Step 2 in Section 4.2) randomly samples $t_B/|A|$ tuples from the larger table B and takes their Cartesian product with all of A . As one may suspect, this strategy may not work very well if we have very large tables or very low fraction of matching pairs in $A \times B$. In such a situation, the current sampling strategy may give us a sample with very few positives.

This possible limitation can be addressed as follows. First, we must select t_B as large as we can without hurting the execution time for matching. In our current system, we assume a very modest infrastructure for matching, and thus set t_B to 3 million. Here our goal was to demonstrate that even with very strict constraints on the size of candidate set, our solution comes up with highly precise rules. In practice, t_B can be easily increased to a few hundred million by using a cluster to speed up the active learning algorithm (as we discuss later). This will ensure that t_B is much larger than $|A|$ or $|B|$.

Secondly, we can use a non-uniform sampling strategy to select positives with a high likelihood. We already have one such strategy and our preliminary results indicate that it works very well even on large datasets or when there is very low positive density. Here is how it works. Given the sample size t_B , we randomly select t_B/m tuples from the table B ($m = 200$). For each selected tuple from B we select m tuples from table A , forming m tuple pairs, and add them to the sample. The m tuples are selected as follows. We “stratify” the tuples in A into two sets: A_1 - tuples that have at least one token in common with the B tuple, and A_2 tuples with no common token. We then randomly pick up to $m/2$ tuples from A_1 , and then randomly pick tuples from A_2 until we have a total of m tuples. At the end of this process we have the desired sample with roughly t_B tuple pairs.

To demonstrate the effectiveness of this strategy, we present in Table 6 the results for a preliminary experiment on Citations dataset. We use the new sampling strategy to sample from different versions of the Citations dataset. To test how well it scales up to large datasets, we create larger versions of the Citations dataset by replicating the tables. Thus, Citations 2X has tables A and B with twice as many tuples as in Citations, while 5X Citations has 5 times the number of tuples. For 2X Citations, $A \times B$ is 4 times that of Citations, while the positive density is half of that for Citations (since the actual number of matching pairs is only 2 times that of Citations). Thus, as we replicate Citations more and more times, the size of $A \times B$ keeps increasing while positive density keeps dropping. In Table 6, we show the number of positives selected in the sample, as we create bigger and bigger versions of the Citations dataset. As we can see, the new sampling strategy gives us a consistently high number of positives, even as the size of $A \times B$ increases and the positive density drops.

Applying the Rules Efficiently: In the current system, when a blocking rule is actually applied to eliminate the ob-

vious non-matching pairs, the rule is evaluated for every pair in $A \times B$. This may work well for now as this is computed over a Hadoop cluster, however, beyond a certain input size it will be too expensive to enumerate all the pairs. Fortunately, there is a way around this problem. Prior work on scaling up blocking has developed techniques such as sorted neighborhood, canopy clustering, and indexing to speed up the application of blocking rules [8]. We are currently working on extending the component for applying the blocking rules to incorporate these techniques.

Efficient Active Learning: The active learning algorithm for training the matcher proceeds iteratively. In each iteration, it processes all the unlabeled pairs in the candidate set and selects the most informative pairs among them (Section 5.2). This step involves computing the entropy for each unlabeled pair in the candidate set. The blocking threshold already ensures that the size of candidate set is no more than t_B . However, when scaling up to very large datasets, we might want to set t_B to a very large value (hundreds of millions).

To ensure that we can efficiently execute matching over a very large candidate set, we need a scalable solution to compute the entropy for pairs in the candidate set. Thankfully, this is a very straightforward problem to parallelize, since we can compute the entropy for each pair independently. Thus, we can easily distribute the entropy computation over a MapReduce cluster and can easily handle candidate sets with hundreds of millions of pairs. There are additional techniques we can use to further speed up this solution, e.g., as the active learning progresses, we can narrow down the set of pairs for which we need to compute the entropy in each iteration.

10.2.2 Improving Matching and Estimation

Improving the Blocking Recall: One way to improve the overall matching performance is by improving the recall for blocking. This could be achieved in two ways. First is to learn a more diverse set of blocking rules. Our current framework separates the process of generating candidate blocking rules, from the evaluation of the candidates to pick the best rules. Thus, we can easily extend the system to add new techniques to generate the rules. In addition to pulling the rules from the forest, we could use other learning techniques to obtain candidate blocking rules. We can even consider directly obtaining simple rules from the crowd, and adding them to our set of candidate rules.

Another way to improve the blocking recall is by better evaluation of the blocking rules. If we can precisely predict the number of errors each candidate rule is going to make, then we can do a better job at picking the best rules.

Improving the Estimation of Matcher Accuracy: The current solution for estimating the precision and recall of the matcher (Section 6.2) works very well, but is not perfect. It relies on the accuracy of the rules used for reduction, and also the accuracy of the labels inferred from crowd provided labels. Next step to would be to make this solution more tolerant to the errors made by the crowd, as well as the imperfection of the reduction rules.

10.2.3 Cost Models

It is highly desirable to develop cost models making the system more cost efficient. For example, given a monetary

budget constraint, how to best allocate it among the blocking, matching, and accuracy estimation step? As another example, paying more per question often gets the crowd to answer faster. How should we manage this money-time trade-off? A possible approach is to profile the crowd during the blocking step, then use the estimated crowd models (in terms of time, money, and accuracy) to help guide the subsequent steps of Corleone.

10.2.4 Other Extensions

Engaging the Crowd: There are a number of ways to improve the interaction with the crowd. First, we can improve the way we manage the workers, and infer the answers. For instance, we can test alternative solutions to infer the answers, such as estimating the worker accuracy and labels in an online fashion. Secondly, we can look at optimizing the time taken by the crowd, together with the accuracy of labels. Third, we can improve the interface used for presenting the pairs to the crowd. Finally, we can extend the system to use multiple crowdsourcing platforms, instead of using only Amazon Mechanical Turk.

Adding New Components: Another direction for extending the current system is to add new components that complement the EM workflow. One such component is a verification component at the end of the current workflow, that takes the final predicted matches and the crowd provided labels as input, and verifies the predicted matches to return only the most trustworthy matching pairs. Some of the techniques developed in recent works on crowdsourcing EM [27, 28, 6] can be used to implement such a verification component.

Another useful addition to the Corleone system would be a pre-processing component that performs data cleaning and normalization operations. This can be extremely useful in further improving the matching accuracy. Finally, a visualization module can make the system much more user-friendly. This can also empower the user to make informed decisions in the middle of workflow execution, e.g., when to stop the execution, or whether to skip a particular step in workflow.

10.2.5 Applying to Other Problem Settings

Finally, it would be interesting to explore how the ideas underlying Corleone can be applied to other problem settings. Consider for example crowdsourced joins, which lie at the heart of recently proposed crowdsourced RDBMSs. Many such joins in essence do EM. In such cases our solution can potentially be adapted to run as hands-off crowdsourced joins. We also note that crowdsourcing typically has helped learning by providing labeled data for training and accuracy estimation. Our work however raises the possibility that crowdsourcing can also help “clean” learning models, such as finding and removing “bad” positive/negative rules from a random forest. Finally, our work shows that it is possible to ask crowd workers to help generate complex machine-readable rules, raising the possibility that we can “solicit” even more complex information types from them. We plan to explore these directions in near future.

11. CONCLUSIONS

We have proposed the concept of hands-off crowdsourcing (HOC), and showed how HOC can scale to EM needs

at enterprises and startups, and open up crowdsourcing for the masses. We have also presented Corleone, a HOC solution for EM, and showed that it achieves comparable or better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost. Our work thus demonstrates the feasibility and promise of HOC, and suggests many interesting research directions in this area.

12. REFERENCES

- [1] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, 2013.
- [2] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, 2010.
- [3] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *SIGKDD*, 2012.
- [4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [5] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.
- [6] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, 2012.
- [7] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.
- [8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, 2007.
- [9] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, 2011.
- [10] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, 2012.
- [11] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *HCOMP*, 2010.
- [12] N. Katariya, A. Iyer, and S. Sarawagi. Active evaluation of classifiers on large datasets. In *ICDM*, 2012.
- [13] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, Feb. 2010.
- [14] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1-2):484–493, 2010.
- [15] J. Le, A. Edmonds, V. Hester, and L. Biewald. Ensuring quality in crowdsourced search relevance evaluation: The effects of training question distribution. In *SIGIR 2010 Workshop on Crowdsourcing for Search Evaluation*, 2010.
- [16] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: a crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.
- [17] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5:13–24, 2011.
- [18] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
- [19] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Active learning for crowd-sourced databases. *CoRR*, abs/1209.3686, 2012.
- [20] H. Park, H. Garcia-Molina, R. Pang, N. Polyzotis, A. Parameswaran, and J. Widom. Deco: a system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993,

- 2012.
- [21] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.
- [22] C. Sawade, N. Landwehr, and T. Scheffer. Active estimation of f-measures. In *NIPS*, 2010.
- [23] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
- [24] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.
- [25] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.
- [26] P. Wais, S. Lingamneni, D. Cook, J. Fennell, B. Goldenberg, D. Lubarov, D. Marin, and H. Simons. Towards large-scale processing of simple tasks with mechanical turk. In *HCOMP*, 2011.
- [27] J. Wang, T. Kraska, M. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.
- [28] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.
- [29] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2010.
- [30] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.
- [31] S. E. Whang, J. McAuley, and H. Garcia-Molina. Compare me maybe: Crowd entity resolution interfaces. Technical report, Stanford University.
- [32] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, 2010.
- [33] C. J. Zhang, L. Chen, H. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9):757–768, 2013.