Doing Real Work with FHE: The Case of Logistic Regression

Jack L.H.Crawford
Queen Mary Univ. of London

Craig Gentry
IBM Research

Shai Halevi
IBM Research

Daniel Platt
IBM Research

Victor Shoup*
NYU

February 19, 2018

Abstract

We describe our recent experience, building a system that uses fully-homomorphic encryption (FHE) to approximate the coefficients of a logistic-regression model, built from genomic data. The aim of this project was to examine the feasibility of a solution that operates "deep within the bootstrapping regime," solving a problem that appears too hard to be addressed just with somewhat-homomorphic encryption.

As part of this project, we implemented optimized versions of many "bread and butter" FHE tools. These tools include binary arithmetic, comparisons, partial sorting, and low-precision approximation of "complicated functions" such as reciprocals and logarithms. Our eventual solution can handle thousands of records and hundreds of fields, and it takes a few hours to run. To achieve this performance we had to be extremely frugal with expensive bootstrapping and data-movement operations.

We believe that our experience in this project could serve as a guide for what is or is not currently feasible to do with fully-homomorphic encryption.

Keywords: Homomorphic Encryption, Implementation, Logistic Regression, Private Genomic Computation

Research partly supported by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office(ARO) under Contract No. W911NF-15-C-0236.

*Work partially done in IBM Research.

Contents

1 Introduction

1.1 Somewhat vs. Fully Homomorphic Encryption

Since Gentry's breakthrough almost a decade ago [13], fully-homomorphic encryption (FHE) was touted as a revolutionary technology, with potential far-reaching implications to cloud computing and beyond. Though only a theoretical plausibility result at first, the last decade saw major algorithmic improvements (e.g., [5, 4, 15]), resulting in many research prototypes that implement this technology (e.g., [18, 12, 6, 10]) and attempt to use it in different settings (e.g., [3, 14, 22, 11, 16, 23], among others).

Nearly all contemporary FHE schemes come in two variants: The basic underlying scheme is *somewhat homomorphic* (SWHE), where the parameters are set depending on the complexity of the required homomorphic operations, and the resulting instance can only support computations up to that complexity. The reason is that ciphertexts are "noisy", with the noise growing throughout the computation, and once the noise grows beyond some (parameter-dependent) threshold the ciphertext can no longer be decrypted. This can be solved using Gentry's bootstrapping technique (at the cost of relying on circular security). In this technique the scheme is augmented with a *recryption* operation to "refresh" the ciphertext and reduce its noise level. The augmented scheme is thus *fully homomorphic* (FHE), meaning that a single instance with fixed parameters can handle arbitrary computations. But FHE is expensive, as the computation must be peppered with expensive recryption operations. So, it is often cheaper to settle for a SWHE scheme with larger parameters (that admit larger noise).

Indeed, with very few exceptions, almost all prior attempts at practical use of HE used only the SWHE variant, fixing the target computation and then choosing parameters that can handle that computation and no more. But SWHE has its limits: as the complexity of the function grows, the SWHE parameters become prohibitively large. In this work, we set out to investigate the practical feasibility of "deep FHE", attempting to answer the fundamental question of FHE's usefulness in practice:

Can FHE realistically be used to compute complex functions?

1.2 Logistic Regression and the iDASH Competition

Over the last few years, competitions by the iDASH center [20] provided a good source of "real world problems" to grind our teeth on. iDASH promotes privacy-preserving approaches to analysis of medical data, and since 2014 they have organized yearly competitions where they present specific problems in this area and ask for solutions using technologies such as differential privacy, secure multi-party computation, and homomorphic encryption.

In the homomorphic-encryption track of the 2017 competition, the problem to be solved was to compute homomorphically the parameters of a logistic-regression model, given encrypted data. The data consists of records of the form $(\vec{x}, y)$ where $\vec{x}_i \in \{0,1\}^d$ represent attributes of an individual (e.g., man or woman, over 40 or not, high blood pressure or not, etc.), and $y \in \{0,1\}$ is the target attribute that we investigate (e.g., whether or not they have a heart disease). A logistic-regression model tries to predict the probability of $y = 1$ given a particular value of $\vec{x}$, postulating that the probability $p_{\vec{x}} \stackrel{\text{def}}{=} \Pr[y = 1|\vec{x}]$ can be expressed as $p_{\vec{x}} = 1/(1 + \exp(-w_0 - \langle \vec{x}, \vec{w'} \rangle))$ for some fixed vector of weights $\vec{w} = (w_0, \vec{w'}) \in \mathbb{R}^{d+1}$. (The term w_0 is typically called an "offset".) Given sample data consisting of n records, our task is to find the weight vector $\vec{w} \in \mathbb{R}^{d+1}$ that best approximates the empirical probabilities (e.g., in the sense of maximum likelihood). For a more detailed exposition see section 2.

In addition to presenting the problem, the iDASH organizers also provided some sample data on which to test our procedures. The data consisted of nearly 1600 records of genomic data, each with 105 attributes (but they also accepted solutions that could only handle much fewer attributes than

this). With so many attributes, this appears firmly outside the scope of SWHE[1], hence we set out to design a solution to the iDASH task using FHE.[2]

1.3 Our Logistic-Regression Approximation Procedure

The starting point for our solution is a closed-form formula that we developed for approximating the logistic-regression parameters. This formula, developed in section 2 below, involved partitioning the records into "buckets", one per value of $\vec{x} \in \{0,1\}^d$, then counting the numbers of $y = 1$ and $y = 0$ in each bucket. These bucket counters are then used to derive a linear system $A\vec{w} = \vec{b}$ whose solution is the vector of weights $\vec{w}$ that we seek. As explained in section 2, computing $A, \vec{b}$ from the bucket counters involve "complicated functions" such as rational inversion and the natural logarithm.

The first issue that we have to deal with, is that our approximation formula only yields valid results in settings where the number of records n far exceeds the number of attributes d. Specifically, it relies on the fraction of $y = 1$ records in each bucket $\vec{x}$ to roughly approximate $p_{\vec{x}}$, so in particular we must have $n \gg 2^d$ to ensure that we have sufficiently many records in each bucket. But we aim at a setting with $d > 100$, which is far outside the validity region of this approximation formula.

We thus added to our solution a "quick-n-dirty" pre-processing phase, in which we homomorphically extract from the d input attributes a set of $k \ll d$ attributes which are likely to be the most relevant ones, then apply the approximation formula only to these k attributes, and set $w_j := 0$ for all the others. Specifically, in our solution we used $k = 5$, since the sample iDASH data had very few attributes with significant w_j coefficients.

This quick-n-dirty procedure involves computing the correlation between each column (attribute) x_j and the target attribute y, this is essentially just computing linear functions. Then we find the indexes $j_1, \ldots, j_k$ of the k columns x_j that are most correlated with y, and extract only these columns from all the records. The high-level structure of our homomorphic procedure is therefore:

1. For each column j, compute $\mathsf{Corr}_j = |\mathsf{Correlation}(x_j, y)|$;
2. Compute $j_1, \ldots, j_k$, the indexes of the k columns with largest Corr_j values;
3. Extract the k columns $j_1, \ldots, j_k$, setting $\vec{x'_i}[1 \ldots k] := (\vec{x_i}[j_1], \ldots \vec{x_i}[j_k])$
4. Compute the bucket counters, for every $\vec{x} \in \{0,1\}^k$ set
$$Y_{\vec{x}} := \left|\{i \le N : \vec{x'_i} = \vec{x} \text{ and } y_i = 1\}\right|, \; N_{\vec{x}} := \left|\{i \le N : \vec{x'_i} = \vec{x} \text{ and } y_i = 0\}\right|.$$
5. Compute $A \in \mathbb{R}^{(k+1)\times(k+1)}$ and $\vec{b} \in \mathbb{R}^{k+1}$ from the $Y_{\vec{x}}$'s and $N_{\vec{x}}$'s.
6. Solve the system $A\vec{w'} = \vec{b}$ for $\vec{w'} \in \mathbb{R}^{k+1}$, then output the coefficients $w_0 := w'_0$, $w_{j_i} := w'_i$ for the columns $j_1, \ldots, j_k$, and $w_j := 0$ for all other columns j.

Jumping ahead, about 55% of the computation time is spent in the first "quick-n-dirty" phase, which is the only part of the computation that manipulates homomorphically the entire input dataset.

1.4 Homomorphic Computation of the Approximation Procedure

We used the HElib library [19] as our back end to evaluate our approximation procedure homomorphically. Devising a homomorphic computation of this procedure brings up many challenges. Here we briefly discuss some of them.

[1] Some entries in the iDASH competition, including the winner, found clever ways to use SWHE for this problem, albeit only for a much smaller number of attributes. See for example [23].

[2] Unfortunately, our solution was not ready in time for the iDASH competition deadline, so we ended up not participating in the formal competition.

Implementing "complex" functions. Obtaining the linear system $A, \vec{b}$ from the bucket counters $Y_{\vec{x}}, N_{\vec{x}}$ involves computing "complex" functions such as rational division, or the natural logarithm. Computing these functions homomorphically, we have two potential approaches: one is to try to approximate them by low-degree polynomials (e.g., using their Taylor expansion), and the other to pre-compute them in a table and rely on homomorphic table lookup.

In this work we opted for the second approach, which is faster and shallower when applicable, but it can only be used to get a low-precision approximation of these functions. In our solution we used six or seven bits of precision, see more details in Section 4.

Homomorphic binary arithmetic and comparisons. Other things that we needed were the usual addition and multiplications operations, but applied to integers in binary representation (i.e., using encryption of the individual bits). Somewhat surprisingly, these basic operations were not discussed much in the literature, not in terms of proper implementations. Computing them homomorphically is mostly a matter of implementing textbook routines (e.g., carry look ahead for addition). But in this context we are extremely sensitive to the computation depth, which is not typical in other implementations. We describe our implementation of these methods and their various optimizations in Section 5.

Deciding on the plaintext space. HElib supports working with different plaintext-space moduli, and different calculations are easier with different moduli. In particular, the correlation computation in the first step is much easier when using a large plaintext space, as this lets us treat it as a linear operation over the native plaintext space. But most other operations above are easier when working with bits.

Here we use some features of the recryption implementation in HElib: When set to recrypt a ciphertext whose plaintext space is modulo 2, HElib uses temporary ciphertexts with plaintext space modulo 2^e for some $e > 2$ (usually $e = 7$ or $e = 8$). In particular it means that HElib can support computation with varying plaintext spaces of the form 2^e, and it also supports switching back and forth between them.

In our procedure, we used a mod-2^{11} plaintext space for computing the initial correlation, then switched to mod-2 plaintext space for everything else.

Setting the parameters. Setting the various parameters for bootstrapping is somewhat of an art form, involving many trade-offs. In our implementation we settled for using the m-th cyclotomic ring with $m = 2^{15} - 1$, corresponding to lattices of dimension $\phi(m) = 27000$. We set the number of levels in the BGV moduli-chain so that at the end of recryption we will still have nine more levels to use. Recryption itself for this value of m takes 20 levels, so we need a total of 29 levels. This means that we used a maximum ciphertext modulus q of size roughly 1030 bits, yielding a security level of just over 80 bits.

Solving linear systems. The last step in the approximation procedure above is to solve a linear system over the rational numbers. Performing this step homomorphically (with good numerical stability) is a daunting task. We considered some "pivot free" methods of doing it, but none of them seemed like it would be a good solution to what we need.

Since this is the last step of the computation, one option is to implement instead a *randomized encoding* of this step, which may be easier to compute. We discuss that option in section 6, in particular describing randomized encoding of the linear-system-solver function, that may be new. However we did not implement that scheme in our solution, instead we settled for a "leaky" solution that simply sends the linear system to be decrypted and solved in the clear.

1.5 The End Result

We implemented homomorphically all aspects of the procedure above, except the final linear-system solver. The program takes a little over four and a half hours on a single core to process the dataset of about 1600 encrypted records and 105 attributes. Over two and a half hours of this time is spent on extracting the five most relevant columns, about 45 minutes are spent on computing the bucket counters, and the remaining hour and 15 minutes is spent computing A and $\vec{b}$ from these counters. We can use more cores to reduce this time, down to just under one hour when using 16 cores. See more details in section 7. In terms of accuracy, our solution yields "area under curve" (AUC) of about 0.65 on the given dataset, which is in line with other solutions that were submitted to the iDASH competition.

1.6 Related Work

Surprisingly, not much can be found in the literature about general-purpose homomorphic implementation of basic binary operations. The first work that we found addressing this issue is by Cheon et al. [9], where they describe several optimizations for binary comparison and addition, in a setting where we can spread the bits of each integer among multiple plaintext slots of a ciphertext. They show procedures that use fewer multiplication operations, but require more rotations. These optimizations are very useful in settings where you can ensure that the bits are arranged in the right slots to begin with. But in our setting, we use these operations as a general-purpose tool, working on the result of previous computation. In this setting, the need for many rotations will typically negate the gains from saving on the number of multiplications. We thus decided to stick to bit-slice implementation throughout our solution, and try to make them use as few operations (and as small depth) as possible.

Other relevant works on homomorphic binary arithmetic are due to Xu et al. [27] and Chen et al. [7], who worked in the same bitslice model as us and used similar techniques. But they only provided partially optimized solutions, requiring deeper circuits and more multiplications than we use. (For example, they only used a size-4 carry-lookahead-adder for addition, and did not use the three-for-two procedure for multiplication.)

In terms of applying homomorphic encryption to the problem of logistic regression, the work of Aono et al. [1] described an interactive secure computation protocol for computing logistic regression, using additively-homomorphic encryption. Mohassel and Zhang [24] also described related secure-MPC protocols (but using garbled circuits, not HE). Wang et al described in [26] a system called HEALER that can compute homomorphically an exact logistic-regression model, but (essentially) only with a single attribute and only with very small number of records (up to 30).

A lot more work on the subject was done as part of the iDASH competition in 2017, but the only public report that we found on it is that of Kim et al. [23]. In this report they describe their implementation, using the somewhat-homomorphic scheme for approximate numbers due to Cheon et al. [8] to implement a homomorphic approximation of logistic regression (with a small number of attributes) using gradient-descent methods.

1.7 Organization

The rest of this report is organized as follows: In section 2 we derive our closed-form approximation formula for logistic regression. Then in section 3 we provide a bird-eye view of our solution, describing the individual steps and explaining how they are implemented. In sections 4 and 5 we describe many of the "toolboxes" that we developed and used as subroutines in this solution, and in section 7 we give performance results of our implementation. Our randomized encoding for the linear-system solver over the rational numbers in developed in section 6, and we conclude with a short discussion in section 8.

2 Logistic Regression and Our Approximation

Logistic regression is a technique to model dependence between related attributes. The input consists of n records (rows), each with $d+1$ attributes (columns), all of the form $(\vec{x}_i, y_i)$ with $\vec{x_i} \in \{0,1\}^d$ and $y_i \in \{0,1\}$. Below we sometimes refer to a fixed value $\vec{c} \in \{0,1\}^d$ as a *category*. (We sometime also refer to the different values $\vec{c} \in \{0,1\}^d$ as "buckets".) The ultimate goal is to estimate the probability $p_{\vec{x}} = \Pr[y = 1|\vec{x}]$. Logistic regression is a model that postulates that this probability is determined as

$$p_{\vec{x}} = \frac{1}{1 + \exp\big(-w_0 - \sum_{i=1}^n x_i w_i\big)} = \frac{1}{1 + \exp\big(-\langle (1|\vec{x}), \vec{w}\rangle\big)}$$

for some fixed vector of weights $\vec{w} \in \mathbb{R}^{d+1}$. The goal of logistic regression, given all the records $\{(\vec{x_i}, y_i)\}_{i=1}^n$, is to find the vector $\vec{w}$ that best matches this data. Below we denote $\vec{x'} \stackrel{\text{def}}{=} (1|\vec{x})$, and we use the expression for $p_{\vec{x}}$ as a function of $\vec{w} \in \mathbb{R}^{d+1}$, namely we denote

$$p_{\vec{x}}(\vec{w}) \stackrel{\text{def}}{=} \frac{1}{1 + \exp(-\langle \vec{x'}, \vec{w}\rangle)} = \frac{\exp(\langle \vec{x'}, \vec{w}\rangle)}{1 + \exp(\langle \vec{x'}, \vec{w}\rangle)} \ . \tag{1}$$

For a candidate weight-vector $\vec{w}$ and some given record $(\vec{x}, y)$, the model probability of seeing this outcome y for the attributes $\vec{x}$ is denoted $P_{\vec{x},y}(\vec{w}) \stackrel{\text{def}}{=} \{1 - p_{\vec{x}}(\vec{w}) \text{ if } y = 0, \ \ p_{\vec{x}}(\vec{w}) \text{ if } y = 1\}$. If we assume that the records are independent and use maximum-likelihood as our notion of "best match", then the goal is to find $\vec{w}^* = \text{argmax}_{\vec{w}} \big(\prod_{i=1}^n P_{\vec{x_i},y_i}(\vec{w})\big) = \text{argmax}_{\vec{w}} \big(\sum_{i=1}^n \ln(P_{\vec{x_i},y_i}(\vec{w}))\big)$.

2.1 A Closed-Form Approximation Formula for Logistic Regression

To get our approximation formula for logistic regression, we partition the data into the 2^d "categories" $\vec{c} \in \{0,1\}^d$. For each category $\vec{c}$, we denote the number of records in that category by $n_{\vec{c}}$, the number of records in that category with $y = 1$ by $Y_{\vec{c}}$, and the number of records with $y = 0$ by $N_{\vec{c}} = n_{\vec{c}} - Y_{\vec{c}}$ ('Y' and 'N' for YES and NO, respectively). We also partition the last sum above into the 2^{d+1} terms corresponding to all the $Y_{\vec{c}}$'s and $N_{\vec{c}}$'s,

$$\sum_{i=1}^n \ln(P_{\vec{x_i},y_i}(\vec{w})) = \sum_{\vec{c} \in \{0,1\}^d} Y_{\vec{c}} \cdot \ln(p_{\vec{c}}(\vec{w})) + N_{\vec{c}} \cdot \ln(1 - p_{\vec{c}}(\vec{w})).$$

Below it is convenient to do a change of variables and consider the "log odds ratio",

$$r_{\vec{c}}(\vec{w}) \stackrel{\text{def}}{=} \ln\big(\tfrac{p_{\vec{c}}(\vec{w})}{1-p_{\vec{c}}(\vec{w})}\big) = \langle \vec{c'}, \vec{w}\rangle,$$

where $\vec{c'} = (1|\vec{c})$. Then, $p_{\vec{c}}(\vec{w}) = 1/\big(1 + e^{-r_{\vec{c}}(\vec{w})}\big)$ and $1 - p_{\vec{c}}(\vec{w}) = 1/\big(1 + e^{r_{\vec{c}}(\vec{w})}\big)$. With this change of variables, we now want to find

$$\vec{w}^* = \text{argmax}_{\vec{w}} \sum_{\vec{c} \in \{0,1\}^d} Y_{\vec{c}} \cdot \ln\left(\frac{1}{1 + e^{-r_{\vec{c}}(\vec{w})}}\right) + N_{\vec{c}} \cdot \ln\left(\frac{1}{1 + e^{r_{\vec{c}}(\vec{w})}}\right). \tag{2}$$

Fix some category $\vec{c} \in \{0,1\}^d$, and consider the term corresponding to $\vec{c}$ in the sum above as a function of $r = r_{\vec{c}}(\vec{w})$ (with $Y_{\vec{c}}, N_{\vec{c}}$ as parameters), namely

$$f_{Y,N}(r) \stackrel{\text{def}}{=} Y \ln\big(\tfrac{1}{1 + e^{-r}}\big) + N \ln\big(\tfrac{1}{1 + e^{r}}\big).$$

To develop our closed-form formula, we approximate $f_{Y,N}(\cdot)$ using Taylor expansion around its maximum point $r_0 = \ln(Y/N)$,

$$f_{Y,N}(r) \;\approx\; \mathsf{someConstant} \;-\; \frac{YN}{2(Y+N)} \cdot \left(r - \ln\left(\frac{Y}{N}\right)\right)^2. \tag{3}$$

(We discuss the validity of this approximation later in this section.) Recall that we are seeking the weight-vector $\vec{w}$ that maximizes Eqn. (2), and hence we can ignore the $\mathsf{someConstant}$ (as well as the $1/2$ factor in $\frac{YN}{2(Y+N)}$) since these do not depend on $\vec{w}$. Hence the value that we seek is

$$\begin{aligned}\vec{w^*} \;&=\; \mathrm{argmax}_{\vec{w}} \left\{ -\sum_{\vec{c}} \underbrace{\frac{Y_{\vec{c}} N_{\vec{c}}}{Y_{\vec{c}} + N_{\vec{c}}}}_{\stackrel{\text{def}}{=} V_{\vec{c}}} \cdot \left(r_{\vec{c}}(\vec{w}) - \underbrace{\ln(Y_{\vec{c}}/N_{\vec{c}})}_{\stackrel{\text{def}}{=} L_{\vec{c}}} \right)^2 \right\} \\ &=\; \mathrm{argmin}_{\vec{w}} \left\{ \sum_{\vec{c}} V_{\vec{c}} \cdot \left(\langle \vec{c'}, \vec{w} \rangle - L_{\vec{c}} \right)^2 \right\}.\end{aligned}$$

We continue by expressing the last expression in matrix form. Let $\vec{V}, \vec{L}$ be 2^d-dimensional column vectors consisting of all the $V_{\vec{c}}$'s and $L_{\vec{c}}$'s, respectively. Also let C_d be a $(d+1) \times 2^d$ 0-1 matrix whose columns are all the $\vec{c'}$ vectors (namely the m'th column is $(1|\mathsf{bin}(m))^t$). For example for $d = 3$ we have

$$C_3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}. \tag{4}$$

Then the expression above can be written in matrix form as

$$F(\vec{w}) \stackrel{\text{def}}{=} \sum_{\vec{c}} V_{\vec{c}} \cdot \left(\langle \vec{c'}, \vec{w} \rangle - L_{\vec{c}} \right)^2 \;=\; \left(\vec{w}^T C_d - \vec{L}^T \right) \times \mathsf{diag}(\vec{V}) \times \left(C_d^T \vec{w} - \vec{L} \right).$$

$F(\vec{w})$ is a quadratic form in $\vec{w}$, and it is minimized at

$$\vec{w^*} = \mathrm{argmin}_{\vec{w}}\, F(\vec{w}) = \left(C_d \times \mathsf{diag}(\vec{V}) \times C_d^T \right)^{-1} \times C_d \times \mathsf{diag}(\vec{V}) \times \vec{L}.$$

This finally gives us our closed-form approximation formula: Given all the records $(\vec{x_i}, y_i)$ we compute all the YES and NO counters for the different categories, $Y_{\vec{c}}$ and $N_{\vec{c}}$, then set $V_{\vec{c}} := Y_{\vec{c}} N_{\vec{c}} / (Y_{\vec{c}} + N_{\vec{c}})$ and $L_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$.

We then let D be the 2^d-by-2^d diagonal matrix with the $V_{\vec{c}}$'s on the diagonal and $\vec{U}$ be the 2^d vector with entries $V_{\vec{c}} \cdot L_{\vec{c}}$, and we compute the approximation $\vec{w^*} := (C_d\ D\ C_d^T)^{-1} \times C_d\ \vec{U}$.

2.2 Validity of the Approximation

It is clear that the approximation procedure above relies on the number of records being sufficiently larger than the number of categories, so that we have enough YES and NO instances in each category. (Indeed if any $Y_{\vec{c}}$ or $N_{\vec{c}}$ is zero then the value $L_{\vec{c}}$ becomes undefined.)

Below we therefore assume that the number of records in each category is very large (i.e., tends to infinity). This implies by the law of large numbers that $Y_{\vec{c}}$ and $N_{\vec{c}}$ (now considered as random variables) can be approximated by normal random variables. In particular they tend to their expected value $p_{\vec{c}} \cdot n_{\vec{c}}$ and $(1-p_{\vec{c}}) \cdot n_{\vec{c}}$, and their log ratio $R_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$ tends to $\ln(p_{\vec{c}}/(1-p_{\vec{c}}))$. Turning that around, it means that we expect $r_{\vec{c}} = \ln(p_{\vec{c}}/(1-p_{\vec{c}}))$ to be close to its observed value $\ln(Y_{\vec{c}}/N_{\vec{c}})$, and therefore the formula from Eqn. (3) using the Taylor expansion of $f(r)$ around $\ln(Y_{\vec{c}}/N_{\vec{c}})$ should be a good approximation. The remaining terms in the Taylor expansion represent a residual contribution that is not "explained" by the model.

Remark 1 *In situations where the system is over-determined, it is possible for the model to spline the data. One way to reduce the impact of this over-fitting is to include an ad hoc penalty to the likelihood against the variance in β. Commonly applied penalty functions include linear L_1, and quadratic L_2 forms. However, it should also be recognized that such penalty functions do not represent sampling variation that the binomial and multinomial distributions seek to capture when sampled from disease/exposure p_j's.*

Remark 2 *In the expansion, genotypes $\{0, 1, 2\}$ and other similar constructions are easily accommodated by the approximation.*

3 Bird-Eye View of Our Solution

Expanding on the description from section 1.3, we now explain our approximate logistic-regression procedure in more detail. We begin in Section 3.1 with a description of the various functions that we want to compute, then in Section 3.2 we describe (still on a high level) how we implement these functions homomorphically.

3.1 The procedure that we implement

Input & Output. The input consists of (the encryption of) n records $(\vec{x_i}, y_i)_{i=1}^n$, with $\vec{x_i} \in \{0,1\}^d$ and $y_i \in \{0,1\}$. Below we view the input as an n-by-d binary matrix X with the i'th row equal to $\vec{x_i}$, and a column vector $\vec{y} \in \{0,1\}^n$ containing the y_i's. The output should be a logistic regression model, consisting of $d+1$ real-valued weights $w_0, w_1, \ldots, w_d$.

Extracting "Significant" Columns. We begin by considering each column of X separately, extracting the the k columns that have the strongest correlation with the target vector $\vec{y}$. Below we use $\mathsf{HW}(\vec{v})$ to denote the Hamming weight of $\vec{v}$, and denote by $X_{|j}$ the j'th column of X. We compute the following quantities:

- Let $Y := \mathsf{HW}(\vec{y})$, i.e., the number of records with $y_i = 1$;
- For $j = 1, \ldots, d$, let $\alpha_j := \mathsf{HW}(X_{|j})$, i.e., count the records with $\vec{x_i}[j] = 1$;
- For $j = 1, \ldots, d$, let $\beta_j := \mathsf{HW}(X_{|j} \wedge \vec{y})$, i.e., records with $\vec{x_i}[j] = y_i = 1$;
- For $j = 1, \ldots, d$, let $\mathsf{Corr}_j := |n \cdot \beta_j - Y \cdot \alpha_j|$, i.e., the correlation magnitude between $\vec{y}$ and $X_{|j}$.

Then we redact the input records, keeping only the k attributes with the strongest correlation to y. Namely we extract from X the submatrix $X' \in \{0,1\}^{n \times k}$, consisting of the k columns with the largest values of Corr_j. In our implementation we used in particular $k = 5$. Let $j_1, \ldots, j_k$ be the indexes of the selected columns, and denote by $(\vec{x'_i}, y_i)_{i=1}^n$ the n redacted records, i.e., the rows of $(X'|\vec{y})$.

Computing category counters, variance, and log-ratio. We accumulate the redacted records in 2^k buckets according to their $\vec{x'_i}$ values. For each category $\vec{c} \in \{0,1\}^k$, we compute $Y_{\vec{c}}$, $N_{\vec{c}}$ as the number of records in that category with $y = 1$, $y = 0$, respectively,

$$Y_{\vec{c}} := \left|\left\{(\vec{x'_i}, y_i) : \vec{x'_i} = \vec{c}, y_i = 1\right\}\right|, \quad N_{\vec{c}} := \left|\left\{(\vec{x'_i}, y_i) : \vec{x'_i} = \vec{c}, y_i = 0\right\}\right|.$$

Then we compute the variance and log-ratio, respectively, as $V_{\vec{c}} := \frac{Y_{\vec{c}} N_{\vec{c}}}{Y_{\vec{c}} + N_{\vec{c}}}$ and $L_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$.

Setting up the linear system. We now arrange the $V_{\vec{c}}$'s and $L_{\vec{c}}$'s in vectors $\vec{V}$, $\vec{L}$: For any $m = 0, 1, \ldots, 2^k - 1$, let $\mathsf{bin}(m) \in \{0,1\}^k$ be the binary expansion of m, we set the m'th entry in $\vec{V}$ to $V_{\mathsf{bin}(m)}$, and the m'th entry in $\vec{L}$ to $L_{\mathsf{bin}(m)}$. Next we compute the coefficients of a linear system from the vectors $\vec{L}$, $\vec{V}$ as follows:

The matrix A. Let C_k be a fixed $(k+1) \times 2^k$ binary matrix, whose m'th column is $(1|\mathsf{bin}(m))^T$. An example for $k = 3$ is illustrated in Eqn. (4). We compute a real $(k+1) \times (k+1)$ matrix $A := C_k \times \mathsf{diag}(\vec{V}) \times {C_k}^T$ (over $\mathbb{R}$), where $\mathsf{diag}(V)$ is the diagonal matrix with $\vec{V}$ on the diagonal.

The vector $\vec{b}$. Let $\vec{U}$ be an entry-wise product of $\vec{V}$ and $\vec{L}$, i.e., $U_m := V_m \cdot L_m$ for all $m \in [2^k]$. We compute the real $(k+1)$-vector $\vec{b} := C_k \times \vec{U}$.

Computing the output. Finally, we solve the linear system $A\vec{w'} = \vec{b}$ (over $\mathbb{R}$) for $\vec{w'} \in \mathbb{R}^{k+1}$, then set the output vector $\vec{w} \in \mathbb{R}^{d+1}$ as follows:

- The "offset" is $w_0 := w'_0$;
- For the selected columns $j_1, \ldots, j_k$ we set $w_{j_\ell} := w'_\ell$;
- For all other columns we set $w_j := 0$.

3.2 Homomorphic Evaluation

We now proceed to give more details on the various steps we used for homomorphic evaluation of the functions above. The description below is still a high-level one, with many of the details deferred to later sections. In particular, this implementation relied on many lower-level tools for homomorphic evaluation of binary arithmetic and binary comparisons that will be described in Section 5, homomorphic table lookup in binary representation that will be described in Section 4, and more.

3.2.1 Parameters and plaintext space

For our native plaintext space we use the cyclotomic ring $\mathbb{Z}[X]/(\Phi_m(X), 2^{11})$, with $m = 32767$ (so $\phi(m) = 27000$). This native plaintext space yields 1800 plaintext slots, each holding an element of $\mathbb{Z}[x]/(F(X), 2^{11})$ for some degree-15 polynomial $F(X)$, irreducible modulo 2^{11}. (In other words, each slot contains the Hensel lifting of $GF(2^{15})$ to a mod-2^{11} ring.)

We stress that HElib includes operations for extracting the bits of an encrypted integer in a mod-2^t plaintext space, so we can always switch to bit operations when needed. The only limitation is that it roughly takes depth t to extract t bits, and we can only use bootstrapping once we have encryption of individual bits. We therefore must ensure that we always have enough homomorphic capacity left to extract the bits of whatever integers we are manipulating.

These 1800 slots are arranged in a $30 \times 6 \times 10$ hypercube, corresponding to the generators $g_1 = 11628 \in \mathbb{Z}^*_m/(2)$ of order 30, $g_2 = 28087 \in \mathbb{Z}^*_m/(2, g_1)$ of order 6, and $g_3 = 25824 \in \mathbb{Z}^*_m/(2, g_1, g_2)$ of order 10.

We note that due to limitations of the BGV encryption scheme that we use, we cannot realistically use a larger plaintext space. Using a large plaintext modulus adds to the noise of operations in the scheme, and above 2^{11} this added noise becomes too hard to deal with. In fact a better optimized implementation would have used a smaller plaintext space of perhaps 2^8 rather than 2^{11}. This would make computing the correlation a little harder, but would reduce the noise everywhere else.

3.2.2 Encrypting the input

As we said in the introduction, computing the correlation is much simpler when working with a large plaintext space, but for other operations it is easier to work with bit representation. We therefore encrypt the input more than one way, as follows:

- We keep two mod-2^{11} ciphertexts as accumulators for the α and β counters, and a few other ciphertexts for packing the raw data itself. Initially all the ciphertexts are initialized to zero. The number of the raw-data ciphertexts depends on the number of records in the dataset: The packing scheme that we use allows each raw-data ciphertext to hold up to 150 records, and we "fill" these ciphertexts one at a time until we encrypt all the records.

- Given a record $(\vec{x_i}, y_i)$, we pack the bits in the next available raw-data ciphertext, using a packing scheme that considers the 27000 coefficients in the native plaintext space as arranged in a $180 \times 150 = (30 \times 6) \times (10 \times 15)$ matrix. The j'th attribute in the record is then stored in the coefficient with index (j, i) in this matrix.

  In even more detail, let us consider the $d+1$ vector $\vec{z_i} = (y_i|\vec{x_i})$ (where we assume that $d < 180$), and the bits of this record will be stored in the raw-data ciphertext of index i div 150. We let $i' = i \bmod 150$, $i_1 := i' \bmod 15$ and $i_2 := i' \text{ div } 15$, and for every $j = 0, 1, \ldots, d$ we also also let $j_1 := j \bmod 6$ and $j_2 := j \text{ div } 6$. Then the bit $\vec{z_i}[j]$ is stored in the slot of index (j_1, j_2, i_2) in the hypercube, in the i_1'st coefficient. To encrypt this record we prepare a fresh ciphertext that encrypts all the bits from $\vec{z_i}$ in the order above (and is otherwise empty), and homomorphically add it to the appropriate raw-data ciphertext.

  Then, we also add the bits $\vec{x_i}[j]$ and $y_i \cdot \vec{x_i}[j]$ to the accumulator ciphertexts α and β. Again we prepare a fresh ciphertext that has each bit $\vec{x_i}[j]$ in the j'th slot (and zero elsewhere) and add it homomorphically to the α, accumulator, and similarly we homomorphically add to the β accumulator a fresh ciphertext with $y_i \cdot \vec{x_i}[j]$ in the j'th slot (and zero elsewhere).

3.2.3 Computing the correlation

Once we have encrypted all the records, we have in the α, β ciphertexts all the counters α_j, β_j (which we assume are sufficiently smaller than the plaintext-space modulus 2^{11}). We also assume that we are given in the clear (a good approximation of) the value Y, i.e. the number of records with $y_i = 1$. [3] Similarly we know in the clear the number of records n, so we would like to just compute homomorphically the linear combination $n \boxtimes \beta \boxminus Y \boxtimes \alpha$. Unfortunately our plaintext space is not large enough for this computation, as we expect the result to exceed 2^{11}. Instead, what we do is use a low-resolution approximation of n, Y, namely we compute the correlation values as

$$\mathsf{Ecorr} = \lceil n/S \rfloor \boxtimes \beta \boxminus \lceil Y/S \rfloor \boxtimes \alpha$$

for an appropriate scaling factor S, chosen just large enough so we can expect the result to fit in 11 bits.

An alternative implementation (that we did not try) is to use sub-sampling. Namely instead of adding all the record data into the accumulators α, β, we can sub-sample (say) only 1/8 of the records to add. This would give us three more bits, and we can even trade it off with the amount of precision in n and Y (i.e., make the scaling factor S smaller as we sub-sample less records). Yet another option would have been to extract the bits of all the integers in α, β and perform the computation using bit operations, bypassing the plaintext-space issue altogether.

[3] This is a valid assumption in the context of medical studies, since the fraction of YES records in the overall population is always given in "Table 1" in such studies.

Once we have the Ecorr ciphertext, we extract the bits to get ciphertexts $\mathsf{Ecorr}_1, \mathsf{Ecorr}_2, \mathsf{Ecorr}_3, \ldots$, where Ecorr_i encrypts the i'th bit of all the correlation numbers (represented as signed integers in 2's-complement). I.e., the j'th slot in Ecorr_i encrypts the i'th bit of the number Corr_j.

Computing the absolute value. Once we have the bits of the Corr_j's, we need to compute their absolute value. Here we simplify things by computing the 1's-complement absolute value (rather than 2's-complement). Namely, for a signed integer in binary representation $x = x_t x_{t-1} \ldots x_0$, we will set $x'_i = x_i \oplus x_t$ $(i \leq t-1)$. Note that this introduces an error if $x < 0$ (since we now have $x' = -x - 1$ rather than $x' = -x$). But we assume that the "relevant" columns have much stronger correlation than the others, so a ± 1 error should not make a difference.

3.2.4 Finding the k most correlated columns

We now come to the most expensive part of our procedure, where we find the indexes of the k columns with largest correlation magnitude.

We note that the correlation computations above were done on packed ciphertexts, in a SIMD manner. This means that we now have a few ciphertexts (one for each bit of precision), with the i'th bit of $|\mathsf{Corr}_j|$ stored in the j'th slot of the i'th ciphertext. We therefore find the top few values by a shift-and-MUX procedure, using a binary comparison subroutine:

- Let $\vec{C}$ be the vector of values that we have, we compare point-wise $\vec{C}$ to $\vec{C} \ll \ell/2$ (where ℓ is the number of slots), homomorphically computing in each slot $j \leq \ell/2$ a bit b_j which is zero if $C_j < C_{j_{\ell/2}}$ and one otherwise. Then we set $\vec{C}' := \vec{b} \boxtimes (\vec{C} \ll \ell/2) + (1 \boxminus \vec{b}) \boxtimes \vec{C}$.

- Then we repeat the process with shifting by $\ell/4$, etc. After $\log(\ell)$ such steps we have transformed the vector into a heap with the MAX value at the root (which is at slot 0).

  We then zero-out the MAX value at slot 0, and repeat the process to get the 2nd-largest value, then the 3rd-largest value, etc. After running this procedure k times, we have our k largest values.

  We remark that this procedure that we implemented does not take any advantage of the fact that after finding the largest value we have the values in a heap rather than in arbitrary ordering. But note that in our SIMD environment we only need $\log(\ell)$ operations to extract the next largest value, the same as extracting a value from a heap. We do not know if there is a SIMD solution that uses less than $\log \ell$ operations per extracted value.

- As described above, this procedure gives the k largest Corr_j values, but our goal here is to compute the argmax, namely the *indexes* of these k largest values.

  To that end, we pack the indexes in the same order as we do the values. Namely we keep another ciphertext that packs the index j in the same slot that Ecorr packs the value Corr_j. Then we perform the comparison on Ecorr, computing the $\vec{b}$ as before, and apply the same shift-and-MUX operations to both Ecorr and the ciphertext containing the indexes. This ensures that when the MAX value arrives in slot 0 in Ecorr, the index of the corresponding column will arrive at slot 0 of the other ciphertext.

Extracting the k most correlated columns. Now that we computed the indexes of the k significant columns, we proceed to extract these columns from the raw-data ciphertexts. Note that with the packing scheme as described above, each column j is packed in all the coefficients of all the slots with hypercube indexes $(j \bmod 6, j \operatorname{div} 6, \star)$.

We therefore implement a homomorphic operation, similar to the shift-and-MUX from above, that given the bits of j, move each slot $(j \bmod 6, j \operatorname{div} 6, i)$ to position $(0,0,i)$, then zero-out all other slots, thus extracting the raw data of column j. We repeat this for column 0 (containing the y_i's) and columns $i_1, \ldots, i_k$.

3.2.5 Computing the category counters

Now that we extracted the data corresponding to the relevant $k+1$ columns, we need to count for every value $\{0,1\}^{k+1}$, how many records $(y_i, \vec{x_i'})$ we have with this value. After the column extraction step above the bits of each column are packed in all the coefficients of some of the slots (namely slots of index $(0,0,\star)$) in several ciphertexts.

As a first step in computing the counters, we distribute the bits of each column j among the slots of one ciphertext C_j, one bit per slot. This is doable since we have 1800 slots per ciphertext, and less than 1800 records in our dataset. (If we had more records we could have used more ciphertexts to pack them, this would not have made a big difference in running time.) Similar to other data movement procedures (e.g., the replicate procedures from [17]), the bit distribution can be done using a shift-and-add approach, and there are some time-vs.-noise trade-offs to be had. In our program we somewhat optimized this step, but more optimizations are certainly possible.

After we have the bits from each column j in the slots of one ciphertext C_j, we proceed to compute in a SIMD manner all the indicator bits $\chi_{i,m}$, for $i = 0, \ldots, n$ and $m \in [2^{k+1}]$, indicating whether the record $\vec{x_i'}, y_i$ belongs to category m. I.e., whether $(y_i | \vec{x_i'}) = \mathsf{bin}(m)$. This is done simply by taking all the "subset products" of the $k+1$ ciphertexts C_j. Namely we compute the product ciphertexts $P_0, P_1, \ldots, P_{2^{k+1}-1}$ as follows:

$$
\begin{aligned}
P_0 &:= (1-C_i) \boxtimes \ \ldots \ \boxtimes (1-C_{k-1}) \boxtimes (1-C_k) \\
P_1 &:= (1-C_i) \boxtimes \ \ldots \ \boxtimes (1-C_{k-1}) \boxtimes \ C_k \\
P_2 &:= (1-C_0) \boxtimes \ \ldots \ \boxtimes \ C_{k-1} \ \boxtimes (1-C_k) \\
P_3 &:= (1-C_0) \boxtimes \ \ldots \ \boxtimes \ C_{k-1} \ \boxtimes \ C_k \\
&\vdots \\
P_{2^{k+1}-1} &:= \ C_0 \ \boxtimes \ \ldots \ \boxtimes \ C_{k-1} \ \boxtimes \ C_k
\end{aligned}
$$

Computing all these 2^{k+1} products is done in depth $\lceil \log_2(k+1) \rceil$, and using "not much more" than 2^{k+1} multiplications, as we describe in Section 4. (Specifically, for our choice of $k = 5$ we use 96 multiplies.)

At this point, each slot i in the ciphertext P_m contains the indicator bit $\chi_{i,m}$. All that is left is to sum up all the slots in each ciphertext P_i (as integers in binary representation), getting the bits of the corresponding counter. In our program we implemented a special-purpose accumulation procedure for this purpose, described in Section 5 (but that procedure is not very well optimized). The accumulation of the P_m's for $m = 0 \ldots, 2^{k-1}$ give the counters N_m, and the accumulation of the P_m's for $m = 2^k, \ldots, 2^{k+1}-1$ gives the counters Y_{m-2^k}.

Our accumulation routine also includes a "transpose-like" operation: In the input we have different categories (buckets) represented by different ciphertexts, with the different rows across the slots. In the output we have the different categories across the slots and different ciphertexts for different bit positions. (We expect seven-bit counters in the output, so we have seven ciphertexts $Q_0, \ldots, Q_6$ encrypting the bits of the counters. The slots in $\sum 2^i Q_i$ give all the counter values, with the m'th counter in the m'th slot.) We terefore need to "transpose" from categories across different ciperhtexts to categories across slots in the same ciphertext. This "transpose-like" operation is handled at the same time as the accumulation: Beginning with all the slots in the input corresponding to bits of the same counter, we gradually accumulate many bits in larger integers, thereby clearing the slots of these

bits so we can pack in these slots the integers for other counters, until we have the integers of all the counters packed across the slots of the result.

3.2.6 Computing the variance and log-ratio

Next we need to compute from Y_m and N_m the values $V_m = \frac{Y_m N_m}{Y_m+N_m}$ and $U_m = V_m \cdot \ln(Y_m/N_m)$. Computing the variance and log-ratio is done using table lookups: As described in Section 4, for some function f that we want to compute, we pre-compute a table T_f such that $T[x] = f(x)$ for all x. These tables are computed with some fixed input and output precision, which means that the values there are only approximate. (In our program we use 7 input bits and 7 output bits for most tables.)

We use tables for three functions in our program, specifically $T_{inv}[x] \approx 1/x$, $T_{inv1}[x] \approx 1/(x+1)$, and $T_{ln}[x] \approx \ln(x)/(x+1)$. Then given Y_m and N_m we compute

$$\begin{aligned} r_m &:= Y_m \cdot T_{inv}[N_m] \approx Y_m/N_m, \\ V_m &:= Y_m \cdot T_{inv1}[r_m] \approx Y_m \cdot \tfrac{1}{Y_m/N_m+1} = Y_m N_m/(Y_m+N_m) \\ U_m &:= Y_m \cdot T_{ln}[r_m] \approx \ln(Y_m/N_m) \cdot Y_m N_m/(Y_m+N_m) \end{aligned}$$

Since the counters are packed in the different slots of the ciphertexts Q_i, then we only need to perform these operations once to compute in SIMD all the V_m's and U_m's.

3.2.7 Computing the matrix A and vector $\vec{b}$

The next step is to compute $A := C_k \times \mathsf{diag}(\vec{V}) \times C_k^T$ and $\vec{b} := C_k \times \vec{U}$ (over the rationals), using the bit representation of the V_m's and U_m's (which in our program are represented by seven bits each). Given the structure of the 0-1 matrix C_k, in our $k = 5$ implementation these computations require computing a relatively small number of subset-sums of these numbers (in binary representation). In particular, for every two bits position $\ell_1, \ell_2 \in \{0 \dots k-1\}$, we need to sum up all the number V_m corresponding to indexes m with bits ℓ_1, ℓ_2 set (i.e, $m_{\ell_1} = m_{\ell_2} = 1$), and we also need one more subset sum of all the numbers V_m of even index ($m_0 = 0$). Similar subset sums should be computed of the numbers U_m, and we pack the numbers U_m, V_m in such a way that the sums for U_m, V_m can be computed together.

Computing all the entries of $A, \vec{b}$ for our case $k = 5$ takes only 16 subset sums. Note that since the different numbers are packed in different slots, then adding two numbers require that we rotate the ciphertext to align the slots of these numbers. Again we carefully packed the numbers in the slots to reduce the number of rotations needed, and this step in its entirety requires 50 different rotation amounts (each applied to all the seven bits of the numbers, for a total of 350 rotation operations).

3.2.8 Solving $A\vec{w'} = \vec{b}$

The final operation that needs to be computed is solving the linear system $A\vec{w'} = \vec{b}$ over the rationals to find $\vec{w'}$. Here, however, we have a problem: recall that the procedures above only compute an approximation of $A, \vec{b}$ (mostly due to our use of low precision in the table-lookup-based implementation of inversion and logarithm). Hence we must use a very numerically-stable method for solving this linear system in order to get a meaningful result, and such methods are expensive.

One solution (which is what we implemented) is to simply send $A, \vec{b}$ back to the client, along with the indexes $j_1, \dots, j_k$ of the significant columns. The client then decrypts and solves in the clear to find $\vec{w'}$ and therefore $\vec{w}$. The drawback of this solution, of course, is that it leaks to the client more information than just the solution vector $\vec{w'}$. In section 6 we describe a solution that prevents this extra leakage, leaking to the client only as much information as contained in $\vec{w'}$, without having to implement homomorphically expensive linear-system solvers. However we did not get around to

implementing this solution in our program. (We remark that implementing it would not have added significantly to the running time.)

3.2.9 Bootstrapping considerations

As it turns out, most of the runtime of our program is spent in the recryption operations, between 66% and 75%. We must therefore be frugal with these operations. Some things that we did to save on them include:

- *Fully packed recryption.* HElib can bootstrap *fully packed* ciphertexts, i.e., ones that encode $\phi(m)$ coefficients in one ciphertext. The ciphertexts that we manipulate in our procedure, however, are seldom fully packed. Hence, whenever we need to perform recryption, we first pack as much data as we can in a single ciphertext, then bootstrap that ciphertext, and unpack the data back to the ciphertexts where it came from.

- *Strategic recryption.* Instead of performing recryption only at the last minute, we check the level of our ciphertexts before every "big step" in our program. For example, before we begin to add two numbers in binary representation, we check all the bit encryptions to ensure that we could complete the operation without needing to recrypt. If any of the input bits is at a low enough level, we pack all the input bits as above and recrypt them all. Then we unpack and perform the entire "big step" without any further recryption operations. This way we ensure that we never need to recrypt temporary variables that are used in internal computations, only the "real data" which is being manipulated.

4 Computing "Complicated Functions" using Table Lookups

As we explained above, we used a solution based on table lookup to implement a low-precision approximation of "complex functions". Namely, for a function f that we need to compute, we pre-compute in the clear a table T_f such that $T_f[x] = f(x)$ for every x in some range. Then given the encryptions of the (bits of) x, we perform homomorphic table lookup to get the (bits of) the value $T_f[x]$.

Building the table. Importantly, implementing a function using table lookup relies on fixed-point arithmetic. Namely the input and output must be encoded with a fixed precision and fixed scaling. In our implementation, we have three fixed-point parameters, precision p, scale s, and a Boolean flag ν that indicates if the numbers are to be interpreted as unsigned ($\nu = \mathsf{false}$) or as signed in 2's complement ($\nu = \mathsf{true}$). Given the parameters (p, s, ν), a p-bit string $(x_{p-1} \dots x_1 x_0)$ is interpreted as the rational number

$$R_{p,s,\nu}(x_{p-1} \dots x_1 x_0) = 2^{-s} \cdot \left(\sum_{i=0}^{p-1} 2^i x_i \ + (-1)^\nu \cdot 2^{p-1} x_{p-1} \right).$$

In our implementation we have two such sets of parameters, (p, s, ν) for the input (i.e., indexes into T), and (p', s', ν') for the output (i.e., values in T). With these parameters, the table will have 2^p entries, each big enough to hold a $2^{p'}$-bit number. In our implementation we pack all the bits of the output *in one plaintext slot*, so we can only accommodate tables with output precision up to the size of the slots.

Preparing the table T_f with parameters $(p, s, \nu, p', s', \nu')$ for a function $f(\cdot)$, each entry in the table consists of a native plaintext element (i.e, an element in $\mathbb{Z}[X]/(\Phi_m(X), p^r)$, in our case $m = 2^{15} - 1$,

$p^r = 2^{11}$). For every index $i \in [2^{p_{in}}]$, we put in $T_f[i]$, and element that has in *every plaintext slot* the bits of the integer z_i such that

$$R_{p',s',\nu'}(\mathsf{bin}(z_i)) = \lceil f\big(R_{p,s,\nu}(\mathsf{bin}(i))\big) \rfloor_{p',s'}$$

where $\lceil x \rfloor_{p',s'}$ rounds the real value x to the nearest point in the set $2^{-s'} \cdot [2^{p'}]$.

Saturated arithmetic. When building the table, we need to handle cases where the function value is not defined at some point, or is too large to encode in p' bits. In these cases, the number that we store in the table will be either the largest or smallest number (as appropriate) that can be represented with the given parameters p', s', ν'. (For example, in the table for $f(x) = 1/x$, the entry $T_{1/x}[0]$ will have the MAXINT value $2^{p'} - 1$ encoded in all the slots.)

Computing all subset-products. The main subroutine in homomorphic table lookup is a procedure that computes all the subset products of a vector of bits. The input is an array of p encrypted bits $\sigma_{p-1}, \ldots, \sigma_1, \sigma_0$, and the output is a vector of 2^p bits ρ_m of all the "subset products" of the σ_i's the their negation, i.e.,

$$\begin{aligned}
\rho_0 &:= (1-\sigma_i)\cdot \ \ldots\ \cdot(1-\sigma_{p-1}) \cdot (1-\sigma_p) \\
\rho_1 &:= (1-\sigma_i)\cdot \ \ldots\ \cdot(1-\sigma_{p-1}) \cdot \ \sigma_p \\
\rho_2 &:= (1-\sigma_0)\cdot \ \ldots\ \cdot\ \sigma_{p-1}\ \cdot(1-\sigma_p) \\
\rho_3 &:= (1-\sigma_0)\cdot \ \ldots\ \cdot\ \sigma_{p-1}\ \cdot\ \sigma_p \\
&\ \ \vdots \\
\rho_{2^p-1} &:= \ \ \sigma_0\ \ \cdot\ \ldots\ \cdot\ \sigma_{p-1}\ \cdot\ \sigma_p
\end{aligned}$$

Namely, for any $m \in [2^p]$, the bit ρ_m is set to $\rho_m := \prod_{m_j=1} \sigma_j \cdot \prod_{m_j=0}(1-\sigma_j)$.

To compute all these products ρ_m we use a "product tree" that on one hand ensure that the multiplication depth remains as low as possible (namely $\lceil \log_2 p \rceil$), and on the other hand tries to use as few multiplication operations as possible. For p power of two, this can be done recursively as follows:

ComputeAllProducts(input: $\sigma_{p-1}, \ldots, \sigma_0$, output: $\rho_{2^p-1}, \ldots, \rho_0$)

1. if $p = 1$ return $\rho_0 := 1 - \sigma_0$, $\rho_1 := \sigma_0$
2. else
2. ComputeAllProducts(input: $\sigma_{p/2-1}, \ldots, \sigma_0$, output: $\rho'_{2^{p/2}-1}, \ldots, \rho'_0$)
4. ComputeAllProducts(input: $\sigma_{p-1}, \ldots, \sigma_{p/2}$, output: $\rho''_{2^{p/2}-1}, \ldots, \rho''_0$)
5. for i, j in $0, \ldots, 2^{p/2}$, set $\rho_{2^{p/2}j+i} := \rho''_j \cdot \rho'_i$.

Essentially the same procedure applies when p is not a power of two, except that it is better to split the array so that the first part is of size power of two (i.e., size 2^ℓ for $\ell = \lceil \log_2 p \rceil - 1$) and the second part is whatever is left.

We comment that this procedure is not quite optimal in terms of the number of multiplications that it uses, but it is not too bad. Specifically the number of multiplications that it uses to compute the 2^p products is only $N(p) = 2^p + 2N(p/2) = 2^p + 2^{p/2+1} + 2^{p/4+2} + \cdots$. One optimization that we have in our program is that we stop the recursion at $p = 2$ rather than $p = 1$, and compute the four output bits using just a single multiplication (rather than four). Namely we set $\rho_3 = \sigma_1\sigma_0$, $\rho_2 = \sigma_1 - \sigma_1\sigma_0$, $\rho_1 = \sigma_0 - \sigma_1\sigma_0$, and $\rho_0 = 1 + \sigma_1\sigma_0 - \sigma_1 - \sigma_0$.

This optimization can in principle be extended to higher values of p, but it gets more complicated. The idea is that the ρ_m's can be computed in terms of the "real subset products" $\tau_m = \prod_{m_j=1} \sigma_j$. The τ_m's can be computed using a recursive formula similar to the one above, except that in the last line if $i = 0$ or $j = 0$ we do not need to multiply. (For $i = 0$ we set $\tau_{2^{p/2}j} := \tau''_j$ and for $j = 0$ we set $\tau_i := \tau'_i$.) Hence the number of products is reduced to $N'(p) = (2^{p/2} - 1)^2 + 2N'(p/2) = 2^p - p - 1$. The problem with this procedure is that recovering the ρ_m's from the τ_m's seems complicated (and the savings are not that large), so we did not attempt to implement it.

Homomorphic table lookup. Once we have a table $T[0, \ldots, 2^p - 1]$ and an implementation of the subset-product procedure above, implementing homomorphic lookups into the table with encrypted p-bit indexes requires just a simple MUX. Namely, we are given p ciphertexts, encrypting the bits σ_i of an index into T. We apply the subset-product procedure above to get all the products ρ_m, then return $\sum_m T[m] \boxdot \rho_m$.

Note that the input ciphertext could be packed, with a different bit $\sigma_{i,j}$ in each slot j of ciphertext i. In this case our lookup procedure would return a SIMD table lookup: the coefficients of the j'th slot of the output will store the bits of $T[x_j]$, where $x_j = \sum_i 2^i \sigma_{i,j}$.

We also remark that it is possible to implement different tables in different slots, so the j'th output slot will contain $T_j[x_j]$ instead of all using the same table T. This will require only a minor change to our procedure for building the table (and no change to the homomorphic lookup procedure), but we did not implement this variant yet.

5 Binary Arithmetic and Comparisons

Much of our logistic-regression procedure manipulates the various variables in their binary representation. To implement these manipulations, we rely on procedures that implement various common low-level operations, such as arithmetic and comparisons in binary representation. In this section we describe our implementation of these low-level operations, which has been integrated into the HElib library.

5.1 Adding Two Integers

One basic operation that we need is adding two integers in binary representation. The input consists of two sequences of ciphertexts, $(a_{t-1}, \ldots, a_1, a_0)$ and $(b_{t-1}, \ldots, b_1, b_0)$, encrypting the bits of two integers a, b, respectively (using padding, we can assume w.l.o.g. that the two integers have the same bit size).[4] The output is the sequence of ciphertexts $(s_{t+1}, \ldots, s_1, s_0)$, encrypting the bits of the sum $s = a + b$. Of course the hard part is to compute the carry bits in the addition, which we do as follows:

- For $i = 0, ..., t - 1$ we compute "generate carry" and "propagate carry" bits, $g_i := a_i b_i$ and $p_i := a_i + b_i$. (Note that at most one of p_i, g_i can be 1.)
- We extend the "generate" and "propagate" bits to intervals, where for any $i \le j$ we have $p_{[i,j]} = \prod_{k=i}^{j} p_k$ and $g_{[i,j]} = g_i \cdot \prod_{j=i+1}^{j} p_k$.
- The carry bit out of position j is $c_j := \sum_{i=0}^{j} g_{[i,j]}$, and the result bits are $s_i := a_i + b_i + c_{i-1}$ for $i = 0, \ldots, t$.

[4] We can have different bits in different plaintext slots of these ciphertexts, so each slot could represent a different integer and the addition will be applied to all of them.

To get all the carry bits c_i, we therefore need to compute all the interval products $g_{[i,j]}$ for all $[i,j] \subseteq [0, t-1]$, which we do using a dynamic-programming approach. Namely we compute for all the intervals of size two, then all intervals of size up to four, etc.

In more detail, given the inputs a_i, b_i we build an *addition DAG* that encodes our plan for what ciphertexts to multiply in what order. This is done to ensure that we consume the smallest number of levels, and use as few multiplications as we can. Note that the input ciphertexts need not be all at the same level, and the plan may vary depending on the input levels.

The DAG has two nodes for every interval $[j,i] \subseteq [0, t-1]$, representing $p_{[i,j]}$ and $g_{[i,j]}$, and each node has two parents which are the nodes that should be multiplied to form the variable of this node. We initialize the nodes in the DAG in the following order:

- First we initialize all the singleton nodes $p_{[i,i]}$, the parents are set to a_i, b_i and the level is set to $\min(\mathsf{lvl}(a_i), \mathsf{lvl}(b_i))$.
- Next we initialize all the other nodes $p_{[i,j]}$ in order of increasing interval size. To initialize $p_{[i,j+1]}$, we compute
$$k = \arg\max_{k \in [i,j]} \left\{ \min\left(\mathsf{lvl}(p_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) \right\}$$
(breaking ties as described later in this section). The parents of $p_{[i,j+1]}$ are set to $p_{[i,k]}$ and $p_{[k+1,j+1]}$, and its level to $\min\left(\mathsf{lvl}(p_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) - 1$.
- Next we initialize all the singleton nodes $g_{[i,i]}$, the parents are set to a_i, b_i and the level is set to $\min(\mathsf{lvl}(a_i), \mathsf{lvl}(b_i)) - 1$.
- Finally we initialize all the other nodes $g_{[i,j]}$ in order of increasing interval size. To initialize $p_{[i,j+1]}$, we compute
$$k = \arg\max_{k \in [i,j]} \left\{ \min\left(\mathsf{lvl}(g_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) \right\}$$
(breaking ties as described later in this section). The parents of $p_{[i,j+1]}$ are set to $g_{[i,k]}$ and $p_{[k+1,j+1]}$, and its level to $\min\left(\mathsf{lvl}(g_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) - 1$.

This procedure ensures that each node ends up at the highest possible level (i.e. the lowest possible multiplication depth), for the given levels of the inputs a_i, b_i. When all the input bits a_i, b_i are at the same level, then the depth is $\lceil \log_2(t+2) \rceil$, since the largest term that we need to compute is the $(t+1)$-product $g_{[0,t-1]} = a_0 b_0 \cdot \prod_{i=1}^{t-1}(a_i + b_i + 1)$.

We note, however, that not all the nodes in the DAG must be computed: Only the nodes $g_{[i,j]}$ are used in the carry calculation, and not every $p_{[i,j]}$ is necessarily an ancestor of some $g_{[i',j']}$. We can hope that by breaking ties in a clever way when computing argmax above, we can minimize the number of nodes $p_{[i,j]}$ that need to be computed, hence reducing the number of multiplications that must be performed. In our implementation, we break ties heuristically by choosing among the highest-level k's the nodes that already have the largest number of children.

The homomorphic addition procedure. Given the input ciphertexts a_i and b_i, we build a DAG as above, and check that the lowest-level node in this DAG is still at a level above zero. If not, then we attempt to recrypt all the input ciphertexts, then re-build the DAG with the new input levels. Once we have a valid DAG, we compute all the $g_{[i,j]}$ nodes and from then add them as needed to compute all the carry bits, and then compute the result bits.

While computing the $g_{[i,j]}$'s, we try to compute the nodes in the DAG lazily, computing each node only when it is needed (either directly for one of the carry bits or indirectly for one of its children), and keeping the intermediate node ciphertexts around only as long as they are still needed. (I.e., as long as they still have some descendants that were not yet computed.) We also use parallelism when we can, computing different nodes using different threads (if we have them).

5.2 Adding Many Integers

When we need to add many integers (all in binary representation), we use the three-for-two method (cf. [21]) to reduce their number: until we only have two integers left, then use the routine from above to add the remaining two numbers.

The three-for-two procedure. Given three integers in binary representation, $(u_{t-1}, \ldots, u_0)$, $(v_{t-1} \ldots v_0)$, $(w_{t-1} \ldots w_0)$, we can add the three bits in each position $u_i + v_i + w_i$ (over the integers), yielding a number between zero and three that can be represented in two bits. Namely $u_i + v_i + w_i = x_i + 2y_i$, where $x_i = u_i + v_i + w_i \pmod 2$ and $y_i = u_i v_i + u_i w_i + v_i w_i \pmod 2$. Adding every triple of bits u_i, v_i, w_i in this manner, we get the two integers $x = (x_{t-1} \ldots x_0)$, and $y = (y_t \ldots y_0 0)$, such that $x + y = u + v + w$ over the integers.

Computing the bits of x involves only additions, and each y_i can be computed using two multiplications and two additions, namely $y_i := u_i v_i + (u_i + v_i) w_i$. Hence x is at the same level as the input numbers u, v, w, and y is one level lower. (Note also that all the x_i's and y_i's can be computed in parallel.)

The add-many-numbers procedure. Given many integers in binary, we apply the three-for-two procedure to them in a tree manner, namely we partition them into groups of three, apply the three-for-two to each group separately, then collect all the resulting pairs (plus whatever leftover numbers were not part of any group) into one list, and repeat the process until only two integers are left. This yields multiplication depth $d \approx \log_{3/2}(n)$ to reduce n numbers into two, while adding at most d to the bitsize of the input integers. Once we have only two integers left, we apply the addition routine from above.

5.3 Integer Multiplication

Given two integers in binary to multiply $a = (a_{t-1} \ldots a_0)$ and $b = (b_{t'-1} \ldots b_0)$, we first compute all the pairwise products $b_i a_j$, and then use the add-many-numbers procedure from above to add the t' integers $2^i b_i \cdot a$. For example when multiplying a 3-bit b by a 4-bit a, we add the numbers

$$\begin{array}{rccccc} b_0 \cdot a = & & & b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 \\ 2b_1 \cdot a = & & b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 \\ 4b_2 \cdot a = & b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 \end{array}$$

When both numbers are unsigned, we always choose $t' \leq t$, namely we let the longer integer be a and the shorter one be b.

Dealing with negative numbers. In our implementation we also implemented a multiplication of a 2s-complement number a by an unsigned number b. In that case we always use the 2s-complement number as a and the unsigned number as b, and we modify the procedure above by computing the sign extension of all the numbers $b_i \cdot a$, namely replicating the top bit in each number all the way to the largest bit position. For example, if we have a 2-bit 2s-complement number $(a_1\ a_0)$ and a three-bit unsigned number $(b_2\ b_1\ b_0)$, then we compute and add the three integers (considered as 2s-complement numbers):

$$\begin{array}{rcccc} b_0 \cdot a = & b_0a_1 & b_0a_1 & b_0a_1 & b_0a_0 \\ 2b_1 \cdot a = & b_1a_1 & b_1a_1 & b_1a_0 & 0 \\ 4b_2 \cdot a = & b_2a_1 & b_2a_0 & 0 & 0 \end{array} .$$

We did not implement a 2s-complement by 2s-complement multiplication, since we did not need it for the current project.

5.4 Comparing Two Integers

The procedure for integer comparison is somewhat similar to integer addition. We have two integers in binary, $a = (a_{t-1}, \ldots, a_1, a_0)$ and $b = (b_{t-1}, \ldots, b_1, b_0)$, and we want to compute the two integers $x = \max(a, b) = (x_{t-1} \ldots x_0)$ and $y = \min(a, b) = (y_{t-1} \ldots y_0)$, as well as the two indicator bits $\mu = (a > b)$ and $\nu = (b > a)$ (note that when $a = b$, both μ, ν are zero).

We begin by computing for every $i < t$ the bits $e_i := a_i + b_i + 1$ (which is 1 iff $a_i = b_i$) and $g_i := a_i + a_i b_i$ (one iff $a_i > b_i$). We then compute the products $e_i^* = \prod_{j \geq i} e_j$ and $g_i^* = g_i \cdot \prod_{j>i} e_j$, and the bits $\tilde{g}_i = \sum_{j \geq i} g_1^*$ (one iff $a_{t-1\ldots i} > b_{t-1\ldots i}$). Computing the products e_i^*, g_i^* is done using a recursive procedure somewhat similar to ComputeAllProducts from Section 4. Finally we compute the results by setting $\mu := \tilde{g}_0$, $\nu := 1 + \tilde{g}_0 + e_0^*$, and for $i = 0, \ldots, t-1$ we set $x_i := (a_i + b_i)\tilde{g}_i + b_i$ and $y_i := x_i + a_i + b_i$.

Note that we use all the g_i^*'s but only e_0^* for computing the output results, hence we somewhat optimized our procedure for computing these products by skipping the computation of e_i^*'s that are never used.

We remark that the last product $(a_i + b_i)\tilde{g}_i$ means that our procedure may use depth one more than the minimum possible. Using the absolutely smallest possible depth is challenging, straightforward solutions would take $O(t^2)$ multiplications (vs. $O(t)$ multiplications in the procedure above). While getting minimal depth with $O(t)$ multiplications is possible in theory, the procedure for doing this is overly complex (and extremely hard to parallelize), so we opted for a simpler procedure with slightly non-optimal depth. (Also, as opposed to the addition procedure from above, the simple procedure that we implemented here does not vary depending on the level of the input ciphertexts for a_i, b_i.)

5.5 Accumulating the bits in a ciphertext

As described in Section 3.2, when computing the category counters we at some point have 64 ciphertexts, with ciphertext C_m encrypting in each slot i the indicator bits $\chi_{i,m}$, and we want to sum-up these indicator bits (over the integers) and compute the 64 counters $P_m \sum_i \chi_{i,m}$. While this is theoretically just an instance of adding many numbers (these numbers being the bits $\chi_{i,m}$), there are two properties of this instance that require special optimization:

- The input bits to be added are not aligned in the same slots of different ciphertexts, but rather spread across the different slots of the same ciphertext.
- We need to perform this add-many-numbers procedure on 64 different lists in parallel, so we have an opportunity to use SIMD operations.

We therefore implemented a special-purpose shift-and-add procedure to do the accumulation, combining the addition operations that we need to make with the "matrix transpose" that transforms the input counter-per-ciphertext with different bits across the slots into a ciphertext-per-bit-position in the output with the different counters across the slots.

At every step in this procedure we keep a current list of (encrypted) arrays of integers in binary representation. Each array in the list is represented by a vector of ciphertexts $(c_0, c_1, \ldots)$, one per bit position, and the integers in the array are the different slots of $\sum_i 2^i \cdot c_i$. Initially the list consists of 64 arrays, each array corresponding to the different slots of one of the input ciphertexts, and the integers all bits (so each array is represented by length-1 vector). As the computation progresses, the integers represent partial sums (so their bitsize is getting larger), correspondingly the arrays have fewer integers in them (since the number of partial sums is getting smaller), and also the number of arrays get smaller (as we pack more counters across the slots).

In each step we perform partial addition, adding each group of r partial sums into a single larger sum. We first apply r rotations to all the ciphertexts representing all the arrays, so as to align the

numbers that we need to add. These rotations mean that we are using fewer slots to hold these partial sums (since we only use one of each r slots as the "pivot" where addition is to take place). So we can pack some number $p \leq r$ of these sums and apply the add-many-numbers procedure to all of them in a SIMD manner. This cuts the number of arrays by a p factor, and change the size of arrays by a p/r factor. (Each array is cut by a factor of r because we add r partial sums into one, but increased by a factor of p as we pack multiple arrays into one.)

The procedure that we actually implemented is slightly different, in that in the first few steps we consider the different bit positions as different arrays (so we always work with bits rather than larger integers) and just remember for each array the power of two that it should be multiplied by. Only after we complete the "transpose" part of this transformation and have just one slot per counter, we add together all the relevant integers (shifted as needed to account for the powers of two). Specifically, we begin with 64 arrays, each containing 1800 single-bit integers. Then we perform these steps: of four steps:

1. In the first step we group $r_1 = 15$ bits together for addition (yielding 4-bit numbers), and pack $p_1 = 14$ arrays together. This yields $4 \cdot \lceil 64/14 \rceil = 20$ new arrays (of bits), and the data for each category counter is spread across $1800/15 = 120$ slots.

2. In the second step we group $r_2 = 12$ bits together for addition (again melding 4-bit numbers), and pack $p_2 = 5$ them together. This yields $4 \cdot \lceil 20/5 \rceil = 16$ arrays of bits, and the data for each category counter is spread across only $120/12 = 10$ slots.

3. In the this step we group $r_2 = 10$ bits together for addition (again yielding 4-bit numbers), and no further packing is needed. This yields again $4 \cdot 16 = 64$ arrays of bits, but now the data for each category counter is all in just one slot position.

4. We note that our current 64 ciphertexts encrypt shifted bits, i.e., bits that should be multiplied by some powers of two. No shift amount corresponds to more than twelve ciphertexts, so we can re-arrange these 64 bits in just 12 integers. Then we call our add-many-numbers procedure to add these 12 integers thereby completing the accumulation of the category counters.

The specific choices of $r_1 = 15, p_1 = 14$ and $r_2 = 12, p_1 = 5$ were made so that the shift operations involved in aligning numbers before addition could be implemented with 1D rotation operations. These operations map directly to automorphisms in the underlying cryptosystem, rather than the more expensive general-purpose shifts.

6 Solving a Linear System

The last thing that our solution needs to do, after setting the linear system $A\vec{w'} = \vec{b}$, is to solve it and output the solution vector $\vec{w'}$. But solving a linear system homomorphically is complex, even if it is only a 6-by-6 system as in our application. Moreover, the linear system that we computed was just an approximation (due mostly to the low-precision inherent in our table-based approach to computing inversion and logarithms). Hence we must ensure that our homomorphic solver is numerically stable, making it harder still.

Instead, in our program we opted for simply sending A and $\vec{b}$ to the client, having the client decrypt and solve in the clear. This "solution", however, leaks information about the input data beyond what is implied by the vector $\vec{w'}$. This extra leakage is perhaps acceptable in the context of our application to logistic regression on medical data, but surely there are applications where such a solution will not be acceptable. So we would like to find a feasible solution that will eliminate the extra leakage, simpler than implementing a homomorphic stable linear solver.

6.1 Randomized Encoding with Rational Reconstruction

An appealing approach for addressing this issue is to use *randomized encoding* (cf. [2]). Namely, consider the function that we want to compute, $f(A, \vec{b}) = A^{-1}\vec{b}$, as a function over the rational numbers with bounded integer inputs. We would like to apply a randomized transformation to the input $u := \mathsf{enc}(A, \vec{b}; R)$, such that (i) it is possible to "decode" $A^{-1}\vec{b}$ from u; (ii) u does not yield any more information[5] on $A, \vec{b}$ than what is implied by $\vec{w'} = A^{-1}\vec{b}$; and (iii) computing $\mathsf{enc}(\cdot)$ is substantially easier than computing $f(\cdot)$ itself.

If we had such randomized encoding, we could choose the randomness R and evaluate homomorphically $\mathsf{enc}(A, \vec{b}; R)$, send the encrypted u back to the client, who could decrypt u and decode $\vec{w'}$ from it. We note that the linear-system-solver function $f(\cdot)$ is in NC1, so theoretically we could apply generic randomized encoding solutions here. This solution yields a very low-depth encoding, but the size of the encoding is exponential in the depth of the circuit for f, so we do not expect them to be practical.

Below we describe a randomized encoding for the linear-system-solver function f, that uses only integer addition and multiplication. This encoding can therefore be implemented using the binary arithmetic routines that we described in Section 5. However we did not implement that idea in our solution, we expect it to be doable but it will add a significant overhead (see below).

Our first observation is that if we wanted to compute the linear-system-solver function modulo some prime q then it would be easy to randomize (when A is invertible): All we need is to choose a random invertible $R \in \mathbb{Z}_q^{n\times n}$ and set $A^* := RA \bmod q$ and $\vec{b^*} := R\vec{b} \bmod q$. On one hand A^* is just a random invertible function modulo q, and on the other hand $(A^*)^{-1}\vec{b^*} = A^{-1}\vec{b} \pmod q$.

However, in our case we want to find the solution over the rational numbers, not modulo some q. Our second observation is that we can apply here the tool of *rational reconstruction* (cf. [25, Ch 4.6]). Recall that rational reconstruction is an efficient procedure (denoted below by $\mathsf{RationalRec}(\cdot)$), such that

$$\forall a, b, q \in \mathbb{Z} \text{ s.t. } |a| \cdot |b| < q/2, \quad \mathsf{RationalRec}(q,\ ab^{-1} \bmod q) = (a, b),$$

provided that ab^{-1} is defined modulo q. In other words, the procedure gets as input a modulus q and an element $z \in \mathbb{Z}_q$, and it is guaranteed to output the unique solution (a, b) to $az = b \pmod q$ satisfying $|a| \cdot |b| < q/2$, if such a solution exists.

In our application we are given $A, \vec{b}$ with some precision p, and the rational solution that we seek is

$$\vec{w'} = A^{-1}\vec{b} = \mathsf{adj}(A)\vec{b}/\det(A).$$

Every entry of $\vec{w'}$ is of the form x/d, where $d = \det(A)$ and x is one entry in $\mathsf{adj}(A)\vec{b}$. Since all the entries in $A, \vec{B}$ are smaller than 2^p, then d and all the entries of $\mathsf{adj}(A)\vec{b}$ are smaller in magnitude than $(\sqrt{n}2^p)^n$. If we choose $q > 2(\sqrt{n}2^p)^{2n} = 2^{2np+1}n^n$, then given the solution $A^{-1}\vec{b} \bmod q$ with entries of the form $xd^{-1} \in \mathbb{Z}_q$, we could use rational reconstruction to get the rational numbers x/d. We could therefore get our randomized encoding by randomizing $A, \vec{b}$ modulo this large q.

But this solution is still not good enough, randomizing mod q implies in particular that we need to implement a homomorphic mod-q operation, which is expensive (even when q is in the clear). Our next observation is that we can replace the reduction mod-q by adding a large enough multiple of q. Recall that for any fixed integer x, if we choose a random s from a large enough domain (relative to $|x|/q$) then the random variable $x + qs$ depends only on the value of $x \bmod q$, and is essentially independent of $x \operatorname{div} q$. Specifically, if we have a bound $|x|/q < B$ and we choose s at random from $[B \cdot 2^k]$, then the result is almost independent of $x \div q$, up to statistical distance of at most 2^k.

[5] This property is formulated by requiring a simulator that can only see $\vec{w'}$ and can output the same distribution as $\mathsf{enc}(A, \vec{b}; R)$

PARAMETERS: dimension n, precision p, security parameter k.
Let $m := \max(n, \lceil \frac{2np+1}{k} \rceil)$

REncSolver (input: Invertible matrix $A \in \mathbb{Z}_{2^p}^{n\times n}$, vector $\vec{b} \in \mathbb{Z}_{2^p}^n$):

1. Choose m random primes $q_1, \ldots, q_m$ of bitsize $n2^k$.
2. For each q_i do
3. Choose a random invertible matrix $R \in \mathbb{Z}_{q_1}^{n\times n}$
4. Choose a random matrix $S_i \in \mathbb{Z}_{n2^{p+k}}^{n\times n}$ and a random vector $\vec{t_i} \in \mathbb{Z}_{n2^{p+k}}^n$.
5. Compute $A_i^* := R_i \times A' + qS_i$ and $\vec{b_i}^* := R_i \times \vec{b'} + q\vec{t_i}$.
6. Output $\{(q_i, A_i^*, \vec{b^*}_i)\}_{i=1}^m$

DecSolver(input: $\{(q_i, A_i^*, \vec{b^*}_i)\}_{i=1}^m$):

7. Solve $\vec{w_i^*} = (A_i^*)^{-1}\vec{b_i^*} \pmod{q}_i$ for all i
8. Use Chinese remaindering to recover $\vec{w^*}$ modulo $q = \prod_i q_i$
9. Apply rational reconstruction and output $\vec{w} := \mathsf{RationalRec}(q, \vec{w^*})$.

Figure 1: Randomized encoding for the rational linear-system solver, $f(A, \vec{b}) = A^{-1}\vec{b}$.

Hence instead of computing homomorphically the reduced matrix $RA \bmod q$, we note that each entry in RA is smaller than $n2^p q$ in absolute value. We can therefore choose a random integer matrix $S \in [n2^{p+k}]$ and compute $RA + qS$ over the integers (and similarly for $\vec{b}$).

This solution is almost plausible, but it requires integer arithmetic with very large numbers, even if n and p are small. In our application we have $n = 6$ and $p = 7$, so $q = 6^6 \cdot 2^{2\cdot 6\cdot 7+1} \approx 2^{100}$. And even choosing a measly statistical parameter $k = 10$, the entries of $RA + qS$ would be integers with about 120 bits.

Our final observation, therefore, is that we could express R and S relative to an appropriate CRT basis, thereby replacing each big-integer operation by a moderate number of operations on much smaller integers. Specifically, we choose many "smallish" primes q_i, for each one choose a random $R_i \in \mathbb{Z}_{q_i}^{n\times n}$ and $S_i \in [n2^{p+k}]^{n\times n}$ and compute homomorphically $A_i^* = R_i A + q_i S_i$ (and similarly for $\vec{b}$). As above, the client who decrypts the A_i^*'s only get $R_i A \bmod q_i$, then compute $RA \bmod q$ (where $q = \prod_{q_i}$ and $R \equiv R_q \pmod{q_i}$) and proceed as before.

One drawback with this approach is that we may end up choosing a modulus q_i that divides $\det(A)$, in which case $R_i A \bmod q_i$ will reveal mod-q_i linear correlations between the columns of A. We therefore must choose the q_i's of size slightly larger than 2^k, to ensure the same level of protection against this threat as we get against leakage from the A_i^*'s. Setting $q_i \approx n2^k$ would mean that we only need to work with integers of total size about $n^2 2^{p+2k}$. In our setting with $n = 6, p = 7$ we can choose the (admittedly weak) $k = 15$ and work with 42-bit numbers, which is expensive but doable. (With 18-bit q_i's, we would need six of them to reach the size of q that we need.) Our randomized encoding procedure is described in Figure 1. From the discussion above we have:

Claim 1 *The function REncSolver from Figure 1 is a randomized encoding for the function $f(A, \vec{b}) = A^{-1}\vec{b}$ over the rational numbers, where A is invertible and $A, \vec{b}$ are bounded.* □

6.2 Are We Still Leaking Too Much?

We end this section by pointing out that the above solution is a randomized encoding for the *exact solution function vector* $A^{-1}\vec{b}$, which by itself may already leak information. In particular it usually

reveals the determinant $\det(A)$ (or a factor of it), just by taking the common denominator of all the entries in the solution vector. In the current application, what we really want to compute is the *limited precision solution* function, that rounds the exact solution to some given precision. (We can even tolerate small error in the solution.) We do not know of any feasible randomized encoding for this limited precision function.

7 Implementation and Performance Results

All of our testing was done on an Intel Xeon E5-2698 v3 (which is a Haswell processor), with two sockets and sixteen cores per socket, running at 2.30GHz. The machine has 250GB of main memory, the compiler was GCC version 4.8.5, and we used NTL version 10.5.0 and GnuMP version 6.0.

Parameters. We worked with the cyclotomic ring $\mathbb{Z}[X]/\Phi_m(X)$ with $m = 2^{15} - 1 = 32767$ (so $\phi(m) = 27000$) The largest modulus q in the moduli-chain was about 1030-bit long, corresponding to about 80 bits of security. The plaintext space was set to $2^{11} = 2048$ (but as we explained in Section 3, most of the computation was done with plaintext space modulo 2). This gave as a total of 1800 plaintext slots, arranged in a $30 \times 6 \times 10$ hypercube with the first two 30×6 being "good dimensions" and the last being a "bad dimension".[6] Each plaintext slot held a degree-15 extension of the ring $\mathbb{Z}_{2^{11}}$ (or an element of $GF(2^{15})$ when we used it for mod-2 computation).

This means that we could pack as many as $15 \cdot 1800 = 27000$ integers in a single ciphertext each up to 11-bit long. But in our application we only packed in each of our ciphertext either up to 27000 bits, or up to 180 11-bit integers, depending on the phase of the computation.

7.1 Results for the Logistic-Regression Application

Single-threaded timing. A single-thread execution of the program took just under five hours, from key-generation and encryption of the data up to and including the computation of the matrix A and vector $\vec{v}$. Homomorphic processing took just under 280 minutes of this time, and fifteen minutes were spent packing and encrypting the raw data. The program used only about 4.5GB of RAM. Only 25% of the processing time was spent on the application logic, and about 75% (210 minutes) was spent in 65 bootstrapping operations (so under 3 minutes per operation). The timing results for the different phases of the computation are described in table 1:

- Computing the correlations and extracting its binary representation (`corrBinary`) took almost no time, only 16 seconds;
- About 125 minutes (45% of the processing time) was spent comparing the correlation numbers and computing the indexes of the five fields most correlated to the disease (`topIndexes`);
- Once we found the indexes, it took 33 minutes (12%) to extract the actual data corresponding to these fields (`extractCols`);
- Then it took 47 minutes (17%) to compute the 64 bucket counters and their binary expansion (`bucketCounters`);
- Computing the vectors $\vec{v}$ and $\vec{y}$ using table lookup operations (`compV&Y`) took another hour (22%);

[6] The distinction between "good" and "bad" dimensions in HElib is that 1D-rotations along a good dimension take a single automorphism, while along a bad dimension it takes two automorphisms and some constant multiplies to zero out some data.

# threads:	1	2	4	8	16	30
corrBinary	.25	.18	.13	.13	.1	.1
topIndexes	125	72	42	35	24	24
extractCols	33	20	13	11	8.5	8.7
bucketCounters	47	28	18	16	12	12
compV&Y	60	35	20	16	10	10
compA&b	12	7.7	5.3	5	3.8	3.8
total	**278**	**163**	**99**	**83**	**59**	**59**
recrypt	210	119	70	56	38	38

Table 1: Timing results (minutes) of different phases of the logistic-regression program

- Finally, computing the matrix A and vector $\vec{b}$ from the vectors $\vec{v}, \vec{y}$ (`compA&b`) took only 12 minutes (4%).

Multi-threaded timing. Multi-threading was very effective in reducing the computation time up to eight threads, but using more threads did not help very much (and above sixteen threads the runtime leveled off completely). The processing time dropped to 83 minutes with eight threads and just under one hour with sixteen threads. The RAM consumption increased somewhat, from 4.5GB with one thread to 5.5GB with sixteen. The fraction of time spend on bootstrapping dropped slightly, from 75% with one thread to 64% with sixteen, indicating that multi-threading during bootstrapping was somewhat more effective than in other parts of the computation. The ratio between different phases of the computation did not change much when switching to multi-threaded implementation. See more details in table 1.

7.1.1 Is this Procedure Accurate Enough?

How good is the solution that we obtained from this procedure? As was done in the iDASH competition itself, we measured our solution using the metric of "area under curve" (AUC). The given data consisted of genomic data variables, and a target attribute representing cancer. A random model is expected to give AUC result of 0.5. One of the attributes in this data was the BRCA gene, and taking only that attribute already gives AUC result close to 0.6. On the other hand even the best plaintext-based logistic-regression model only yields AUC of about 0.7 on that dataset. Hence the game for this dataset was to get as far above 0.6 as possible.[7] We tested our solution by running it on sub-sampled data from the training dataset, and the AUC results were usually close to 0.65. This appears similar to other solutions that were submitted to the iDASH competition.

7.2 Timing results for the Various Components

Since our application used a large setting of parameters ($m = 2^{15-1}$) and spent most of its time bootstrapping, the performance results above do not tell the story of how the different components perform for smaller parameters or when bootstrapping is not needed. These numbers are reported in Table 2, with (a) reporting the number of native multiplications and circuit depth for the different operations, while in (b)-(d) we report some performance numbers in various settings.

The addition operations we tested added two n-bit numbers to get their $n+1$-bit sum, while the multiplication operations multiplied two n-bit numbers but only computed the lower n bits of the

[7]These numbers are said to be typical for genomic data, but it probably means that this is not a good dataset on which to develop an approximation procedure. Still this is what we had, so this is what we used.

result. In the tests below we varied the security level (when processing 8-bit numbers), the number of input bits (at security level 125), and also tested the effect of multithreading.

The runtime of the operations range from a few seconds for addition and comparison in the smaller settings to about one minute for multiplication and table lookup in the larger setting. Some trends that can be seen in these numbers include the following:

- As expected, the running times of the various operations grow quasi-linearly with the cyclotomic index m (which is more or less proportional to the security level).
- As the input bitsize grows, the number of native multiplications (and hence the running time) is roughly quadratic for addition, linear for comparison, and roughly $n^{2.3}$ for multiplication. (For table lookup, of course the number of products grows exponentially with the bitsize, since the table itself grows exponentially with the number of bits in the index.)
- Our table lookup implementation is embarrassingly parallel, and indeed we get nearly linear speedup in the number of threads. For the other operations the speedup is less pronounced. From one to eight threads we only get more or less $3X$ speedup, and above eight threads there are almost no additional gains.

8 Conclusions and Discussion

In this work we investigated the question of whether "full blown FHE" can be used for a realistic use case. We devised a procedure to compute an approximate logistic regression model on encrypted data, and demonstrated that this can be achieved in a matter of a few hours (or even just one hour if we use multithreading).

In the course of this work we developed many new tools for homomorphic computations. Many of these tools are general-purpose (such as binary arithmetic, table lookup, etc.), but some are specific to the current setting (e.g., specific data packing and movement schemes). Our experience in this work leads us to believe that the answer to our motivating question is "Yes, but just barely."

We stress that *the main roadblock is not performance*: devising a logistic regression model in a matter of hours may be perfectly acceptable in many settings. (And clusters or hardware acceleration can sometimes be brought to bear as well.) The main problem was the lack of good development and support tools, developing an FHE application feels a lot like programming using only assembly language. (Indeed the reason we did not submit our work to the iDASH competition last year was because it was not debugged in time.)

Using FHE in real-world settings will require much more library and development support, and many more FHE toolboxes beyond the few that we implemented in this work. We believe that this is an important project, and expect to continue working along these directions.

References

[1] Y. AONO, T. HAYASHI, L. T. PHONG, and L. WANG. Privacy-preserving logistic regression with distributed data sources via homomorphic encryption. *IEICE Transactions on Information and Systems*, E99.D(8):2079–2089, 2016.

[2] B. Applebaum. Randomized encoding of functions. In *Cryptography in Constant Parallel Time*, Information Security and Cryptography, pages 19–31. Springer, 2014.

[3] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu. Private database queries using somewhat homomorphic encryption. In *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 2013.

(a) Number of native multiplications and circuit depth for different bit sizes

	addition		comparison		table lookup		multiplication	
bitsize	# mults	depth	# mults	depth	# mults	depth	# mults	depth
4	12	3	17	3	18	2	14	3
8	45	4	37	4	292	3	79	6
12	96	4	58	4			205	8
16	166	5	81	5			411	10

(b) Performance for single-threaded 8-bit operations at different security settings

security	cyclotomic	addition		comparison		table lookup		multiplication	
param	m ($\phi(m)$)	time	RAM	time	RAM	time	RAM	time	RAM
70	8191 (8190)	1.8	153	1.4	142	12.8	342	2.9	180
85	11441 (10752)	3.7	277	2.9	262	28.5	568	6.2	318
210	15709 (15004)	3.8	295	3.0	275	28.5	639	6.3	343
440	32767 (27000)	9.2	610	7.2	576	68.2	1232	15.0	697

(c) Multithreaded performance for 8-bit operations $m = 15709$ (security=210).

	addition		comparison		table lookup		multiplication	
# threads	time	RAM	time	RAM	time	RAM	time	RAM
1	3.8	295	3.0	275	28.5	639	6.3	343
2	3.7	306	1.8	282	14.9	646	5.0	350
4	2.7	315	1.2	300	8.1	658	3.3	369
8	1.8	347	1.0	341	4.6	691	2.0	400
16	1.1	350	1.0	339	2.8	741	1.9	470
32	1.1	353	1.0	334	1.9	867	1.9	607

(d) Performance for single-threaded operations with different input sizes (encrypted at level 13, $m = 15709$, security parameter 125).

	addition		comparison		table lookup		multiplication	
bitsize	time	RAM	time	RAM	time	RAM	time	RAM
4	1.3	359	2.0	355	2.5	375	1.4	368
8	5.5	415	4.4	387	43.1	937	9.1	486
12	12.0	482	6.8	419			23.7	636
16	21.3	564	9.4	451			47.4	880

Table 2: Complexity measures and performence results. Time in seconds, RAM in MB.

[4] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):13, 2014.

[6] H. Chen, K. Laine, and R. Player. Simple encrypted arithmetic library - SEAL v2.1. In *Financial Cryptography Workshops*, volume 10323 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2017.

[7] J. Chen, Y. Feng, Y. Liu, and W. Wu. Faster binary arithmetic operations on encrypted integers. In *WCSE'17, Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering*, 2017.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[9] J. H. Cheon, M. Kim, and M. Kim. Search-and-compute on encrypted data. In *International Conference on Financial Cryptography and Data Security*, pages 142–159. Springer, 2015.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

[11] A. Costache, N. P. Smart, S. Vivek, and A. Waller. Fixed-point arithmetic in SHE schemes. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 401–422. Springer, 2016.

[12] L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[13] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[14] C. Gentry, S. Halevi, C. S. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2015.

[15] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Part I*, pages 75–92. Springer, 2013.

[16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016.

[17] S. Halevi and V. Shoup. Algorithms in HElib. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[18] S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.

[19] S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. https://github.com/shaih/HElib/, Accessed September 2014.

[20] Integrating Data for Analysis, Anonymization and SHaring (iDASH). https://idash.ucsd.edu/.

[21] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. A)*, pages 869–941. MIT Press, Cambridge, MA, USA, 1990.

[22] A. Khedr, P. G. Gulak, and V. Vaikuntanathan. SHIELD: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Trans. Computers*, 65(9):2848–2858, 2016.

[23] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure logistic regression based on homomorphic encryption. Cryptology ePrint Archive, Report 2018/074, 2018. https://eprint.iacr.org/2018/074.

[24] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[25] V. Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.

[26] S. Wang, Y. Zhang, W. Dai, K. Lauter, M. Kim, Y. Tang, H. Xiong, and X. Jiang. Healer: homomorphic computation of exact logistic regression for secure rare disease variants analysis in gwas. *Bioinformatics*, 32(2):211–218, 2016.

[27] C. Xu, J. Chen, W. Wu, and Y. Feng. Homomorphically encrypted arithmetic operations over the integer ring. In F. Bao, L. Chen, R. H. Deng, and G. Wang, editors, *Information Security Practice and Experience*, pages 167–181, Cham, 2016. Springer International Publishing. https://ia.cr/2017/387.