Privacy-Preserving Classifier Learning

Justin Brickell and Vitaly Shmatikov

The University of Texas at Austin, Austin TX 78712, USA

Abstract. We present an efficient protocol for the privacy-preserving, distributed learning of decision-tree classifiers. Our protocol allows a user to construct a classifier on a database held by a remote server without learning any additional information about the records held in the database. The server does not learn anything about the constructed classifier, not even the user's choice of feature and class attributes.
Our protocol uses several novel techniques to enable oblivious classifier construction. We evaluate a prototype implementation, and demonstrate that its performance is efficient for practical scenarios.

Keywords: Privacy, Secure Multiparty Computation, Data Mining

1 Introduction

Privacy-preserving data analysis is one of the most important applications of secure multi-party computation. In this paper, we develop a privacy-preserving version of a fundamental data-analysis primitive: an algorithm for constructing or learning a *classifier*. Classifiers, such as decision trees, are a mainstay of data mining and decision support [24]. Given a database with multiple attributes (an attribute can be thought of as a column in a database schema), a classifier *predicts* the value of a "target" or "class" attribute from the values of "feature" attributes. One can also think of a classifier as assigning records to certain classes (defined by the value of the class attribute) on the basis of their feature attributes. A popular machine-learning task is to automatically learn a classifier given a training set of records labelled with class attributes. Classifiers built in this way are used for marketing and customer relationship management, development of better recommendation algorithms and services, clinical studies, and many other applications.

We focus on the problem of securely constructing a classifier in a two-party setting where one party provides a database, while the other party provides the parameters of the classifier that it wants to construct from the records in the database. This is a common situation in law-enforcement, regulatory, and national-security settings, where the entity performing the analysis (for example, an agency investigating irregular financial transactions) does not want to reveal which patterns it is mining the database for (for example, to prevent the target of investigation from structuring their transactions so as to avoid scrutiny). Confidentiality of the resulting classifier is also important in scenarios where both the data-analysis techniques and the output of the analysis

process constitute potentially valuable intellectual property, *e.g.*, when mining patient databases in clinical studies, constructing expert systems and diagnostic frameworks, and so on.

The key privacy properties that the protocol for privacy-preserving classifier learning must guarantee are, informally, as follows. First, the records from which the classifier is constructed should remain confidential from the party who obtains the classifier (except for the information which is inevitably revealed by the classifier tree itself). Second, the data owner should not learn anything about the classifier which has been constructed. While the algorithm for constructing the classifier is standard (*e.g.*, ID3), its parameters—(i) which attributes are used as features?, (ii) which attributes are used as class attributes?, (iii) if the classifier is being constructed only on a subset of database records, what is the record selection criterion?—should remain hidden from the data owner. Note that the latter requirement precludes the data owner from simply computing the classifier on his own.

Previous work on privacy-preserving classifier learning [16,28,29] focused on a very different problem in which the resulting classifier is revealed to *both* parties. This greatly simplifies the protocol because the classifier can be constructed using the standard recursive algorithm—since both parties learn the resulting classification tree, revealing each node of the tree to both parties as it is being constructed does not violate the privacy property. This is no longer true in our setting, which presents a nontrivial technical challenge.

Existing protocols cannot be used in practical scenarios where confidentiality of the classifier is essential. For example, a national-security agency may want to mine records of financial transactions without revealing the classified patterns that it is looking for (defined by its choice of feature and class attributes and of a certain subset of individuals in the database). Other scenarios include construction of a recommendation algorithm from transactional data without revealing it prematurely (*e.g.*, the Netflix Prize competition [22]); clinical studies involving competing medical institutions, each of which is fiercely protective both of their patient data *and* their analysis techniques (which subset of patients to look at, which symptoms to focus on, and so on), because the latter can lead to patentable and potentially lucrative diagnostic methods; expert systems, where the classifier constitutes valuable intellectual property; remote software fault diagnostics [6]; and many others.

In this paper, we use the same basic framework of secure multi-party computation (SMC) as the original paper on privacy-preserving data mining by Lindell and Pinkas [16] and aim to provide the same level of cryptographic security guarantees. We emphasize, however, that (i) our desired privacy properties (in particular, confidentiality of the resulting classifier) are very different and more challenging because the techniques of [16] no longer work; (ii) we allow, but do not assume or require that the data are partitioned between the two parties; and (iii) unlike [16], we provide a prototype implementation and performance measurements in order to evaluate the scalability of the SMC-based approach to privacy-preserving data classification.

Our contributions. We present a cryptographically secure protocol for privacy-preserving construction of classification trees. The protocol takes place between a user and a server. The user's input consists of the parameters of the classifier that he wishes to construct: which data attributes (columns) to use as feature attributes, which as the class attribute, and, optionally, which predicate on records (rows) to use in order to select only a subset of the database records for the classifier construction. The server's input is a relational database. We assume that the schema of the database (*i.e.*, names of attributes and the values they can take) is public, but that the actual records are private.

The user's protocol output is a classification tree constructed from the server's data. The server learns nothing from the protocol; in particular, he does not learn the parameters of the classification algorithm, not even which attributes have been used when constructing the classifier. We re-iterate that the latter requirement precludes the server from computing the classifier on his own, and also makes existing protocols inapplicable. Our protocol exploits the structure of the classifier-construction algorithm in a fundamental way. In each node of the classification tree, the records are "split" based on the value of some attribute. In order to pick the best attribute for this purpose, the tree-construction algorithm must, in each node of the tree, count the number of records that fall into several categories. In contrast to [16], the database owner should not learn how many of his own records fall into each category, so we must perform this computation in a privacy-preserving manner. If done naïvely, using generic techniques, the computational cost of the resulting protocol would be prohibitive.

Our key technical innovation is to build the tree "one tier at a time" by simultaneously counting the categories for an entire tier of nodes rather than for a single node. By partitioning the categories into mutually exclusive groups, we are able to compute the counts for a whole tier of nodes using the same number of secure circuit evaluations as we would have needed for a single node. This enables a substantial performance gain which bridges the gap between theoretical and practical efficiency.

Our final contribution is to measure the scalability of our prototype implementation and evaluate its performance on realistic datasets. While theoretical protocol designs in the SMC framework abound, actual implementations have been very rare. This makes it difficult to determine whether these (theoretically sound) techniques can actually be applied, even given modern computing power, to anything other than toy examples. Our performance measurements show the limits of SMC-based privacy-preserving data analysis.

2 Related Work

Classifier learning is one of the most fundamental tasks in data mining and machine learning [20,24]. The privacy-preserving version of the problem was addressed by Lindell and Pinkas [16]. We use the same framework of secure multi-party computation as [16] and provide the same level of cryptographic security. Note, however, that [16] solves a different

problem, where the database is *horizontally partitioned* between the parties, and both participants learn the resulting decision tree. By contrast, we consider an (arguably, much more common) problem where one party may hold all of the data, and a second party wishes to construct a classifier which is not revealed to the first party. Our protocol allows the data to be arbitrarily partitioned between the parties, while still maintaining the property that only one party learns the resulting decision tree.

This distinction is not superficial and has important technical ramifications. The fact that *both* parties learn the classifier is used in an essential way in [16] to implement recursive tree construction: because all nodes of the tree are revealed to all parties as part of the final classifier, the algorithm is allowed to reveal the nodes in intermediate steps. Our problem, where the classifier is *not* revealed to the data owner, cannot be solved using the techniques of [16] and requires more sophisticated algorithms.

Other techniques for privacy-preserving classifier construction [8, 9, 28, 30] also assume that both parties learn the classifier. Therefore, they cannot be applied to our problem setting. Unlike [8, 28], our solution is accompanied by a practical implementation and does not require a third-party server.

In *randomized* databases, statistical noise is added to individual data entries in order to hide their values. Agrawal and Srikant considered the problem of privacy-preserving classifier construction in this setting [3], but their privacy definition as well as several subsequent definitions were very weak [2, 10]. The SuLQ (Sub-Linear Queries) framework enables construction of ID3 classifiers from perturbed data with adequate privacy guarantees [4]. Our approach is different in that our trees are constructed on the original, unperturbed data, and are thus more precise. It can also be applied even to relatively small databases where the sub-linearity constraint would restrict the approach of [4] to a very small number of queries. Furthermore, queries are made in the clear in the SuLQ framework, so only the privacy of the server is guaranteed. In contrast, our approach guarantees the privacy of the user's input.

Another class of techniques for privacy-preserving data publishing is based on k-anonymity [7, 26, 27]. In this approach, some of the attributes (so-called "quasi-identifiers") are transformed so that each attribute tuple occurs at least k times in the anonymized database, while other attributes are released untouched. k-anonymous databases can be used for classifier construction [14]. Limitations of k-anonymity include the fact that it cannot be applied to high-dimensional data [1], k-anonymous databases can reveal individuals' sensitive attributes [15, 18] and/or whether a given individual has an entry in the database [21, 25], and anonymity is not guaranteed against adversaries with background knowledge [18, 19] or even adversaries who simply know the k-anonymization algorithm [32]. This paper provides an alternative way of constructing classifiers that does not involve releasing the data to untrusted users.

An orthogonal problem to *learning* decision trees is that of *evaluating* decision trees so that the data owner does not learn the tree which is being evaluated (*i.e.*, evaluation is oblivious). Recent solutions include [6, 11]. We use a (substantially modified) oblivious tree evaluation protocol
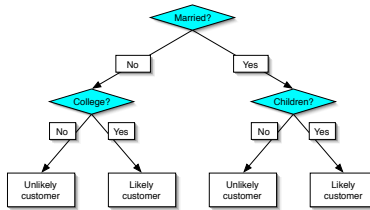
Fig. 1. Example decision tree.

of [6] as a building block. It provides better efficiency in our setting than [11], where each decision node can only examine a single bit.

3 Cryptographic Tools

Our construction employs several standard cryptographic tools, including secure circuit evaluation (SCE) and homomorphic encryption. We only utilize them for the lowest-level computations in our protocol, and, furthermore, we use SCE in a non-black-box fashion. For the standard secure circuit evaluation, we use a compiler [13] which, given a circuit description, generates a corresponding "garbled circuit" following Yao's method [17,31]. Where an additively homomorphic encryption scheme is needed, we use the Paillier cryptosystem [23].

Our protocol also requires a subprotocol for privacy-preserving *evaluation* of decision trees, described in Appendix A. To achieve practical efficiency, we carefully design circuit logic to allow the same set of inputs to be used across multiple garbled-circuit evaluations, which reduces the number of costly oblivious transfers.

4 Problem Formulation

4.1 Decision-tree learning

A *classifier* takes as input a *record* (or transaction) consisting of several attribute values, and outputs a classification label which categorizes the record. *Decision trees* are a common type of classifier. Each internal node in a decision tree examines a single attribute and redirects evaluation to one of several child nodes based on the value of that attribute. Once a leaf node is reached, the classification label contained therein is outputted as the result of classification. Fig. 1 shows an example decision tree that could be used by a marketing department to determine whether a consumer is likely to buy a company's product.

Decision-tree classifiers can be constructed manually by a human expert with domain knowledge, but algorithms for *decision-tree learning* are increasingly popular (*e.g.*, see Algorithm 1). Given a database of records tagged with classification labels, the algorithm constructs the decision tree recursively from the top down. At the root node, the algorithm

Algorithm 1: The (non-private) recursive decision-tree learning algorithm.

considers every attribute and measures the quality of the split that this attribute will provide (see below). The algorithm chooses the "best" attribute and partitions all records by the value of this attribute, creating a child node for each partition. The algorithm is then executed recursively on each partition.

Two popular measures of the "quality" of a split are information gain and the Gini index. Information gain is used in the ID3 and C4.5 algorithms [24], while the Gini index is used in the CART algorithm [5]. Information gain can be computed privately using the $x \log x$ protocol from [16]. Our privacy-preserving protocol for decision-tree learning can use either, but the private computation of the Gini index is more efficient, so we will focus on it.

In the following, suppose that the class attribute (*i.e.*, the target of classification) can assume k different values $c_1, ..., c_k$ and that the candidate splitting attribute A can assume m different values $a_1, ..., a_m$. Denote by $p(c_i)$ the portion of the records whose attribute $C = c_i$, by $p(a_i)$ the portion of the records whose attribute $A = a_i$, and by $p(c_i|a_j)$ the portion of the records that have both attribute $C = c_i$ and attribute $A = a_i$.

The Gini index GINI(A) is computed as:

$$1 - \sum_{i=1}^{k} \left(p(c_i)\right)^2 - \sum_{j=1}^{m} p(a_j) \sum_{i=1}^{k} p(c_i|a_j) \left(1 - p(c_i|a_j)\right) \tag{1}$$

If we use the notation $n(c_i)$ for the *number* of records with attribute $C = c_i$, then we can rewrite (1) as:

$$1 - \sum_{i=1}^{k} \left(\frac{n(c_i)}{|T|}\right)^2 - \sum_{j=1}^{m} \frac{n(a_j)}{|T|} \sum_{i=1}^{k} \frac{n(c_i|a_j)}{|T|} \left(1 - \frac{n(c_i|a_j)}{|T|}\right) \tag{2}$$

Multiplying this equation by $|T|^3$ gives:

$$|T|^3 - \sum_{i=1}^{k} (n(c_i))^2 |T| - \sum_{j=1}^{m} n(a_j) \sum_{i=1}^{k} n(c_i|a_j) (|T| - (c_i|a_j)) \qquad (3)$$

Since the number of records $|T|$ is fixed, we can compare the Gini index of different attributes using only multiplication and addition. These operations can be easily computed in a privacy-preserving manner using Yao's garbled-circuits method.

4.2 Distributed decision-tree learning

Conventional decision-tree learning is performed by a single user. The user has access to some database T and chooses the set of feature attributes $\mathcal{R}$, the class attribute C, and the number of tiers D. In this paper, we focus on a *distributed* setting, where the database T resides on a server and a remote user chooses $\mathcal{R}$, C, and D. We emphasize that for real-world databases, where the total number of attributes is fairly large, $\mathcal{R}$ may be only a small subset of attributes. For example, attributes of T may include hundreds of demographic features, and the user may be interested only in a handful of them for classification purposes.

In the distributed setting, both parties may have privacy concerns. The server wishes to reveal no more about T than is necessarily revealed by a decision tree based on T. The user, on the other hand, may not wish to reveal which feature attributes $\mathcal{R}$ and class attribute C he selected for the purposes of constructing a classifier.

We assume that several parameters are known to both parties: $|T|$, the number of records in the database; $\mathcal{A}$, the set of all attributes in the database; the set $a_1, ..., a_m$ of possible values for each attribute $A \in \mathcal{A}$; $|\mathcal{R}|$, the number of feature attributes selected by the user; and D, the depth of the decision tree to be constructed.

Branching factor. In the general case, the record database T may contain nominal attributes whose domains have different sizes. For instance, a consumer database may have 2 possible values for the "sex" attribute, and 50 possible values for the "state of residence" attribute. We refer to the number of different values that an attribute can assume as its *branching factor*, because it determines the number of children for each internal node corresponding to that attribute.

When the decision tree is computed in a privacy-preserving manner, all internal nodes must have the same number of children in order to prevent the server from learning any information about which attribute is considered in a given node. Therefore, all attributes must have the same branching factor m. As a pre-processing step, attributes can be padded with unused values so that all attributes have the same branching factor. For simplicity, we assume that each attribute value is encoded as an integer between 0 and $m-1$, and can thus be represented using $\log_2 m$ bits.

5   Privacy-Preserving Decision-Tree Learning

Our protocol takes place between a server in possession of a database T and a user who wishes to build a classifier for class attribute C based on a set $\mathcal{R}$ of feature attributes. The tree is constructed from the root down, as in the conventional algorithm shown in Fig. 1. Unlike the conventional algorithm, ours is non-recursive. Instead, the tree is constructed one tier at a time. When processing tier i, m^i pending nodes are considered. In the final tier, the pending nodes are transformed into leaf nodes with classification labels in them; in all intermediate tiers, they become internal decision nodes, where the attribute for making the decision is chosen based on the data in T. We now describe the protocol, which is divided into four phases.

5.1   Phase 1: Sharing the attribute values

The set of attributes $\mathcal{A}$ found in the database T may be far larger than the set $\mathcal{R} \cup \{C\}$ of attributes that are relevant to tree construction. For an attribute $R_i \in \mathcal{R} \cup \{C\}$ and a record $t \in T$, $t[R_i]$ refers to the attribute value for attribute R_i in record t. For each record $t \in T$ and for each relevant attribute $R_i \in \mathcal{R} \cup \{C\}$, Phase 1 enables the user and the server to learn shares $t[R_i]_U$ and $t[R_i]_S$ such that $t[R_i]_U + t[R_i]_S \pmod{m} = t[R_i]$. This is done using the oblivious attribute selection technique from [6], which is outlined below:

1. For all $A_i \in \mathcal{A}$, the server encrypts $t[A_i]$ using an additively homomorphic encryption scheme, and sends $E[t[A_i]]$ to the user.
2. User creates a blinding value b_i for each relevant attribute R_i, and uses the homomorphic property to add b_i and $t[R_i]$ under encryption. User sends $E[b_i + t[R_i]]$ to the server. User's random share is $t[R_i]_U = -b_i \mod m$.
3. Server decrypts to obtain $b_i + t[R_i]$ and stores $t[R_i]_S = b_i + t[R_i] \mod m$.

We use a blinding value b_i at least 80 bits longer than the $(\log_2 m)$-bit value $t[R_i]$ so that it statistically hides $t[R_i]$. The shares $t[R_i]_U$ and $t[R_i]_S$ will be used in later phases as inputs to small Yao circuits that are generated by the user and evaluated by the server. Therefore, the server needs to learn the random wire keys representing his input shares $t[R_i]_S$ in the circuit. As usual, this is done via a 1-out-of-2 oblivious transfer for each of the $\log_2 m$ bits in the $t[R_i]_S$ values, where the server's input is the jth bit of $t[R_i]_S$, and the user's input is the pair of wire keys representing, respectively, 0 and 1 on the input wire corresponding to this bit.

Unlike the standard Yao protocol, the same input-wire keys are used for multiple circuits. The oblivious transfers can thus be done only once per protocol execution instead of once per circuit evaluation. This results in a substantial performance improvement, since the bulk of computation in secure circuit evaluation is spent on the oblivious transfers.

After performing these preliminary steps, the user participates in the PRIVATEDECISIONTREE$(\mathcal{R}, C, D, T)$ protocol with the server, which starts Phase 2.

(a) Incomplete Tree (b) Augmented Tree
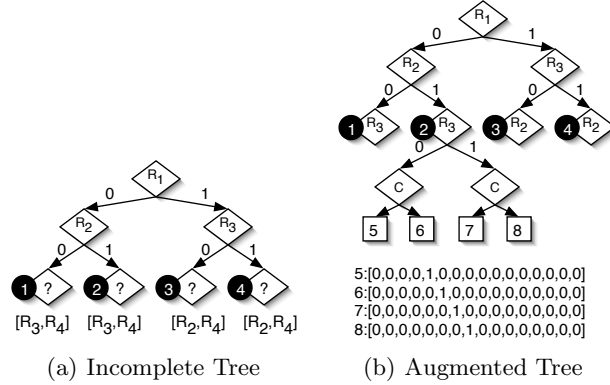
Fig. 2. An incomplete decision tree with 4 pending nodes, and the same tree augmented with a feature attribute and class attribute

We also observe that our protocol can be applied not only in the case where the server holds the entire database, but also for any vertical or horizontal partitioning of the database between the user and the server. If the database is partitioned, the steps described above are carried out only for the attribute values held by the server. For each value held by the user, the user simply splits it into two random shares and sends one of them to the server. Regardless of the database partitioning, after Phase 1 every attribute value of every record is shared between the user and the server.

5.2 Phase 2: Computing category counts

Phase 2 is shown in Algorithm 2 as lines 3–6.

Let d be the depth of the current tier. Within this tier there are m^d pending nodes, and of the original $|\mathcal{R}|$ feature attributes, only $|\mathcal{R}| - d$ remain as candidates to be chosen as the splitting attribute for each pending node because d attributes have already been used. The set of candidate attributes for splitting at a pending node n depends on which attributes were already encountered on the path from the root node to n, and may thus be different for each pending node. For example, Fig. 2(a) shows a tree entering Phase 2 on tier 2. The path to the 3rd pending node consists of the edges $R_1 = 1$ and $R_3 = 0$, so attributes R_1 and R_3 are no longer available as candidates for this node. The candidates for the 3rd pending node are $[R_2, R_4]$, while the candidates for the 2nd pending node are $[R_3, R_4]$,

Let $T(n)$ be the set of records that satisfy the preconditions of node n (for the 3rd pending node in Fig. 2(a), these are records with $R_1 = 1$ and $R_3 = 0$). Let $\{R_{n_1}, ..., R_{n_{|\mathcal{R}|-d}}\}$ be the set of candidate attributes for node n. Finally, let $T_k(n : i, j)$ be the set of records in $T(n)$ that have $R_{n_k} = i$ and $C = j$. To determine the quality of the split that would be provided by choosing R_{n_k} as the splitting attribute for this node, it is necessary to compute $|T_k(n : i, j)|$ for all possible values of i and j ($0 \leq i, j \leq m$).

For any choice of n, i, and j, the user can build a decision tree to determine whether a given record is in $T_k(n : i, j)$. Using oblivious decision-tree evaluation, the user and the server can then learn shares of $|T_k(n : i, j)|$ without either revealing his private inputs. The problem with this naïve approach is that determining the quality of splitting on a single attribute R_{n_k} requires $m^d \cdot m^2$ oblivious decision-tree evaluations on each record in T (one for each choice of n, i, and j).

Our construction is significantly more efficient because it iterates over the database only *once* by counting $m^d \cdot m^2$ different mutually exclusive categories simultaneously. The key observation is that for each record $t \in T$, there is a unique pending node n such that $t \in T(n)$. Furthermore, for each $t \in T(n)$ and $0 \le k \le |\mathcal{R}| - d$, there are unique i, j such that $t \in T_k(n : i, j)$. Therefore, our construction builds a classifier to determine for which values of n, i, and j the record t belongs to $T_k(n : i, j)$.

To do this, we augment the partially constructed tree P by replacing each pending node with a depth-two subtree that considers attributes C and R_{n_k}. Fig. 2(b) shows the result of augmenting the tree from Fig. 2(a) when $k = 1$. (To avoid clutter, the augmented portion is only shown for the 2nd pending node.) The $m^d \cdot m^2$ leaves of the tree contain vectors of length $m^d \cdot m^2$ as their labels. Each leaf is reachable by records in $T_k(n : i, j)$ for a unique choice of n, i, and j, and the vector used as its label has a single "1" in the position corresponding to $T_k(n : i, j)$ and "0" elsewhere.

Once the augmented tree $P' = P.\textsc{AugmentWithAttAndClass}(k, C)$ has been constructed, the user and server engage in a privacy-preserving decision-tree evaluation protocol for each record $t \in T$. To support oblivious evaluation, the tree must be transformed as follows (see [6] and Appendix A for details). Each node other than the root is encrypted with a random key. Each internal node is replaced by a small Yao circuit that takes as its input the user's and server's shares $t[R_i]_U$ and $t[R_i]_S$ of the relevant attribute values $t[R_i]$ for each $R \in \mathcal{R}$, and outputs the index and decryption key for the appropriate child node. Each leaf node has as its label a vector of $m^d \cdot m^2$ values, encrypted using a user-created instance of an additively homomorphic encryption scheme. As described above, the vector has "1" in the position corresponding to its category, and "0" in all other positions. Note that although the same tree is applied to every record, it must be freshly transformed into a secure tree for each oblivious evaluation.

As the result of oblivious evaluation of augmented trees, the server learns a vector of $m^d \cdot m^2$ ciphertexts. All but one are encryptions of "0." The sole ciphertext encrypting "1" occurs in the position corresponding to the category of the record (of course, the server cannot tell which ciphertext this is). By summing up these vectors under encryption, the server obtains ciphertexts encrypting the counts $|T_k(n : i, j)|$. The server must then transform these encrypted counts into additive random shares (mod $|T|$), using the same technique as in Sect. 5.1.

The following subroutines are used during Phase 2:

– $P.\textsc{AugmentWithAttAndClass}$. This method is executed by the user, and adds two tiers to the tree P: one for the attribute R_{n_k} (different for each pending node) and one for the class attribute C.

```
User's Input:
R, the set of feature attributes (|R > D)
C, the class attribute
D, the desired maximum depth
Server's Input: T, the set of records converted into random wire values.
User's Output: P, a decision tree to classify C from R
PROT:PRIVATEDECISIONTREE(user: R, C, D server: T)
 1:  P = new tree
 2:  for d=0 to D − 1 do
 3:    for k=1 to |R| − d do
 4:       P′=P.AUGMENTWITHATTANDCLASS(k, C)
 5:       (|T_k(...)|_U, |T_k(...)|_S) = PROT:ENCRYPTEDCOUNTS(user: P′ server: T)
 6:    end for
 7:    for n=1 to m^d do
 8:       for k=1 to |R| − d do
 9:          (Q_U^k, Q_S^k) = PROT:COMPUTEQUALITY(|T_k(n : ...)|_U, |T_k(n : ...)|_S)
10:       end for
11:       bestatt = PROT:ARGMAX (user: Q_U^1, ..., Q_U^{|R|−d} server: Q_S^1, ..., Q_S^{|R|−d})
12:       In P, make node n an internal node splitting on attribute R_{n_{bestatt}}
13:    end for
14:  end for
15:  P′ = P.AUGMENTWITHCLASS(C)
16:  (|T(...)|_U, |T(...)|_S) = PROT:ENCRYPTEDCOUNTS(user: P′ server: T)
17:  for n=1 to m^D do
18:    bestclass = PROT:ARGMAX(user: |T(n : *, 1)|_U, ..., |T(n : *, m)|_U
                                 server: |T(n : *, 1)|_S, ..., |T(n : *, m)|_S)
19:    In P, make node n a leaf node with label bestclass
20:  end for
21:  return  p
```

Algorithm 2: The private "one-tier-at-a-time" decision-tree learning protocol

- PROT:ENCRYPTEDCOUNTS. This protocol between the user and the server results in the user and server holding shares for the counts $|T_k(n : i, j)|$ for n=1 to m^d, i=1 to m, and j=1 to m. Pseudocode is given in Algorithm 3.

5.3   Phase 3: Selecting the highest-quality split

Phase 3 is shown in Algorithm 2 as lines 7–13.

After Phase 2, the user and the server share counts $|T_k(n : i, j)|$ for all pending nodes n in the tier, and for all values of k, i, and j. This enables them to compute $Gini(R_{n_k})$ for each node n using (3), but over $T(n)$ rather then the entire record set T. The user and server must compute

$$|T(n)|^3 - \sum_{j=1}^{m} |T(n)| \, |T_k(n : *, j)|^2 -$$

$$\sum_{i=1}^{m} |T_k(n : i, *)| \sum_{j=1}^{m} |T_k(n : i, j)| \, (|T(n)| - |T_k(n : i, j)|) \quad .$$

User's Input: A decision tree P with k leaf-nodes. The label of leaf i is a k-length vector with $E[1]$ in position i and $E[0]$ in all other positions.

Server's Input: A record set T for which each bit of each attribute value has been converted into a random wire value.

Output: Let $K = \sum_{t \in T} P(t)$ be the k-length vector whose ith entry is the number of records in T landing in leaf node i. The user's and server's outputs are shares K_U and K_S of K.

ENCRYPTEDCOUNTS(P, T)

1: $K \leftarrow$ length k vector with each entry set to $E[0]$
2: **for each** $t \in T$ **do**
3:     $J \leftarrow$ PRIVATETREEEVAL(P, t)
4:     $K \leftarrow K + J$ under encryption
5: **end for**
6: Split each component of K into shares (mod $|T|$); user decrypts his share

Algorithm 3: Protocol to determine how many records fall into each of k categories

Where $|T(n : *, j)|$ is the number of nodes in $T(n)$ with class attribute $C = j$, and $|T_k(n : i, *)|$ is the number of nodes in $T(n)$ with attribute $R_{n_k} = i$. These values, along with $|T(n)|$, can be computed from the shares $|T_k(n : i, *)|_{\{U,S\}}$ which the user and server hold.

Given the shares (mod $|T|$) of all inputs, a simple circuit produces shares (mod $|T|^3$) of $\mathrm{G}ini(R_{n_k})$ for each node n and for each k, using only addition and multiplication. For each pending node n, these shares are then fed into another garbled circuit. This circuit determines which attribute R_{n_k} provides the best split quality. The user updates the tree P with this information, by replacing the pending node n with an internal node that splits on the attribute R_{n_k}.

The following subroutines are used during Phase 3:

- PROT:COMPUTEQUALITY. This protocol uses a garbled circuit to compute the Gini index for node n and attribute R_{n_k}. This protocol takes as input shares (mod $|T|$) of the m^2 counts $|T_k(n : i, j)|$, and returns shares (mod $|T|^3$) of the Gini index.
- PROT:ARGMAX. This protocol takes as input shares of values $v_1, ..., v_n$ and provides the user with an index m such that v_m is greater than or equal to all other values. The server learns nothing.

5.4   Phase 4: Constructing the bottom tier

Phase 4 is shown in Algorithm 2 as lines 17–20.

Phase 4 completes the decision tree P by adding the correct labels to its leaf nodes. Each leaf node n should have as its label the most common classification value among the records in $T(n)$. Similar to Phase 2, we can find the most popular classification value for all leaf nodes at once. The incomplete tree P is augmented with a single extra tier which examines the classification node C. Then ENCRYPTEDCOUNTS provides the user and server with the shares of the the counts $|T(n : *, j)|$ for n=1 to m^D and j=1 to m. Next, a garbled circuit finds the value c such that $|T(n : *, c)|$ is maximal, and makes it the label for node n.

The following subroutines are used during Phase 4:

- P.AugmentWithClass. This is executed by the user and adds one additional tier to the tree P for the class attribute C.
- Prot:EncryptedCounts. Same as in Phase 2, and provides the user and server with shares (mod $|T|$) of $|T(n : *, j)|$.
- Prot:ArgMax. Same as in Phase 3.

5.5 Security properties

Due to space constraints, we omit the detailed security argument. We use the same secure multi-party computation framework as the original protocol by Lindell and Pinkas [16] (which applied to a different decision-tree learning problem, as explained above). Just like [16], our basic protocol is secure against a passive attacker. Note that the decision tree resulting from protocol execution has a rich structure and may reveal a substantial amount of information about the database to the user. As is standard in the SMC framework, we do not prevent privacy violations that occur as a result of the protocol output; instead, we guarantee that no *additional* information is revealed.

If the underlying oblivious transfer protocol (used by the data owner to obtain wire-key representations of the records in his database) is secure against an actively malicious chooser, and the server's homomorphic encryption scheme (an instance of which is used during the the oblivious attribute selection protocol) can be verified as well-formed by the user, then our protocol is also secure against an actively malicious data owner. Recall that the data owner plays the role of an (oblivious) circuit evaluator in our protocol.

To obtain security against an actively malicious user, it is necessary to ensure that (a) the oblivious transfer protocol is secure against an actively malicious sender, (b) the user's instance of the homomorphic encryption scheme (used when obliviously counting the sizes of record categories) can be verified as well-formed by the server, and (c) the server can verify that the garbled circuits created by the user are well-formed. Note that the latter can be achieved at a constant additional cost under certain number-theoretic assumptions (*e.g.*, see [12]).

6 Performance

Recall that there are $|\mathcal{A}|$ attributes, each of which has a branching factor of m; $|\mathcal{R}|$ feature attributes; $|T|$ transactions, and depth D. In evaluating the performance of our protocol, we distinguish between online and offline computations. Offline computations include generating m^{d+2} homomorphic encryptions of "0" for each of the $|T|(|\mathcal{R}| - d)$ augmented decision trees used at tier d (user); generating homomorphic encryptions of $|T||\mathcal{A}|$ attributes for oblivious attribute selection (server); garbling of circuits to compute the Gini index and ArgMax (server), and garbling of circuits to compute attribute selection (user). Note that the number of gates in these circuits depends on $|\mathcal{A}|$ and $|T|$ (Gini), $|T|$ and m (ArgMax), and $|\mathcal{A}|$ and m (attribute selection).

	sym. enc	sym. dec	homo. dec	homo. add	OTs	eval
user	114s	0s	7.1s	0.07s	185.2s	4.2s
server	0s	171s	8.0s	41.9s		12.7s

Table 1. Runtime for the "cars" dataset from the UC Irvine repository.

The following cryptographic operations must be performed online once per protocol execution: $|T||\mathcal{R}|$ homomorphic additions for oblivious attribute selection (user); $|T||\mathcal{R}|$ homomorphic decryptions (server); and $|T|(|\mathcal{R}|+1)\log m$ 1-out-of-2 oblivious transfers so that the server can learn wire values for his attribute shares. In addition, the following are performed online to construct tier d (with m^d nodes): symmetric encryption of $(\sum_{h=1}^{d+2} m^h)$ garbled nodes for each of $|T|(|\mathcal{R}|-d)$ augmented decision trees (user); $d+2$ symmetric decryptions and evaluations of garbled attribute selection circuits for each of $|T|(|\mathcal{R}|-d)$ augmented decision trees (server); $|T|(m^{d+2})$ homomorphic additions (server); m^{d+2} homomorphic decryptions (user); evaluation of $m^d(|\mathcal{R}|-d)$ garbled circuits to compute the Gini index at tier d (user); and evaluation of m^d garbled circuits for ArgMax at tier d (user).

Because performance is often a concern when using secure multi-party computation techniques, we evaluated a prototype Java implementation of our protocol. Fig. 3 shows how the online time required by our protocol depends on several parameters of the decision-tree learning problem: the branching factor, the number of feature attributes, the number of tiers, and the number of records. Online time is *independent* of the number of attributes. This makes our protocol especially well-suited to scenarios where the set of feature attributes is a relatively small subset drawn from a very large set of total attributes. Note that this is the common case for databases with demographic information.

To evaluate performance on real-world data, we applied our protocol to the "cars" dataset from the UC Irvine machine-learning repository. This dataset has 1728 records and 7 attributes with a branching factor of 4. We chose to build a tree with 5 feature attributes and 2 tiers. Table 1 shows the time consumed by different online components of our protocol. This experiment demonstrates that, unlike generic techniques, our protocol can be successfully applied to problem instances of realistic size.

7 Conclusions

The field of privacy-preserving data mining has two approaches to the problem of executing machine-learning algorithms on private data. One approach sanitizes the data through suppression and generalization of identifying attributes and/or addition of noise to individual data entries. The sanitized version is then published so that interested parties can run any data-mining algorithm on it.

The other approach is to use cryptographically secure multi-party computation techniques to construct protocols that compute the same answer as would have been obtained in the non-private case. This approach has
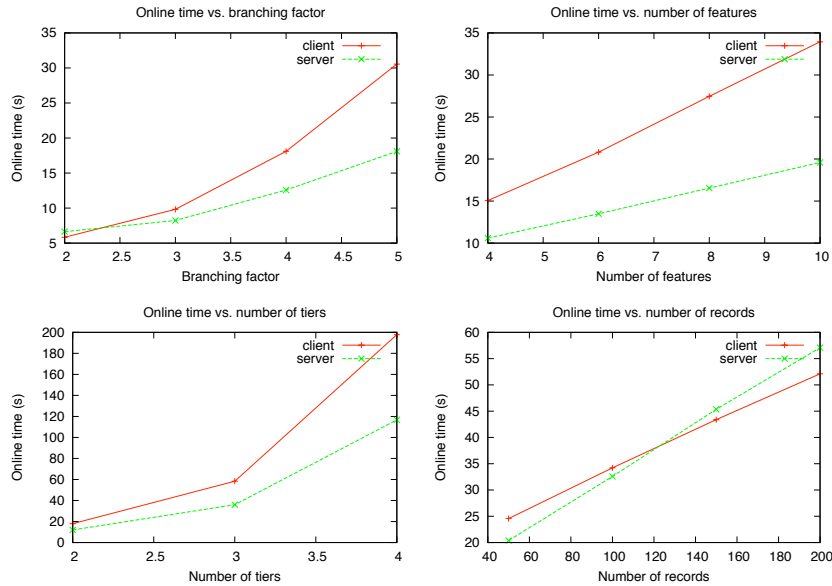
Fig. 3. Online performance of the prototype implementation

typically been applied when the relationship between the parties is symmetric: for example, the database is partitioned between them and the result of the protocol execution is that both parties learn the same output based on the joint database. By contrast, in the sanitization approach, the parties executing the data-mining algorithms do not have any data of their own, while the database owner obtains no output at all.

Even if the data-mining algorithms are the same (*e.g.*, classifier learning), the privacy-preserving versions for the two settings are substantially different. We argue that settings where data are asymmetrically distributed and only one party learns the output are very natural in real-world scenarios. In this paper, we show that it is possible to apply secure multiparty computation techniques to these scenarios. Our protocol requires several technical innovations (such as the ability to obliviously compute the sizes of several record categories in a single pass over the database). Unlike most designs in the literature, our protocol has been implemented, and we demonstrated that it can be efficiently applied even to problem instances of realistic size.

References

1. C. Aggarwal. On k-anonymity and the curse of dimensionality. In *VLDB*, 2005.

2. D. Agrawal and C. Aggarwal. On the design and quantification of privacy-preserving data mining algorithms. In *PODS*, 2001.

3. R. Agrawal and R. Srikant. Privacy-preserving data mining. In *SIGMOD*, 2000.

4. A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *PODS*, 2005.

5. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

6. J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *CCS*, 2007.

7. V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-anonymity. *Secure Data Management in Decentralized Systems*, 2007.

8. W. Du and Z. Zhan. Building decision tree classifier on private data. In *ICDM*, 2002.

9. C. Dwork and K. Nissim. Privacy-preserving data mining on vertically partitioned databases. In *CRYPTO*, 2004.

10. A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy-preserving data mining. In *PODS*, 2003.

11. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *TCC*, 2007.

12. S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, 2007.

13. L. Kruger. Sfe-tools. `http://pages.cs.wisc.edu/~lpkruger/sfe/`, 2008.

14. K. LeFevre, D. DeWitt, and R. Ramakrishnan. Workload-aware anonymization. In *KDD*, 2006.

15. N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and ℓ-diversity. In *ICDE*, 2007.

16. Y. Lindell and B. Pinkas. Privacy preserving data mining. *J. Cryptology*, 15(3):177–206, 2002.

17. Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. `http://eprint.iacr.org/2004/175`, 2004.

18. A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

19. D. Martin, D. Kifer, A. Machanavajjhala, J. Gehrke, and J. Halpern. Worst-case background knowledge for privacy-preserving data publishing. In *ICDE*, 2007.

20. T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

21. M. Nergiz, M. Atzori, and C. Clifton. Hiding the presence of individuals from shared database. In *SIGMOD*, 2007.

22. Netflix. Netflix Prize. `http://www.netflixprize.com/`, 2006.

23. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

24. J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

25. V. Rastogi, D. Suciu, and S. Hong. The boundary between privacy and utility in data publishing. In *VLDB*, 2007.

26. P. Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. on Knowledge and Data Engineering*, 13(6), 2001.

27. L. Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, 2002.
28. J. Vaidya and C. Clifton. Privacy-preserving decision trees over vertically partitioned data. In *DBSec*, 2005.
29. J. Vaidya, M. Kantarcioglu, and C. Clifton. Privacy-preserving Naive Bayes classification. *The VLDB Journal*, 17(4), 2008.
30. Z. Yang, S. Zhong, and R. Wright. Privacy-preserving classification of customer data without loss of accuracy. In *SDM*, 2005.
31. A. Yao. How to generate and exchange secrets. In *FOCS*, 1986.
32. L. Zhang, S. Jajodia, and A. Brodsky. Information disclosure under realistic assumptions: Privacy versus optimality. In *CCS*, 2007.

A   Privacy-preserving evaluation of decision trees

Our subprotocol for privacy-preserving evaluation of decision trees is inspired by [6], with several substantial modifications. In [6], attributes can take one of a large number of different values, and each internal node selects one of two children based on a threshold comparison. In this paper's setting, each attribute takes one of m values (m is relatively small), and internal nodes have m children—one for each attribute value. The privacy requirement is that this evaluation should be oblivious: the evaluator should not learn anything about the structure of the tree except the total number of nodes and the length of the evaluation path, nor *which* of his attributes were considered during evaluation. To achieve the former, the tree is represented as a set of encrypted nodes; decrypting each node reveals the index of the next node (which depends on the value of the attribute considered in the parent node) and the corresponding decryption key. To hide which attribute is considered in each node, the "oblivious attribute selection" protocol [6] splits each of the attributes that will be used during evaluation into two random shares. The circuit creator receives one share and the evaluator receives the other, without learning to which of his attributes this share corresponds.

Each oblivious evaluation of an internal node results in moving control to one of the m child nodes. Unlike in [6], where each node has only two children, we are no longer able to encode the indices and decryption keys for all possible child nodes in the garbled values corresonding to a single output wire. Instead, we use $\log_2$ output wires for every internal node. Each such node is implemented as a circuit which reassembles the two shares of the attribute a considered in this node ($a^E + a^C = a \mod m$, where a^E is the circuit evaluator's share, and a^C is the circuit creator's) and outputs the value of a using $\log_2 m$ output wires. As in the standard Yao's construction, each wire has two random keys associated with it, representing, respectively, 0 and 1. These random keys are used to encrypt a table with m randomly permuted rows (observe that there is a 1:1 correspondence between the rows, all possible values of a, and all possible combinations of bit values on the $\log_2 m$ output wires). For each value of a, the encrypted row contains the index of and the decryption

key for the appropriate next node in the evaluation, encrypted under the output-wire keys corresponding to the bit representation of a.

For instance, suppose that $m = 4$, so that each attribute takes values from 0 to 3, and thus each internal node in the tree has 4 children. We represent each node by a gate with two output wires, w_0, w_1. Let w_i^0 and w_i^1 be the random keys representing, respectively, 0 and 1 values for wire i. If the bit representation of a is $\alpha\beta$, then evaluating this gate reveals to the evaluator w_0^α and w_1^β. Note that the evaluator does not learn a.

Let h_a be the string containing the index and the decryption key for the child node corresponding to the attribute value a. The gate is accompanied by a random permutation of the following 4 ciphertexts: $\{\{h_0\}_{w_0^0}\}_{w_1^0}$, $\{\{h_1\}_{w_0^1}\}_{w_1^0}$, $\{\{h_2\}_{w_0^0}\}_{w_1^1}$, $\{\{h_3\}_{w_0^1}\}_{w_1^1}$. Observe that the keys w_0^α and w_1^β decrypt exactly one row of this table, namely, the row corresponding to a. By decrypting it, the evaluator can proceed to the correct child node.

We need another technical trick so that the decision-tree evaluation protocol can be efficiently invoked multiple times on the same set of attributes. Recall that as the result of oblivious attribute selection, the evaluator has a random share for each of his attributes that will be used in some internal decision node. For internal nodes, we have the evaluator provide as input shares $a_1^E, ..., a_r^E$ of *all* attribute values, while the creator's input is the index i of the attribute considered by the node, and the creator's share a_i^C of this attribute value. The output of the circuit is $a_i^E + a_i^C \pmod{m}$.

With this circuit logic, the evaluator's input is the same for all nodes of all trees created during our protocol. This enables a substantial efficiency gain. Instead of generating random wire keys for each bit of the evaluator's input into each circuit (as in the standard Yao's method), we generate them once, and then re-use this representation for the evaluator's input wires in all circuits. This allows us to perform only a single set of oblivious transfers to provide the evaluator with the the wire keys corresponding to his input bits. These wire keys are then used in all of the garbled circuits.

B   Horizontal Selection

In many applications of decision-tree learning, the user wants to construct a classifier using the records defined by a certain predicate, *i.e.*, from a *horizontal* subset of the database. In other words, the user selects not only a subset of columns to use as features, but also a subset of records (rows), and the protocol should construct a classifier using the data in the selected records only.

This is motivated by real-world scenarios. For example, a proprietary database may contains records for diverse individuals living throughout a nation, while the user is interested in building a marketing classifier only for consumers from a particular region or those belonging to a particular demographic. In this scenario, the user may wish to keep his record selection criterion private so as to avoid revealing his marketing strategy to competitors. Previous protocols for privacy-preserving decision-tree

learning cannot solve this problem because, by their design, they reveal the resulting classifier to all protocol participants.

In this scenario, we assume that the user does not have a vertical partition of the database, and, since he does not have access to the database, cannot explicitly specify the indices of the records which satisfy his selection criterion. Instead, he must choose them *implicitly* by providing a selection predicate to be evaluated on all records in the database. The user wants to keep this predicate private from the server. Depending on the scenario, the number of records which satisfy the predicate may need to be revealed to the user, to the server, to both, or to neither.

We will outline an extension to our protocol for the variant in which the number of satisfying records is revealed to the user but not to the server. This variant has some useful properties: the user may not believe that the classifier is of high quality if it is based on too few records (thus it is helpful for the user to know how many records were used in constructing the tree), while the server learns a significant amount of information about the user's predicate if he learns the number of records which satisfy the predicate (thus the user may prefer to have this number hidden from the server). This particular variant does present some privacy risks to the server: if the predicate, which is hidden from the server, selects a very small subset of records, then the resulting decision tree will leak a lot of information about the records in the selected subset.

The extension involves two components: (1) an additional phase of the protocol, in which the user learns the indices of all records in the database that satisfy his selection predicate, and (2) a slight change to the category-counting phase to ensure that the records *not* selected by the user's predicate are not counted as belonging to any category, and thus do not participate in determining the best attributes for each internal decision node of the classifier.

To determine the indices of the records that satisfy the predicate, the user and the server engage in an instance of the oblivious decision-tree evaluation protocol described in Appendix A. The user's predicate is represented as a decision tree which evaluates a record and labels it with true if it satisfies the predicate and false otherwise. This decision tree is then obliviously evaluated for each record in the database T. The protocol of Appendix A guarantees that the results are revealed only to the user, and not to the data owner.

The records *not* satisfying the predicate (*i.e.*, those which the user's predicate evaluated to false) should not be used when constructing the classifier. Recall from Sect. 5.2 that in order to determine the best splitting attribute for each internal node of the classifier, the user builds decision trees whose labels are vectors of ciphertexts that all encrypt "0," except for a single ciphertext—in the position corresponding to the record's category—that encrypts "1." For the records that he wants to "turn off," the user simply constructs the tree where the labels contain encryptions of "0" only. This effectively means that the corresponding record is not included in any of the $T_k(n : i, j)$ categories, and thus has no influence on the Gini index computation which is used to find the best splitting attribute.