# The Case for Cost-Sensitive and Easy-To-Interpret Models in Industrial Record Linkage

Sheng Chen
Stevens Institute of Technology
Hoboken, NJ 07030, USA
schen5@stevens.edu

Andrew Borthwick
Intelius Data Research
Bellevue, WA 98004, USA
aborthwick@intelius.com

Vitor R. Carvalho
Intelius Data Research
Bellevue, WA 98004, USA
vcarvalho@intelius.com

## ABSTRACT

Record Linkage (RL) is the task of identifying two or more records referring to the same entity (e.g., a person, a company, etc.). RL systems have traditionally handled all input record types in the same way. In an industrial setting, however, business imperatives (such as privacy constraints, government regulation, etc.) often force RL systems to operate with extremely high levels of false positive/negative error rates. For instance, false positive errors can be life threatening when identifying medical records, while false negative errors on criminal records can lead to serious legal issues. In this paper we introduce RL models based on *Cost Sensitive Alternating Decision Trees (ADTree)*, an algorithm that uniquely combines boosting and decision trees algorithms to create shorter and easier-to-interpret linking rules. These models present a two-fold advantage when compared to traditional RL approaches. First, they can be naturally trained to operate at industrial precision/recall operating points. Second, the shorter output rules are so clear that it can effectively explain its decisions to non-technical users via score aggregation or visualization. Experiments show that the proposed models significantly outperformed other baselines on the desired industrial operating points, and the improved understanding of the model's decisions led to faster debugging and feature development cycles. We then describe how we deployed the model to a commercial RL system with several billion personal records covering nearly the entire U.S. population as input, and obtained a 6:1 ratio of input records to output profiles, with an estimated 99.6%/86.2% precision/recall trade-off. This system was then deployed in a commercial e-commerce website, as well as to the subdomain of linking criminal records, obtaining an impressive 99.7%/82.9% precision/recall overall trade-off.

## Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning


*9th International Workshop on Quality in Databases* August 29, 2011, Seattle, WA


## General Terms

Algorithms, Performance

## Keywords

Data Cleaning, Record Linkage, Cost Sensitive

## 1. INTRODUCTION

Due to the critical importance of databases in today's economy, the quality of the information (or the lack thereof) stored in databases can have significant cost implications to conduct business [15]. For instance, poor quality customer data has been estimated as costing US businesses $611B annually in posting, printing, and staff overhead [14]. In the scenario in which we are consolidating information from multiple data sources, it is always desirable to create an error-free database through locating and merging duplicate records belonging to the same entity. These duplicate records could have many deleterious effects, such as preventing discoveries of important regularities [7], and erroneously inflating estimates of the number of entities [34]. Unfortunately, this cleaning operation is frequently quite challenging due to the lack of a universal identifier that would uniquely identify each entity.

The study of quickly and accurately identifying duplicates from one/multiple data source(s) is generally recognized as *Record Linkage*(RL)[18]. Synonyms in database community include *record matching* [3], *merge-purge* [21], *duplicate detection* [26], and *reference reconciliation* [13]. RL has been successfully applied in census databases [34], biomedical databases [12], and web applications such as reference disambiguation of the scholarly digital library *CiteSeerX* [1] and online comparison shopping [6].

The general approach to record linkage is to first estimate the similarity between corresponding fields to reduce or eliminate the confusion brought by typographical errors or abbreviations. A straightforward implementation of similarity function could be based on edit distance, such as the Levenshtein distance [24]. After that, a strategy for combining these similarity estimates across multiple fields between two records is applied to determine whether the two records are a match or not. The strategy could be rule-based, which generally relies on domain knowledge or on generic distance metrics to match records [15]. However, a common practice is to use Machine Learning (ML) techniques, to treat the similarity across multiple fields as a vector of *features* and "learn" how to map them into a match/unmatch binary decision. ML techniques that have been tried for RL include

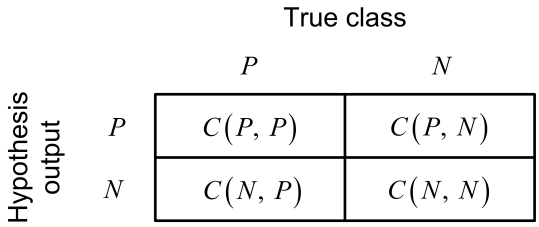


**Figure 1: A cost matrix for cost sensitive learning**

Support Vector Machines [7], decision trees [30], maximum entropy [8], or composite ML classifiers tied together by boosting or bagging [36].

Despite its importance in producing accurate estimation of duplicates in databases, insufficient attention has been given to tailoring ML techniques to optimize the performance of industrial RL systems. In this paper, we propose cost sensitive extensions of the Alternating Decision Tree (ADTree) [16] algorithm to address these problems. Cost Sensitive ADTrees (CS-ADTree) are a novel improvement to the ADTree algorithm which is well-suited to handle business requirements to deploy a system with extremely different minimum false-positive and false-negative error rates. Specifically, our method assigns biased misclassification costs for positive class examples and negative class examples, which makes it fundamentally different from existing ML techniques used by most record linkage frameworks.

Because ADTrees outputs a single tree with shorter and easy-to-ready rules, another key advantage of the proposed method is that it can effectively explain its decisions, even to non-technical users, using simple score aggregation and/or tree visualization. Even for very large models with hundreds of features, score aggregation can be straightforwardly applied to perform *feature blame assignment* — i.e., consistently calculate the importance of each feature on the final score of any decision. We show that improved understanding of these models has led to faster debugging and feature development cycles.

This work is organized as follows. Section 2 describes the concept of cost-sensitive learning and discusses how it is relevant to industrial RL. Then Section 3 discusses the design of reliable features for our system. After that, we describe ADTrees and introduce the CS-ADTree algorithms in Section 4. The experimental performance of the proposed algorithm is presented in Section 6, along with a comparison with alternative ML techniques for record linkage. We discuss our results and cover related work on record linkage in Section 7, and conclude the paper in Section 8

## 2. COST-SENSITIVE LEARNING

One common motivation for cost sensitive learning is the scenario of training a classifier on a data set which contains a significantly unequal distribution among classes. This sort of problem usually consists of *relative imbalances* and *absolute imbalances* [20]. Absolute imbalances arise in data sets where minority class examples are definitely scarce and under-represented, whereas relative imbalances are indicative of data sets in which minority examples are well represented but remain severely outnumbered by majority class examples.

A second motivation for cost sensitive learning is when there are different "business costs" (real-world consequences) between false positive and false negative errors. Cost sensitive learning for the binary classification problem can be best illustrated by a cost matrix adhering to data that consists of two classes: positive class $P$ and negative class $N$. For the sake of convenience, in the rest of this paper we refer to examples/instances belonging to the positive and negative classes as positive examples/instances and negative examples/instances, respectively. In the person record linkage context, a positive example is a pair of records which represent the same person. A negative example is when the pair of records represent two different people.

The cost matrix in Fig. 1 demonstrates the cost of four different scenarios of classifying an instance into either positive class $P$ or negative class $N$. The correct classifications reside on the diagonal line of the cost matrix and have a cost of 0, i.e., $C(P,\ P) = C(N,\ N) = 0$. Traditional reports on Record Linkage work often assign equal costs to misclassifying a positive instance into a negative instance and misclassifying a negative instance into a positive instance, i.e., $C(P,\ N) = C(N,\ P)$. This works perfectly fine when the positive class and negative class are of equal interest. Nevertheless, this is rarely true in the real business world. For instance, failure to identify a credit fraud case would bring a much higher expense than the reverse case.

In a RL industrial setting, most applications are more averse to false positives than false negatives. For instance, the industrial database studied in this paper was driving an application which displayed the history of addresses a person had resided at and jobs they had held, among other things. In this business, a false negative would generally mean that we would fail to list a true address or true job title that a person had had. However, it is considered far worse to make a false positive error whereby we would say that a person had lived at an address or held a job that pertained to someone else. Similar business tradeoffs are described in [28] where a false positive in record linkage on a children's immunization database can lead to a child not being vaccinated for a dangerous disease, whereas a false negative leads to a child receiving a redundant, but harmless, vaccination.

A classifier created for record linkage has the task of classifying a pair of records in a database into *match* or *unmatch*. Although academic systems typically target *f-measure* (the harmonic mean of *precision* and *recall*), which weights false positive and false negative errors equally [4], as discussed previously, industrial applications typically consider false positives to be much more expensive than false negatives. Hence industrial systems will frequently seek to maximize *recall* while ensuring that *precision* is at least $\pi$ for some $\pi \in [0, 1]$[3]. As an example, in one database that we were targeting, there was a requirement of $\pi \geq 0.985$, where $\pi = 0.996$ was a typical value. This implies that misclassifying no-match cases, and thus making a false positive error, should be much more expensive than misclassifying match cases (yielding a false negative) in terms of cost, which gives rise to the study of applying cost sensitive learning to tackle record linkage problems in this paper. To the best of our knowledge, this is the first work applying cost sensitive learning methods to Record Linkage.

## 3. FEATURE DESIGN

It is common sense that to build a strong record linkage system, coming up with a proper feature representation is

| ID | Company | Address | | Name | Phone # |
|---|---|---|---|---|---|
| Person 1 | Evanston Chiropractic Life Center Inc | 5716 S Evanston Ave | IN, 46123 | Ronald Eugene Raulston, ***Sr.*** | 219-821-8900 |
| Person 2 | Evanston Chiropractic Life Center Inc | 5716 S Evanston Ave | IN, 46123 | Ronald Eugene Raulston, ***Jr.*** | 219-821-8900 |

**Figure 2: An example of a pair of person profile records**

of critical importance. To accomplish that goal, most existing work concentrates on designing similarity functions estimating the similarity levels between corresponding fields in a pair of records. Although this kind of method can efficiently capture some of the semantic similarity between a pair of record despite various levels of distortions of textual strings, it leaves out some crucial signals. For instance, a rare name match in records should be considered more likely to be a duplicate than a common name match (e.g. two "Hannah Philomene's" are more likely to the same than two "John Smith's"). More generally, our experience is that achieving the highest levels of accuracy requires the design of at least dozens of heterogenous features so that the system can attempt to exploit every possible signal that is detectable by the feature designer. Given these requirements, working with a machine-learning algorithm which produces a human-understandable run-time model greatly facilitates feature development and debugging.

An illustration of our feature design starts with the example of a pair of person profile records shown in Fig. 2. Although the pair only differs by one character across all fields, it can be clearly identified as a non-match by anyone familiar with American naming conventions, since it is apparent that these two individuals are likely father and son. However, this example would be predicted as a match by many record linkage platforms because of the degree of textual similarity. Our first feature is thus *name_suffix*, which is a categorical feature with three possible values: *match* if the name suffixes match (i.e. John Smith Jr. vs. John Smith Jr.), *differ* if the name suffixes differ, and *none* if this feature does not apply (e.g. if one or both records do not contain a name suffix).

The second feature is related to name frequency, which is *global_name_frequency*. It is a numeric feature characterizing frequency of the name in the population. Note that if the names do not match, we consider the feature value in that case to be positive infinite. This feature is negatively correlated to the record linkage decision, i.e., a large value of *global_name_frequency* would decrease the likelihood that the two records match.

The third feature is telephone number match, i.e., *phone_match*, which is a categorical feature. US phone numbers can be segmented into three parts. Records matching on different parts or different conjunctions of parts of phone number should be considered duplicates with varied likelihood. Besides a *no_match* where phone numbers are different in all three parts, Fig. 3 illustrates the other four feature values *phone_match* can take. The advantage of this feature is that in lieu of determining the cost of variation in different part of phone numbers either manually [27] or adaptively [7], we directly push the different match scenarios into the ML algorithm as feature values to make the algorithm figure out the appropriate strategy for "ranking" different match cases.

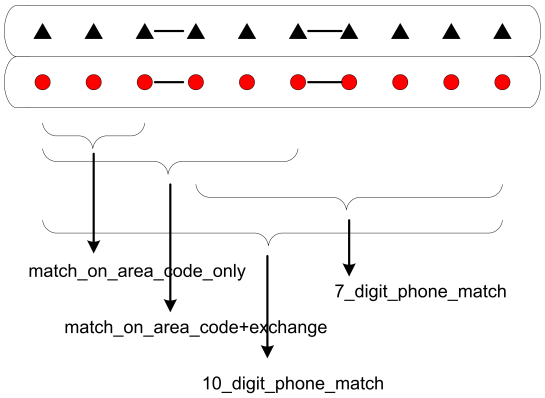


**Figure 3: Feature values of *phone_match***

A partial list of other features used in this system include the following:

| Feature Name | Explanation |
|---|---|
| street_address_match | returns "true" if street name and house number match |
| birthday_difference | returns the number of days separating the birthdays |
| regional_population | returns the population of the region that the two records have in common or positive infinity if not in the same region. (rationale: two records known to share a New York City address are less likely to match than two addresses sharing a Topeka, Kansas address) |
| name_matches | returns number of matching names (rationale: particularly useful for criminal matching, where criminals may match on multiple aliases) |

**Table 1: Other Features**

As can be seen from these feature examples, the construction of a high-precision, industrial-strength record linkage system requires a lengthy list of complex features which are the result of extensive feature engineering. It is thus important that the machine learning algorithm supports the feature development process.

# 4. ALGORITHMS

## 4.1 Alternating Decision Trees (ADTrees)

The *ADTree* algorithm [16] is a combination in a peculiar way of the *decision tree* [9] and *boosting* [17] algorithms. There are two kinds of nodes in an ADTree. One is the splitter node which specifies the condition that should be tested for the instance. The other is the prediction node, which assigns a real-valued *score* to instances satisfying conditions at different points. An ADTree can be split multiple times at any point, i.e., it can attach more than one splitter node at any single prediction node. This is different from a decision tree since (1) generally, a decision tree can only be split

once at each point, and (2) the split can be only performed at the bottom of the tree, i.e., the prediction leaves, in the *progress of tree generation*. Upon determining the class label of an instance, ADTree sums up the score of the prediction nodes of *all* paths on which the condition specified by the splitter nodes are all satisfied by the instance. The sign of the summation is the class label of the instance. Note that a conventional decision tree also decides the class label of an instance by going through a path in the tree hierarchy. The difference is that there is just *one* path and only the prediction leaf at the bottom determines the class label of the instance.

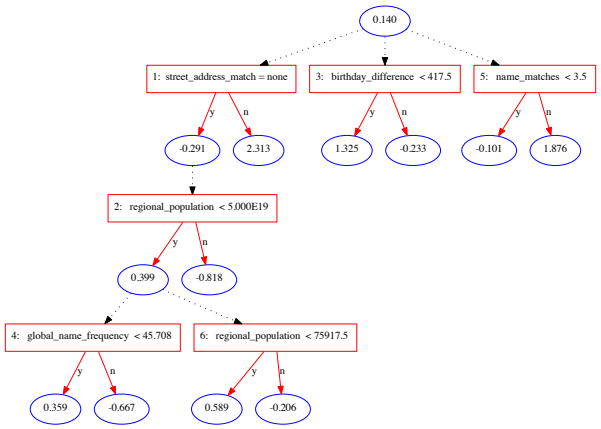


**Figure 4: A tiny ADTree for person linkage**

A toy example of a person-linking ADTree, shown in Fig. 4, was built from a database of over 50,000 labeled examples consisting of a mixture of criminal and non-criminal person records. There are five features illustrated in this model which were described in Section 4. Let's suppose the model is considering the pair of records shown in Table 2.

For this example, we first note that the root prediction node starts us off with a score of 0.140. The relatively small positive score indicates a slight predominance of positive ("match") examples in the training corpus. We then see that Feature 1, street_address_match, returns "none" (yes:-0.291), so we test Feature 2, which returns 3.3 million, since that is the population of the Seattle Metropolitan Area. Since feature 2 returned yes (+0.399), we now query features 4 and 6. Feature 4 returns 1000, the frequency of "Robert Jones" and thus answers no (-0.667), while Feature 6 tests the "regional population" feature again and decides no (-0.206) this time since the population is not less than 75,917.5. Feature 3 returns 14, which is less than 417.5, giving a "yes" decision (1.325). Feature 5 returns 1 since there is only one matching name, so the decision is also "yes" (-0.101).

Summing these values, the class label is thus determined through calculating $0.140 - 0.291 + 0.399 - 0.667 - 0.206 + 1.325 - 0.101 = 0.599$, i.e. a relatively low-confidence match. Note also that we can determine the amount of "blame" to assign to each feature by summing the prediction nodes of each feature. Consequently, we can see that "regional_population" contributes 0.399 - 0.206 = 0.193 to the final score, which is very helpful in feature engineering.

Finally, note the following two benefits of ADTrees. Firstly, if Feature 1 had returned "no" because *street_address_match* did not return "none", we would not have tested features 2, 4, and 6, thus reaping a major gain in run-time performance. Secondly, note how the ADTree seamlessly mixed real-valued features with a boolean feature ( street_address _match ), a property with simplifies development and facilitates the incorporation of heterogenous features in a record linkage model.

As a second example, consider the *name_suffix* feature described in the previous section. Due to the importance of this feature, ADTree generally puts it somewhere near the root node. The example in Fig. 9 illustrates this. "*name_suffix=differ*" is a child of the root node and is the eighth feature chosen. If the name suffix differs, we decrease the score (making a *match* less likely) by −2.89 and don't query any additional features. On the other hand, if the suffixes don't differ, the score is basically unaffected (adding 0.039 is trivial in the context of the other, much larger values) and we query many other features. Note that one additional feature is whether "*name_suffix=match*", which gets a strong positive value of 1.032 if true.

## 4.2 Cost Sensitive Alternating Decision Tree

In this section, we formulate a novel implementation of ADTree equipped with cost sensitive learning, called *Cost Sensitive Alternating Decision Tree* (CS-ADTree). To the best of our knowledge, this is the first adaptation of ADTree to a framework of cost sensitive learning and also the first attempt at applying a cost sensitive ML technique to the problem of record linkage.

Cost sensitive learning additionally assigns a cost factor $c_i \in (0, \infty)$ to each example $\boldsymbol{x}_i, y_i$ in the database to quantify the cost of misclassifying $\boldsymbol{x}$ into a class label other than $y_i$. In the standard boosting algorithm, the weight distribution over the data space is revised in an iterative manner to reduce the total error rate. Cost sensitive learning operates in a similar manner, but it operates over a space of examples in which the weight distribution has been updated in a biased manner towards examples with higher costs. According to the "translation theorem" [35], the classifier generated in this way will be conceptually the same as the one that explicitly seeks to reduce the accumulated misclassification cost over the data space. We will examine three different methods of biasing the training data to account for the business preference for false negatives vs. false positives. The baseline weight update rule of ADTree is [16],

$$w_i^{t+1} \leftarrow w_i^t \cdot e^{-r_t(\boldsymbol{x}_i) y_i}$$

In the method that we focus on in this paper, AdaC2 [33], the weight update rule for ADTree is modified as,

$$w_i^{t+1} \leftarrow c(i) \cdot w_i^t \cdot e^{-r_t(\boldsymbol{x}_i) y_i}$$

where $c(i) = c_+ \cdot \mathrm{I}(y_i = +1) + c_- \cdot \mathrm{I}(y_i = -1)$. Note $c_+$ and $c_-$ are the misclassification cost associated with positive an negative class examples, respectively, and $\mathrm{I}(\bullet) = 1$ when condition $\bullet$ is met and 0 otherwise.

The intuition here is straightforward. We hope weights of the examples with higher costs are increased *faster* than those with lower costs. According to Lemma 1 (omitted

| | Record 1 | Record 2 | Comment |
|---|---|---|---|
| Name | Robert Jones | Robert Jones | Name Frequency = 1000 |
| Birthday | 3/1/1966 | 3/15/1966 | Birthday difference = 14 |
| Address | 121 Walnut St. | 234 Chestnut St. | |
| City/State | Seattle, WA | Seattle, WA | Population of Seattle region = c. 3.3 million |

**Table 2: An example comparison of two records**

**Algorithm 1** Cost Sensitive Alternating Decision Tree

**Inputs:**

1: database $\mathcal{S} = \{(\boldsymbol{x}_1, y_1, c_1), \ldots, (\boldsymbol{x}_n, y_n, c_n)\}$, where $\boldsymbol{x}_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$, and $c(i) = c_+ \cdot \mathrm{I}(y_i = +1) + c_- \cdot \mathrm{I}(y_i = -1)$.
2: weights $\boldsymbol{W} = \{w_1^0, \ldots, w_n^0\}$, where $w_i^0 = 1$.
/* uniform distribution initially */
3: $\mathcal{D} \leftarrow \{all\ possible\ conditions\}$.
4: $N \leftarrow$ Number of iterative rounds.
5: $\eta \leftarrow$ smooth factor.

**Procedure:**

6: $r_0 \leftarrow \mathrm{T}$ with score $\frac{1}{2} \ln \frac{c_+ \cdot \mathcal{W}_+(\mathrm{T}) + \eta}{c_- \cdot \mathcal{W}_-(\mathrm{T}) + \eta}$
/* T: precondition is *true* for all examples. */
7: $\mathcal{P}_0 \leftarrow r_0$ /* precondition set */
8: **for** $t : 1 \rightarrow N$ **do**
9: $d_1, d_2 = \underset{d_1, d_2}{\operatorname{argmin}}(\mathrm{Z})$
s.t. $\mathrm{Z} = [2(\sqrt{\mathcal{W}_+(d_1 \cap d_2)\mathcal{W}_-(d_1 \cap d_2)}$
$+\sqrt{\mathcal{W}_+(d_1 \cap \neg d_2)\mathcal{W}_-(d_1 \cap \neg d_2)}) + \mathcal{W}(\neg d_1)]$
$d_1 \in \mathcal{P}_t,\ d_2 \in \mathcal{D}$
10: $\alpha_1 = \frac{1}{2} \ln \frac{c_+ \cdot \mathcal{W}_+(d_1 \cap d_2) + \eta}{c_- \cdot \mathcal{W}_-(d_1 \cap d_2) + \eta} \rightarrow d_1 \cap d_2$
/* $\alpha_1$ is the score associated with $d_1 \cap d_2$ */
11: $\alpha_2 = \frac{1}{2} \ln \frac{c_+ \cdot \mathcal{W}_+(d_1 \cap \neg d_2) + \eta}{c_- \cdot \mathcal{W}_-(d_1 \cap \neg d_2) + \eta} \rightarrow d_1 \cap \neg d_2$
/* $\alpha_2$ is the score associated with $d_1 \cap \neg d_2$ */
12: $r_t \leftarrow d_1 \cap d_2 \bigoplus d_1 \cap \neg d_2$
/* $r_t(\boldsymbol{x})$ is a new splitter node with two associated prediction nodes. */
13: $\mathcal{P}_{t+1} \leftarrow \{\mathcal{P}_t, r_t\}$
14: $w_i^{t+1} \leftarrow c(i) \cdot w_i^t \cdot e^{-r_t(\boldsymbol{x}_i) y_i}$
/* update example weights */
15: **end for**
16: **return** Classifier for unseen unlabeled instances:
$\mathcal{H}(\boldsymbol{x}) = \mathrm{sgn}(\sum_{t=0}^{N} r_t(\boldsymbol{x}))$

for space reasons), the learning focus of ADTree will be biased towards examples with higher costs. Given the modified weight update rule, the original equation for calculating scores $\alpha_1$ and $\alpha_2$ at each iterative round no longer guarantees the error rate could be decreased fastest. Inspired by the induction of optimized $\alpha$ for AdaC2, $\alpha_1$ and $\alpha_2$ can be modified to $\alpha_1 = \frac{1}{2} \ln \frac{c_+ \mathcal{W}_+(d_1 \cap d_2)}{c_- \mathcal{W}_-(d_1 \cap d_2)}$ and $\alpha_2 = \frac{1}{2} \ln \frac{c_+ \mathcal{W}_+(d_1 \cap \neg d_2)}{c_- \mathcal{W}_-(d_1 \cap \neg d_2)}$, where $d_2$ is the condition set by splitter node $r_t$, and $d_1$ is the condition that an example has to meet in order to reach $r_t$. The splitter node $r_t$ can also be interpreted as rule in that an example $\boldsymbol{x}$ arriving at $r_t$ would be assigned score $\alpha_1$ if it satisfies both $d_1$ and $d_2$, i.e., $d_1 \cap d_2$, and $\alpha_2$ if it satisfies $d_1$ but not $d_2$, i.e., $d_1 \cap \neg d_2$.

Now that we have modified the weight update rule and equations for calculating prediction scores $\alpha_1$ and $\alpha_2$, we formulate CS-ADTree in Algorithm 1.

The cost factor $c(i)$ can also be put in other spots of the weight update equation. For instance, it can be put inside the exponential term $e^{-r_t(\boldsymbol{x}_i) y_i}$. When example $\boldsymbol{x}$ is misclassified by $r_t$, we have $\mathrm{sgn} - r_t(\boldsymbol{x}_i) y_i > 0$. A high cost inside the exponential term would thus increase the weight of the example in exponential order. This is the idea of AdaC1 [33]. Its weight update rule and prediction score equations of AdaC1 can be adapted for ADTree as,

$$w_i^{t+1} \leftarrow w_i^t \cdot e^{-c(i) \cdot r_t(\boldsymbol{x}_i) y_i}$$

$$\alpha_1 = \frac{1}{2} \ln \frac{\mathcal{W}(d_1 \cap d_2) + c_+ \cdot \mathcal{W}_+(d_1 \cap d_2) - c_- \cdot \mathcal{W}_-(d_1 \cap d_2)}{\mathcal{W}(d_1 \cap d_2) - c_+ \cdot \mathcal{W}_+(d_1 \cap d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap d_2)}$$

$$\alpha_2 = \frac{1}{2} \ln \frac{\mathcal{W}(d_1 \cap \neg d_2) + c_+ \cdot \mathcal{W}_+(d_1 \cap \neg d_2) - c_- \cdot \mathcal{W}_-(d_1 \cap \neg d_2)}{\mathcal{W}(d_1 \cap \neg d_2) - c_+ \cdot \mathcal{W}_+(d_1 \cap \neg d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap \neg d_2)}$$

The cost factor can also be put both inside and outside the exponential term in the weight update equation, which gives rise to AdaC3 [33]. Its weight update rule and prediction score equations can be adapted for ADTree as,

$$w_i^{t+1} \leftarrow c(i) \cdot w_i^t \cdot e^{-c(i) \cdot r_t(\boldsymbol{x}_i) y_i}$$

$$\alpha_1 = \tfrac{1}{2} \times \ln \frac{c_+ \cdot \mathcal{W}_+(d_1 \cap d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap d_2) + c_+^2 \cdot \mathcal{W}_+(d_1 \cap d_2) - c_-^2 \cdot \mathcal{W}_-(d_1 \cap d_2)}{c_+ \cdot \mathcal{W}_+(d_1 \cap d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap d_2) - c_+^2 \cdot \mathcal{W}_+(d_1 \cap d_2) + c_-^2 \cdot \mathcal{W}_-(d_1 \cap d_2)}$$

$$\alpha_2 = \tfrac{1}{2} \times \ln \frac{c_+ \cdot \mathcal{W}_+(d_1 \cap \neg d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap \neg d_2) + c_+^2 \cdot \mathcal{W}_+(d_1 \cap \neg d_2) - c_-^2 \cdot \mathcal{W}_-(d_1 \cap \neg d_2)}{c_+ \cdot \mathcal{W}_+(d_1 \cap \neg d_2) + c_- \cdot \mathcal{W}_-(d_1 \cap \neg d_2) - c_+^2 \cdot \mathcal{W}_+(d_1 \cap \neg d_2) + c_-^2 \cdot \mathcal{W}_-(d_1 \cap \neg d_2)}$$

Despite the fact that cost sensitive learning can manifest itself in different forms in boosting, we chose to create CS-ADTree by integrating AdaC2 into ADTree's training procedure since the weight updating rule of AdaC2 weighs each example by its associated cost item directly [33], which naturally fits the algorithm into the realm of the translation theorem. On the other hand, AdaC1 and AdaC3 both have cost item placed inside exponential term which is lack of direct connection to the weight update. In addition, our preliminary empirical studies showed that AdaC2 performed the best across a number of benchmarks. That being said, we would like to further explore the performance of ADTree with the cost sensitive learning frameworks of AdaC1 and AdaC3 in future work.

# 5. EVALUATION PARAMETERS

Most of the time the metrics for evaluating a classifier's performance on record linkage problem are *precision*, *recall* and their harmonic mean, *f-measure* — metrics obtained by fixing the classifier's decision threshold to zero. A confusion matrix shown in Fig. 5 is always helpful to fully capture what these metrics stand for. A quick impression is that a confusion matrix is very similar to a cost matrix. The difference is that what resides in the matrix is the number of instances satisfying the scenario specified by the row and column indexes. Therefore TP (True Positive) and TN (True Negative) are the number of correctly classified positive and negative instances, respectively. $FP$ (False Positive) and $FN$ (False Negative) are the number of instances *falsely* classified into positive class and negative class, respectively. Considering a matched pair of records as a positive example, and an unmatched pair of records as a negative example,

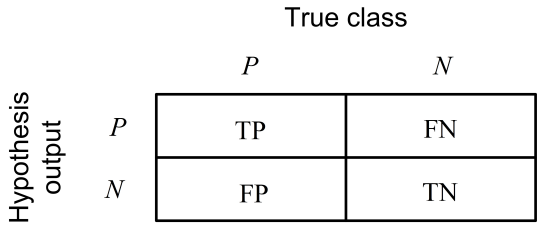


**Figure 5: A confusion matrix for binary classification**

*precision*, *recall*, and *f-measure* for record linkage are defined as,

$$precision = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$recall = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$f\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

From the definitions, *precision* measures the *accuracy* of a classifier's predictions of positive instances, while *recall* tests the *completeness* of a classifier's coverage of real positive instances. It is clear that there is a trade-off between *precision* and *recall*. This can be seen by the explanation that if all instances are frivolously predicted as positive, *recall* is maximized to be 1, whereas *precision* is equal to the proportion of positive examples in the database, which is very bad. To this end, *f-measure* is frequently used to measure the overall performance of a classification algorithm when the relative importance of recall and precision are evenly balanced.

*Customization of evaluation metrics* Good performance at a single threshold such as zero may lead to unwanted subjectivity. Recent studies show that the performance of a classifier should be evaluated upon a range of thresholds [19]. An appealing property of ADTree is that instead of merely outputting a hard class label, it gives a score value, $\sum_{t=0}^{N} r_t(\boldsymbol{x})$, to estimate the confidence level of its decision. While ADTree generally uses the sign function to classify unlabeled instances, other numerical value can also be used to serve as the *threshold*. In this case, the classifier of ADTree can be represented as $\mathcal{H}(\boldsymbol{x}) = \text{sgn}(\sum_{t=0}^{N}(r_t(\boldsymbol{x})) - d)$, where $d \neq 0$. Since the business priority for the industrial database on which we were working was to keep *precision* above a threshold which was in excess of 99%, *precision* in a high range was of much more interest to us and we thus focused on metrics other than *f-measure*. In this study, we start by setting the threshold, $d$, to be large enough to make $precision = 1$, and then tune down the threshold to progressively decrease *precision* down to 0.985 in steps of 0.001 to determine the *recall* that we are able to achieve at varying high *precision* levels.

*Choice of cost ratio* Some business applications do have strategies to decide costs for different examples. For instance, the amount of money involved in a transaction can be used to quantify the cost related to it. Nevertheless for many other applications such as record linkage, the only prior knowledge available is the biased interest towards one class over the other. In industrial applications, this is usually expressed as a requirement that the *precision* be at least $\pi$, where in our applications, we generally had $\pi \geq 0.985$ and a more typical requirement for $\pi$ was in excess of 0.995. Note that two parameters influence the *precision* of the classifier, the cost factor, $C$, and the the required *precision*, $\pi$. We have found that a reasonable heuristic is to adjust your cost factor $C$ so that the threshold, $d$, that yields your desired *precision*, $\pi$ is close to 0, this generally yields close to optimal *recall* at $\pi$. We give an informal theoretical justification for this heuristic in section 6.1.

# 6. EXPERIMENT

*Objectives* In this section, we present experimental results of ADTree and CS-ADTree for comparing records of person profile databases. The objectives here are to 1. demonstrate the effectiveness of ADTree and CS-ADTree on classifying pairs of record into match/unmatch for record linkage; 2. illustrate how the run-time representation of ADTree can enable humans' interpretation of the classifier derived by ADTree; 3. demonstrate the competitiveness of ADTree/CS-ADTree with alternative ML techniques heavily used by most existing record linkage frameworks; and 4. Show how CS-ADTree demonstrates superior performance to ADTree at both very high precision requirements.

*Implementation* In this study, ADTree, CS-ADTree, boosted decision tree, and decision tree algorithms are all implemented based on the JBoost platform [2]. The SVM algorithm is implemented based on SVM-Light [22].

## 6.1 Performance of ADTree/CS-ADTree

In our initial experiments, we used a database $\mathcal{S}$ with match/unmatch labels assigned by expert internal annotators which had $20,541$ pairs of person profile records characterized by more than 42 features including those described in Section 3. Since the cost of hiring experts is relative high, we later switched to Amazon's Mechanical Turk (MT) System which provides a significantly cheaper and faster way to collect label data from a broad base of non-expert contributions over the web [32]. A drawback of this method is that the database is likely to become noisy since it is possible that some workers on MT ("Turkers") are purely spammers and some are lacking in the necessary domain knowledge for the task.

We report average results for a 10-fold cross validation. For these initial experiments, we held the number of boosting iterative rounds, $\mathcal{T}$ fixed at 200 and we used a cost factor of $\mathcal{C} = 4$ based on earlier experiments showing these to be optimal values over the precision range $[0.985, 1]$.

With $T$ and $C$ determined, the record linkage performance of CS-ADTree, ADTree, and CS-ADTree with cost-sensitive learning frameworks of AdaC1 and AdaC3 on our initial person profile database $\mathcal{S}$ are studied. Their average *recall* are given in Fig. 6, which clearly shows that CS-ADTree performs the best across all methods under comparison. Average *recall* can sometime be misleading when the physical P-R curves of two methods under comparison cross in P-R space [19]. In this case, it would be nonsensical to conclude one method is better than the other based on the area under P-R curve which is conceptually equal to the averaged *recall*. To that end, we also plot P-R curves of all method under comparison in Fig 7. One can clearly see that CS-ADTree can consistently contain other methods in terms of P-R curve, which validates the supremacy of CS-ADTree.

One perspective on how CS-ADTree is able to achieve this

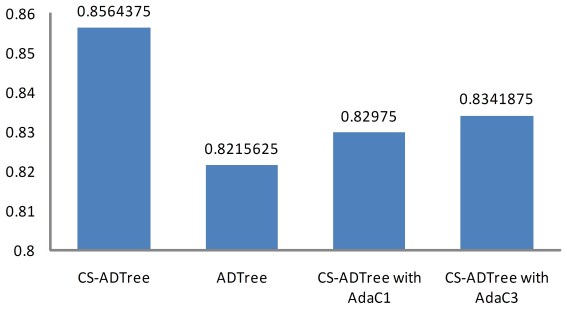


**Figure 6: Comparison of average *recall*'s for ADTree/CS-ADTree**

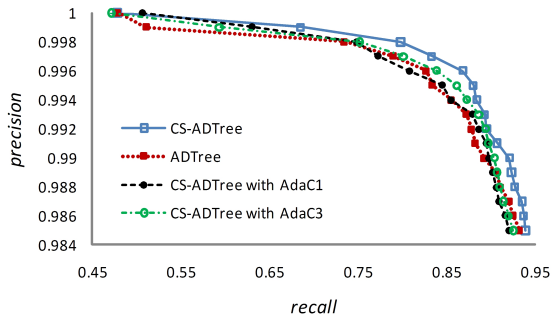


**Figure 7: P-R curves of ADTree/CS-ADTree**

superior performance can be seen in Fig. 8. CS-ADTree achieves a given level of *precision* at a much lower threshold than the other three methods. It is particularly instructive to look at CS-ADTree as compared to ADTree. CS-ADTree achieves 0.995 *precision* at a threshold of roughly 0, while ADTree achieves this at about 4. An intuition as to CS-ADTree's superiority is that the model is seeking to push positive examples to have scores above 0 and negative scores below 0. In the case of CS-ADTree, we are able to put our threshold, $d$, at about the same threshold as the model's threshold, whereas with ADTree, the model is not "aware" that the threshold of interest to us is 4.

To demonstrate how run-time representation of CS-ADTree

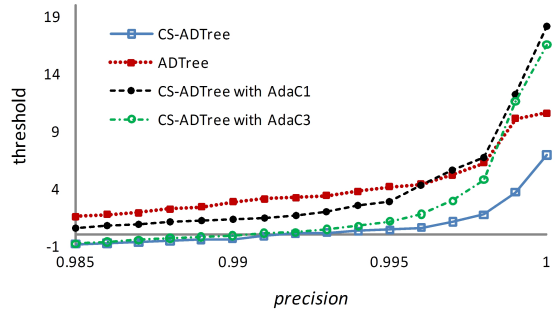


**Figure 8: Threshold range of ADTree/CS-ADTree w.r.t high *precision* range**

can help a practitioner better understand the record linkage problem of his/her database in a human-interpretable way, we present two partial snapshots of a CS-ADTree generated by our initial labeled database $\mathcal{S}$ in Fig. 9 and Fig. 10. In Fig. 9, one can clearly see that the decision node of *name_suffix* resides right below the root node. When *name_suffix=differ*, the corresponding path ends immediately and assigns a large negative score to the instance under consideration, otherwise CS-ADTree would go on to check many other features. This is exactly as predicted by our discussion for the *name_suffix* feature in Section 3. Note also that this tree can be very efficiently computed if the features are lazily evaluated. If the name suffix differs, there is no need to compute the values for features 14 - 18, which all have *name_suffix* $\neq$ "*differ*" as a precondition (in fact, 87 out of the 100 features in this sample CS-ADTree have this as a precondition). More hierarchical levels are involved in the example in Fig. 10. If the value of feature *birthday_difference* for a pair of records is relatively small, (the birthdays are less than 432.5 days apart), CS-ADTree would terminate the corresponding path by just examining the value of *birthday_difference*. This is intuitive because having nearly matching birthdays is a strong "match" indicator. We don't need to ask further questions to reach a decision. Otherwise it asks if *birthday_difference* is greater than $5 \times 10^{19}$, i.e., *infinity*, which is how we indicate a null value (birthday isn't present on one or the other records). In this case, CS-ADTree would continue to check a number of other features to determine the match/unmatch status of the pair of records. So in both cases, the derived model is easy to understand and can be seen to be doing something reasonable.

*Efficiency* Our experiments also show that ADTree and CS-ADTree are efficient at training and run-time. Given 50 candidate features, a training set size 11,309 example pairs, and a pre-computed feature vector, we trained the system on JBoost using CS-ADTree for 200 iterations in 314 seconds. On a major run of the system using a Python implementation and a 100 node model, the model performed 180 record pair comparisons per second per processor, including both the time to compute feature computations and the computation in the CS-ADTree. All statistics were on an Intel Xeon 2.53 GHz processor with 60 GB of RAM.

## 6.2 Performance of ADTree/CS-ADTree with active learning

The initial database $\mathcal{S}$ is amplified by picking records from a data pool with size $8 \times 10^6$ which are then presented to Turkers for labeling. Denoting the pairwise classifier trained on $\mathcal{S}$ to be $\mathcal{H}$, the principle of choosing records are based on three active learning approaches listed as follows,

- The first approach serves as our baseline active learning model, which is basically to randomly choose $r \in \mathcal{S}$, s.t., $\mathcal{H}(r) > -3$. The reason we choose $-3$ as the threshold to pick records is based on the empirical observation that examples in $\mathcal{S}$ that are given a prediction score by $\mathcal{H}$ greater than $-3$ exhibit an appropriate proportion between positive and negative examples, i.e., negative examples shouldn't be excessively outnumbered by positive examples to avoid Turkers from clicking "match" all the way through.
- The second approach is based on the randomized parameter approach described in [30]. Instead of always splitting with the feature of minimum loss $Z$, we enable CS-ADTree

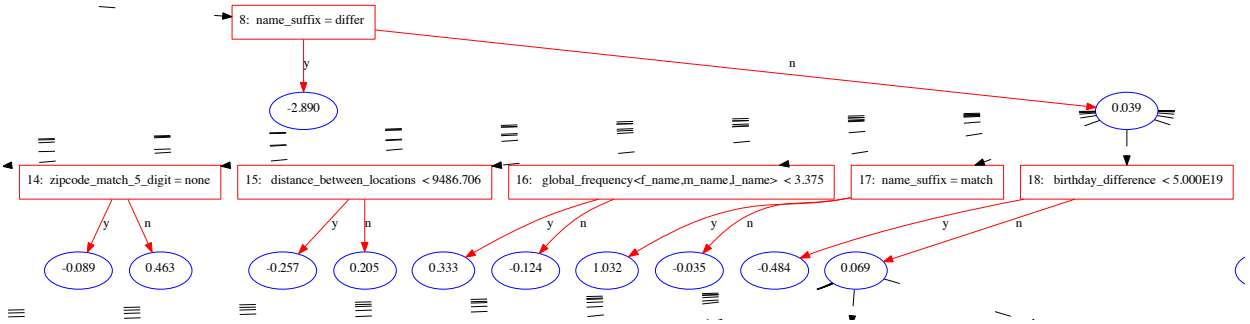


**Figure 9: Partial CS-ADTree. If *name_suffix=differ*, we get a strong negative score and don't query additional features.**

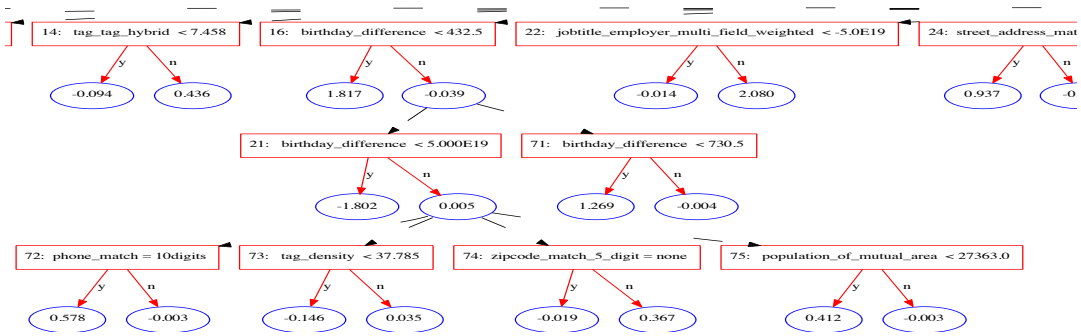


**Figure 10: Partial CS-ADTree. Note how we assign different scores to different birthday distances and query additional features if birthday difference is unknown (infinite)**

to always *randomly* choose 1 of the 3 features with minimum losses across all features. A committee of 5 CS-ADTree's trained on $\mathcal{S}$ is then created as $\{\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3, \mathcal{H}_4\}$, records $r \in \mathcal{S}$ with minimal absolute committee score, i.e., $|\sum_{i=0}^{4}(\mathcal{H}_i(r))|$, will be picked for class labeling.

• The last approach is pretty straightforward. We choose records $r \in \mathcal{S}$ with minimal absolute prediction score by $\mathcal{H}$, i.e., $|\mathcal{H}(r)|$, since it represents a small *margin* given the natural threshold 0 which signifies the uncertainty of $\mathcal{H}$ for classifying the particular instance. This approaches resonates with our argument that a model that yields our desired *precision*, $\pi$ at a threshold close to 0 will have close to optimal *recall*.

We pick 1000 pairs of records from the data pool using each of the three active learning approaches, which yields 3000 records in total. We also pick another 5000 pairs of records using active learning approach 1 to serve as the independent data set $\mathcal{E}$ for evaluating the learning models. The P-R curves of CS-ADTree trained on $\mathcal{S}$ plus the records picked by each of three active learning approaches are given in Fig. 11, which clearly shows active learning approaches 2 and 3 perform better than the baseline approach.

Expanded by the 3000 pairs of records picked by active learning approaches, $\mathcal{S}$ now becomes noisier since it is unavoidable that class labels assigned by Turkers will be partially erroneous. Using ADTree and CS-ADTree to derive classifiers from $\mathcal{S}$, Fig. 12 shows the P-R curves of applying these classifiers on $\mathcal{E}$, which clearly indicates CS-ADTree exhibits improved performance compared to ADTree for record

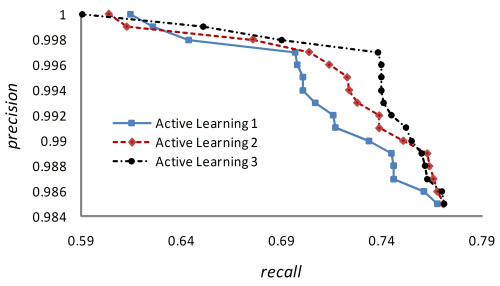


**Figure 11: P-R curves of 3 active learning approaches**

linkage of noisy database.

In order to demonstrate the competitiveness of ADTree/CS-ADTree against other popular ML techniques used by record linkage designers, we also apply decision tree (DT), boosted DT ($T = 200$), and SVMs to classify pairs of records in $\mathcal{E}$. The kernel function selected for SVMs is the *Gaussian kernel* with $\sigma = 0.1$. Empirical study discovers that SVMs cannot efficiently handle the original feature representation, i.e., a mixture of categorical and numerical features. Thus we apply two strategies to transform the features to cater for SVMs' needs. Categorical features are all transferred into discrete integers for both strategies. For numerical features, strategy 1 is uniform bucketing which evenly divides its data range into 5 buckets, and transforms the feature value into

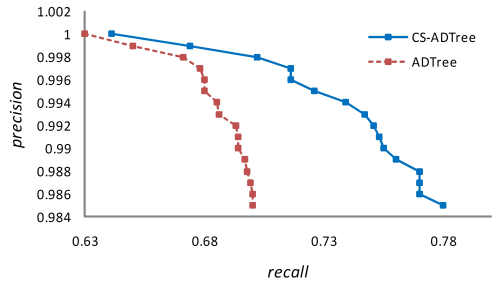


**Figure 12: P-R curves of ADTree and CS-ADTree on $\mathcal{E}$**

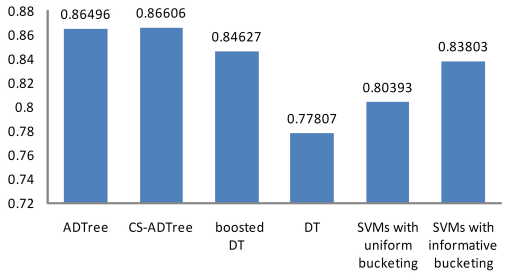


**Figure 13: Histogram of *f-measure*'s for ADTree, CS-ADTree, and alternative ML techniques on $\mathcal{E}$**

the integral index of the bucket it falls in. Strategy 2 is informative bucketing which efficiently makes use of the information provided by the CS-ADTree structure after training. If numerical feature $f$ is split on values of $p_0 < p_1 < \ldots < p_n$ in the training process of CS-ADTree, any feature value $v$ of $f$ will be rewritten into integer $i$, s.t., $p_i < v < p_{i+1}$. Using 0 as the threshold for ADTree/CS-ADTree, Fig. 13 shows the *f-measure* of ADTree, CS-ADTree, boosted decision tree, decision tree, and SVM's using two bucketing strategies. It is obvious that ADTree and CS-ADTree both perform better than alternative ML techniques on record linkage of $\mathcal{E}$.

## 7. RELATED WORK

Since record linkage has been a well-studied topic, we recommend interested readers to [15] and [18] for a comprehensive survey. Due to the importance of feature representation, similarity function design is at the core of many record linkage studies [23]. The most straightforward one is the Levenshtein distance [24] which counts the number of *insert*, *remove*, and *replace* operations when mapping string A into B. Considering the unbalanced cost of applying different operations in practice, [27] modifies the definition of edit distance to explicitly allow for the cost customization by designers. In recent years similarity function design is increasingly focused on adaptive methods. [29] [7] proposed similar stochastic models to learn the cost factors of different operations for edit distance. Rooted in the spirit of fuzzy match, [11] considers the text string at the tuple level and proposes a probabilistic model to retrieve the $K$ nearest tuples w.r.t an input tuple received in streamed format. Exploiting the similarity relation hidden under a big umbrella of linked pairs, [5] iteratively extracts useful information from the pairs to progressively refine a set of similarity functions. [10] introduces the similarity functions from probabilistic information retrieval and empirically studies their accuracy for record linkage.

However, whatever ingenious methods may be used for similarity functions, it is necessary to integrate all of these field-level similarity judgments into an overall match/no-match decision. Various learning methods have been proposed for this task. [7] proposed stacked SVMs to learn and classify pairs of record into match/unmatch, in which the second layer of SVMs is trained on a vector of similarity values that are output by the first layer of SVMs. [25] considers the records in a database as nodes of a graph, and applies a clustering approach to divide the graph into an adaptively determined number of subsets, in which inconsistencies among paired records are expected to be minimized. [31] instead considers features of records as nodes of a graph. Matched records would excite links connecting corresponding fields, which could be used to facilitate other record comparisons. A well-performing pairwise classifier depends on the representativeness of the record pairs selected for training, which calls for an active learning approach to efficiently pick informative paired records from a data pool. [30] described several committee-based active learning approaches for record linkage. Considering the efficiency concern of applying an active learning model on a data pool with quadratically large size, [3] proposed a scalable active learning method that is integrated with *blocking* to alleviate this dilemma.

## 8. CONCLUSION

In this paper, we proposed to study record linkage of databases by ADTree. Considering that an essential problem of record linkage is that the business costs of misclassifying matched pair and unmatched pair are extremely biased, we further propose CS-ADT which assigns a higher misclassification cost for matched pair in the process of training ADTree. Experiments show CS-ADTree and ADTree perform extremely well on a clean database and exhibit superior performance on noisy database compared with alternative ML techniques. We also demonstrate how the run-time representation of ADTree/CS-ADTree can facilitate human understandability of learned knowledge by the classifier and yield a compact and efficient run-time classifier.

## 9. REFERENCES

[1] *CiteSeerX*. `http://citeseerx.ist.psu.edu/`.
[2] *JBoost Project*. `http://jboost.sourceforge.net`.
[3] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, pages 783–794, 2010.
[4] J. Artiles, A. Borthwick, a. S. S. J. Gonzalo, and E. Amigó. Weps-3 evaluation campaign: Overview of the web people search clustering and attribute extraction tasks. In *CLEF-2010*.
[5] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proc. ACM SIGMOD workshop on Research issues in data mining and knowledge discover*, pages 11–18, 2004.
[6] M. Bilenko and S. Basu. Adaptive product normalization: Using online learning for record linkage in comparison shopping. In *ICDM*, pages 58–65, 2005.

[7] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.

[8] A. Borthwick, M. Buechi, and A. Goldberg. Key concepts in the choicemaker 2 record matching system. In *ACM KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[9] L. Breiman, J. Friedman, and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[10] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In *SIGMOD*, pages 353–364, 2007.

[11] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[12] P. Christen and T. Churches. *Febrl - Freely Extensible Biomedical Record Linkage*. Australian National University, 0.4 edition.

[13] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96, 2005.

[14] W. Eckerson. Data warehousing special report: Data quality and the bottom line. *Applications Development Trends*, 2002.

[15] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[16] Y. Freund and L. Mason. The alternating decision tree algorithm. In *ICML*, pages 124–133, 1999.

[17] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Computer and System Sciences*, 55(1):119–139, 1997.

[18] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. Technical report, CSIRO Mathematical and Information Sciences, 2003.

[19] D. J. Hand. Measuring classifier performance: a coherent alternative to the area under roc curve. *Machine Learning*, 77(1):103–123, 2009.

[20] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE TKDE*, 21(9):1263–1284, 2009.

[21] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. In *KDD*, pages 9–37, 1998.

[22] T. Joachims. *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*. MIT-Press, 1999.

[23] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: Similarity measures and algorithms. In *SIGMOD*, pages 802–803, 2006.

[24] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[25] A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

[26] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[27] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biology*, 48(3):443–453, 1970.

[28] V. Papadouka, P. Schaeffer, A. Metroka, A. Borthwick, and et al. Integrating the new york citywide immunization registry and the childhood blood lead registry. *J. Public Health Management and Practice*, 10:S72, 2004.

[29] E. S. Ristad and P. N. Yianilos. Learning string edit distance. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[30] S. Sarawagi and A. Bhamidipaty. Interative deduplication using active learning. In *KDD*, pages 269–278, 2002.

[31] P. Singla and P. Domingos. Multi-relational record linkage. In *Proc. ACM KDD Workshop on Multi-Relational Data Mining*, pages 31–48, 2004.

[32] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast - but is it good?: evaluating non-expert annotations for natural language tasks. In *EMNLP*, pages 254–263, 2008.

[33] Y. Sun, M. S. Kamel, A. K. C. Wong, and Y. Wang. Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognition*, 40(12):3358–3378, 2007.

[34] W. E. Winkler. Overview of record linkage and current research directions. Technical report, Bureau of the Census, 2006.

[35] B. Zadrozny, J. Langford, and N. Abe. Cost-sensitive learning by cost-proportionate example weighting. In *ICDM*, pages 435–442, 2003.

[36] H. Zhao and S. Ram. Entity identification for heterogeneous database integration: a multiple classifier system approach and empirical evaluation. *Information Systems*, 30(2):119–132, 2005.