

Corleone: Hands-Off Crowdsourcing for Entity Matching

Chaitanya Gokhale¹, Sanjib Das¹, AnHai Doan^{1,2},
Jeffrey F. Naughton¹, Narasimhan Rampalli², Jude Shavlik¹, Xiaojin Zhu¹

¹University of Wisconsin-Madison, ²@WalmartLabs

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.

Categories and Subject Descriptors

H.2 [Database Management]: Systems

Keywords

Crowdsourcing; Entity Matching; Active Learning

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., [2, 5, 15, 7]). 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., [30, 31, 6, 33, 34]).

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” [12].

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

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD’14, June 22–27, 2014, Snowbird, UT, USA.

Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...\$15.00.

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.

We then describe *Corleone*, a HOC solution for EM (named after Don Corleone, the fictional 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 [26] 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, 23, 20]). 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 [7, Chapter 7]). 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., [30, 31, 6, 33, 34, 27]) and industry (e.g., CrowdFlower, CrowdComputing, and SamaSource). Current works use the crowd to verify predicted matches [30, 31, 6], finds the best questions to ask the crowd [33], and finds the best UI to pose such questions [34]. 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., [5, 7]). These solutions however do not employ crowdsourcing, and still require a developer (e.g., to write and apply rules, create training data, and build indexes). 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 [7, Chapter 7]. Here the works closest to ours are those that use active learning [24, 2, 3, 22]. These works however either do not use crowdsourcing (requiring a developer to label training data) (e.g., [24, 2, 3]), or use crowdsourcing [22] 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 [14, 25]. [14] 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. [25] 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 [8]. In the database community, the work [9, 23, 20] build crowdsourced RDBMSs. Many other works crowdsource joins [19], find the maximal value [11], collect data [28], match schemas [21], and perform data mining [1] and analytics [18].

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

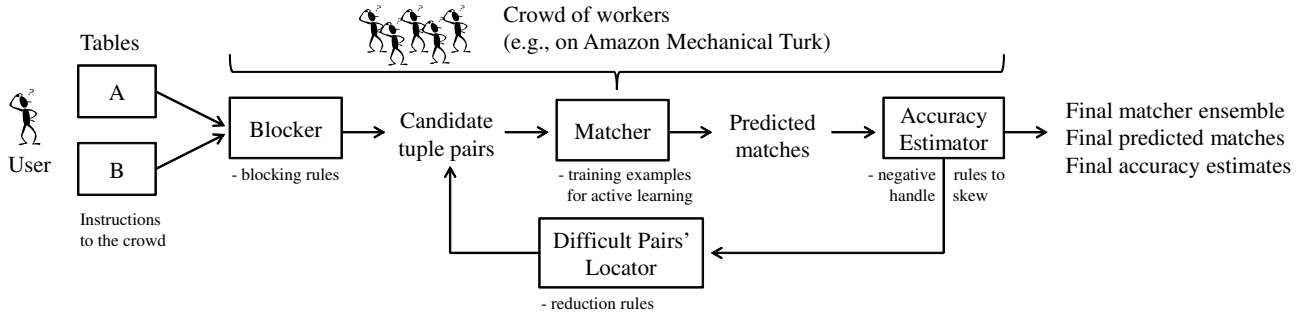


Figure 1: The Corleone architecture.

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
3. 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. \square

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

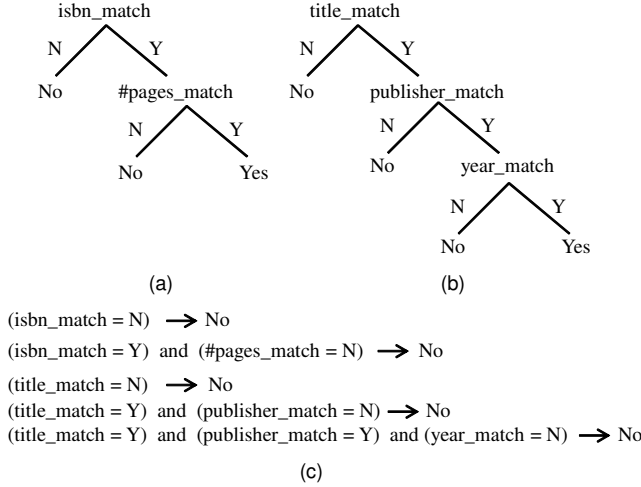


Figure 2: (a)-(b) A toy random forest consisting of two decision trees, and (b) negative rules extracted from the forest.

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 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 in detail.

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.

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 , 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 uni-

formly in B , then the above strategy ensures that S has a reasonable number of positive pairs. We show empirically later that this simple sampling strategy is effective; exploring better sampling strategies is an ongoing work.

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. [7, Chapter 4.2]. 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 [24]. 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.

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 $|cov(R, S)|$. Of course, we cannot compute $prec(R, S)$ because we do not know the true labels of examples in $cov(R, S)$. 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 $prec(R, S) \leq |cov(R, S) - T| / |cov(R, S)|$. 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 $\epsilon = Z_{1-\delta/2} \sqrt{\left(\frac{P(1-P)}{n}\right) \left(\frac{m-n}{m-1}\right)}$ [32]. 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.

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.

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. We omit further detail for space reasons.

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 , 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.

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 pair cost. The tuple pair cost is the cost of applying rule R to a tuple pair, 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 the matching accuracy, while minimizing the crowdsourcing cost. To do this, we use active learning. 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 convert 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. In what follows we use the terms example, pair, and feature vector interchangeably, when there is no ambiguity.

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. We use the default values $k = 10$ and $m = \log(n) + 1$ of the random forest learner in the Weka package (cs.waikato.ac.nz/ml/weka). Once trained, applying a random forest classifier means applying the k decision trees, then taking the majority vote.

EXAMPLE 5.1. Consider matching book tuples (*title, authors, isbn, publisher, pages, year*). Then we may generate features such as *isbn_match, title_match, etc.* A tuple pair $\langle (Data\ mining, Joe\ Smith, 1321, Springer, 234, 2013), (Data\ mining, Joseph\ Smith, 1324, Springer, 234, 2013) \rangle$ then can be converted into a feature vector with *isbn_match* =

N , $title_match = Y$, etc. Given a set of such feature vectors, together with label “matched”/“not matched”, we may learn a random forest such as the one shown in Figure 2. \square

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 for the crowd to label (currently set to 20, after experimenting with different q values on AMT). 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 [26]:

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

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. 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.

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 ,

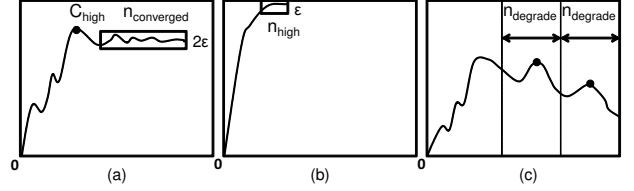


Figure 3: Typical confidence patterns that we can exploit for stopping.

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 3.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 3.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 3.b). 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 [32].

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

37,273 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 following. 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 :

1. *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} .
2. *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 to estimate the cost of each option (to be discussed below), then select the one with the lowest cost.
3. *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,

(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 re-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}). This is akin to mid-query re-optimization in RDBMSs: given a SQL query, find a good execution plan, partially evaluate the plan, then use the newly gathered statistics to re-optimize to find a potentially better plan, and so on.

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} .

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 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”, by sampling b examples from the set C (currently $b = 50$). 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 omit further details for space reasons.

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.



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 4: A sample question to the crowd.

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. Crowds 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 4 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 [17] and expectation maximization [13]. They often require a large number of answers to work well, and it is not yet clear when they outperform simple solutions, e.g., majority voting [29]. 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, 35, 19].

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 ap-

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.

pears 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 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 incur 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 [16]. 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 being involved).

9.1 Overall Performance

Accuracy and Cost: We begin by examining the overall performance of Corleone. The first five columns of Table

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 [30, 15]
Citations	89.9	94.3	92.1	\$69.5	2082	90.4	84.3	87.1	93.0	91.1	92.0	88-92 [16, 15, 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.

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.

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 $20\% \times 180,382 = 36,076$ 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. Baseline 1 achieves low accuracy because its training set is too small. 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.

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, [15] reports 92-97% F_1 for several works that they compare. Furthermore, CrowdER [30], 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, [16, 15, 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

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.

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.

9.3 Additional Experimental Results

We have run a large number of additional experiments to extensively evaluate *Corleone*. For space reasons, we will briefly summarize them here, deferring the detailed results to a forthcoming technical report [10].

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. 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.

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.

Sensitivity Analysis: We have run extensive sensitivity analysis for *Corleone*. Of these, the most interesting is on varying the labeling accuracy of the crowd. To do this, we use the random worker model in [13, 11] 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.

9.4 Evaluating and Setting System Parameters

Finally, we discuss how we evaluated and set system parameters (see the technical report [10] for more details). 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 millions, 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 the technical report 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 & FUTURE WORK

Our goals with this paper are to introduce the novel concept of hands-off crowdsourcing, to describe *Corleone*, the very first HOC solution to EM, and to establish the feasibility and promise of HOC, via extensive experiments with *Corleone*. While these goals have been achieved, it is also clear that *Corleone* is just a starting point for HOC research, and can be significantly extended in many ways.

First, sensitivity analysis shows that *Corleone* is relatively robust to small changes in parameter values (see Sections 9.3-9.4), but we need solutions to select optimal ones. Second, the *Corleone* components are highly modular and each

can be significantly improved further, e.g., developing better sampling strategies for blocking, better ways to use the crowd to evaluate rules, and better accuracy estimation methods. Third, 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. Fourth, it is critical to examine how to run Corleone on a Hadoop cluster, to scale up to very large data sets.

Fifth, 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 note also 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.

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. ACKNOWLEDGMENTS

We thank NSF for supporting this work under grant IIS 1018792.

13. 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, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier Science, 2012.
- [8] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.
- [9] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, 2011.
- [10] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. Technical report, UW-Madison, 2014. <http://pages.cs.wisc.edu/~cgokhale/corleone-tr.pdf>.
- [11] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won? Dynamic max discovery with the crowd. In *SIGMOD*, 2012.
- [12] P. Hanrahan. Analytic DB technology for the data enthusiast. SIGMOD Keynote, 2012.
- [13] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on Amazon mechanical turk. In *HCOMP*, 2010.
- [14] N. Katariya, A. Iyer, and S. Sarawagi. Active evaluation of classifiers on large datasets. In *ICDM*, 2012.
- [15] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, Feb. 2010.
- [16] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1-2):484–493, 2010.
- [17] 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.
- [18] 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.
- [19] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 5:13–24, 2011.
- [20] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
- [21] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, 2008.
- [22] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Active learning for crowd-sourced databases. *CoRR*, abs/1209.3686, 2012.
- [23] 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.
- [24] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.
- [25] C. Sawade, N. Landwehr, and T. Scheffer. Active estimation of f-measures. In *NIPS*, 2010.
- [26] B. Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.
- [27] 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.
- [28] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.
- [29] 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.
- [30] J. Wang, T. Kraska, M. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.
- [31] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.
- [32] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2010.
- [33] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.
- [34] S. E. Whang, J. McAuley, and H. Garcia-Molina. Compare me maybe: Crowd entity resolution interfaces. Technical report, Stanford University.
- [35] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: Exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, 2010.