Homomorphic Training of 30,000 Logistic Regression Models

Flavio Bergamaschi[1], Shai Halevi[2], Tzipora T. Halevi[3], and Hamish Hunt[1]

[1] IBM Research, UK, {flavio,hamishhun}@uk.ibm.com
[2] IBM Research, NY, USA, shaih@alum.mit.edu
[3] Brooklyn College, NY, USA, thalevi@nyu.edu

Abstract. In this work, we demonstrate the use the CKKS homomorphic encryption scheme to train a large number of logistic regression models simultaneously, as needed to run a genome-wide association study (GWAS) on encrypted data. Our implementation can train more than 30,000 models (each with four features) in about 20 minutes. To that end, we rely on a similar iterative Nesterov procedure to what was used by Kim, Song, Kim, Lee, and Cheon to train a single model [14]. We adapt this method to train many models simultaneously using the SIMD capabilities of the CKKS scheme. We also performed a thorough validation of this iterative method and evaluated its suitability both as a generic method for computing logistic regression models, and specifically for GWAS.

Keywords. Approximate numbers, Homomorphic encryption, GWAS, Implementation, Logistic regression

1 Introduction

In the decade since Gentry's breakthrough [9] we saw rapid improvement in homomorphic encryption (HE) techniques. What started as a mere theoretical possibility is now a promising technology on its way from the lab to the field. Many of the real-world problems to which this technology was applied originated in the yearly competitions that are organized by the iDASH center [12]. These competitions, organized annually since 2014, pose specific technical problems related to privacy preserving analysis of medical data and ask for solutions using specific technologies. Some of the problems posed in the 2017 and 2018 instalments dealt with training logistic regression (LR) models on encrypted data.

In 2017, the task was to devise a single model. Many solutions were suggested that perform this task in a matter of a few minutes to a few hours [14,6,1,15,8,11]. In particular, the winning entry in the 2017 competition was due to Kim et al. [14], using the HE scheme due to Cheon et al. [7] (which we call below the CKKS scheme). For 2018, the goal was to train a very large number of models, as needed for a Genome-Wide Association Study (GWAS). In GWAS a large number of markers are simultaneously tested for their association with some condition (such as a specific disease). The modus operandi in GWAS is to devise a large number of LR models at once, each model using only a handful of markers, then test these models to see which of them have good predictive power. For the 2018 competition, the iDASH organizers provided a dataset with over 10,000 markers, noting that "implementation of linear or logistic regression based GWAS would require building one model for each SNP, which requires a lot of time." (SNP is a single genomic marker.) Instead they suggested to use the *semi-parallel algorithm* of Sikorska et al. [18] for that purpose.

1.1 Our Work

The goal of the current work was to show that "building one model for each SNP" can actually be accomplished with reasonable resources, by a careful adaptation of the techniques used in the iDASH competition from 2017. Specifically, we implemented a solution along the same lines as the procedure used by Kim et al. [14] with some corrections and optimizations. Our implementation is able to compute more than 30,000 LR models in parallel taking only about 20 minutes.

This work consists of two parts: In one part, we adapted the iterative procedure of Kim et al. [14] to the setting of GWAS, using the SIMD capabilities of the CKKS HE scheme, to compute a large number of models simultaneously. In the other part, we performed a thorough validation of this iterative method, evaluating its suitability both as a generic method for computing LR models and specifically for GWAS.

Adapting and validating the iterative procedure. The iterative procedure of Kim et al. [14] applies Nesterov's accelerated gradient descent [16] with a very small number of iterations (and uses the CKKS cryptosystem to run it on encrypted data). While Kim et al. evaluated the accuracy of their method, the GWAS setting raises some other demands that were not evaluated in [14]. For one thing, in [14] they only devised a handful of models on data with a rather strong signal, whereas in GWAS we need to devise many thousands of models on data that ranges from having very strong to very weak signal (and many in between). Moreover, for GWAS we had to train the model on encrypted data, and also *evaluate it homomorphically* by computing the log-likelihoods ratio.

We found that some details of the iterative procedure had to be adapted to this setting. One notable issue was that when the data was not balanced (e.g., with more 0's than 1's), as the signal weakens the model tends to degenerate to the constant predictor that always says zero (hence getting a *recall value* of zero). In our tests, we found that sub-sampling the training data to ensure that it is balanced resulted in much better recall values with almost no effect on the accuracy of the model. We also found and fixed a few minor mistakes and inconsistencies in the procedure from [14] and its evaluation, see section 3.

In the tests ran, we compared the adequacy of the iterative procedure (in terms of ordering the genomic markers by relevance) to that of the semi-parallel algorithm. We concluded that the ordering in both methods are mostly equivalent, but the iterative procedure often provided better model parameters. We also compared the approximate LR models of the iterative procedure to the LR models computed by Matlab's `glmfit` function. Surprisingly, even when we use very few iterations, the resulting LR models are just as predictive as the ones produced by Matlab, see section 3.1.

We used several datasets of very different characteristics for testing. One was the genome dataset provided by the iDASH team. Others include the Edinburgh myocardial infarction dataset [13] (also used by Kim et al.), a credit-card fraud dataset [2] [17], and the dataset related to the sinking of the RMS Titanic [3].[4]

Homomorphic implementation. Like many contemporary homomorphic encryption schemes, the CKKS approximate number scheme of Cheon et al. [7] supports Single-Instruction-Multiple-Data (SIMD) operations: ciphertexts in CKKS encrypt vectors of numbers and each homomorphic operation induces element-wise operations on the corresponding vectors. This provides the basis of our GWAS procedure: simply pack the parameters of the different models in different entries of these vectors, then use the SIMD structure to run the iterative procedure on all of them in parallel. Specifically in our setting, we used ciphertexts that can pack upto $2^{15} = 32768$ numbers, so we can compute that many models in parallel.

Implementing this approach requires some care, particularly with regards to RAM consumption. We need to ensure that the computation fits in the available RAM as packed ciphertexts are typically large. A notable optimization described in section 4.2 takes advantage that CKKS ciphertexts can pack a vector of *complex numbers* (not just real numbers). Our optimization uses that fact to reduce the number of operations by almost a factor of two by packing twice as many real numbers in each ciphertext, but paying some price in larger noise accumulation.

We also mention that our CKKS implementation, done over the HElib engine [10], differs from other implementations in some details (which makes working with it a little easier). These details are described in section 4. As mentioned above, using this implementation we can compute all the LR models for a GWAS with upto 2^{15} markers and three clinical variables in about twenty minutes.

We note that the running time would grow nearly quadratically with the number of clinical variables: since to train models with more variables we also need more records, then the size of the input matrix grows quadratically with the number of variables. With three clinical variables we were able to train the models in under 20 minutes, and a back-of-an-envelope calculation indicates that we could handle 8-10 clinical variables in about an hour.

Organization In section 2, we provide some background on LR, GWAS, and Nesterov's Accelerated Gradient Descent [16]. In section 3, we provide details on our variant of the iterative procedure, and our testing methodology and results. In section 4, we describe the implementation of this procedure on encrypted data and provide various runtime measurements.

[4] The last three datasets are much smaller than we would like. Nonetheless, they contain features with strong signal and others with very weak signal, so we can still use them to evaluate the GWAS setting.

2 Background

2.1 Logistic Regression

Logistic regression (LR) is a machine-learning technique trying to predict one attribute (condition) from other attributes. In this work, we only deal with the case where the condition that we want to predict is binary (e.g., sick or healthy). The data that we get consists of n records (rows) of the form $(y_i, \boldsymbol{x_i})$ with $y_i \in \{0,1\}$ and $\boldsymbol{x_i} \in \mathbb{R}^d$. We would like to predict the value of $y \in \{0,1\}$ given the attributes $\boldsymbol{x}$, and the logistic regression technique postulates that the distribution of y given $\boldsymbol{x}$ is given by

$$\Pr[y=1|\boldsymbol{x}] = \frac{1}{1+\exp\big(-w_0 - \sum_{i=1}^{n} x_i w_i\big)} = \frac{1}{1+\exp\big(-\boldsymbol{x'}^T\boldsymbol{w}\big)},$$

where $\boldsymbol{w}$ is some fixed $(d+1)$-vector of real weights that we need to find, and $\boldsymbol{x'_i} = (1|\boldsymbol{x_i}) \in \mathbb{R}^{d+1}$. Given the training data $\{(y_i, \boldsymbol{x_i})\}_{i=1}^{n}$, we thus want to find the vector $\boldsymbol{w}$ that best matches this data, where the notion of "best match" is typically maximum likelihood. Using the identity $1 - \frac{1}{1+\exp(-z)} = \frac{1}{1+\exp(z)}$, we therefore want to compute (or approximate)

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \left\{ \prod_{y_i=1} \frac{1}{1+\exp\big(-\boldsymbol{x'_i}^T\boldsymbol{w}\big)} \cdot \prod_{y_i=0} \frac{1}{1+\exp\big(\boldsymbol{x'_i}^T\boldsymbol{w}\big)} \right\}.$$

The last condition can be written more compactly: let $y'_i = 2y_i - 1 \in \{\pm 1\}$ and $\boldsymbol{z}_i = y'_i \cdot \boldsymbol{x'}_i$, then our goal is to compute/approximate

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \left\{ \prod_{i=1}^{n} \frac{1}{1+\exp\big(-\boldsymbol{z_i}^T\boldsymbol{w}\big)} \right\} = \arg\min_{\boldsymbol{w}} \left\{ \sum_{i=1}^{n} \log\big(1+\exp(-\boldsymbol{z_i}^T\boldsymbol{w})\big) \right\}.$$

For a candidate weight vector $\boldsymbol{w}$, we denote the (normalized) *loss function* for the given training set by

$$J(\boldsymbol{w}) \overset{\text{def}}{=} \frac{1}{n} \cdot \sum_{i=1}^{n} \log\big(1+\exp(-\boldsymbol{z_i}^T\boldsymbol{w})\big), \tag{1}$$

and our goal is to find $\boldsymbol{w}$ that minimizes that loss.

Gradient Descent and Nesterov's Method. In this work, we use a variant of the iterative method used by Kim et al. in [14] based on Nesterov's accelerated gradient descent [16]. Let σ be the sigmoid function $\sigma(x) \overset{\text{def}}{=} 1/(1+e^{-x})$, it can be shown that the gradient of the loss function with respect to $\boldsymbol{w}$ is

$$\nabla J(\boldsymbol{w}) = -\frac{1}{n} \sum_{i=1}^{n} \frac{1}{1+\exp(\boldsymbol{z_i}^T\boldsymbol{w})} \cdot \boldsymbol{z_i} = -\frac{1}{n} \sum_{i=1}^{n} \sigma\big(-\boldsymbol{z_i}^T\boldsymbol{w}\big) \cdot \boldsymbol{z_i}. \tag{2}$$

Nesterov's method initializes two evolving vectors (e.g., to the average of the input records), then in each iteration it computes

$$\begin{aligned} \boldsymbol{w}^{(t+1)} &= \boldsymbol{v}^{(t)} - \alpha_t \cdot \nabla J(\boldsymbol{v}^{(t)}), \\ \boldsymbol{v}^{(t+1)} &= (1-\gamma_t) \cdot \boldsymbol{w}^{(t+1)} + \gamma_t \cdot \boldsymbol{w}^{(t)}, \end{aligned} \tag{3}$$

where α_t, γ_t are scalar parameters that change from one iteration to the next. (α is the learning rate and γ is called the moving average smoothing parameter, see section 3 for how they are set.)

Approximating the Sigmoid. As in [14], we use low-degree polynomials to approximate the sigmoid function in a bound range around zero. We use the same degree-3 and degree-7 approximation polynomials in the interval $[-8, +8]$, namely

$$SIG3(x) \overset{\text{def}}{=} 0.5 - 1.2\left(\frac{x}{8}\right) + 0.81562\left(\frac{x}{8}\right)^3 \text{ and} \tag{4}$$

$$SIG7(x) \overset{\text{def}}{=} 0.5 - 1.734\left(\frac{x}{8}\right) + 4.19407\left(\frac{x}{8}\right)^3 - 5.43402\left(\frac{x}{8}\right)^5 + 2.50739\left(\frac{x}{8}\right)^7$$

2.2 Genome-Wide Association Study (GWAS)

In genetic studies, LR is often used for Genome-Wide Association Study (GWAS). Such studies take a large set of genomic markers (SNPs) and determine which of them are associated with a given trait. A GWAS typically considers one condition variable (e.g., sick or healthy), a small number of clinical variables (such as age, gender, etc.) and a large number of SNPs. For each SNP separately, the study builds a LR model that tries to predict the condition from the clinical variables and that one SNP, then tests how good that model is at predicting the condition. (The clinical variables are sometimes called covariates, below we use these terms almost interchangeably.)

Assessing a Model: Likelihood Ratio, *p*-values, Accuracy, Recall. One way to evaluate the quality of a LR model is to compute its loss function $J(\boldsymbol{w})$, eq. (1). We note that this number has a semantic meaning, it is the logarithm of the likelihood of the training data according to the LR model (with parameters $\boldsymbol{w}$). This number can then be used to compute the likelihood-ratio-test (LRT)[5] which is sometimes called the "p-value" of the model. We note that eq. (1) is not the only formula used for computing p-values (indeed the iDASH competition organizers used a different formula for it). However, at least according to Wikipedia, the LRT is "the recommended method to calculate the p-value for logistic regression" (cf. [19]).

Another way to evaluate the model is to use it for prediction and test how well it performs. Typically, you would divide your dataset into training and

[5] The LRT measures how much more likely we are to observe the training data if the true probability distribution of the y_i's is what we compute in the model vs. the probability to observe the same training data according to the null hypothesis in which the y_i's are independent of the $\boldsymbol{x}_i$'s.

test data, devise the model on the training data, then use it on the test data to predict the value of the y_i's (predicting $y_i = 1$ if $\Pr[y = 1|\boldsymbol{x}_i] > 1/2$ and $y_i = 0$ otherwise). The fraction of correct predictions is called the *accuracy* of the model. It is common to use *five-fold testing* where the procedure above is repeated five times, each time choosing 80% of the records for training and the rest for testing, then averaging the accuracy values of the five runs.

Overall accuracy may not always be a good measure of performance. For example, if 90% of the records in our dataset have $y_i = 0$ then even the constant predictor $y = 0$ will have 90% accuracy. We therefore also test the *recall* of the model, which is its success probability over *only the records with* $y_i = 1$. This too is typically measured with a five-fold testing.

3 The Logistic Regression Iterative Procedure

The LR procedure that we used is similar to the one used by Kim et al. [14], but we had to make some changes and correct a few inaccuracies:

Balancing the input. We observed that when the input dataset is unbalanced, the model obtained from the iterative procedure is highly biased as well, sometimes to the point of having recall value of zero. Our program therefore trains the model always on a random subset of the input dataset where 50% of the records have $y_i = 0$ and 50% have $y_i = 1$. This simple solution corrects the recall values of the resulting models and in our tests it only has a very minor effect on their accuracy.

We remark that this solution can be applied even when the data is encrypted, for example, by storing the $y = 0$ encrypted records separately from the $y = 1$ records. This of course will reveal the y value of all the records, but nothing else about them. If we want to hide also the y value of the records and if we know *a priori* the fraction p of records with $y = 1$, then we could just choose at random which records to use in the study during encryption. For example, if $p < 1/2$ we can choose each $y = 1$ record with probability one and each $y = 0$ record with probability $p/(1-p)$.

The number of iterations. The number of iterations that we can perform is very limited as we are using a somewhat-homomorphic encryption scheme to implement the procedure on encrypted data. We denote this number by τ, and in our implementation and tests we used $\tau = 7$ iterations.

Initializing the evolving state. Since we need to use a small number of iterations, the initial values of $\boldsymbol{v}, \boldsymbol{w}$ is important to the convergence of the weights. Our tests show that setting them as the average of the inputs (i.e., $\boldsymbol{v}^{(0)} = \boldsymbol{w}^{(0)} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{z}_i$) yields better results than choosing them at random.[6]

The α and γ parameters. The learning-rate parameter α was set just as in [14], namely in iteration $t = 1, \ldots, \tau$ we used $\alpha_t = 10/(t+1)$.

[6] This form of initialization differs from the description in [14], but it is consistent with the code shared online by the authors.

For the moving average smoothing parameter γ, Kim et al. stated in [14] that they used $\gamma \in [0,1]$, but positive γ values result in bad performance of the Nesterov algorithm. Instead, we used negative values for gamma as suggested in [5]: Setting $\lambda_0 = 0$, we compute for $t = 1, \ldots, \tau$

$$\lambda_t = \frac{1 + \sqrt{1 + 4\lambda_{t-1}^2}}{2} \text{ and } \gamma_t = \frac{1 - \lambda_{t-1}}{\lambda_t}.$$

The values of γ for the first few steps are therefore $\gamma \approx (1, 0, -0.28, -0.43, -0.53, -0.6, -0.65, \ldots)$

Precision. We tested our procedure in order to decide how much precision is needed since the CKKS scheme only offers limited precision. Our tests found no significant difference in performance, even with only six bits of precision (i.e. error of upto 2^{-7} per operation). We therefore decided to set the precision parameter for the homomorphic scheme at $r = 8$, corresponding to 2^{-8} error. As there was no real effect, we ran most of our plaintext tests below with full precision.

Computing the log-likelihood. In addition to computing the model parameters, we extended the procedure from above to also compute the loss function (i.e., the log-likelihood of the resulting model). For this purpose, we needed to approximate also the log-sigmoid function using a low-degree polynomial, in particular we used the degree-4 approximation in the range $[-8, 8]$ (obtained using Python's `numpy.polyfit`):

$$LOGSIG4(x) \overset{\text{def}}{=} 0.000527x^4 - 0.0822x^2 + 0.5x - 0.78 \approx \log(\sigma(x)) \qquad (5)$$

We then approximate the log-likelihood of each model $\boldsymbol{w}$ as $LOSS(\boldsymbol{w}) \approx -\sum_{i=1}^{n} LOGSIG4(\boldsymbol{z}_i^T \boldsymbol{w})$.

3.1 Experimental Evaluation

We evaluated our procedure across multiple parameters and settings, and compared it to alternative procedures. When attempting such evaluation, it is important to ensure that the procedure is not over-engineered to fit just one type of data, so we run our tests against four different datasets with very different characteristics (though not every test was run on every dataset). These datasets included the iDASH 2018 dataset for correlating cancer with genomic markers, a credit-card fraud dataset, the Edinburgh dataset for correlating heart attacks with various tests and symptoms, and a dataset for correlating various passenger characteristics with the rate of survival in the Titanic disaster. See appendix B for more details on these datasets.

Sorting the columns in order of relevance. One focus of this work is GWAS-like procedures where we want to filter out the irrelevant columns. Hence, many of our tests examined *the order of relevance* of the different columns rather than the actual model parameters for their respective models. In these tests, we computed all the LR models, one model per SNP (all models containing the clinical variables), approximated the log-likelihood for each, and ordered the SNPs in decreasing order of their log-likelihood. This order is our procedure's estimate of the order of relevance of the different SNPs to the condition. We then used the following methodology to evaluate the "quality" of this ordering:

– We applied the Matlab implementation of LR to re-compute the LR model on the same data, using the `glmfit` function. The resulting LR models (one per SNP) could be different than those produced by our iterative procedure;
– Next, we ran a five-fold test on the data using Matlab's `glmval` to compute the predicted condition values, and computed the accuracy and recall for each model;
– Finally, we plotted the accuracy and recall values against the order of columns from our iterative procedure.

If the procedure works well, we expect a decreasing order of accuracy and recall, since the first models in the order are supposed to correspond to the most relevant SNPs, and hence to the highest accuracy and recall values.

We compared the column ordering from our iterative procedure to the ordering generated by the p-values of the semi-parallel algorithm of Sikorska et al. [18]. (We used the R implementation provided by the iDASH organizers for that purpose). We also tested the column ordering using the accuracy and recall results as produced by the Matlab LR models (using `glmfit` and `glmval`), plotting them against the column ordering of the semi-parallel algorithm.

We stress that for both orderings, we plot the exact same accuracy and recall numbers, i.e. the ones corresponding to the Matlab LR models. The only difference is the order in which we plot these numbers.

Evaluating the model parameters. Since our procedure yields not only the log-likelihood (a.k.a. p-value) for each model but also the model parameters themselves, we ran a few tests to examine how well these models perform. Namely, we compared the accuracy and recall of our models with those of the Matlab LR models on the same data. Again, we plotted all the accuracy and recall results in the order of columns of our iterative procedure.

Different approximations of the sigmoid function. We tested our iterative procedure in two settings, one using nine iterations with the degree-3 approximation of the sigmoid, and the other using seven iterations with the degree-7 approximation. While there were no significant differences in the order of SNPs produced by the two variants, the model parameters produced by the degree-7 approximation were often improved than those produced by the degree-3 approximation. We therefore ran most of our tests using only the seven-step degree-7 approximation.

3.2 Accuracy and Recall Results

Here, we summarize the test results. All of these tests were run with our iterative procedure using the degree-7 approximation of the sigmoid function and $\tau = 7$

iterations. All accuracy and recall results below were obtained using five-fold testing (described in section 2.2). One point that needs care when running a five-fold test, is ensuring that the test data has similar characteristics to the training data: Some datasets are collected from multiple sources, hence the first records may have very different characteristics than the last ones. (In particular the data provided for the iDASH competition had that problem.) Randomizing the order of the records in the dataset before running the test fixes this.

Comparing the column ordering, iterative vs. semi-parallel. As we explained above, we run our iterative procedure and the semi-parallel algorithm from [18] side by side on the same data, computing the p-values from each and ordering the SNPs according to these p-values. We then used the Matlab implementation of LR to compute the accuracy and recall values of the model corresponding to each SNP (with the same clinical variable), and plotted these accuracy and recall values in the two orders.

The results for the iDASH dataset are depicted in fig. 2. For that dataset, the two orders more or less coincide for the most relevant 1500 columns or so (out of 10643). For the next 1500 columns, the orders are no longer the same. Moreover, while the accuracy results are very similar, the iterative ordering yields better recall values than the semi-parallel ordering. The last 8000 columns no longer contain much information on the condition variable, hence the ordering of these columns is essentially random. We also ran the same test for the credit-card fraud dataset, which contains only 30 columns. Here while the two orders identify the same top nine, middle seven, and bottom fourteen columns, the ordering within each of the first two groups was somewhat more accurate for the semi-parallel algorithm than for the iterative method.

Comparing the iterative vs. Matlab models. Next, we tried to evaluate the quality of the models generated by the iterative method to the standard LR models of Matlab. Since the iterative method with so few steps is only a crude approximation, we expected the Matlab model to perform better, but wanted to check by how much. We therefore computed for each column the accuracy and recall values of both models (iterative vs. Matlab), and plotted them against the p-value ordering from our procedure. (See the full version for a plot of the results.)

To our surprise, the crude approximated model computed by the iterative method performed at least as well (and sometimes better) than the LR model that Matlab computed for the same data. We can see that the iterative model has some bias for outputting $y = 1$, resulting in better recall and somewhat worse accuracy values. For example, notice that around the 1000'th SNP the iterative model has recall value of 1, while the Matlab model's recall values are capped around 0.9 (with essentially the same accuracy). We ran the same test also on the Titanic dataset, and again the iterative models did about as well (and sometimes better) than the Matlab models.

The Edinburgh dataset. We also ran our iterative procedure on the Edinburgh datasets computing the accuracy/recall for the Matlab model for each column and plotting these values against the p-value ordering of the columns as produced by the iterative procedure.

3.3 Conclusions and Some Comments

Summarizing the tests above, the iterative procedure that we used produces models which are competitive to what we get from Matlab, and that the relevance order that we get from our p-values is just as reasonable as the one obtained by the semi-parallel algorithm. While the semi-parallel algorithm is faster (especially when there are many covariates), for a small number of covariates the iterative procedure has reasonable performance. A reasonable conclusion to draw is that one should still run the semi-parallel algorithm in the context of GWAS, but use the iterative model if it is desired to also get the actual LR models (in addition to ordering the columns by relevance).

In this context, the semi-parallel algorithm assumes that the model weights for the covariates are more or less the same when you devise a model for just the covariates as when you devise a model for the covariates and a single SNP. For the iDASH dataset, this was true for most SNPs (since most SNPs were not correlated with the condition at all), but our tests showed that it seems to **not** be true for the most relevant SNPs. This observation implies that while the semi-parallel algorithm is a good screening tool to filter out the irrelevant SNPs (for which the assumption on the covariates should hold), it probably should not be used to compute the model parameters for the more relevant SNPs.

Finally, during our work we encountered two minor bugs/inconsistencies in the literature, notified the relevant authors, and document them in appendix A.

4 Homomorphic Evaluation of the LR Procedure

To evaluate the procedure from section 3 on encrypted data, we used the CKKS approximate-number HE scheme of Cheon et al. [7], which we implemented in the HElib library [10]. The underlying plaintext space of this scheme are complex numbers (with limited precision), and the scheme can pack many such complex numbers in a single ciphertext. In section 4.3 below, we briefly describe some details of our HElib-based implementation, see the original work [7] for details about the scheme itself. The API provided by our implementation is as follows:

Parameters. Security parameter λ, plus two functionality parameters: The packing parameter ℓ determines how many complex numbers can be encoded in a single ciphertext, and the accuracy parameter r determines the supported precision. Operations of the scheme are accurate up to additive noise of magnitude bounded by 2^{-r}. We refer to entries in the encrypted vectors as *plaintext slots.*

Noisy Encoding. The native objects manipulated in the CKKS scheme belong to an algebraic ring (specifically algebraic integers in cyclotomic number fields). The scheme provides routines to encode and decode plaintext complex vector

$\boldsymbol{v} \in \mathbb{C}^\ell$ into and out of that ring. However the encoding is noisy, which introduces additive errors of magnitude up to 2^{-r} in each entry.

Encryption, decryption, and homomorphic operations. Once encoded in the "native ring," data can be encrypted and decrypted using the public and secret keys, respectively.

- The scheme supports addition and multiplication operations, both plaintext-to-ciphertext and ciphertext-to-ciphertext, including element-wise addition/multiplication on the underlying complex vectors. Providing $w_t = u_t + v_t$ for every entry t for addition, and similarly $w_t = u_t \cdot v_t$ for multiplication.
- There are procedures (which are essentially free) for multiplying and dividing ciphertexts by real numbers, namely setting $v_t = u_t \cdot x$ or $v_t = u_t/x$ for all t.
- Included is the support for "homomorphic automorphisms". Our application uses automorphisms for computing complex conjugates. Namely, given an encoded (or encrypted) vector $\boldsymbol{u}$, the conjugate operation outputs a similarly encoded/encrypted vector $\boldsymbol{v}$ such that $v_t = \bar{u}_t$ for every entry t. Used to homomorphically extract the real and imaginary parts, via $\mathsf{im}(x) = (x - \bar{x})/2\mathsf{i}$ and $\mathsf{re}(x) = (x + \bar{x})/2$ (with i denoting the imaginary square root of -1).

All the operations above (including encoding and encryption) accrue additive errors. Namely, an operation can return a vector $\boldsymbol{v}'$ that differs from the intended result $\boldsymbol{v}$, with the guarantee that for every entry t we have $|v_t - v'_t| \le 2^{-r}$.

4.1 The Homomorphic LR Procedure

The input to the LR procedure consists of n records, each containing k covariates (i.e., clinical variables such as age or gender), N genomic markers (or SNPs), and a single binary condition variable (sick or healthy). Our solution is tailored for the case where k is small (up to five), N is large (many thousands) and the number of records is moderate (hundreds to a few thousands).

Our goal is to compute N (approximate) LR models, one per SNP, where the t'th model includes parameters for all the k clinical variables and the (single) t'th SNP. As described in section 3, our approach follows the approach by Kim et al. [14]. Namely, we run an iterative method using Nesterov's algorithm and a low-degree approximation of the sigmoid function implemented on top of the CKKS approximate-number homomorphic encryption scheme [7].

The main difference is that we use the inherent SIMD properties of CKKS to compute all the N models at once: We run the LR computation in a *bitslice mode*, where we pack the data into a number of N-vectors with the t'th entry in each vector corresponds to the t'th model. Each input record has $k + 2$ input ciphertexts: One for the condition variable (with all the slots holding the same condition bit), one for each of the covariates (with all the slots holding the same covariate value), and one more ciphertext for all the SNPs (with the different SNPs in the different slots).

```
Input($n$-by-$(k+2)$ matrix $C$)
1.  $\boldsymbol{w} := \boldsymbol{v} := \frac{1}{n}\sum_{i=1}^{n} C_{i-}$   // initialize evolving state to average of the rows in $C$
2.  Repeat for $\tau$ steps:        // run the iterative process
3.      $\boldsymbol{x} := C \times \boldsymbol{v}^T$              // $\boldsymbol{x}$ is a dimension-$n$ column vector
4.      $\boldsymbol{y} := SIG7(\boldsymbol{x})$            // approximate the sigmoid on each entry of $\boldsymbol{x}$
5.      $\boldsymbol{g} := -\boldsymbol{y}^T \times C$             // the gradient $\boldsymbol{g}$ is a dimension-$(k+2)$ row vector
6.      Compute $\alpha, \gamma \in \mathbb{R}$ for this step  // see details in section 3
7.      $\boldsymbol{w'} := \boldsymbol{v} + \alpha \cdot \boldsymbol{g}$
8.      $\boldsymbol{v} := \gamma \cdot \boldsymbol{w} + (1-\gamma) \cdot \boldsymbol{w'}$          // $= \gamma\boldsymbol{w} + (1-\gamma)\boldsymbol{v} + \alpha(1-\gamma)\boldsymbol{g}$
9.      $\boldsymbol{w} := \boldsymbol{w'}$
10. $\boldsymbol{x} := C \times \boldsymbol{v}^T$                  // compute the log likelihood of the model
11. $\boldsymbol{y} := LOGSIG4(\boldsymbol{x})$            // approximate the log-sigmoid on each entry of $\boldsymbol{x}$
12. $u = \sum_{i=1}^{n} y_i$                   // the log-likelihood
13. output $\boldsymbol{w}$ and $u$              // output the resulting model weights and log likelihood
```

Fig. 1. The homomorphic logistic regression procedure

We denote by C the $n \times (k+2)$ matrix of input ciphertexts, where each row i corresponds to an input record and each column j corresponds to a model parameter.[7] Given the input matrix C, we evaluate homomorphically the iterative Nesterov-based procedure described in section 3 for as many steps as our parameters allow. Our main solution uses seven iterations, each employing a degree-seven approximation of the sigmoid function. The homomorphic procedure is described on a high-level in fig. 1 with details discussed below.

Fitting the computation in RAM. Note that as described in fig. 1, each iteration of the main loop requires two passes over the input matrix C, one for computing $C \times \boldsymbol{v}$ in Line 3 and another to compute $\boldsymbol{y} \times C$ in Line 5. If C does not fully fit in memory, then each iteration would require swapping it twice in and out of main memory. Instead, partitioning C into bands that fit in RAM requires a single pass over it in each iteration. Let $I_1, I_2, \ldots, I_b$ be a partition of the row indexes $[n]$ and let $C_{I_1}, \ldots, C_{i_b}$ be the corresponding partition of the rows of C (and similarly $\boldsymbol{x}_{I_1}, \ldots, \boldsymbol{x}_{i_b}$ be the partition of the entries of $\boldsymbol{x}$, and the same for $\boldsymbol{y}$). We replace lines 3-5 by the following computation:

[7] Another "hidden" dimension are the slots $t = 1, \ldots, N$ in each ciphertext, but since our computation is completely SIMD then we can ignore that dimension.

```
[...]
2.  Repeat for $\tau$ steps:                   // run the iterative process
2a.     $\boldsymbol{g} := \mathbf{0}$
2.      For $h = 1$ to $b$                     // go over the bands of $C$
3′.         $\boldsymbol{x}_{I_h} := C_{I_h} \times \boldsymbol{v}^T$        // $\boldsymbol{x}_{I_h}$ is part of $\boldsymbol{x}$
4′.         $\boldsymbol{y}_{I_h} := SIG7(\boldsymbol{x}_{I_h})$     // approximate the sigmoid on each entry of $\boldsymbol{x}$
5′.         $\boldsymbol{g} := \boldsymbol{g} - \boldsymbol{y}_{I_h}^T \times C_{I_h}$  // the contribution of $C_{I_h}$ to the gradient
6.      [...]                                  // continue with the update of $\boldsymbol{v}, \boldsymbol{w}$ as before
```

Computing the log likelihood. As we explained in section 2.2, after computing the model parameters $\boldsymbol{w}$ we need to also evaluate this model by computing its p-value, i.e, the loss function from eq. (1). This computation is very similar to the computation of the gradient, but here we use the approximation of the log-sigmoid $LOGSIG4$ instead of the $SIG7$ approximation of the sigmoid itself. Namely we first compute $\boldsymbol{x} := C \times \boldsymbol{w}$, then $\boldsymbol{y} := LOGSIG4(\boldsymbol{x})$, and finally sum up (or average) the entries in the vector $\boldsymbol{y}$.

4.2 Fewer Multiplications Via Complex Packing

We implemented a second variant of our solution, which is faster and uses half the number of ciphertexts, but adds more noise per iteration. This was done by packing the data more tightly, utilizing both the real part and the imaginary part of each plaintext slot, thus encrypting two input records in each ciphertext (one in the real part of all the slots and the other in the imaginary parts). Specifically, let $z_{2i-1,j}, z_{2i,j}$ be the two real values that were encrypted in the two ciphertexts $C_{2i-1,j}, C_{2i,j}$ in the matrix C from above. In the new variant we instead use a single ciphertext $C'_{i,j}$, encrypting the complex value $z'_{i,j} = z_{2i-1,j} + \mathsf{i} \cdot z_{2i,j}$ (with i the imaginary square root of -1). Let $C' = [C'_{i,j}]$ be the resulting ciphertext matrix, and $N' = \lceil N/2 \rceil$ be the number of rows in the matrix C'.

During the computation we maintain the evolving state vectors $\boldsymbol{v}, \boldsymbol{w}$ as *real vectors* (i.e., their imaginary part is zero). This sometimes requires spliting the encrypted complex numbers into their real and imaginary parts (using the conjugate operation mentioned above). For example, we initialize the evolving state by computing the average of the (complex) rows of C'. Then we split the result into its real and complex parts and average the two.

Similarly, we sometimes also need to assemble two real values into a complex one, just by computing $z_c = z_r + \mathsf{i} \cdot z_i$ homomorphically. These split and assemble operations cause this variant to accrue more noise than before. However, it uses half as many input ciphertexts and roughly half as many operations per iteration of the Nesterov algorithm.

Computing the gradient. The most interesting aspect of this complex-packed procedure is the computation of the gradient in Steps 3-5 from fig. 1. The multiplication in Step 3 is quite straightforward: since $\boldsymbol{v}$ encrypts a real vector, we can compute $\boldsymbol{x'} := C' \times \boldsymbol{v}^T$ just as before and the multiplication by $\boldsymbol{v}$ operates separately on the real and imaginary parts of C'.

To apply the sigmoid function, requires spliting the resulting $\boldsymbol{x'}$ into its real and imaginary components and compute the sigmoid approximation on each of them separately. Namely, we set $\boldsymbol{x}_r := \mathsf{re}(\boldsymbol{x})$ and $\boldsymbol{x}_{\mathsf{i}} := \mathsf{im}(\boldsymbol{x})$, then $\boldsymbol{y}_r := SIG7(\boldsymbol{x}_r)$ and $\boldsymbol{y}_{\mathsf{i}} := SIG7(\boldsymbol{x}_{\mathsf{i}})$. To save on noise, we fold into the sigmoid computation some of the multiply-by-constant operations from splitting $\boldsymbol{x'}$.

More interesting is how to compute the product $\boldsymbol{y} \times C'$ from Step 5 with our tightly packed version of the ciphertext matrix. Here we use the happy coincidence that for complex numbers we have $(a + \mathsf{i}b)(a' - \mathsf{i}b') = aa' + bb' + \mathsf{i} \cdot \mathsf{something}$, giving us the inner product $\langle (a, b), (a', b') \rangle$ in the real part. We therefore pack $\boldsymbol{y'} := \boldsymbol{y}_r - \mathsf{i} \cdot \boldsymbol{y}_{\mathsf{i}}$, compute $\boldsymbol{g'} = -\boldsymbol{y} \times C'$, and the real part of $\boldsymbol{g'}$ turns out to be exactly the gradient vector that we need. To see this, recall that for all i, j we have $y'_j = y_{2j-1} - \mathsf{i} \cdot y_{2j}$ and $C'_{i,j} = z_{2j-1} + \mathsf{i} \cdot z_{2j}$, and therefore

$$\begin{aligned} g'_j = \sum_{i=1}^{N'} y'_i \cdot C'_{i,j} \; &= \; \sum_{i=1}^{N/2} \big(y_{2i-1} - \mathsf{i} \cdot y_{2i}\big) \cdot \big(z_{2i-1,j} + \mathsf{i} \cdot z_{2i,j}\big) \\ &= \sum_{i=1}^{N/2} \big(y_{2i-1} \cdot z_{2i-1,j} + y_{2i} \cdot z_{2i,j} + \mathsf{i} \cdot \mathsf{something}\big) \; = \; \big(\sum_{i=1}^{N} y_i \cdot z_{i,j}\big) + \mathsf{i} \cdot \mathsf{something}'. \end{aligned}$$

We complete the gradient computation just by extracting the real part, $\boldsymbol{g} := \mathsf{re}(\boldsymbol{g'})$. This new gradient calculation performs half as many multiplications in the inner-product steps (3 and 5), the same number of operations in the sigmoid step 4, and a few more operations to split and recombine complex vectors from real and imaginary parts. Since the inner product operations are by far the most expensive parts of each Nesterov computation, this saves nearly half of the overall number of multiplications. However, in our tests it only saved about 20% of the running time. (We think that this discrepancy is partially because we worked harder on optimized the standard procedure than the complex packed one.)

4.3 Implementing CKKS in HElib

The CKKS scheme from [7] is a Regev-type cryptosystem, with a decryption invariant of the form $[\langle \mathsf{sk}, \mathsf{ct} \rangle]_q = \tilde{\mathsf{pt}}$, where sk, ct are the secret-key and ciphertext vectors, respectively, $[\cdot]_q$ denotes reduction modulo q into the interval $[-q/2, q/2]$, and $\tilde{\mathsf{pt}}$ is an element that encodes the plaintext and includes also some noise.

The CKKS scheme is similar in many ways to the BGV scheme from [4]: both schemes use an element $\tilde{\mathsf{pt}}$ of low norm, $|\tilde{\mathsf{pt}}| \ll q$, and the homomorphic operations are implemented almost exactly the same in both. The difference between these schemes lies in the way they interpret the element $\tilde{\mathsf{pt}}$, i.e., how it is decoded into plaintext pt and noise e: We tend to think in the BGV of the low-order bits of $\tilde{\mathsf{pt}}$ as pt and the high-order bits as e, and in CKKS it is the other way around. Specifically, the BGV decodes $\tilde{\mathsf{pt}} = \mathsf{pt} + p \cdot \mathsf{e}$, where p is the plaintext space modulus and $|\mathsf{pt}| < p$, whereas CKKS decodes $\tilde{\mathsf{pt}} = \mathsf{e} + \Delta \cdot \mathsf{pt}$ where Δ is some scaling factor and (hopefully) $|\mathsf{e}| < \Delta$.

This difference in interpretation of $\tilde{\mathsf{pt}}$ implies very different plaintext algebras for the two schemes: While BGV deals with integral plaintext elements modulo p, in CKKS the plaintext elements are complex numbers with limited precision. Some other (rather small) differences between the homomorphic operations in BGV and CKKS are related to the way the scaling factor Δ is handled:

- The plaintext modulus p in BGV typically does not change throughout the computation, but the scaling factor Δ in CKKS does vary: Specifically, Δ is squared on multiplication and is scaled via modulus switching.
- In both CKKS and BGV, ciphertexts can only be added when they are defined relative to the same modulus q. However, it is also important for CKKS addition that they have the same scaling factor Δ.

Our CKKS implementation in HElib relies on the same chassis as the BGV cryptosystem that supports the required homomorphic operations and handles any cyclotomic field.[8] Differently from the way it is described in [7], the HElib implementation does not rely on the application to use explicit scaling, instead the library can automatically scale all the ciphertexts as needed. Each ciphertext in our implementation is tagged with both a noise estimate and the scaling factor Δ and the library uses these tags to decide how and when to scale these ciphertexts using modulus-switching. These scaling decisions balance the need to scale the ciphertexts down before multiplication to keep the noise small with the need to keep the scaling factor Δ sufficiently larger than the noise element e.

The cryptosystem is initialized with an accuracy parameter r that from the application perspective roughly means the additive noise terms in the various operations is bounded by 2^{-r} in magnitude. The library tries to ensure that operations with added noise term η will only be applied to ciphertexts with scaling factors $\Delta \geq \eta \cdot 2^r$. Note, that this logic only "does the right thing" when the complex values throughout the computation are close to one in magnitude. For smaller values, the requested accuracy bound will typically not be enough, while for larger values the implementation will spend too much resources trying to keep the precision way too high. The logic works quite well for the LR procedure (section 3) where indeed all the encrypted quantities are kept at size $\Theta(1)$.

4.4 Performance of the Homomorphic Procedure

We tested the running time and memory consumption in a few different settings, depending on the number of available threads, and the number of bands in the matrix C. (As we explained in section 4.1, using more bands is useful when the machine has limited RAM and cannot fit all the encrypted input ciphertexts in memory at once.) We also tested the complex packing optimization from section 4.2 vs. the "standard" way of packing only real numbers in the slots.

These tests were run on a machine with Intel E5-2640 CPU running at 2.5GHz, with 2×12 cores, 64 GB memory (split 32GB for each chip in a NUMA configuration), and 15MB cache. The software configurations (on Ubuntu

[8] Our logistic regression procedure uses a power of two cyclotomic field for efficiency.

16.04.5) included HElib commit `dbaa108b66c5` from Sep 2018, NTL version 11.3.2, GMP version 6.1.2, and Armadillo version 9.200.7. All compiled with gcc 8.1.0 including our LR code.

Parameters. The parameters were chosen so as to get at least 128 security level while having enough levels to complete seven iterations (followed by computing the log likelihood of the resulting model). Specifically, the largest modulus in the chain had $|q| = 900$ bits, and the scheme was instantiated over the m'th cyclotomic field with $m = 2^{17} = 131072$ (so the dimension of the relevant lattice was $\phi(m) = 65536$). This setting gave us estimated security level of 142 bits. These parameters give us $\phi(m)/2 = 32768$ slots in which to pack data, so we could compute up to 32768 LR models in parallel.

The results that we describe below were measured on the iDASH 2018 dataset, where each model has three clinical variables and a single SNP. This dataset had only 10643 SNPs, so we only packed that many numbers in the slots, but the performance numbers are not affected by the number of "empty slots," we would have identical results even if all 32767 slots were filled.

On the other hand, the number of records in the training set does influence the running time (as well as the memory consumption). Here we used the fact that small LR models can be computed accurately by sub-sampling the data. The common "one in ten" rule of thumb states that a model with k features requires at least $10k$ records with 0 and $10k$ records with 1. Since in these tests we had four features in each model (three clinical variables and one SNP), and since we sub-sampled the data to get 50% 0's and 50% 1's, then we needed at least 80 total record, and we run all our tests on 100 records in the training test.

Without the complex-packing optimization, each iteration of the Nesterov procedure took four levels in the modulus chain. This is a little surprising, as each iteration includes a degree-7 polynomial sandwiched between two vector-matrix multiplications so we expect it to take five levels rather than four. The reason is that we used 44 bits "wide" levels, and the noise management logic of HElib performed two consecutive operations at the same level. This indicates some waste in the HElib noise management. With complex packing, we could only perform six iterations with the same parameters as each iteration of the Nesterov procedure used an average six levels.

Results. The results are described in Tables 1 and 2. The optimization of using complex packing cuts the input-reading time in half (as there are half as many ciphertexts), but only reduces the running time by about 20% (for the same number of iterations). There is approximately a linear speedup when the number of threads is increased from one to twelve, but not more due to cache contention on the testing server architecture. The memory requirements grow slowly with the number of threads, twelve threads consumed $1.5\times$ to $2\times$ more memory than a single threads.

Parallelization vs. run-time, seven iterations

# threads	read input time	training time	RAM consumption
1	435 sec	8847 sec	24GB
2	220 sec	4190 sec	26GB
6	78 sec	1673 sec	28GB
12	44 sec	1202 sec	30GB
24	44 sec	1128 sec	33GB

Table 1. CPU time and RAM consumption of the "standard packing" method with seven iterations and a single band, vs. number of threads

Standard vs. complex packing, six iterations

packing	# threads	read input time	training time	RAM consumption
standard	1	464 sec	7620 sec	24GB
	2	223 sec	3677 sec	26GB
	6	79 sec	1449 sec	28GB
	12	44 sec	1128 sec	30GB
	24	40 sec	1016 sec	33GB
complex	1	223 sec	5960 sec	13GB
	2	111 sec	2998 sec	14GB
	6	42 sec	1242 sec	16GB
	12	25 sec	859 sec	18GB
	24	23 sec	818 sec	24GB

Table 2. CPU time and RAM consumption with six iterations and a single band, both complex and standard packing, vs. number of threads

5 Conclusions

In this work, we demonstrated that the CKKS cryptosystem [7] can be used to implement homomorphic training of a very large number of logistic regression models simultaneously in a reasonable amount of time.

For that purpose, we adopted the iterative method used by Kim et al. [14] based on Nesterov's accelerated gradient descent. Our implementation can train simultaneously over 30,000 small models, each with four variables, in about 20 minutes. We estimate that the same number of models with 8-10 variables can be trained in about an hour. We also provided extensive evaluation of this iterative procedure, testing it on a number of different datasets and comparing its predictive power with a few alternatives. Our tests show that this method is competitive.

References

1. C. Bonte and F. Vercauteren. Privacy-preserving logistic regression training. *BMC Medical Genomics*, 11((Suppl 4)), 2018. `https://doi.org/10.1186/s12920-018-0398-y`.
2. G. Bontempi, A. D. Pozzolo, O. Caelen, and R. A. Johnson. Credit Card Fraud Detection. Technical report, Université Libre de Bruxelles, 2015.
3. A. Bootwala. Titanic for Binary logistic regression. `https://www.kaggle.com/azeembootwala/titanic/home`.

4. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.
5. S. Bubeck. ORF523: Nesterov's accelerated gradient descent. `https://blogs.princeton.edu/imabandit/2013/04/01/acceleratedgradientdescent`, accessed January 2019, 2013.
6. H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter. Logistic regression over encrypted data from fully homomorphic encryption. *BMC Medical Genomics*, 11((Suppl 4)), 2018. `https://doi.org/10.1186/s12920-018-0397-z`.
7. 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.
8. J. L. H. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. Doing real work with FHE: the case of logistic regression. In M. Brenner and K. Rohloff, editors, *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018*, pages 1–12. ACM, 2018. `https://eprint.iacr.org/2018/202`.
9. 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.
10. S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. `https://github.com/shaih/HElib/`, Accessed January 2019.
11. K. Han, S. Hong, J. H. Cheon, and D. Park. Efficient logistic regression on large encrypted data. Cryptology ePrint Archive, Report 2018/662, 2018. `https://eprint.iacr.org/2018/662`.
12. Integrating Data for Analysis, Anonymization and SHaring (iDASH). `https://idash.ucsd.edu/`.
13. R. L. Kennedy, H. S. Fraser, L. N. McStay, and R. F. Harrison. Early diagnosis of acute myocardial infarction using clinical and electrocardiographic data at presentation: derivation and evaluation of logistic regression models. *European Heart Journal*, 17(8):1181–1191, Aug. 1996. Data obtained from `https://github.com/kimandrik/IDASH2017/tree/master/IDASH2017/data/edin.txt`.
14. A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(4):83, Oct 2018.
15. M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Med. Inform.*, 6(2):e19, 2018. DOI: 10.2196/medinform.8805. Also available from `https://eprint.iacr.org/2018/074`.
16. Y. Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87 of *Applied Optimization*. Springer US, 2004. `https://doi.org/10.1007/978-1-4419-8853-9`.
17. A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. Calibrating Probability with Undersampling for Unbalanced Classification. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 159–166, Dec. 2015.
18. K. Sikorska, E. Lesaffre, P. J. Groenen, and P. H. Eilers. GWAS on your notebook: fast semi-parallel linear and logistic regression for genome-wide association studies. *BMC Bioinformatics*, 14:166, 2013.
19. Logistic regression. `https://en.wikipedia.org/wiki/Logistic_regression#Discussion`. Accessed January 2017.

A Corrections in the Literature

During our work we encountered two minor bugs/inconsistencies in the literature. We have notified the relevant authors and document these issues here:

- The Matlab code used in the iDASH **2017** competition had a bug in the way it computed the recall values, computing it as $\frac{false\ positive+true\ positive}{false\ negative+true\ positive}$ instead of $\frac{true\ positive}{false\ negative+true\ positive}$.
- Some of the mean-squared-error (MSE) results reported in [14] seem inconsistent with their accuracy values: For the Edinburgh dataset, they report accuracy value of 86%, but MSE of only 0.00075. We note that 86% accuracy implies MSE of at least $0.14 \cdot (0.5)^2 = 0.035$ (likely a typo).

B The Datasets that We Used

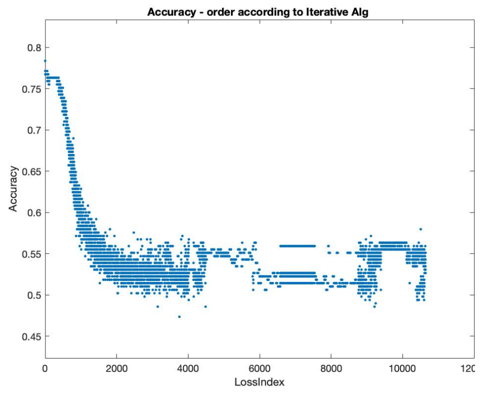
Recall that we tested the iterative procedure against a few different datasets, to ensure that it is not "tailored" too much to the characteristics of just one type of data. We had some difficulties finding public datasets that we could use for this evaluation, eventually we converged on the following four:

- The iDASH 2018 dataset, as provided by the organizers of the competition, is meant to correlate various genetic markers with the risk of developing cancer. It consists of 245 records, each with a binary condition (cancer or not), three covariates (age, weight, and height), and 10643 markers (SNPs). The last 120 records were missing the covariates, so we ran our procedure by replacing each missing covariate by the average of the same covariate in the other records.
- A credit card dataset [2] attempts to correlate credit-card fraud with observed characteristics of the transaction. This dataset has 984 records each with thirty columns.
- The Edinburgh dataset [13] correlates the condition of Myocardial Infarction (heart attack) in patients who presented to the emergency room in the Edinburgh Royal Infirmary in Scotland with various symptoms and test results (e.g., ST elevation, New Q waves, Hypoperfusion, depression, vomiting, etc.). The same dataset was also used to evaluate the procedure of Kim et al. [14]. The data includes 1253 records, each with nine features.
- The Titanic dataset [3], consisting of 892 records with sixteen features, correlating passenger's survival in that disaster with various characteristics such as gender, age, fare, etc.

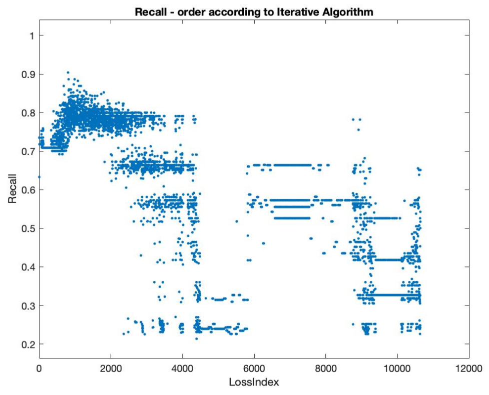
The first dataset comes with a distinction between SNPs and clinical variables, but the other three have just the condition variable and all the rest. We had to decide which of the features (if any) to use for covariates. We note that whatever feature we designate as covariate will be present in all the models, so choosing a feature with very high signal will make the predictive power of all the models very similar. We therefore typically opted to choose for a covariate the features which is *least correlated* with the condition. We also ran the same test with no covariates, and the results were very similar.

C Model Evaluation Figures

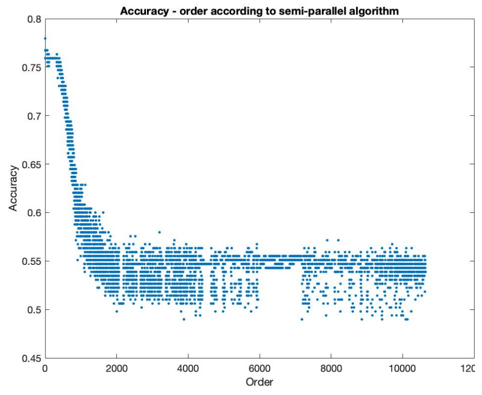
iDASH data column ordering: iterative vs. semi-parallel



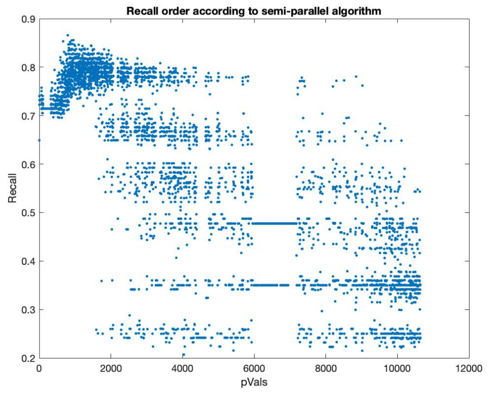
(a) Accuracy, iterative Order



(b) Recall, iterative Order



(c) Accuracy, semi-parallel order



(d) Recall, semi-parallel order

Fig. 2. Accuracy/recall of the Matlab LR models for the iDASH 2018 dataset ordered according to the p-value order of the iterative procedure (top) or the semi-parallel algorithm (bottom).