Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning

Phillipp Schoppmann
Humboldt-Universität zu Berlin
schoppmann@informatik.hu-berlin.de

Adrià Gascón
The Alan Turing Institute / University of Warwick
agascon@turing.ac.uk

Mariana Raykova
Google
mpr2111@columbia.edu

Benny Pinkas
Bar-Ilan University
benny@pinkas.net

ABSTRACT

Exploiting data sparsity is crucial for the scalability of many data analysis tasks. However, while there is an increasing interest in efficient secure computation protocols for distributed machine learning, data sparsity has so far not been considered in a principled way in that setting.

We propose sparse data structures together with their corresponding secure computation protocols to address common data analysis tasks while utilizing data sparsity. In particular, we define a Read-Only Oblivious Map primitive (ROOM) for accessing elements in sparse structures, and present several instantiations of this primitive with different trade-offs. Then, using ROOM as a building block, we propose protocols for basic linear algebra operations such as Gather, Scatter, and multiple variants of sparse matrix multiplication. Our protocols are easily composable by using secret sharing. We leverage this, at the highest level of abstraction, to build secure protocols for non-parametric models (k-nearest neighbors and naive Bayes classification) and parametric models (logistic regression) that enable secure analysis on high-dimensional datasets. The experimental evaluation of our protocol implementations demonstrates a manyfold improvement in the efficiency over state-of-the-art techniques across all applications.

Our system is designed and built mirroring the modular architecture in scientific computing and machine learning frameworks, and inspired by the Sparse BLAS standard.

KEYWORDS

secure computation, machine learning, sparsity

ACM Reference Format:
Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. 2019. Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3339816

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3339816

1 INTRODUCTION

Machine Learning (ML) techniques are today the de facto way to process and analyze large datasets. The popularity of these techniques is the result of a sequence of advances in several areas such as statistical modeling, mathematics, computer science, software engineering and hardware design, as well as successful standardization efforts. A notable example is the development of floating point arithmetic and software for numerical linear algebra, leading to standard interfaces such as BLAS (Basic Linear Algebra Subprograms): a specification that prescribes the basic low-level routines for linear algebra, including operations like inner product, matrix multiplication and inversion, and matrix-vector product. This interface is implemented by all common scientific computing frameworks such as Mathematica, MATLAB, NumPy/SciPy, R, Julia, uBLAS, Eigen, etc., enabling library users to develop applications in a way that is agnostic to the precise implementation of the BLAS primitives being used. The resulting programs are easily portable across architectures without suffering performance loss. The above libraries, and their variants optimized for concrete architectures, constitute the back-end of higher-level machine learning frameworks such as TensorFlow and PyTorch.

In this work, we present a framework for privacy-preserving machine learning that provides privacy preserving counterparts for several basic linear algebra routines. The new tools that we construct mirror the techniques of scientific computing which leverage sparsity to achieve efficiency. Our framework enables computation on inputs that are shared among several different parties in a manner that does not require the parties to reveal their private inputs (only the output of the computation is revealed). In settings where the input parties are regulated by strict privacy policies on their data, such privacy guarantees are a crucial requirement to enable collaborations that are beneficial for all participants. There are many example scenarios that have these characteristics: hospitals that want to jointly analyze their patients' data, government agencies that want to discover discrepancies across their databases, companies that want to compute on their joint user data, and many others.

The main novel aspect of our work is exploiting data sparsity for scalability, by tailoring the basic operations in our framework for that purpose. This functionality is analogous to the one provided by the *Sparse* BLAS interface [13], a subset of computational routines in BLAS with a focus on unstructured sparse matrices. The constructions that we develop for the basic building blocks in our framework are cryptographic two-party computation protocols,

which provide formal privacy guarantees for the computation [15]. We optimize our protocols for the setting where the sparsity level of the input data is not a sensitive feature of the input that needs to be kept secret. This is the case in many datasets where a bound on a sparsity metric is public information. For example, text data where the maximum length of the documents in the training dataset is public, or genomic data, as the number of mutations on a given individual is known to be very small. Similarly to Sparse BLAS implementations, sparsity allows us to achieve substantial speed-ups in our secure protocols. These efficiency gains translate to the efficiency of the higher-level applications that we develop in our framework, as is described in Section 1.1.

Sparse linear algebra on clear data relies on appropriate data structures such as coordinate-wise or Compressed Sparse Row (CSR) representations for sparse matrices. For the MPC case, we develop a similar abstract representation, which we call *Read-Only Oblivious Map* (ROOM). A significant aspect of this modular approach is that our alternative back-end implementations of the ROOM functionality immediately lead to different trade-offs, and improvements, with regards to communication and computation. This also allows MPC experts to develop new efficient low-level secure computation instantiations for the ROOM primitive. These can then be seamlessly used by non-experts to develop higher level tools in a way that is agnostic to many of the details related to secure computation. Such usage of our framework will be similar to to how data scientists develop high-level statistical modeling techniques while benefiting from the high performance of back-ends of ML frameworks.

1.1 Contributions

We present a modular framework for secure computation on sparse data, and build three secure two-party applications on top of it. Our secure computation framework (depicted in Figure 1) emulates the components architecture of scientific computing frameworks. We define a basic functionality and then design and implement several secure instantiations for it in MPC; we build common linear algebra primitives on top of this functionality; and then we use these primitives in a black-box manner to build higher level machine learning applications. More concretely, we present the following contributions:

(1) A Read-Only Oblivious Map (ROOM) data structure to represent sparse datasets and manipulate them in a secure and composable way (Section 4).

(2) Three different ROOM protocol instantiations (Section 4.2) with different trade-offs in terms of communication and computation. These include a basic solution with higher communication and minimal secure computation (Basic-ROOM), a solution using sort-merge networks that trades reduced communication for additional secure computation (Circuit-ROOM), and a construction that leverages fast polynomial interpolation and evaluation (Poly-ROOM) to reduce the secure computation cost in trade-off for local computation, while preserving the low communication.

(3) We leverage our ROOM primitive in several sparse matrix-vector multiplication protocols (Section 5.2), which are optimized for different sparsity settings. We also show how to implement sparse gather and scatter operations using our ROOM primitive.

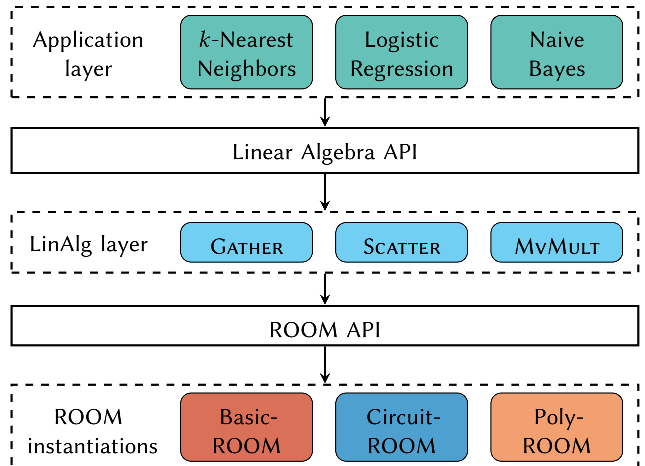


Figure 1: Components of our system.

(4) We build three end-to-end ML applications using our framework. The resulting protocols significantly improve the state of the art, as discussed below.

Our three chosen applications are k-nearest neighbors (Section 6.1), stochastic gradient descent for logistic regression training (Section 6.2), and naive Bayes classification (Appendix B). We evaluate the performance of these applications on real-world datasets (Section 8) and show significant improvements over the state of the art:

- For k-NN, previous work [35] already exploits sparsity using a hand-crafted sparse matrix multiplication protocol. We show that using our protocols with the appropriate choice of the ROOM primitive can reduce the online running time by up to 5x.
- Our sparse stochastic gradient descent implementation improves upon the total runtime of the dense protocol by Mohassel and Zhang [29] by factors of 2x–94x, and improves communication by up to 215x.
- Our protocol for naive Bayes classification scales to datasets with tens of thousands of features, while the previous best construction by Bost et al. handled less than a hundred [5].

2 OVERVIEW AND SETUP

How to exploit sparsity, and implications for privacy. Two properties of real-world data representations used for automated analysis are (a) their high dimensionality and (b) their sparsity. For example, the Netflix dataset [4] contains ~480K users, ~17K movies, but only ~100 million out of ~8.5 billion potential ratings, less than 2%. In another common machine learning benchmark, the 20Newsgroups dataset [34], the training data consists of just over 9000 feature vectors with 10^5 dimensions, but less than 100 (0.1%) non-zero values in each vector. Finally, in Genome-Wide Analysis Studies (GWAS), where the goal is to investigate the relationship between genetic mutations and specific diseases, the relevant mutations are limited to only about 5 million locations, while a full human genome contains ~3.2 billion base pairs [10].

To cope with memory limits, and speed up computations on sparse data in general, several data structures have been developed that exploit sparsity in the data by only storing relevant (i.e., non-zero) values. For a vector $\mathbf{v}$, a straightforward approach is to store only pairs $((i, v_i))_{v_i \neq 0}$. For sparse matrices, this generalizes to Compressed Sparse Row representation, where all rows are successively stored in the above fashion, and an additional row-index array stores pointers to the beginning of each row. Linear algebra libraries such as SciPy and Eigen provide implementations of these sparse vectors and matrices [17, 21], and databases for genomic data use similar sparse storage formats [11].

Note that sparse data representation does not only reduce the storage overhead, but is also the basis for more efficient algorithms. For example, a matrix-vector product, where the matrix is stored as CSR, is independent of the number of columns in the original data and only depends on the number of rows and the number of non-zero values in the matrix. For the examples above, where columns correspond to hundreds of thousands of features, this saves large amounts of computation.

In this paper we show how to obtain the same benefits from sparsity in the secure distributed ML setting, revealing only the sparsity metric of the underlying data while hiding where in the input the sparsity occurs. There are many scenarios where the sparsity metric can be revealed safely without compromising privacy guarantees: that value might already be public (as with the GWAS example above), or a reasonable upper bound can be set in advance. The main challenge is hiding the *locations* of the non-zero values in the data, which are revealed in the plaintext algorithms for the above sparse data structures. Revealing those indices can leak private information. For example, in the common bag-of-words representation for text data, words in the input vocabulary correspond to columns of a sparse matrix. Revealing the columns where a particular row is non-zero would reveal the words contained in the training document corresponding to that row.

In the remainder of this section, we concretely state our privacy requirements and threat model, and introduce necessary notation and preliminaries.

Threat model. We consider a two-party computation setting with semi-honest parties, and the security of our protocols holds in the common simulation-based paradigm for secure computation [15].

Our computations are over matrices and vectors over a finite domain. In all cases we assume that the sparsity metric of the input set is public. For different protocols this metric will be either the total number of zeros, or the total number of zero rows or columns. In settings where we apply our protocols to collections of rows of a dataset, i.e. batches, the sparsity metric is revealed about the batch. In the context of our applications, the real-world interpretation of the sparsity metric is straightforward. For example, in our logistic regression application, the sparsity metric corresponds to revealing an upper bound on the number of different words in each batch of 128 documents.

The preprocessing model for MPC. Some secure computation protocols adopt the *online-offline* paradigm, which splits the computation work into an offline stage that can be executed before the inputs are available, and an online stage that depends on the concrete input values. We do not optimize our constructions for the online-offline setting but rather focus on minimizing the *total cost* of the protocol.

Related work: Custom MPC Protocols for Sparse Data. Exploiting sparsity for efficiency has been explored before in some concrete privacy preserving applications. The work of Nikolaenko et al. [31] develops a protocol for privacy preserving collaborative filtering, a typical application on the Netflix dataset described above. By disclosing a bound on the number of movies rated by a user and exploiting sorting networks, the proposed solution significantly improves on the naive approach that considers all user-movie pairs.

GraphSC [30] is a framework for secure computation that supports graph parallelization models. The design crucially relies on oblivious sorting, as it enables efficiently running computations expressed as sparse graphs while hiding communication patterns across processors. Another application of oblivious sorting to MPC on sparse graphs is given by Laud [24], albeit in a different threat model (three parties with honest majority).

Finally, as mentioned in Section 1.1, Schoppmann et al. [35] propose a protocol for k-nearest neighbors classification that relies on sparse matrix multiplication.

All of these works rely on oblivious sorting networks [20] and task-specific optimizations. Our ROOM primitive (Section 4) abstracts away from the concrete application by providing a generic interface for secure sparse lookups. In the case of k-NN [35], we show that this directly translates to a significant improvement in the online running time.

3 TOOLS AND NOTATION

Vectors and Matrices. We denote matrices using capital letters such as $\mathbf{M}$, and vectors by letters $\mathbf{u}, \mathbf{v}, \mathbf{r}$. Matrices and vectors are indexed as $\mathbf{M}_{i,j}$ and $\mathbf{u}_i$, and by $\mathbf{M}_{[i..j]}$ we mean the sub-matrix of $\mathbf{M}$ containing rows in the interval $[i..j]$. Our matrices and vectors will take values in a finite domain, which we simply denote as $\mathbb{Z}_{2^\sigma}$, to emphasize that σ bits are needed to encode an element of the domain.

Secret Sharing. We denote a shared value x by $[\![x]\!]$. As all our protocols involve two-party computations between parties P_1 and P_2, $[\![x]\!]$ can be seen as a pair $([\![x]\!]_{\mathsf{P}_1}, [\![x]\!]_{\mathsf{P}_2})$ from which each party has the corresponding entry. Our focus will be on arithmetic secret sharing, where $[\![x]\!]_{\mathsf{P}_1}$ and $[\![x]\!]_{\mathsf{P}_2}$ are values in $\mathbb{Z}_{2^\sigma}$, and x is recovered as $[\![x]\!]_{\mathsf{P}_1} + [\![x]\!]_{\mathsf{P}_2}$. Secret shares are extended to vectors and matrices by sharing each of their entries, and we denote shared matrices and vectors as $[\![\mathbf{M}]\!]$ and $[\![\mathbf{u}]\!]$.

Garbled Circuits. Sometimes in our protocols we state that the parties *engage in a secure computation*. By this we mean that they run a generic protocol for the specified functionality. In all such cases we rely on a garbled circuit protocol [25, 38]. Some of our protocols rely on secure evaluations of a pseudorandom function, which we denote as $\mathsf{F}_K(x)$, for key K and input x.

4 BASIC PRIMITIVE: ROOM

We define Read-Only Oblivious Maps (ROOMs) as a 2-party functionality between a server and a client. For fixed finite sets $\mathcal{K}$ and $\mathcal{V}$ – which we call the domain of keys and values, respectively – the server holds a list of key-value pairs $\mathbf{d} = ((x_1, v_{x_1}), \ldots, (x_n, v_{x_n}))$,

Parties: Server, Client.

Inputs:

Server: key-value pairs $\mathbf{d} \in (\mathcal{K} \times \mathcal{V})^n, \beta \in \mathcal{V}^m$.

Client: Query $\mathbf{q} \in \mathcal{K}^m$.

Outputs (shared): $[[\mathbf{r}]]$ such that $\mathbf{r} \in \mathcal{V}^m$ and $\forall j \in [m]$:

$$\mathbf{r}_j = \begin{cases} v_{\mathbf{q}_j} & \text{if } (\mathbf{q}_j, v_{\mathbf{q}_j}) \in \mathbf{d} \\ \beta_j & \text{otherwise.} \end{cases}$$

Figure 2: Functionality of Shared Output ROOM.

with *unique* keys $x_i \in \mathcal{K}$ and values $v_{x_i} \in \mathcal{V}$, and the client holds a query $(q_1, \ldots, q_m)$, with $q_j \in \mathcal{K}$.

The output of a ROOM is an array $(r_1, \ldots, r_m)$, where for each q_j, if q_j is a key in $\mathbf{d}$ then r_j is equal to the corresponding value, namely $r_j = v_{q_j}$. Otherwise r_j gets a default value $\beta_j \in \mathcal{V}$ chosen by the server (and which might be different for each index j). This mirrors common implementations of a map data structure: for example, in Python `d.get(k, val)` returns the value associated with the key `k` in dictionary `d` if `k` is found, and a default value `val` otherwise. In Java, `d.getOrDefault(k, val)` does the same thing.

Figure 2 formalizes this functionality. Note that, as the output is secret-shared among the two parties, the question of whether the indexes in the client can be chosen adaptively by the client is not relevant. In some cases, when we want a single party to obtain the output, we write *designated output ROOM*. This variant can be trivially implemented by having one party send their shares to the other, or – as all our concrete implementations have a generic MPC phase at the end – omitting the secret-sharing step.

4.1 Existing primitives

Before introducing our instantiations of ROOM, we overview what differentiates our ROOM functionality from existing primitives.

ROOM is related to Private Set Intersection (PSI) (see [33] and references therein). However, the ROOM functionality requires selecting data items based on key comparison and thus not every PSI protocol will directly imply ROOM. In addition, PSI protocols leak the size of the intersection to the client, while it is crucial that a ROOM protocol does not reveal how many indexes in the query were found in the database. Still, extending recent developments on *labeled PSI* [7] and PSI with shared output [9] to the ROOM setting seems to be a promising approach for future improvements.

ROOM can also be constructed using Oblivious RAM [16]. However, ROOM does not need support for writes, and thus the resulting solution will have much overhead that can be avoided.

Private Information Retrieval (PIR), in its *symmetric* variant [14], is another primitive relevant to ROOM. The Keyword PIR variant [8] considers the setting of a database that may not contain items at all indices, which is required for ROOM. Finally, while batching techniques that allow the execution of multiple queries have been developed for PIR [1], they do not directly apply to the keyword variant and also do not always have good concrete efficiency. Thus, from a PIR perspective, our ROOM techniques could be interpreted

Let $\mathbf{d} \in (\mathcal{K} \times \mathcal{V})^n$, $\beta \in \mathcal{V}^m$, $\mathbf{q} \in \mathcal{K}^m$, and K be a PRF key.

Inputs:

Server: $\mathbf{d}, \beta$, K.

Client: $\mathbf{q}$.

Output (shared): $[[\mathbf{r}]] \in \mathcal{V}^m$

ROOM Protocol:

(1) For $i \in \mathcal{K}$, Server encrypts $\mathbf{c}_i \leftarrow (v_i \oplus \mathsf{F}_\mathsf{K}(i))$, where

$$v_i = \begin{cases} val & \text{if } (i, val) \in \mathbf{d} \\ \perp & \text{otherwise.} \end{cases}$$

Server sends $(\mathbf{c}_i)_{i \in \mathcal{K}}$ to Client.

(2) For each $i \in [m]$, the parties run a secure two-party computation where Client inputs $\mathbf{c}_{\mathbf{q}_i}$ and $\mathbf{q}_i$ and Server inputs K and β_i. The secure computation decrypts $\mathbf{c}_{\mathbf{q}_i}$ as $v = \mathbf{c}_{\mathbf{q}_i} \oplus \mathsf{F}_\mathsf{K}(\mathbf{q}_i)$, and reveals shares $[[\mathbf{r}]]$ to Client and Server where

$$\mathbf{r}_i = \begin{cases} v, & \text{if } v \neq \perp, \\ \beta_i, & \text{otherwise} \end{cases}$$

Figure 3: Basic-ROOM Protocol.

as improvements on batched symmetric keyword PIR with shared output.

4.2 Instantiations of ROOM

This section presents three instantiations of the ROOM functionality (Figure 2). As described in Section 4, they can be easily transformed into the designated output variant. The first two constructions are based on generic MPC techniques, while the third instantiation also leverages techniques for oblivious selection using polynomial interpolation.

A naive approach for constructing a ROOM protocol requires *mn* comparisons since each of the client's queries may be present in the server's database. Our ROOM instantiations reduce this many-to-many comparison problem to one-to-one comparisons. The asymptotic behavior of our proposed instantiations of ROOM is presented in Table 1. The online cost distinguishes between local computation and generic MPC computation because the latter has a significantly higher overhead in practice, and hence this distinction is essential for the asymptotics to reflect concrete efficiency.

4.2.1 Basic-ROOM. Our Basic-ROOM protocol, presented in Figure 3, is a baseline construction that does not exploit sparsity in the database $\mathbf{d}$, and instead expands the whole domain of keys $\mathcal{K}$. Namely, the server computes and sends an encrypted answer for each potential query in $\mathcal{K}$. However, as shown in Table 1, the linear dependency on $|\mathcal{K}|$ is limited to the local computation performed by the parties during initialization, whereas the more costly online MPC computation only depends on the length of the ROOM query.

LEMMA 4.1. *The protocol in Figure 3 is a secure instantiation for the ROOM functionality with the following overhead: The initialization includes $O(|\mathcal{K}|)$ work for the server, and $O(|\mathcal{K}|)$ communication to*

Data Structure	*Initialization*			*Answer a query of length m*	
	MPC Runtime	*Local Runtime*	*Comm.*	*MPC Runtime*	*Local Runtime*
Basic-ROOM	-	$O(\lvert\mathcal{K}\rvert)$ (server)	$O(\lvert\mathcal{K}\rvert)$	$O(m)$	$O(m)$ (server and client)
Circuit-ROOM	-	-	-	$O((n+m)\log(n+m))$	$O(n\log(n))$ (server) and $O(m\log(m))$ (client)
Poly-ROOM	-	$O(n\log^2(n))$ (server)	$O(n)$	$O(m)$	$O((m+n)\log^2(n))$ (client)

Table 1: Cost of initializations and execution of our instantiations of ROOM, for a database $\mathsf{d} \in (\mathcal{K} \times \mathcal{V})^n$ held by the server, and a query q of length m. Initialization is defined as preprocessing independent from the query q. We assume the security parameter is constant, and we also do not show factors $\log(\mathcal{K})$ and $\log(\mathcal{V})$. The order of the communication for the online phase (answer length m query) is the same as the MPC Runtime for that phase in all cases.

send the encrypted database to the client. The online phase has an $O(m)$ overhead for the MPC protocol.

Security Sketch. The security of the PRF implies that the client does not learn anything about database items due to the initialization. The secure computation in the next step guarantees that both parties learn only shares of the result.

4.2.2 *Circuit-ROOM.* Our second protocol for ROOM uses secure computation and leverages the following observation. We can compute the ROOM functionality by doing a join between the server's data and the query on their key attribute and then computing a sharing of the vector of the corresponding data items from the server's input. A common algorithm for performing equality joins in databases is the *sort-merge join* [40], where elements of each relation are first sorted individually by the join attribute. Subsequently, the two sorted lists are merged (as in merge sort), yielding a combined list where elements from both tables with equal keys are adjacent. This combined list only needs to be scanned once in order to retrieve all elements of the joined table. In the ROOM setting, note that only the last two steps, merge and iteration, depend on data from both parties, as sorting can be performed locally. This makes this algorithm particularly useful for MPC, since merging of n elements can be performed using a circuit of size $O(n \log n)$ [2], and the circuit for comparing adjacent pairs is linear. A similar approach has been taken in previous works [20, 35] for Private Set Intersection and sparse matrix multiplication, respectively. We call this ROOM instantiation Circuit-ROOM, and describe it in Figure 4.

Note that we can assume without loss of generality that $\mathbf{d}$ and $\mathbf{q}$ are sorted. If they are not, we can extend the MPC protocol that we construct to first compute the sorting with a small $O(m\log(m))$ additive overhead.

The secure computation first arranges the inputs into vectors of triples $\mathbf{w}^{\mathrm{C}}$ and $\mathbf{w}^{\mathrm{S}}$, which consequently are merged by the first and then third component into a vector $\mathbf{v}$. Entries with matching keywords are adjacent in $\mathbf{v}$, and the third component of such entries indicates to which party they belong, i.e., indices greater than 0 indicate entries originally in the client's input.

In Step 2) the protocol computes vectors $\mathbf{b}$ and $\mathbf{c}$. Each entry of these vectors stores information about the result of comparing two adjacent entries of $\mathbf{v}$. In particular, $\mathbf{b}$ stores the selected value (i.e., answer), depending on whether keys of such entries matched. The vector $\mathbf{c}$ stores whether the i-th pair of compared adjacent entries involves a key from $\mathbf{q}$. In that case, $\mathbf{c}_i > 0$, as it corresponds to the index of that key in $\mathbf{q}$. Otherwise $\mathbf{c}_i = 0$. If $\mathbf{c}_i > 0$ then the computation must return an answer in $\mathbf{b}$. The answer is the corresponding value from $\mathbf{d}$ if a match was found, or the corresponding value from β if no match was found.

Next, in Step 3) $\mathbf{b}$ and $\mathbf{c}$ are obliviously shuffled to avoid leakage induced by relative positions of their entries. This is analog to the shuffle step in Sort-Compare-Shuffle PSI [20].

Finally, in step 4), entries of $\mathbf{b}$ that correspond to comparisons with keys from the client are output in shares between the parties, along with the corresponding entries in $\mathbf{c}$. This allows the parties to map their output shares back to the order of the inputs. Note that the shuffling in step 3) makes sure that the indexes at which elements are revealed do not leak any information to either party.

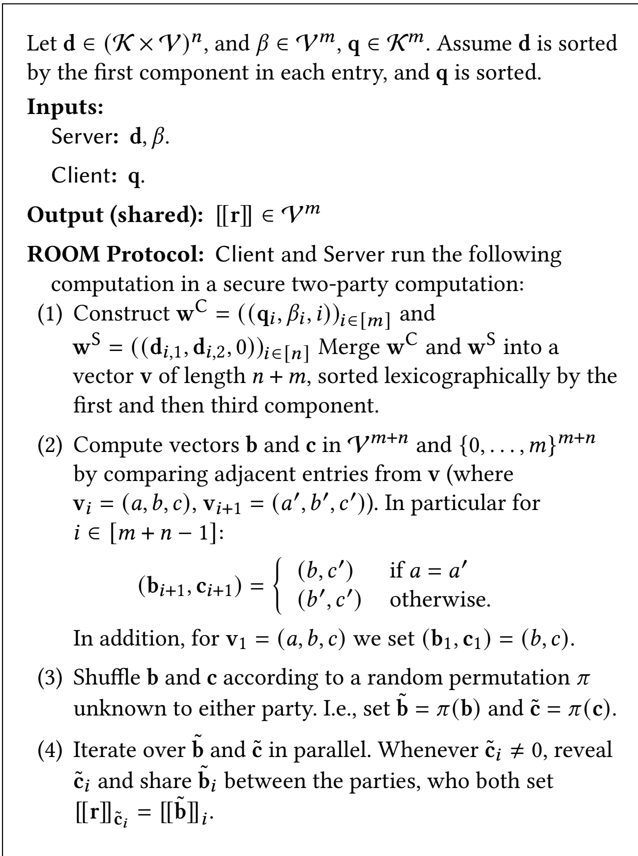
Lemma 4.2. *The protocol in Figure 4 is a secure instantiation for the ROOM functionality with the following overhead. The client and the server run a secure two-party computation protocol whose main bottleneck is computing $O((n+m)\log(n+m))$ comparisons. Additionally, local computations cost $O(n\log n)$ for the server and $O(m\log m)$ for the client.*

The security claim in the above lemma follows directly since our protocol is entirely done in MPC and any additional information revealed beyond the output shares is indistinguishable from random.

4.2.3 *Poly-ROOM.* Finally, as our main instantiation for ROOM, we present a protocol that has MPC runtime similar to Basic-ROOM (independent of n and linear on m), but avoids the dependence on the key domain in initialization.

The main insight for our new construction is that the server can construct a polynomial which evaluates, for inputs which are keys of items in the server's database, to outputs which are encrypted versions of the corresponding values in the server's database. The encryptions are done with a key that is only known to the server. The resulting polynomial is of degree n and is therefore a concise representation of the data. At the same time, the polynomial looks pseudorandom (since it is an interpolation over pseudorandom points), and therefore hides the points which have non-zero values.

The server sends this polynomial to the client. The client then evaluates the polynomial on its inputs and learns m outputs. For each of the client's keys present in the database, the client obtains the encrypted version of the corresponding database value. The two parties then run a secure computation that decrypts each value that the client obtained, checks if it decrypts correctly (i.e., ends with a fixed string of zeros), and reveals to the client either a value


Let $\mathbf{d} \in (\mathcal{K} \times \mathcal{V})^n$, and $\beta \in \mathcal{V}^m$, $\mathbf{q} \in \mathcal{K}^m$. Assume $\mathbf{d}$ is sorted by the first component in each entry, and $\mathbf{q}$ is sorted.

Inputs:

Server: $\mathbf{d}, \beta$.

Client: $\mathbf{q}$.

Output (shared): $[[\mathbf{r}]] \in \mathcal{V}^m$

ROOM Protocol: Client and Server run the following computation in a secure two-party computation:

(1) Construct $\mathbf{w}^{\mathsf{C}} = ((\mathbf{q}_i, \beta_i, i))_{i \in [m]}$ and $\mathbf{w}^{\mathsf{S}} = ((\mathbf{d}_{i,1}, \mathbf{d}_{i,2}, 0))_{i \in [n]}$ Merge $\mathbf{w}^{\mathsf{C}}$ and $\mathbf{w}^{\mathsf{S}}$ into a vector $\mathbf{v}$ of length $n + m$, sorted lexicographically by the first and then third component.

(2) Compute vectors $\mathbf{b}$ and $\mathbf{c}$ in $\mathcal{V}^{m+n}$ and $\{0, \ldots, m\}^{m+n}$ by comparing adjacent entries from $\mathbf{v}$ (where $\mathbf{v}_i = (a, b, c)$, $\mathbf{v}_{i+1} = (a', b', c')$). In particular for $i \in [m + n - 1]$:

$$(\mathbf{b}_{i+1}, \mathbf{c}_{i+1}) = \begin{cases} (b, c') & \text{if } a = a' \\ (b', c') & \text{otherwise.} \end{cases}$$

In addition, for $\mathbf{v}_1 = (a, b, c)$ we set $(\mathbf{b}_1, \mathbf{c}_1) = (b, c)$.

(3) Shuffle $\mathbf{b}$ and $\mathbf{c}$ according to a random permutation π unknown to either party. I.e., set $\tilde{\mathbf{b}} = \pi(\mathbf{b})$ and $\tilde{\mathbf{c}} = \pi(\mathbf{c})$.

(4) Iterate over $\tilde{\mathbf{b}}$ and $\tilde{\mathbf{c}}$ in parallel. Whenever $\tilde{\mathbf{c}}_i \neq 0$, reveal $\tilde{\mathbf{c}}_i$ and share $\tilde{\mathbf{b}}_i$ between the parties, who both set $[[\mathbf{r}]]_{\tilde{\mathbf{c}}_i} = [[\tilde{\mathbf{b}}]]_i$.

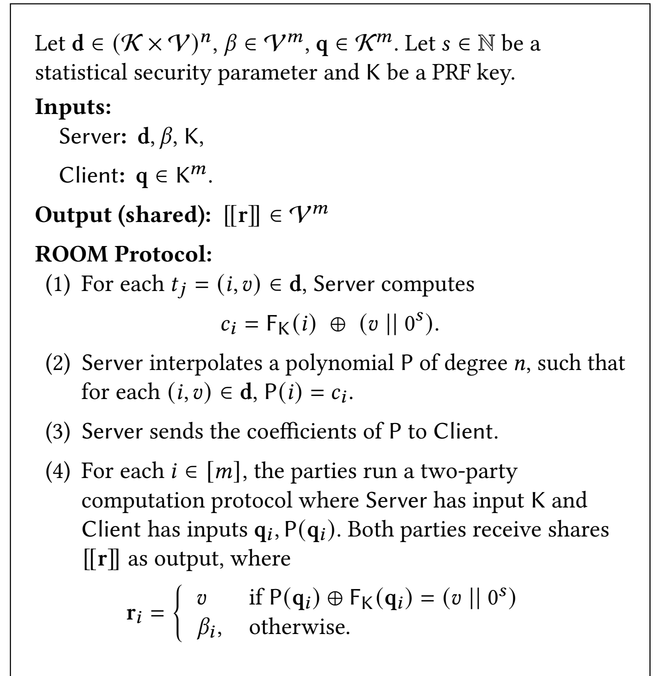
Figure 4: Circuit-ROOM Protocol.

in the database or a default value from β depending on the result of that check. Note that the check passes if the key in the client's query is in $\mathbf{d}$.

Lemma 4.3. *The protocol in Figure 5 is a secure instantiation for the ROOM functionality. The initialization cost includes $O(n \log^2 n)$ work for the server and communication of $O(n)$ to send the polynomial to the client. The online cost of the protocol has $O(m)$ cost for the MPC execution and then $O((m+n)\log^2 n)$ local computation for the client.*

The overhead of the local computation is based on running efficient algorithms for polynomial interpolation and multi-point evaluation, which interpolate a polynomial of degree n in time $O(n \log^2 n)$, and evaluate such a polynomial on n points also in time $O(n \log^2 n)$ [28] (we used an implementation of these protocols in our experiments).

Security Sketch. The security of the construction follows from the fact that the polynomial that the client receives is pseudorandom since the encryptions c_i used in step 2) are pseudorandom. The rest of the computation is implemented in an MPC protocol.


Let $\mathbf{d} \in (\mathcal{K} \times \mathcal{V})^n$, $\beta \in \mathcal{V}^m$, $\mathbf{q} \in \mathcal{K}^m$. Let $s \in \mathbb{N}$ be a statistical security parameter and K be a PRF key.

Inputs:

Server: $\mathbf{d}, \beta, \mathsf{K}$,

Client: $\mathbf{q} \in \mathsf{K}^m$.

Output (shared): $[[\mathbf{r}]] \in \mathcal{V}^m$

ROOM Protocol:

(1) For each $t_j = (i, v) \in \mathbf{d}$, Server computes

$$c_i = \mathsf{F}_{\mathsf{K}}(i) \ \oplus \ (v \mid\mid 0^s).$$

(2) Server interpolates a polynomial P of degree n, such that for each $(i, v) \in \mathbf{d}$, $\mathsf{P}(i) = c_i$.

(3) Server sends the coefficients of P to Client.

(4) For each $i \in [m]$, the parties run a two-party computation protocol where Server has input K and Client has inputs $\mathbf{q}_i, \mathsf{P}(\mathbf{q}_i)$. Both parties receive shares $[[\mathbf{r}]]$ as output, where

$$\mathbf{r}_i = \begin{cases} v & \text{if } \mathsf{P}(\mathbf{q}_i) \oplus \mathsf{F}_{\mathsf{K}}(\mathbf{q}_i) = (v \mid\mid 0^s) \\ \beta_i, & \text{otherwise.} \end{cases}$$

Figure 5: Poly-ROOM Protocol.

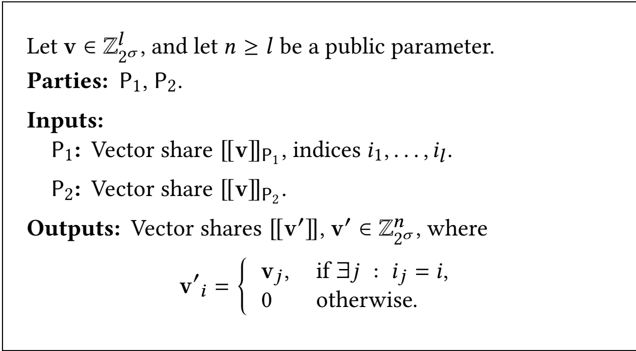
5 ROOM FOR SECURE SPARSE LINEAR ALGEBRA

In this section we present efficient two-party protocols for several common sparse linear algebra operations, which leverage the ROOM functionality in different ways. Similar to how sparse BLAS operations are presented in [13], we first focus on lower-level primitives (Gather and Scatter), and then use them to implement higher-level functionality, namely matrix-vector multiplication. However, we stress that our goal is not to provide implementations of each function described in [13], but instead focus on the operations necessary for the applications described in Section 6.

5.1 Gather and Scatter

Intuitively, the Gather and Scatter operations correspond, respectively, to a sequence of indexed reads from a sparse array, and a sequence of indexed writes into a sparse array. More concretely, Gather takes a vector of indices $\mathbf{q} = (i_1, \ldots, i_n)$ and a (usually sparse) vector $\mathbf{v}$, and returns the vector $\mathbf{r} = (\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_n})$ that results from *gathering* the values from $\mathbf{v}$ at the indices in $\mathbf{q}$. Scatter on the other hand takes a vector of values $\mathbf{v}$, a vector of indices $\mathbf{q} = (i_1, \ldots, i_n)$, and a vector $\mathbf{u}$, and updates $\mathbf{u}$ at each position i_j with the new value $\mathbf{v}_j$.

We transfer the two operations to the two-party setting as follows. Given a sparse vector $\mathbf{v}$ held by Party P_2, and a set of query indices $\mathbf{q}$ held by Party P_1, Gather$(\mathbf{v}, \mathbf{q})$ returns additive shares of a dense vector $\mathbf{v}'$, with $\mathbf{v}'_i = \mathbf{v}_{\mathbf{q}_i}$. It is clear that this is equivalent to a ROOM query with P_2 and P_1 as the server and client, and inputs $\mathbf{q}$, $\mathbf{d} = \{(i, \mathbf{v}_i) \mid \mathbf{v}_i \neq 0\}$, and $\beta = 0$.



Let $\mathbf{v} \in \mathbb{Z}^{l}_{2^\sigma}$, and let $n \geq l$ be a public parameter.
Parties: $\mathsf{P}_1, \mathsf{P}_2$.
Inputs:
P_1**:** Vector share $[[\mathbf{v}]]_{\mathsf{P}_1}$, indices $i_1, \ldots, i_l$.
P_2**:** Vector share $[[\mathbf{v}]]_{\mathsf{P}_2}$.
Outputs: Vector shares $[[\mathbf{v}']]$, $\mathbf{v}' \in \mathbb{Z}^{n}_{2^\sigma}$, where

$$\mathbf{v}'_i = \begin{cases} \mathbf{v}_j, & \text{if } \exists j \; : \; i_j = i, \\ 0 & \text{otherwise.} \end{cases}$$

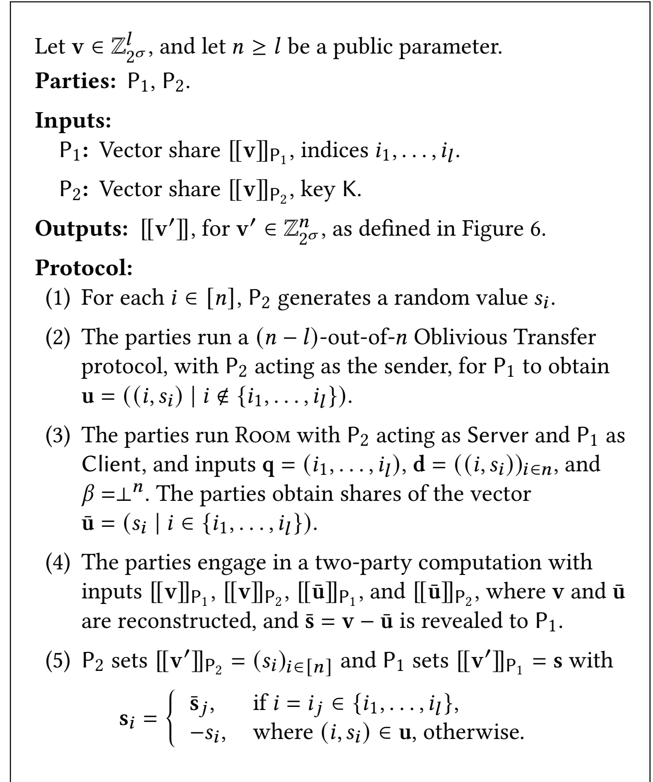
Figure 6: ScatterInit with shared inputs.

For Scatter, we focus on a variant where $\mathbf{u}$ is zero. We call this functionality ScatterInit. Given a dense vector $\mathbf{v}$ and a set of indices $\mathbf{q}$, both of size l, and an integer $n \geq l$, ScatterInit$(\mathbf{v}, \mathbf{q}, n)$ returns a vector $\mathbf{v}'$ of length n such that $\mathbf{v}'_{\mathbf{q}_i} = \mathbf{v}_i$ for all $i \in [l]$, and $\mathbf{v}'_j = 0$ if $j \notin \mathbf{q}$. As in the case of gather, we are interested in secure protocols for ScatterInit that output $\mathbf{v}'$ additively shared. Regarding the inputs, we focus on the case where the input vector is also secret-shared between the parties, while $\mathbf{q}$ is held by one party, and n is a public parameter. The reason for this setting will become apparent when we present our protocol for row-sparse matrix multiplication in Section 5.2.2, as it uses ScatterInit as a sub-protocol.

We formally define the functionality for ScatterInit in Figure 6. A naive implementation using generic MPC would take prohibitive $O(nl\sigma)$ communication and computation. A protocol with reduced communication can be obtained from Function Secret Sharing (FSS) [6], but its local computation remains in $O(nl)$ which turns out to be the bottleneck in practice. In the next paragraph, we describe a version that is *concretely efficient*, trading asymptotic communication complexity against computation time. The two alternatives above are described in Appendix A.

ScatterInit from ROOM and OT Extension. Our protocol is described in Figure 7. The idea is that P_2 generates its random output share $[[\mathbf{v}']]_{\mathsf{P}_2}$ and then P_1 and P_2 execute an MPC protocol from which P_1 obtains (a) all entries of $[[\mathbf{v}']]_{\mathsf{P}_2}$ at indices not in $\{i_1, \ldots, i_l\}$, and (b) its share of the output for the remaining indices, which is obtained securely from P_2's share of the output and the shared input vector $\mathbf{v}$. For (a) we use a well-known $(n-l)$-out-of-n OT protocol that we describe in the next paragraph. For (b) we use a ROOM query followed by a two-party computation where P_2's output share is reconstructed in the MPC (step 4 in Figure 7) and used to mask $\mathbf{v}$ to produce P_1's output share. Note that Basic-ROOM is the natural instantiation to use in this setting.

The $(n-l)$-out-of-n OT in step 2 is implemented using a folklore protocol that requires n invocations of 1-out-of-2 OTs and an l-out-of-n Shamir secret sharing of a PRF key. It works by the sender encrypting each of its n inputs using a key K_{OT}, and then letting the receiver learn in each of the n OTs either an encrypted input or a share (in l-out-of-n secret sharing) of K_{OT}. This forces the receiver



Let $\mathbf{v} \in \mathbb{Z}^{l}_{2^\sigma}$, and let $n \geq l$ be a public parameter.
Parties: $\mathsf{P}_1, \mathsf{P}_2$.
Inputs:
P_1**:** Vector share $[[\mathbf{v}]]_{\mathsf{P}_1}$, indices $i_1, \ldots, i_l$.
P_2**:** Vector share $[[\mathbf{v}]]_{\mathsf{P}_2}$, key K.
Outputs: $[[\mathbf{v}']]$, for $\mathbf{v}' \in \mathbb{Z}^{n}_{2^\sigma}$, as defined in Figure 6.
Protocol:
(1) For each $i \in [n]$, P_2 generates a random value s_i.
(2) The parties run a $(n-l)$-out-of-n Oblivious Transfer protocol, with P_2 acting as the sender, for P_1 to obtain $\mathbf{u} = ((i, s_i) \mid i \notin \{i_1, \ldots, i_l\})$.
(3) The parties run Room with P_2 acting as Server and P_1 as Client, and inputs $\mathbf{q} = (i_1, \ldots, i_l)$, $\mathbf{d} = ((i, s_i))_{i \in n}$, and $\beta = \perp^n$. The parties obtain shares of the vector $\bar{\mathbf{u}} = (s_i \mid i \in \{i_1, \ldots, i_l\})$.
(4) The parties engage in a two-party computation with inputs $[[\mathbf{v}]]_{\mathsf{P}_1}$, $[[\mathbf{v}]]_{\mathsf{P}_2}$, $[[\bar{\mathbf{u}}]]_{\mathsf{P}_1}$, and $[[\bar{\mathbf{u}}]]_{\mathsf{P}_2}$, where $\mathbf{v}$ and $\bar{\mathbf{u}}$ are reconstructed, and $\bar{\mathbf{s}} = \mathbf{v} - \bar{\mathbf{u}}$ is revealed to P_1.
(5) P_2 sets $[[\mathbf{v}']]_{\mathsf{P}_2} = (s_i)_{i \in [n]}$ and P_1 sets $[[\mathbf{v}']]_{\mathsf{P}_1} = \mathbf{s}$ with

$$\mathbf{s}_i = \begin{cases} \bar{\mathbf{s}}_j, & \text{if } i = i_j \in \{i_1, \ldots, i_l\}, \\ -s_i, & \text{where } (i, s_i) \in \mathbf{u}, \text{ otherwise.} \end{cases}$$

Figure 7: Our ScatterInit protocol based on ROOM.

to learn at least l shares of the key, and therefore at most $n-l$ values.

Security Sketch. We argue that the view of each party after each step in the protocol includes only random values in addition to its inputs. This true after the first two steps since first P_2 generates random shares and then, using the security of the oblivious transfer, P_1 obtains a subset of them while P_2 obtains nothing. The security properties of the ROOM protocol guarantee that the shares that each party obtains after the third step are also indistinguishable from random from that party's view. In the secure computation in step four, P_1 obtains values that depend on the random masks $\bar{\mathbf{u}}$. The fact that P_1 does not know $\bar{\mathbf{u}}$, together with the guarantees of the secure computation, ensures the output is indistinguishable from random for P_1. The last step involves only local computation for each party and thus does not change their views.

We will now use the protocols from the previous section to build sparse matrix-vector multiplication protocols in the next section. Concretely, we will use Gather for column sparsity (Section 5.2.1), and ScatterInit for row sparsity (Section 5.2.2).

5.2 Sparse Matrix-Vector Multiplication

Throughout this subsection we consider party P_1 holding a private matrix $\mathbf{M} \in \mathbb{Z}^{n \times m}_{2^\sigma}$, with exactly l nonzero columns or rows, depending on the context. Party P_2 holds a private vector $\mathbf{v} \in \mathbb{Z}^{m}_{2^\sigma}$ with

k nonzero entries. The value of σ is at least 64 in standard ML applications, and potentially more in settings that require additional precision to represent real numbers using fixed point arithmetic, or secret shares. The goal of all protocols is to compute the vector $\mathbf{Mv}$ of length n, *additively shared* between P_1 and P_2. This allows us to easily integrate these protocols as part of higher-level secure protocols, such as the solutions to machine learning problems presented in Section 6. While we assume that n, m, l, k and σ are public, no additional information is revealed to the parties.

The underlying theme of our protocols is different private reductions of sparse matrix-vector multiplication to the dense case. The goal of such reductions is to avoid multiplications by zero, and hence have the cost of the dense multiplication be dependent only on l and k, instead of the total size of the sparse dimension. Therefore, the last step in our protocols will be to use a sub-protocol for two-party dense matrix-vector multiplication. As discussed in Section 2, efficient dedicated protocols for this functionality have been recently presented. This includes solutions based on precomputed triples by Mohassel and Zhang [29], as well as solutions based on homomorphic encryption [22], and server-aided OT [26]. In our protocols in this section we will refer to a generic dense matrix multiplication protocol Dense-Mult, as well as to a generic ROOM protocol Room. The concrete functionality of Dense-Mult takes as input a matrix from one party and a vector from the other party and computes their product as an additive share. In our implementation, we use the protocols from [29].

Parties: P_1, P_2.

Inputs:

P_1**:** Matrix $\mathbf{M} \in \mathbb{Z}_{2^\sigma}^{n \times m}$, with l nonzero columns.

P_2**:** Vector $\mathbf{v} \in \mathbb{Z}_{2^\sigma}^{m}$, with k nonzero entries.

Outputs: $[[\mathbf{Mv}]] = ([[\mathbf{Mv}]]_{\mathsf{P}_1}, [[\mathbf{Mv}]]_{\mathsf{P}_2})$

Protocol:

(1) P_1 sets $\mathbf{q} = (i_1, \ldots, i_l)$, the (sorted) list of indexes of non-zero columns in $\mathbf{M}$.

(2) P_1 and P_2 run Gather$(\mathbf{v}, \mathbf{q})$ to obtain shares $([[\mathbf{v}']]_{\mathsf{P}_1}, [[\mathbf{v}']]_{\mathsf{P}_2})$ of $\mathbf{v}'$, which is the restriction of $\mathbf{v}$ to the indexes in $\mathbf{q}$.

(3) P_1 *locally* computes $\mathbf{M}'$ sub-matrix of $\mathbf{M}$ containing only the nonzero columns.

(4) P_1 and P_2 run Dense-Mult with inputs $\mathbf{M}'$ and $[[\mathbf{v}']]_{\mathsf{P}_2}$ and obtain shares $([[\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}]]_{\mathsf{P}_1}, [[\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}]]_{\mathsf{P}_2})$ of $\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}$.

(5) P_2 sets the output $[[\mathbf{Mv}]]_{\mathsf{P}_2}$ to $[[\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}]]_{\mathsf{P}_2}$ and P_1 sets $[[\mathbf{Mv}]]_{\mathsf{P}_1}$ to $[[\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}]]_{\mathsf{P}_1} + \mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_1}$.

Figure 8: Our first protocol for column-sparse matrix and vector multiplication.

5.2.1 *Column-Sparse Matrix.* We propose two protocols for the case where $\mathbf{M}$ is sparse in the second dimension (i.e., there is a small number of non-zero columns). These have different tradeoffs depending on the relationship between the sparsity of $\mathbf{M}$ and $\mathbf{v}$. Note that matrix-vector multiplication, where the matrix is sparse in its columns, can be viewed as a generalization of sparse vector inner product, and thus the following protocols can also be used for this functionality.

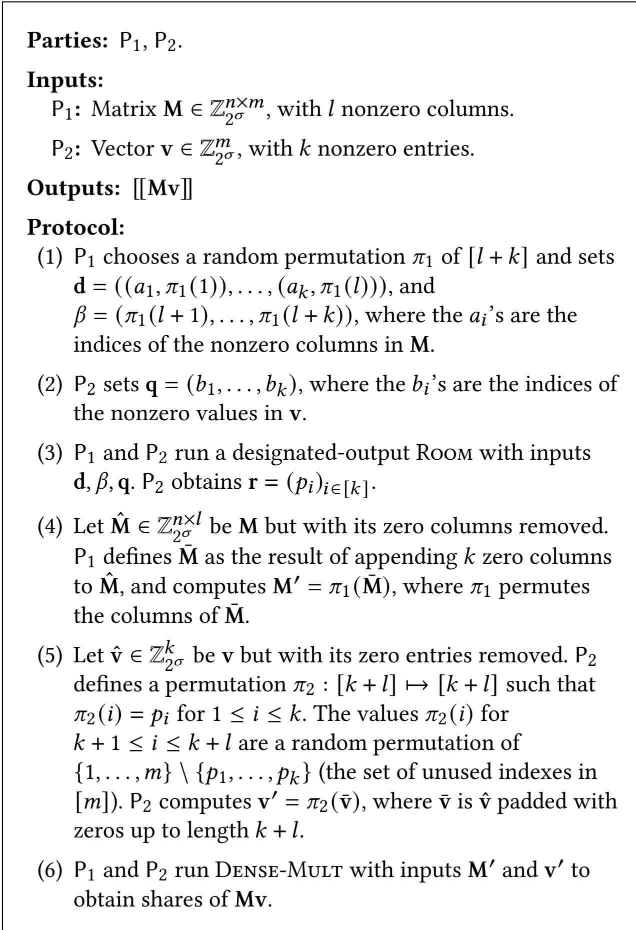
Our first protocol is shown in Figure 8. Let $\mathbf{q} = (i_1, \ldots, i_l)$ be the indexes of the non-zero columns in $\mathbf{M}$. The goal of the sparse-to-dense reduction here is to replace the computation of $\mathbf{Mv}$ by the computation of $\mathbf{M}'\mathbf{v}'$, where $\mathbf{M}'$ is the sub-matrix of $\mathbf{M}$ containing only the non-zero columns $i_1, \ldots, i_l$, and $\mathbf{v}'$ is the restriction of $\mathbf{v}$ to the indices in $\mathbf{q}$. Party P_1 can compute $\mathbf{M}'$ locally. The two parties then call the Gather protocol to obtain shares $([[\mathbf{v}']]_{\mathsf{P}_1}, [[\mathbf{v}']]_{\mathsf{P}_2})$ of $\mathbf{v}'$. At this point the parties invoke the dense matrix multiplication protocol to compute $\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}$. Further, P_1 locally computes $\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_1}$ and adds the result to its share of $\mathbf{M}'[[\mathbf{v}']]_{\mathsf{P}_2}$. As a result, both parties obtain shares of $\mathbf{M}'\mathbf{v}'$. The security of the complete protocol follows directly from the security of the protocols for ROOM and dense multiplication.

There are two drawbacks of the above protocol: (a) the space of values of the ROOM sub-protocol coincides with the domain of the elements of P_2's input vector, $\mathbb{Z}_{2^\sigma}$. This is a problem in high-precision settings where $\sigma > 64$, which are not uncommon in ML applications where real numbers are encoded in fixed point. (b) the length of the ROOM query q is l, which is the sparsity of the server's input matrix. In many settings, the vector v has less non-zero values than M, so $k < l$. That is why we would like to have our ROOM query to only be of the smaller size k, which for two of our constructions directly translates into a speed-up in MPC time (see Table 1). However, if we simply have P_1 act as the server and put the non-zero columns of $\mathbf{M}$ in the ROOM protocol as the database, while $\mathbf{v}$ becomes the query, the values in the ROOM protocol become huge, as it would hold vectors of length n, namely the first dimension of $\mathbf{M}$.

Our next protocol, shown in Figure 9, solves both issues (a) and (b), by relying on a technique based on correlated permutations introduced by Schoppmann et al. [35], which we exploit here by means of the ROOM construction. First, our protocol ensures that the server's input to the ROOM functionality are elements in $\mathcal{K} \times \mathcal{K}$, thus avoiding the dependence on σ. Second, it allows us to swap the roles of P_1 and P_2 in the ROOM protocol, allowing us to choose them depending on the relationship between k and l, as well as other nonfunctional requirements induced by computation and communication limitations of P_1 and P_2.

These two optimizations come at the cost of replacing the input size to the dense multiplication sub-protocol from $2ln\sigma$ to $2(l+k)n\sigma$. Hence, in practice the actual values of $\min(l, k)$, n, and σ determine a trade-off between the protocols in Figure 8 and Figure 9.

The intuition behind the construction in Figure 9 is as follows. Let $\hat{\mathbf{M}}$ and $\hat{\mathbf{v}}$ be the result of removing zero columns and entries of $\mathbf{M}$ and $\mathbf{v}$, as defined in Figure 9, and let $\bar{\mathbf{M}}$ and $\bar{\mathbf{v}}$ be $\hat{\mathbf{M}}$ and $\hat{\mathbf{v}}$ padded with k zero columns and l zeroes, respectively. Now consider a trusted third party that provides party P_i with a random permutation π_i such that, after permuting columns of $\bar{\mathbf{M}}$ and $\bar{\mathbf{v}}$ according to π_1 and π_2 they are "well aligned", meaning $\pi_1(\bar{\mathbf{M}})\pi_2(\bar{\mathbf{v}}) = \mathbf{Mv}$. Note that it is crucial that π_1 and π_2 look random to P_1 and P_2 respectively. To achieve that, the third party generates *random* π_1 and π_2 subject to the constraint that $\forall i \in [l+k] : \pi_1(i) = \pi_2(i) \Leftrightarrow i \in (A \cap B)$, where



Parties: P_1, P_2.

Inputs:

P_1**:** Matrix $\mathbf{M} \in \mathbb{Z}_{2^\sigma}^{n \times m}$, with l nonzero columns.

P_2**:** Vector $\mathbf{v} \in \mathbb{Z}_{2^\sigma}^{m}$, with k nonzero entries.

Outputs: $[[\mathbf{Mv}]]$

Protocol:

(1) P_1 chooses a random permutation π_1 of $[l+k]$ and sets $\mathbf{d} = ((a_1, \pi_1(1)), \ldots, (a_k, \pi_1(l)))$, and $\beta = (\pi_1(l+1), \ldots, \pi_1(l+k))$, where the a_i's are the indices of the nonzero columns in $\mathbf{M}$.

(2) P_2 sets $\mathbf{q} = (b_1, \ldots, b_k)$, where the b_i's are the indices of the nonzero values in $\mathbf{v}$.

(3) P_1 and P_2 run a designated-output Room with inputs $\mathbf{d}, \beta, \mathbf{q}$. P_2 obtains $\mathbf{r} = (p_i)_{i \in [k]}$.

(4) Let $\hat{\mathbf{M}} \in \mathbb{Z}_{2^\sigma}^{n \times l}$ be $\mathbf{M}$ but with its zero columns removed. P_1 defines $\bar{\mathbf{M}}$ as the result of appending k zero columns to $\hat{\mathbf{M}}$, and computes $\mathbf{M}' = \pi_1(\bar{\mathbf{M}})$, where π_1 permutes the columns of $\bar{\mathbf{M}}$.

(5) Let $\hat{\mathbf{v}} \in \mathbb{Z}_{2^\sigma}^{k}$ be $\mathbf{v}$ but with its zero entries removed. P_2 defines a permutation $\pi_2 : [k+l] \mapsto [k+l]$ such that $\pi_2(i) = p_i$ for $1 \leq i \leq k$. The values $\pi_2(i)$ for $k+1 \leq i \leq k+l$ are a random permutation of $\{1, \ldots, m\} \setminus \{p_1, \ldots, p_k\}$ (the set of unused indexes in $[m]$). P_2 computes $\mathbf{v}' = \pi_2(\bar{\mathbf{v}})$, where $\bar{\mathbf{v}}$ is $\hat{\mathbf{v}}$ padded with zeros up to length $k+l$.

(6) P_1 and P_2 run Dense-Mult with inputs $\mathbf{M}'$ and $\mathbf{v}'$ to obtain shares of $\mathbf{Mv}$.

Figure 9: Column-sparse matrix and sparse vector multiplication protocol.

A and B are the sets of indexes of nonzero columns and values in $\mathbf{M}$ and $\mathbf{v}$.

The idea of [35] is to implement the above third party functionality in MPC using garbled circuits. Our protocol in Figure 9 implements this functionality in an different way using the ROOM primitive as follows. P_1 acts as the ROOM's server with inputs $\mathbf{d} = ((a_1, \pi_1(1)), \ldots, (a_k, \pi_1(l)))$ and $\beta = (\pi_1(k+1), \ldots, \pi_1(l+k))$, where π_1 is a random permutation of $[k+l]$ chosen by P_1, and P_2's query is simply $\mathbf{q} = (b_1, \ldots, b_k)$ (see steps 1 and 2 in Figure 9). The outputs for the two parties from the ROOM protocol are *secret shares* of the array $\mathbf{r} = (p_i)_{i \in [k]}$. Party P_1 provides P_2 with their share of the p_i's so that P_2 can reconstruct π_2. Note that this computation is independent of both n and σ, in contrast to the protocol in Figure 8. Moreover, it provides flexibility to exchange the roles of the server and client in the ROOM protocol achieving the second goal defined above.

The security of this construction follows from the security of the ROOM protocol and the dense multiplication. The output of the ROOM allows party P_2 to obtain the evaluation of a random permutation π_1 on its non-zero entries. The rest of the protocol

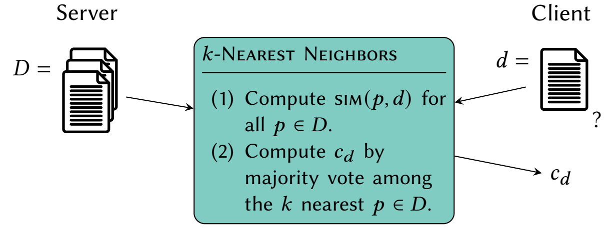


Figure 10: Setting for our secure k-NN application. We focus on a single server, although the general protocol allows D to be distributed among several servers [35].

involves local computations until the final secure computation for the dense multiplication.

5.2.2 Row-Sparse Matrix. We now consider the case where $\mathbf{M}$ is sparse in its first dimension. In our solution to this variant P_1 defines $\mathbf{M}'$ as the matrix resulting from removing all zero rows from $\mathbf{M}$. Then the parties run a protocol to compute shares of the vector $\mathbf{r} = \mathbf{M}'\mathbf{v}$ of length l. For this, we can either use the protocol from Figure 8, if $\mathbf{v}$ is sparse, or we can rely on dense multiplication. In any case, $\mathbf{r}$ now contains all non-zero values of the desired result, but its dimensions do not match $\mathbf{Mv}$. However, note that $\mathbf{Mv}$ can be recovered from $\mathbf{r}$ by inserting $n-l$ zeros between the values of $\mathbf{r}$ at positions corresponding to zero rows in the original matrix $\mathbf{M}$. This can directly be achieved by running ScatterInit$(\mathbf{r}, \mathbf{i}, n)$, where $\mathbf{i}$ contains the indexes of non-zero rows in $\mathbf{M}$.

In our higher-level applications, which we describe next, we will use the matrix-vector multiplications described here in various ways. In particular, we use the column-sparse protocol from Figure 9 for k-NN classification, and the column-sparse and row-sparse protocols from Figure 8 and the previous section for logistic regression.

6 APPLICATIONS

We consider three applications to exemplify the features of our framework. These include non-parametric data analysis tasks (naive Bayes and k-nearest neighbors) as well as parametric data analysis tasks (logistic regression trained by stochastic gradient descent (SGD)). Besides showing the flexibility of our framework, our motivation to select this concrete set of applications is to enable comparisons with previous works: secure naive Bayes classification was studied by Bost et al. [5], Schoppmann et al. [35] recently proposed a custom protocol for k-nearest neighbors that exploits sparsity, and the SecureML work by Mohassel and Zhang [29] is the state of the art in secure two-party logistic regression learning with SGD.

Due to space considerations, and since it mostly consists of ROOM queries, we present our Naive Bayes classification in Appendix B. The remainder of this section focuses on k-NN and two-party SGD, both of which make use of the advanced matrix multiplication protocols we presented in Section 5.2.

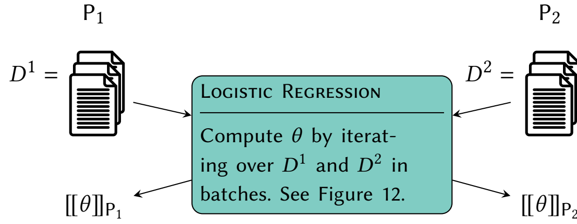


Figure 11: Setting for two-party logistic regression. Each party holds a set of labeled documents, and the output is a model that is secret-shared between the two parties.

6.1 Similarity Computation and k-Nearest Neighbors

In secure k-NN [35], we have a database D of labeled documents distributed among one or multiple servers, where each $d \in D$ is represented as a feature vector using TF-IDF encoding. A client wants to classify a document d against D. The k-NN classification algorithm, which is parameterized by a constant k, achieves that goal by (a) computing $\textsc{sim}(d, p)$, for each $p \in D$, and (b) assigning a class c_d to d as the result of a majority vote among the classes of the k most similar documents according to the similarities computed in step (a). See Figure 10 for a schematic depiction of this setting.

In [35], the authors observe that for the commonly used cosine similarity metric, secure k-NN reduces to one two-party sparse matrix-vector multiplication between the client and each of the data holders, followed by a generic MPC phase for the majority vote. Their construction for similarity computation is equivalent to the protocol in Figure 9 instantiated with Circuit-ROOM, which we use as a baseline in our experimental evaluation (Section 8.1). Note that this approach already exploits sparsity in the data.

However, our modular framework also allows us to use different ROOM primitives to achieve better efficiency in the online phase, while matching the speedups of [35] in the offline phase (cf. Section 2). In Section 8.1, we show that our Poly-ROOM indeed improves the online phase by up to 5x.

6.2 Logistic Regression Training

A drawback of non-parametric approaches like k-NN is that each query depends on the entire training database. To circumvent this, parametric approaches such as logistic regression first train a smaller *model* θ, which is then used to answer classification queries faster.

Here, we assume a two-party setting where parties P_1 and P_2 hold a horizontally partitioned database, i.e., each party holds a *sparse* dataset, where P_i holds $\mathbf{X^i} \in \mathbb{R}^{n_i \times d}, \mathbf{y^i} \in \mathbb{R}^{n_i}$, with $\mathbf{X^i}$ being the set of n_i records with d features and $\mathbf{y}^i$ being the vector of corresponding binary target labels. This corresponds to a training dataset of size $n \times d, n = n_1 + n_2$ distributed among P_1 and P_2, and the goal is to build a shared model θ that is able to accurately predict a target value for an unlabeled record, while keeping the local training datasets private (cf. Figure 11).

A widely used algorithm for building this kind of model is *mini-batched stochastic gradient descent (SGD)*. Here, the empirical loss of the model θ is minimized by iteratively refining it with an update

Parties: $\mathsf{P}_1, \mathsf{P}_2$.

Inputs:

P_1: $D^1 = (\mathbf{X}^1, \mathbf{y}^1), \mathbf{X}^1 \in \mathbb{R}^{n \times d}, \mathbf{y}^1 \in \{0,1\}^n$,

P_2: $D^2 = (\mathbf{X}^2, \mathbf{y}^2), \mathbf{X}^2 \in \mathbb{R}^{n \times d}, \mathbf{y}^2 \in \{0,1\}^n$.

Output: Shared model $[[\theta]]$.

Protocol:

1: $\theta^0 = (0)_{i \in [d]}$
2: **for** T epochs **do**
3: **for** $i \in \lfloor \frac{n}{b} \rfloor$ **do**
4: **for** $j \in [2]$ **do**
5: $\mathbf{B^j} \leftarrow \mathbf{X^j}_{[i..i+b]}$
6: $[[\mathbf{u^j}]] \leftarrow \textsc{MvMult}(\mathbf{B^j}, [[\theta]])$
7: $[[\mathbf{v^j}]] \leftarrow \textsc{Sigmoid}([[\mathbf{u^j}]])$
8: $[[\mathbf{w^j}]] \leftarrow [[\mathbf{v^j}]] - \mathbf{y^j}_{[i..i+b]}$
9: $[[\mathbf{g^j}]] \leftarrow \textsc{MvMult}(\mathbf{B^j}^\top, [[\mathbf{w^j}]])$
10: **end for**
11: $[[\mathbf{g}]] \leftarrow \frac{1}{2b}([[\mathbf{g}_1]] + [[\mathbf{g}_2]])$
12: $[[\theta]] \leftarrow [[\theta]] - \eta[[\mathbf{g}]]$
13: **end for**
14: **end for**

Figure 12: Secure two-party gradient descent on sparse distributed training data.

rule of the form $\theta^{i+1} = \theta^i - \eta \mathbf{g}$, for a step size η and a gradient quantity $\mathbf{g}$. The training dataset is partitioned in so-called *mini-batches*, each of which is used to compute a model update: In a forward pass, the prediction loss of the current batch is computed, and the gradient $\mathbf{g}$ of that loss is obtained as the result of a backward pass.

Figure 12 shows a secure protocol for two-party SGD training. It relies on a secure matrix multiplication protocol $\textsc{MvMult}$, and an approximation of the logistic function $\textsc{Sigmoid}(\mathbf{x}) = 1/(1 + e^{\mathbf{x}})$ introduced by Mohassel and Zhang [29], implemented with a garbled circuit. The security of the protocol follows from the fact that θ is always kept secret shared, and secure implementations of the two sub-protocols above.

We now discuss how the calls to $\textsc{MvMult}$ in lines 6 and 9 of the protocol are instantiated with our protocols from Section 5.2. First note that, as the $\mathbf{X^j}$'s are sparse, so will be their mini-batches $\mathbf{B^j}$ contributed by P_1 or P_2 in line 5. In fact, as common mini-batch sizes are as small as 64 or at most 128, the mini-batches will be sparse in their columns. We show that this is the case in the context of concrete real-world datasets in Section 8. Hence, the call to $\textsc{MvMult}$ in line 6 involves a column-sparse matrix and a dense vector, and thus we choose our protocol from Figure 8 instantiated with Basic-ROOM. The choice of Basic-ROOM is justified by the fact that the keys of $[[\theta]]_{\mathsf{P}_2}$ span the whole key domain $\mathcal{K} = [d]$, as it is a secret share, and hence Basic-ROOM does not incur unneccesary overhead in this case. On the other hand, the computation of $\mathbf{g^j}$ in line 9 is a multiplication between a row sparse matrix and a dense vector, for which we use our protocol from Section 5.2.2.

In Section 8.2, we compare the runtimes of our sparse implementation to those reported in [29]. With the exception of the smallest dataset, we improve computation time by a factor of 2x–11x (LAN) and 12x–94x (WAN), and communication by 26x–215x (LAN) and 4x–10x (WAN).

7 IMPLEMENTATION OF OUR FRAMEWORK

For our implementation, we follow the general architecture presented in Figure 1. For each layer of abstraction, we define generic interfaces that are then matched by our concrete implementations. This allows, for example, to use the same matrix multiplication function for different ROOM instantiations, which in turn simplifies development and makes sure our framework can be extended seamlessly.

Most of our library is written as generic C++ templates that abstract away from concrete integer, vector and matrix types. This allows us to use Eigen's expression templates [18], and thus avoid unnecessary local matrix operations. For generic two-party computation based on garbled circuits, we use Obliv-C [39]. As a PRF, we use the AES-128 implementation in Obliv-C by Doerner [12]. The fast polynomial interpolation and evaluation that we need for Poly-ROOM and ScatterInit is done using Yanai's FastPolynomial library [37]. The code for our framework is publicly available for download.[1]

8 EXPERIMENTAL EVALUATION

Given the large number of parameters and tradeoffs that our framework exhibits, a complete layer-by-layer evaluation of all components from Figure 1 with all ranges of useful parameters is both infeasible and not very useful. Instead, we chose to run experiments on only two abstraction layers: ROOM micro-benchmarks, which allow to compare our constructions with each other and with future improvements, and entire applications, which allow us to compare against previous work on application-specific protocols. We present our micro-benchmarks in Appendix C.2, and focus on our applications for the remainder of this section.

We implement each of the applications presented in Section 6 in our framework. All of our applications are implemented end-to-end, meaning they take the sparse feature vectors as inputs and return the desired class or shared regression model as output. For Naive Bayes, we refer to Appendix B.1.1.

We analyze three real-world datasets that represent common classification tasks: sentiment analysis [27], topic identification [34], and language detection [36]. Table 2 summarizes the properties of each of the datasets, including the average number of features of single documents. We also report, for reference, the classification accuracies that can be achieved using the different methods outlined in Section 6: logistic regression, naive Bayes, and k-nearest neighbors. These were obtained in the clear using out-of-the-box Scikit-Learn [32] model fitting, without any sophisticated hyperparameter tuning.

For the Movie Reviews and 20Newsgroups datasets, features correspond to words, using a TF-IDF representation. We assume a public vocabulary of 150000 words for the first two datasets (Movie reviews and 20newsgroups). This number has been used in previous work [35] and corresponds to the size of a large dictionary of English words. For the language classification task, n-grams of n consecutive characters are used instead. We assume the set of characters is public.

[1] https://github.com/schoppmp/room-framework

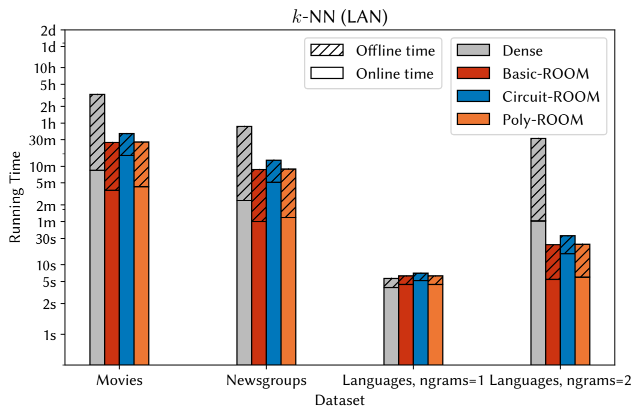


Figure 13: Running times of the matrix-vector multiplication needed for a single k-NN classification.

All our experiments are performed on Azure DS14 v2 instances with 110 GB of memory each, using a single core. We note that the memory bottleneck in our experiments is the "dense" case that we use as a baseline, not our ROOM-based implementations. For LAN experiments, we use instances in the same region, while for WAN experiments, we place one in the US and one in Europe. The measured roundtrip time was 0.7ms in the LAN setting, and 85ms in the WAN setting. The average data transfer rates were 2.73Gbit/s and 245Mbit/s, respectively. As in previous work [29, 35], 64-bit integers are used for $\mathcal{K}$ and $\mathcal{V}$. Garbled circuits are run with 80-bit security, due to Obliv-C. For Poly-ROOM, we use $s = 40$ bits of statistical security.

8.1 k-Nearest Neighbors

For k-NN, the efficiency bottleneck is the computation of scores of the query document with respect to each training sample, which reduces to a secure matrix-vector multiplication where the matrix is sparse in its columns and the vector is sparse. Thus, we can implement this protocol using Figure 9, instantiated with any of our ROOMs.

As observed in Section 6.1, Schoppmann et al. [35] solve this problem using an approach similar to Circuit-ROOM, which is why we use it as the baseline here. In most of our experiments (Figure 13), their approach turns out to be faster in than a simple dense multiplication.

Our new constructions using Basic-ROOM and Poly-ROOM achieve a similar improvement over the dense case (up to 82x) when it comes to total time, and at the same time a 2–5x improved online time compared to [35]. Note that the online time includes top-k selection (implemented using generic MPC) and multiplication of the reduced matrices, both of which are the same for [35] and us. If we focus on the reduction time alone, our new approaches are up to 40x faster.

Dataset	*Documents*	*Classes*	*Nonzero Features*		*Accuracy*		
			Single (avg.)	*Total*	*Log. Regression*	*Naive Bayes*	*k-NN*
Movies [27]	34341	2	136	95626	0.88	0.85	(*) 0.74
Newsgroups [34]	9051	20	98	101631	0.73	0.76	0.57
Languages [36], ngrams=1	783	11	43	1033	0.96	0.87	0.96
Languages [36], ngrams=2	783	11	231	9915	0.99	0.99	0.99

Table 2: Real-world datasets used in the experiments. These comprise a variety of classification tasks such as sentiment analysis of movie reviews (Movies), topic identification (Newsgroups), and language identification (Languages). For the latter, we also investigate the effect of analyzing larger n-grams instead of single characters.
(*) k-NN was trained on a subsample of $10K$ examples due to memory limitations.

Dataset	*Offline Time*		*Total Time*		*Offline Communication*		*Total Communication*	
	SecureML	*Ours*	*SecureML*	*Ours*	*SecureML*	*Ours*	*SecureML*	*Ours*
Movies	6h17m45.06s	**14m19.29s**	6h29m28.37s	**2h43m46.09s**	4.8 TiB	**186.25 GiB**	4.8 TiB	**187.42 GiB**
Newsgroups	1h39m33.66s	**3m34.55s**	1h42m38.14s	**42m37.68s**	1.26 TiB	**46.5 GiB**	1.26 TiB	**47.63 GiB**
Languages, ngrams=1	3.56s	**1.76s**	**5.9s**	29.89s	789.88 MiB	**390.75 MiB**	790.9 MiB	**500.61 MiB**
Languages, ngrams=2	1h1m16.34s	**13.07s**	1h3m7.12s	**6m17.51s**	796.82 GiB	**2.83 GiB**	797.85 GiB	**3.69 GiB**

Table 3: Comparison of our approach with SecureML [29] in the LAN setting. Offline times are extrapolated from the results reported in [29]. In all experiments, we use a batch size of 128. The total time represents a full training epoch, including forward pass, sigmoid activation function, and backward pass.

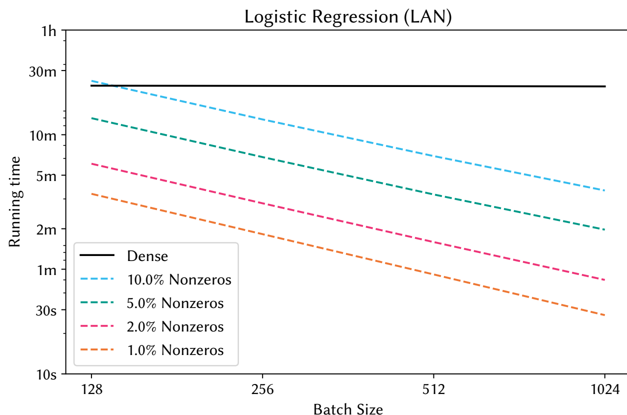


Figure 14: Total running time of a Stochastic Gradient Descent (SGD) training epoch for logistic regression. We use synthetic datasets with 1024 documents per server, a vocabulary size of 150k, and varying sparsity per batch.

8.2 Logistic Regression Training

For each of our datasets, we also evaluate the time needed to build a logistic model using our protocol from Figure 12. We compare two approaches. One uses the state-of-the-art dense matrix multiplication protocol to instantiate MvMult (cf. lines 6 and 9 in Figure 12), which is the extension of Beaver triples [3] to matrices proposed in [29]. The second approach uses our sparse matrix multiplication protocols from Figure 8 and Section 5.2.2 for forward- and backward pass, respectively. We measure the online running time of a full run using both approaches, as well as the total amount of data transfered. We also estimate offline times using the measurements provided by Mohassel and Zhang [29, Table II], and we present it together with the total time that includes both phases in Table 3.

While the dense solution [29] achieves fast online computation, this comes at a significant offline computation and communication cost, requiring hours, even days in the WAN setting (Appendix C.1), and sometimes terabytes of communication. Our solution on the other hand saves a factor of 2x–11x in total runtime and a factor 26x–215x in communication in all reasonably large datasets (in "Languages, ngrams=1", SecureML is faster in total time, but both executions take just a few seconds).

Finally, we investigate how our solution scales with different dataset sparsities and the batch size used for training. For that, we run experiments on synthetic datasets. We use 1024 documents for each of the two servers, and vary the batch size between 128 and 1024. For each batch, we set the number of nonzero values between 1% and 10%. For comparison, the sparsity of a batch of 128 documents from the Movies or Newsgroups datasets is about 3%.

The results are shown in Figure 14. It can be seen that our sparse implementation benefits a lot from increasing the batch size. However, increasing the batch size will also increase the number of nonzeros per batch in real datasets, albeit sub-linearly. Thus, the batch size can be optimized to account for the sparsity of the dataset being used for training. Research on training ML models in the clear suggests larger batch sizes can be used without losing accuracy [19], and we conjecture that this allows us to achieve even better speedups than those reported in Table 3, at the same level of accuracy. However, in order to stay functionally equivalent to previous work [29], we omit such optimizations at this point.

9 CONCLUSION

Privacy preserving techniques for machine learning algorithms have a wide range of applications, which most often need to handle large inputs, and thus scalability is crucial in any solution of practical significance. Exploiting sparsity is heavily leveraged by existing computation frameworks to achieve scalability, and in many settings some sparsity metric of the data is already public. This can be leveraged in the setting of privacy-preserving data analysis, not only at the application level, but also in terms of lower-level operations.

A practical and principled approach to this problem calls for a modular design, where in analogy to the components architecture in scientific computing frameworks, algorithms for linear algebra are built on top of a small set of low-level operations. In this paper we proposed a framework that takes a first step towards this vision: we started by defining sparse data structures with efficient access functionality, which we used to implement fast secure multiplication protocols for sparse matrices, a core building block in numerous ML applications.

By implementing three different applications within our framework, we demonstrated the efficiency gain of exploiting sparsity in the context of secure computation for non-parametric (k-nearest neighbors and Naive-Bayes classification) and parametric (logistic regression) models, achieving manyfold improvement over the state of the art techniques. The existing functionalities in our framework represent main building blocks for many machine learning algorithms beyond our three applications. At the same time, our modular framework can be easily extended, opening the ROOM to future improvements.

ACKNOWLEDGMENTS

Phillipp Schoppmann was supported by the German Research Foundation (DFG) through Research Training Group GRK 1651 (SOAMED). Adrià Gascón was supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1, and funding from the UK Government's Defence & Security Programme in support of the Alan Turing Institute. Mariana Raykova's work on this paper was done while at Yale University supported in part by NSF grants CNS-1633282, 1562888, 1565208, and DARPA SafeWare W911NF-16-1-0389. Benny Pinkas was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

REFERENCES

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *IEEE Symposium on Security and Privacy*. IEEE, 962–979.

[2] Kenneth E. Batcher. 1968. Sorting Networks and Their Applications. In *AFIPS Spring Joint Computing Conference (AFIPS Conference Proceedings)*, Vol. 32. Thomson Book Company, Washington D.C., 307–314.

[3] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 576. Springer, 420–432.

[4] James Bennett, Stan Lanning, et al. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*, Vol. 2007. New York, NY, USA, 35.

[5] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *NDSS*. The Internet Society.

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT (2) (Lecture Notes in Computer Science)*, Vol. 9057. Springer, 337–367.

[7] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *ACM Conference on Computer and Communications Security*. ACM, 1223–1237.

[8] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private Information Retrieval by Keywords. *IACR Cryptology ePrint Archive* 1998 (1998), 3.

[9] Michele Ciampi and Claudio Orlandi. 2018. Combining Private Set-Intersection with Secure Two-Party Computation. In *SCN (Lecture Notes in Computer Science)*, Vol. 11035. Springer, 464–482.

[10] 1000 Genomes Project Consortium et al. 2015. A global reference for human genetic variation. *Nature* 526, 7571 (2015), 68.

[11] Kushal Datta, Karthik Gururaj, Mishali Naik, Paolo Narvaez, and Ming Rutar. 2017. GenomicsDB: Storing Genome Data as Sparse Columnar Arrays. White Paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/genomics-storing-genome-data-paper.pdf

[12] Jack Doerner. [n. d.]. The Absentminded Crypto Kit. https://bitbucket.org/jackdoerner/absentminded-crypto-kit/

[13] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 239–267.

[14] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. 2000. Protecting Data Privacy in Private Information Retrieval Schemes. *J. Comput. Syst. Sci.* 60, 3 (2000), 592–629.

[15] Oded Goldreich. 2009. *Foundations of Cryptography: Volume 2, Basic Applications* (1st ed.). Cambridge University Press, New York, NY, USA.

[16] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[17] Gaël Guennebaud, Benoît Jacob, et al. [n. d.]. Eigen: Sparse matrix manipulations. https://eigen.tuxfamily.org/dox/group__TutorialSparse.html

[18] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org

[19] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *NIPS*. 1729–1739.

[20] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *NDSS*. The Internet Society.

[21] Eric Jones, Travis Oliphant, Pearu Peterson, et al. [n. d.]. Sparse matrices (scipy.sparse) – SciPy v1.1.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/sparse.html

[22] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*. USENIX Association, 1651–1669.

[23] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP (2) (Lecture Notes in Computer Science)*, Vol. 5126. Springer, 486–498.

[24] Peeter Laud. 2015. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *PoPETs* 2015, 2 (2015), 188–205.

[25] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptology* 22, 2 (2009), 161–188.

[26] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM Conference on Computer and Communications Security*. ACM, 619–631.

[27] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *ACL*. The Association for Computer Linguistics, 142–150.

[28] R. Moenck and Allan Borodin. 1972. Fast Modular Transforms via Division. In *SWAT (FOCS)*. IEEE Computer Society, 90–96.

[29] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 19–38.

[30] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 377–394.

[31] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. 2013. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security*. ACM, 801–812.

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[33] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur.* 21, 2 (2018), 7:1–7:35. https://doi.org/10.1145/3154794

[34] Jason Rennie and Ken Lang. 2008. The 20 Newsgroups data set. http://qwone.com/~jason/20Newsgroups/

[35] Phillipp Schoppmann, Lennart Vogelsang, Adrià Gascón, and Borja Balle. 2018. Secure and Scalable Document Similarity on Distributed Databases: Differential Privacy to the Rescue. *IACR Cryptology ePrint Archive* 2018 (2018), 289.

[36] The Scikit-learn authors. [n. d.]. Scikit-learn language identification dataset. https://github.com/scikit-learn/scikit-learn/tree/master/doc/tutorial/

text_analytics/data/languages
[37] Avishay Yanai. [n. d.]. FastPolynomial. https://github.com/AvishayYanay/FastPolynomial
[38] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. IEEE Computer Society, 162–167.
[39] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptology ePrint Archive* 2015 (2015), 1153.
[40] Jingren Zhou. 2009. Sort-Merge Join. In *Encyclopedia of Database Systems*. Springer US, 2673–2674.

A BASELINE PROTOCOLS FOR SCATTERINIT

A.1 Naive solutions

One direct way to implement the the functionality of Figure 6 is using generic MPC such as garbled circuits. The approach requires a circuit of size $O(nl\sigma)$ that for each $i \in [n]$ selects the ith output among the all possible values $(0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-l})$. Hence a solution based on generic MPC constructions would require $O(nl\sigma)$ communication and computation. Alternatively, one can rely on additive homomorphic encryption to enable P_1 to distribute the encrypted values of $\mathbf{r}$ into the right positions of encrypted $\mathbf{r}'$ and then execute a protocol with P_2 to obtain shares of $\mathbf{r}'$ in the clear. This approach requires $O(n)$ computation and $O((n+l)L)$ communication where L is the length of a ciphertext of additively homomorphic encryption, which adds considerable expansion to the length of the encrypted value.

A.2 FSS-based ScatterInit

Function Secret Sharing (FSS). Our construction below uses as a building block function secret sharing (FSS) [6]. While this primitive provides functionality for general functions, we use its instantiation for point functions, which also implies private information retrieval (PIR). A point function $f_{\alpha,\beta}(x)$ with domain $[n]$ is defined as $f(\alpha) = \beta$ and $f(i) = 0$ for all $i \neq \alpha$. A function secret sharing scheme has two algorithms FSS.KeyGen and FSS.Eval. The key generation produces two keys $(\mathsf{K}_\mathsf{C}^\mathsf{FSS}, \mathsf{K}_\mathsf{S}^\mathsf{FSS}) \leftarrow \mathsf{FSS.KeyGen}(\alpha, \beta)$ that when evaluated, satisfy $\mathsf{FSS.Eval}(\mathsf{K}_\mathsf{C}^\mathsf{FSS}, i) = \mathsf{FSS.Eval}(\mathsf{K}_\mathsf{S}^\mathsf{FSS}, i)$ if $i \neq \alpha$ and $\mathsf{FSS.Eval}(\mathsf{K}_\mathsf{C}^\mathsf{FSS}, \alpha) \oplus \mathsf{FSS.Eval}(\mathsf{K}_\mathsf{S}^\mathsf{FSS}, \alpha) = \beta$, otherwise.

First, we present a solution for the ScatterInit functionality of Figure 6 in the case $l = 1$, and then discuss how to extend it to any value of l. The two parties will generate in a secure computation a pair of FSS keys (k^1, k^2) with the following properties. Let $[[\mathbf{v}_i]]_1 = \mathsf{F}_{k^1}(i)$ and $[[\mathbf{v}_i]]_2 = \mathsf{F}_{k^2}(i)$ for $i \in [n]$, then we have that $\forall j \neq i_1 : [[\mathbf{v}_j]]_1 = [[\mathbf{v}_j]]_2$, and $[[\mathbf{v}_{i_1}]]_1 \oplus [[\mathbf{v}_{i_1}]]_2 = \mathbf{r}_1$. This computation can be done several times in parallel to obtain a protocol for the case $l > 1$, where each party XORs locally its output vectors from all l executions. This protocol has $O(l\log(n))$ communication, $O(l\log(n))$ computation in MPC, and $O(ln)$ local computation for each party. One drawback of this approach is that the result would be xor-shared, and for our applications we require additive shares for efficiency, as we will perform arithmetic operations. A naive conversion from xor to additive shares in a circuit would require $O(n)$ additions, bumping up the computation and communication to be linear in n, which is prohibitive for the values of n to be found in some realistic data analysis (see Section 8). This overhead can be avoided as follows: starting with the case $l = 1$. Similarly to above the parties generate FSS keys (k^1, k^2) with the difference that $[[\mathbf{v}_{i_1}]]_1 \oplus [[\mathbf{v}_{i_1}]]_2 = \mathbf{x}$ instead of $\mathbf{r}_1$, where $\mathbf{x}$ is a random value not known to either party, while $\forall j \neq i_1 : [[\mathbf{v}_j]]_1 = [[\mathbf{v}_j]]_2$. Subsequently, the parties run a garbled circuit protocol with inputs $[[\mathbf{r}_1]]_1$, $c_1 = \bigoplus_{j \in [n]}([[\mathbf{v}_j]]_1)$, and $[[\mathbf{v}_{i_1}]]_1$ from P_1, and $[[\mathbf{r}_1]]_2$ and $c_2 = \bigoplus_{j \in [n]}([[\mathbf{v}_j]]_2)$ from P_2. This secure computation (a) computes $[[\mathbf{v}_{i_1}]]_2$ as $c_1 \oplus c_2 \oplus [[\mathbf{r}_1]]_2$, (b) reconstructs $\mathbf{r}_1$, and (c) reveals $s = \mathbf{r}_1 - [[\mathbf{v}_{i_1}]]_2$ to P_1. Finally, to obtain an additive share of the intended sparse vector $\mathbf{r}'$, P_1 sets $[[\mathbf{r}'_j]]_1 = -[[\mathbf{v}_j]]_1$, for all $j \neq i_1$, and $[[\mathbf{r}'_{i_1}]]_1 = s$, while P_2 sets $[[\mathbf{r}'_j]]_2 = [[\mathbf{v}_j]]_2$, for all $j \in [n]$.

Running this protocol several times gives a protocol for the general case ($l > 1$) from Figure 6 with $O(l\log(n))$ communication, $O(l\log(n))$ computation in MPC, and $O(ln)$ local computation for each party. The garbled circuit sub-protocol is extremely efficient, as it requires l additions and $2l$ XORs, where the latter can be performed locally with the Free-XOR optimization [23].

B NAIVE BAYES APPLICATION

This section describes our third application that we built using the functionality from our framework, which was omitted from the main paper body due to space constraints.

B.1 Secure Naive Bayes Classification

A naive Bayes classifier is a non-parametric supervised classification algorithm that assigns to an item d (for example a document) the class c in a set of potential classes C (for example {spam, no-spam}) that maximizes the expression $\textsc{score}(c) = P(c) \cdot \Pi_{t \in d} P(t|c)$, where $t \in d$ denotes the database features present in the feature representation of d. A common approach to keep underflows under control is to use logs of probabilities. This transforms the above expression into $\textsc{score}(c) = \log(P(c)) + \sum_{t \in d} \log(P(t|c))$.

In Naive Bayes, $P(c)$ is estimated as $P(c) = N_c/N$, namely the number of items N_c of class c in the dataset, divided by the dataset size N. $P(t|c)$ is estimated as $P(t|c) = T_{c,t}/N_c$, namely the number of occurrences (or score) $T_{c,t}$ of feature t in items of class c, normalized by the total number of examples of class c in the training dataset. Additionally, Laplace smoothing is often used to correct for terms not in the dataset, redefining $P(t|c)$ to be $P(t|c) = (T_{c,t}+1)/(N_c+N)$

A secure two-party naive Bayes classification functionality is defined as follows: a server holds the dataset D that consists of n items with k features. Each item in the dataset is labeled with its class from the set C of potential classes. Hence, the server holds the values $P(t|c), P(c)$ defined above. A client wants to obtain a label for an item d. This needs to be done in a privacy preserving manner where only the client learns the output label and the server learns nothing.

The work of Bost et al. [5] presented a solution to the above problem using Paillier encryption and an argmax protocol based on additive homomorphic encryption. Our ROOM functionality provides a direct solution for this two-party problem, in which the server reveals an upper bound of its number of features. This solution works as follows: for each class $l \in C$, the server and the client invoke the ROOM functionality with input values $\log(P(t|l))$ for all keys t, as well as default values $1/(N_c+N)$ for the server, and query $(t)_{t \in d}$ for the client. This gives the parties additive shares of the vector $(\log(P(t|l)))_{t \in d}$. Then, the parties can compute locally shares of the vector $(\textsc{score}(l))_{l \in C}$, which contains the scores of d with respect to all classes. Finally, the class with highest score,

Dataset	Offline Time		Total Time		Offline Communication		Total Communication	
	SecureML	Ours	SecureML	Ours	SecureML	Ours	SecureML	Ours
Movies	4d16h34m55.36s	**4h18m52.67s**	4d18h35m26.99s	**9h29m23.53s**	19.19 GiB	**761.73 MiB**	19.33 GiB	**1.92 GiB**
Newsgroups	1d5h40m20.66s	**1h5m15.15s**	1d6h9m32.82s	**2h26m21.82s**	5.01 GiB	**190.38 MiB**	5.13 GiB	**1.31 GiB**
Languages, ngrams=1	1m7.18s	**35.65s**	**1m39.36s**	3m18.35s	3.4 MiB	**1.9 MiB**	**4.42 MiB**	111.76 MiB
Languages, ngrams=2	18h15m18.24s	**4m1.77s**	18h34m34.36s	**13m36.84s**	3.05 GiB	**11.71 MiB**	4.08 GiB	**893.73 MiB**

Table 4: Comparison of our approach with SecureML [29] in the WAN setting. See also Table 3.

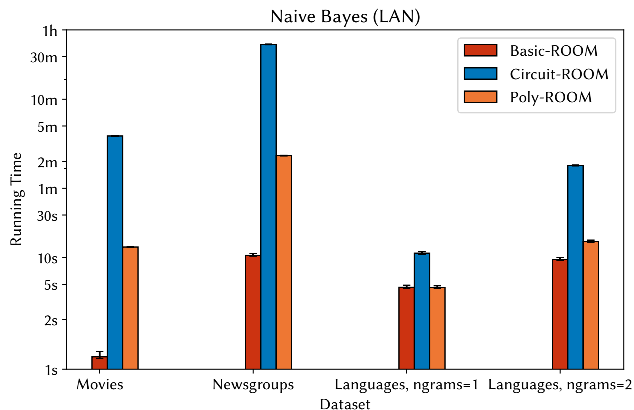


Figure 15: Running times for computing the conditional probabilities for a single Naive Bayes classification for each of the datasets in Table 2 and each of our ROOM constructions. Note that for Basic-ROOM, the vocabulary needs to be public. For private vocabularies, Poly-ROOM is the fastest in all cases. Error bars indicate 95% confidence intervals.

which will be revealed only to the client, can be computed using any generic MPC protocol involving only $|C|$ comparisons.

B.1.1 Experiments on Naive Bayes. We implemented our protocol described above, which consists of (i) a ROOM query for each of the $|C|$ potential classes composed with (ii) a protocol for securely computing the argmax of $|C|$ values. We do not include the latter here, since it only depends on the number of classes, which is usually significantly smaller than the dataset sizes (cf. Table 2). The runtimes, for each of the datasets, are shown in Figure 15 and Figure 16 (left). In both the LAN and WAN settings, Poly-ROOM generally outperforms Circuit-ROOM. This is consistent with the results from the previous section, since the queries are extremely sparse (i.e., m is small). Note that neither the Circuit- nor the Poly-ROOM require a public vocabulary. If the vocabulary is public, then Basic-ROOM can be used as well. The plots in Figure 15 show the runtime for a public vocabulary of size 150000. In the LAN setting this gives a huge advantage: for the Movie Reviews dataset with >95k features, our protocol takes less than 2s. In contrast, the total classification time for a dataset with only 70 features took over 3 seconds in [5].

C ADDITIONAL EXPERIMENTAL RESULTS

C.1 Experiments in the WAN Setting

Figure 16 and Table 4 show the results of the experiments from Section 8 in the WAN.

C.2 ROOM Micro-Benchmarks

Table 1 presents the runtimes for Circuit-ROOM and Poly-ROOM and how they depend on the database size n and the query size m. We first measure the runtimes of each algorithm for a range of parameters $n \in \{500, 5000, 50000\}$ and $m \in \{0.1n, 0.2n, \ldots, n\}$. The results can be seen in Figure 17. Each plot corresponds to one choice of n, while values of m are given on the x-axes. The runtime of both ROOM variants increases as m grows, but Circuit-ROOM is outperformed by Poly-ROOM as n increases, as long as $m < n$. The reason is that the complexity of Circuit-ROOM depends significantly on n, as its runtime is dominated by oblivious merging and shuffling, which scales with the sum $m + n$. The time of Poly-ROOM in mainly determined by m while Circuit-ROOM remains more stable across the choices of m.

To investigate the cutoff point between the two instantiations, as well as their performance *relative to each other*, we benchmark them on a grid of exponentially increasing parameter choices. We then fit functions of the runtime to the collected data, which gives us a model to estimate the performance even for parameter choices not directly measured.

Figure 18 shows the results in the LAN and WAN settings, respectively. For each set of choices for m and n, the color in the plot indicates the relative performance of our two algorithms. Intuitively, in regions where one of those two colors is prevalent, the corresponding algorithm is the optimal choice for that setting. Regions in between (turquoise) correspond to parameters where both of our algorithms perform equally well.

In the LAN setting, Poly-ROOM clearly wins in all cases where $m < n$, while Circuit-ROOM is only viable for large queries on small databases. In the WAN, this effect is slightly reduced, but still visible.

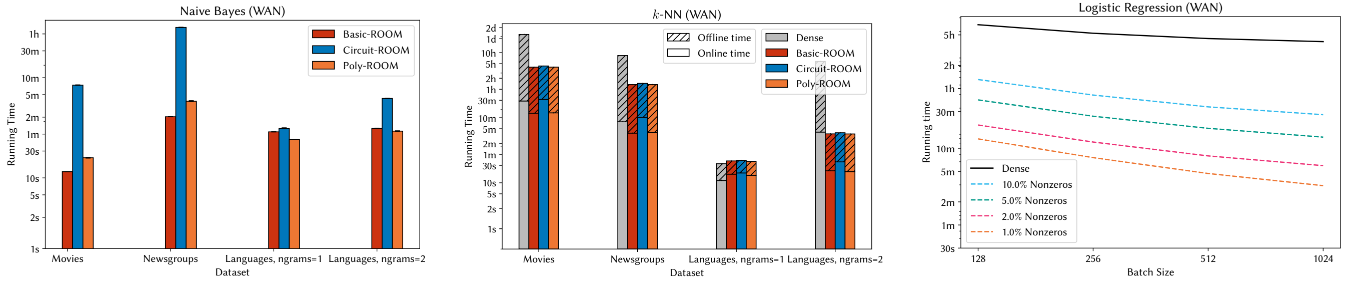


Figure 16: (Left) Running time of a Naive Bayes query in the WAN. See also Figure 15. (Middle) Running time of a k-NN query in the WAN. See also Figure 13. (Right) Total running time of an SGD training epoch for logistic regression with varying document sparsity. Note that unlike in the LAN, we can use SecureML's homomorphic encryption-based offline phase [29] here that also benefits from larger batches. See also Figure 14.

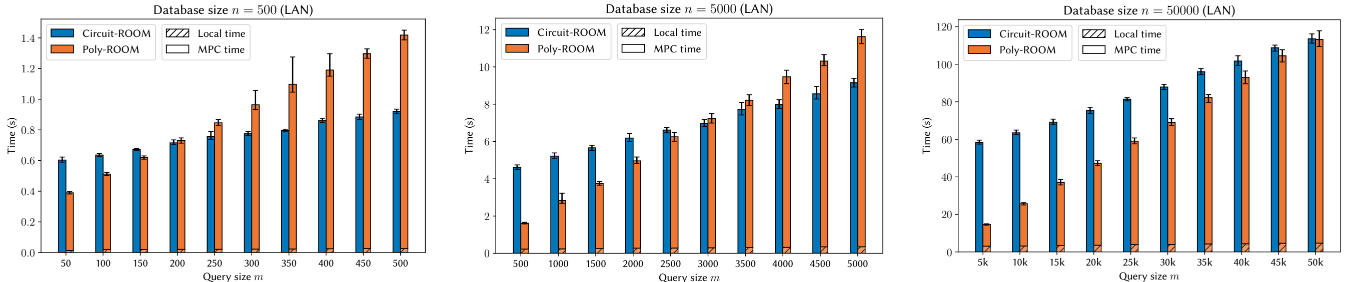


Figure 17: Measured running times of each of our ROOM constructions in the LAN setting, for several choices of query size and database size. We distinguish between local time (for time spent doing local computation) and MPC time, for running time of MPC sub-protocols. Error bars indicate 95% confidence intervals.

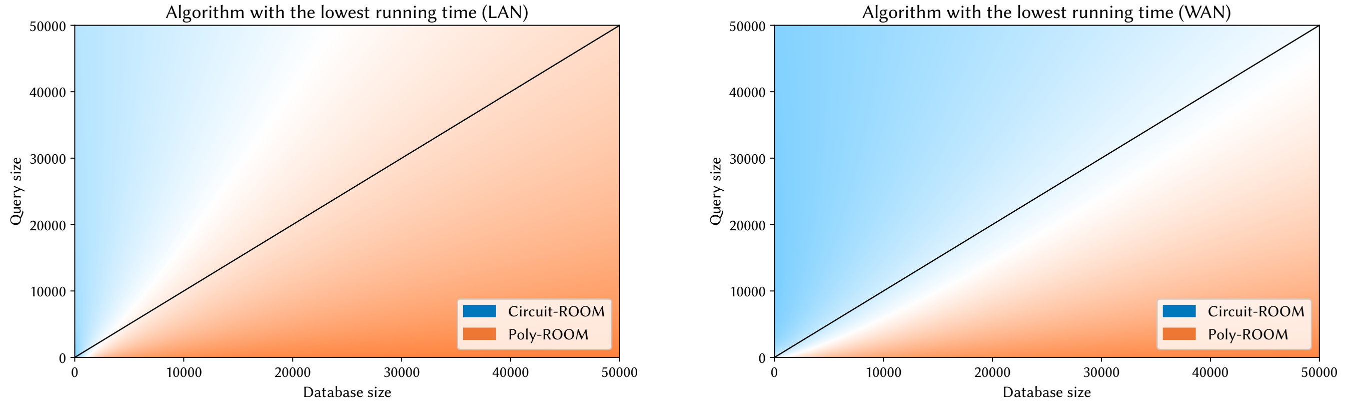


Figure 18: Estimated performance of our two instantiations of sparse ROOM in the LAN (left) and WAN (right) settings. Running times were measured for length m queries to a ROOM of size n, with $m \in \{2^i \mid i \in \{0, \ldots, 13\}\}$ and $m \in \{2^i \mid i \in \{0, \ldots, 18\}\}$. Then, for each of our algorithms, a model of the running time was computed using nonlinear least-squares from `scipy.optimize.curve_fit`, where the function to be fitted was chosen according to the asymptotics in Table 1. Each pixel was computed by averaging over the colors corresponding to each algorithm, weighted by the inverse of their respective running times. Thus, the dominant color of a region corresponds to the algorithm that performs the best in that setting.