Notes for Secure Linear and Logistic Regression

Rob Hall

August 3, 2009

1 Secure Multi-Party Computation

We assume the presence of $P \geq 2$ parties who are wish to compute linear regression on the total of their data. Hence we consider the "functionality" (see [**?**]) which maps the data of each party into the regression parameter vector β.

$$\{(X_1, y_1), (X_2, y_2), \cdots (X_P, y_P)\} \rightarrow \{\beta, \beta, \cdots \beta\} \tag{1}$$

The left side of the arrow is a pair for each party, with $X_i \in \mathbb{R}^{n \times d}$ and $y_i \in \mathbb{R}^n$. The right hand side represents P copies of the parameter, so that each party receives the same output. In this work we consider a setting where each party has an "additive share" of the dataset. That is, $\sum_i X_i = X$ and $\sum_i y_i = y$ where X and y correspond to the design matrix and response vector of the combined data on which the linear regression is performed. This subsumes all the partitioning schemes for the database (e.g., vertical and horizontal partitioning which are the cases considered in [**?**]) as in these cases for each element, one party holds the value and the remaining parties hold zero. Furthermore this setup is applicable in a case where parties may have overlapping data, and the regression is to be learned by using a linear function of the overlapping data (e.g., a weighted average) as a kind of measurement error model.

In this work we build up a protocol for computing (1) which is secure in the presence of "semi-honest" parties. That is, parties who obey the protocol (and do not try to e.g., inject malformed data) but keep a transcript of all the messages they receive. Intuitively, a protocol is secure in this setting whenever the intermediate messages give no information about the secret inputs of other parties.

Formally, a protocol is secure so long as there exists (for each party) a polynomial time algorithm which, given the input and output of that party, outputs a random transcript of message which are *computationally indistinguishable* from the messages received by that party during the computation. The definition of computational indistinguishability is discussed in [**?**]. In essence, if the distribution of the sequence of messages depends only on the private input and output of each party then we may simulate messages by drawing from this distribution (so long as the random number generator returns samples which are computationally indistinguishable from draws from the distribution). The existence of a simulator shows that intermediate messages do not depend on the private input of other parties, and so the protocol is secure in the sense that parties gain no more information about each other's private inputs than that revealed by the output of the protocol. Note that this type of security is "orthogonal" to that studied in [**?**], which seeks to prevent leakage of secret information in the parameter vector.

An example of a protocol which does not achieve this definition of security is one where all parties send their data to party 1, who computes the parameter locally on the combined data and then sends it back to all other parties. In this case the messages received by party 1 consist of the data of other parties, in general it is impossible to simulate these messages given only the input and output belonging to party 1. In the next sections we give a protocol for performing Newton's method on the logistic regression objective in a way which is secure in the presence of semi-honest parties.

2 Additive Sharing

In our construction we make extensive use of additive secret sharing. The idea is to divide a quantity of interest a into P random numbers a_i (called "shares"), where the i^{th} party holds a_i, and where $\sum_i a_i = a$. If the a_i are distributed uniformly in the field (e.g., $a_i \in \mathbb{Z}_m = \{0, 1, \cdots, m-1\}$) then any subset of the a_i will reveal nothing about a (in fact the sum over any subset is still uniform in this field). We use this construction to keep all intermediate quantities secret during evaluation of the covariance matrix and its inverse. Although the joint distribution of the a_i concentrates on the linear subspace corresponding to the secret value, marginally the shares are uniform and depend on no parameters. Hence we may easily simulate messages based on these shares since the marginal distributions will be known. We next show how to compute additive shares of all the intermediate quantities, using the abstract definition of additive shares. Our intermediate values will be real numbers and not integers, so in section 3 we show how to approximate (arbitrarily well) the computations in modular arithmetic on $\mathbb{Z}_m$ for some large m.

2.1 Linear Algebra on Random Shares

For performing Newton's method it is essential to perform linear algebraic operations on random shares, for example computing shares of the Newton step from shares of the gradient and inverse Hessian. In this section we describe how we may compute sums and products of quantities represented as random shares. Using these constructions we may compute inner and outer products of vectors of random shares, and hence also matrix multiplies etc.

Computing shares of the sum of two secret quantities a and b involves only local computation, a party may add his shares a_i and b_i together to get a random share of the quantity $a+b$. Obtaining random shares of the product of two secret quantities is more involved:

$$ab = \sum_i a_i \sum_j b_j = \sum_i a_i b_i + \sum_{j \neq i} a_i b_j \tag{2}$$

The elements of the first sum on the right side may be computed locally by each party, however the second sum requires products between random shares held by different parties. For terms in the second sum, we use a sub-protocol for computing random shares of the products $a_i b_j$, so that these random shares may be added locally. This protocol is described below. If for each product $a_i b_j$ we obtain a random share $c_{i,j} + d_{j,i} = a_i b_j$ (for party i and j respectively) then the second sum of (2) may be written:

$$\sum_{j \neq i} a_i b_j = \sum_{j \neq i} c_{i,j} + d_{j,i} = \sum_{j \neq i} c_{i,j} + d_{i,j} = \sum_i \sum_j c_{i,j} + d_{i,j} \tag{3}$$

So each party may compute a share of the last sum from his shares of the individual products.

2.2 Computing Shares of a Product

We use a protocol for computing random shares of each product $a_i b_j$ which depends upon homomorphic encryption. We use the encryption system due to Paillier, which is summarized below:

Security Parameter The cryptosystem has a parameter which we call k which determines the difficulty of breaking the encryption. The parameter here corresponds to the length of the public key in bits, and a reccomended value is at least 1024.

Setup Party 1 generates his private key $n = pq$ and $\lambda = \mathrm{lcm}(p-1, q-1)$. He generates a random parameter g which satisfies $\gcd(L(g^\lambda \bmod n^2), n) = 1$ where $L(x) = n^{-1}(x-1)$. The public key is the pair (n, g), this is sent to party 2.

Encryption The encryption of a number a using the public key (n, g) is $\mathrm{Enc}_{(n,g)}(a) = g^a r^n \bmod n^2$, where r is drawn uniformly at random from $\mathbb{Z}_n$.

Decryption The decryption of a ciphertext c is $\frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$.

Addition Two ciphertexts may be multiplied together to obtain the encryption of the sum, as $\mathrm{Enc}_{(n,g)}(a) \cdot \mathrm{Enc}_{(n,g)}(b) \bmod n^2 = \mathrm{Enc}_{(n,g)}(a + b \bmod n)$. Furthermore note $\mathrm{Enc}_{(n,g)}(a) \cdot g^b \bmod n^2 = \mathrm{Enc}_{(n,g)}(a + b \bmod n)$ and so a plaintext may also be added to an encrypted value.

Multiplication A ciphertext may be taken to the power of a plaintext, yielding an encryption of the products: $\mathrm{Enc}_{(n,g)}(a)^b \bmod n^2 = \mathrm{Enc}_{(n,g)}(a \cdot b \bmod n)$.

Semantic Security Paillier's cryptosystem is "semantically secure" which means that the distribution of encryptions of two different values are computationally indistinguishable. This implies that ciphertexts may be simulated based on only the public key.

We may then compute random shares of the product $a_i b_j$ without exposing one party's private variable to the other. A protocol which achieves this is shown in figure 1.

- **Input** Party 1 has $a_i \in \mathbb{Z}_m$, party 2 has $b_j \in \mathbb{Z}_m$ for $m^2 < 2^k$.
- **Setup** Party 1 generates the key and shares the public key with party 2.
- **Step 1** Party 1 computes $\text{Enc}(a_i)$ and sends it to party 2.
- **Step 2** Party 2 draws s_1 uniformly at random from $\{m^2, \cdots, n\}$ and computes $\text{Enc}(a_i)^b_j \cdot g^{n-s_1} \bmod n^2 = \text{Enc}(a_i b_j + n - s_i \bmod n) = \text{Enc}(a_i b_j - s_i \bmod n)$. He sends this value to party 1.
- **Step 3** Party 1 decrypts the message to obtain his share $s_2 = a_i b_j - s_i \bmod n$.
- **Output** Party 1 computes $o_1 = s_1 - n \bmod l$ and party 2 computes $o_2 = s_2 \bmod l$ and $o_1 + o_2 \bmod l = a_i b_j \bmod l$.

Figure 1: A protocol for computing shares of a product mod l.

To show that this protocol is correct, note that at step 3, we have that $s_1 + s_2 \equiv a_i b_j \mod n$. Since $s_i \in \mathbb{Z}_n$ we have that $0 \leq s_1 + s_2 \leq 2n$, and so either $s_1 + s_2 = a_i b_j$ or $s_1 + s_2 = a_i b_j + n$. Since in step 2 we constructed s_1 to be greater than m^2, which is greater than $a_i b_j$, we see that the latter condition holds. Therefore the sum of outputs mod l is:

$$o_1 + o_2 \bmod l = s_1 + s_2 - n \bmod l = a_i b_j + n - n \bmod l = a_i b_j \bmod l \qquad (4)$$

and so the protocol computes random shares of $a_i b_j \bmod l$.

To show that the protocol is secure in the face of semi-honest parties, we construct a simulator for the view of each party during execution. for party 2, the only mesage received is the encryption of party 1's value in step 1. Since Paillier's cryptosystem is semantically secure (cite) we may simulate encrypted values by encrypting a random value, using the public key which is known to party 2. For the view of party one, we consider the decryption of the value received in step 3. This value is not uniform on $\mathbb{Z}_n$ since the noise added was in the interval $\{m^2, \cdots, n\}$, so the value s_2 is uniform on a subset of $\mathbb{Z}_n$ which depends on the variable. Nevertheless, we contend that since the subset consists of the overwhelming majority of $\mathbb{Z}_n$, the distribution is computationally indistinguishable from a uniform distribution. Denoting the distribution of s_2 as P_n^m, and the uniform distribution on $\mathbb{Z}_n$ by U_n, consider the variation distance:

$$\frac{1}{2}\sum_{x=0}^{n} |P_n^m(x) - U_n(x)| = \frac{m^2}{n} \leq \frac{m^2}{2^k}$$

We have that the variation distance between the two distributions is bounded above by a negligible function of the security parameter k, which implies that the two distributions are computationally indistinguishable (see oded book). Therefore we may simulate values from U_n for the message received by party 1 (technically it should also be encrypted using his public key). Note that the security parameter is already prescribed to be greater than 1024. Later, we will suggest using $m = 2^{128}$, and so we have that the variation distance will be 2^{-768}. The protocol would have to be run an immense number of times (with the same input values) for party 2 to gain significant information.

3 A Fixed-Point Arithmetic Scheme

So far we have a complete system for securely computing functions on the ring $\mathbb{Z}_m$, where sums are computed locally, and products are computed using the protocol of figure 1 and setting $l = m$. We next show how functions on $\mathbb{R}$ may be approximated under certain conditions.

We chose a parameter $p < m$ and then use the mapping $f : \mathbb{Z}_m \to F \subset \mathbb{R}$ defined by:

$$f(a) = \frac{1}{p}\begin{cases} a & a \leq \frac{m}{2} \\ a - m & a > \frac{m}{2} \end{cases} \tag{5}$$

We also have a function from the image of f back to $\mathbb{Z}_m$ defined by $g(a) = pa \bmod m$, which we may use to define addition and multiplication in F in terms of the respective operations in $\mathbb{Z}_m$:

$$g(a) + g(b) \bmod m = pa + pb \bmod m = g(a + b) \tag{6}$$

$$g(a) \cdot g(b) \bmod m = p^2 ab \bmod m = p \cdot g(ab) \tag{7}$$

Since the multiplication introduces an extra factor of p, we propose to perform multiplication in mod pm and then divide by p:

$$\frac{g(a) \cdot g(b) \bmod pm}{p} = \frac{p^2 ab \bmod pm}{p} = pab \bmod m = g(ab) \tag{8}$$

Therefore we may use the constructions of the previous section to compute in the image F. Since we deal with real input values, we project them to F by chosing the element which is closest, which introduces error less than p^{-1}. Furthermore, when multiplying using the construction above, we implicitly projected the result back into F, which implies a loss of precision of no more than p^{-1}. Note that when the values used are large, the wrapping effect of the modulo can introduce catastrophic errors into the computations (essentialy overflow and underflow), so we must be sure that the quantities we compute stay away from the boundary.

When computing with random shares (for two parties), we have $a_1 + a_2 = a + \hat{a}m$ where $\hat{a}$ takes on the value 1 if $a_1 + a_2 \geq m$ or 0 otherwise. Multiplying shares under the fixed point arithmetic scheme gives:

$$\begin{aligned}
\frac{(a_1+a_2)(b_1+b_2) \bmod pm}{p} &= \frac{(a+\hat{a}m)(b+\hat{b}m) \bmod pm}{p} \\
&= \frac{p^2ab + ma\hat{b} + mb\hat{a} + m^2\hat{a}\hat{b} \bmod pm}{p} \\
&= pab \bmod m + \frac{m}{p}(a\hat{b} + b\hat{a} + m\hat{a}\hat{b}) \\
&= g(ab) + \text{err}
\end{aligned}$$

Therefore if either $\hat{a}$ or $\hat{b}$ is 1, then the above procedure yields incorrect results (note this was not a problem for computing on integers since $p = 1$ means the error term is a multiple of m which "goes away" in the modulus). However we may ammend the procedure by having each party keep track of a share of $\hat{a}$ so that $\hat{a}_1 + \hat{a}_2 = \hat{a}$ where $+$ means exclusive or. That way, we may remove the error term from the result by subtraction. We may then break the product into random shares, and also obtain random shares of the binary flag for the product.

After exectuting the protocol of figure 2 each party has a "xor share" of the binary flag, and an additive share of p^2ab. Dividing the latter shares by p yields the correct product.

4 Secure Matrix Inverse

We use a matrix inversion routine which is entirely built up of matrix multiplications and subtractions, so that we may use the constructions of the preceding sections to implement it securely. We obtain the reciprocal of a number a without necessitating any actual division by an application of Newton's method to the function $f(x) = x^{-1} - a$. Iterations follow $x_{s+1} = x_s(2 - ax_s)$, which requires multiplication and subtraction only.

It turns out that the same scheme may be applied to matrix inversion, e.g., see [**?**] and references therein. A numerically stable, coupled iteration for computing A^{-1}, takes the form:

$$\begin{array}{lclclcl}
X_{s+1} &=& 2X_s - X_sM_s &,& X_0 &=& c^{-1}I\ , \\
M_{s+1} &=& 2M_s - M_s^2 &,& M_0 &=& c^{-1}A,
\end{array} \tag{9}$$

where $M_s = X_sA$, and c is to be chosen by the user. A possible choice, leading to a quadratic convergence of $X_s \to A^{-1}$ ($M_s \to I$), is $c = \max_i \lambda_i(A)$. In our actual implementation we instead consider the trace (which dominates the largest eigenvalue, as the matrix in question is positive definite), since we can compute shares of the trace from shares of the matrix locally by each party. To compute c^{-1} we use the same iteration, with scalars instead of matrices. For this iteration we initialize with an arbitrarily small $\epsilon > 0$ (as convergence depends on the magnitude of the initial value being lower than that of the inverse we compute). We use the constructions of section **??** to iterate through (9) until convergence. As $M_s \to I$, we check for convergence by considering the absolute difference between the trace of M_s and the data dimension d, and we may evaluate the function $1\{|\mathrm{tr}(M_s) - d| > \epsilon\}$ on random shares of the trace of M_s using the same form as (**??**).

5 Securely deciding $\geq$ for "2s Compliment" Integers

Evaluating $1_{\{a \geq b\}}$ securely is another sub-protocol which is required, both for the approximation to the logistic sigmoid, and for checking convergence in newtons method (both for likelihood optimization and for matrix inversion). In this section we will describe a method which works for integers in a ring $\mathbb{Z}_m$ for $m = 2^k$, and note that it will work equally well for both integers and fixed point reals as described above.

First note that $a \geq b$ implies that $a - b \geq 0$, hence it suffices to determine the sign of the difference. Assuming one party starts with a and the other with b, we see that we may decide the predicate based on the most significant bit of the difference $a - b$, along with the bit indicating whether there is a "carry out" of the top of the sum. The exclusive-or of the two bits yields the predicate. Therefore the problem is easier than the problem of securely computing random shares of the carry bit and most significant bit of the difference. Replacing b with $-b$ this is the same problem as deciding those bits of the sum, for which we give a simple protocol. The main idea is to simulate a ripple-carry adder (a simple circuit composed of "and" and "xor" gates) using Yao's protocol. This way, each party gets a share of the bits of the sum, so that when "xor-ed" together they reveal the sum.

The main component of a ripple carry adder is a smaller circuit called a full adder. This circuit computes the sum and carry flag for the sum of two bits, ad a "carry in" flag which indicates a carry coming from a lower bit position. We have that the sum bit $s = a + b + c_{\mathrm{in}}$ where $+$ is xor, and the carry out bit $c_{\mathrm{out}} = 2^{-1}(a + b + c_{\mathrm{in}} - s)$ where $+$ means the usual addition. The protocol below gives "xor shares" of the sum and carry out bit, from the input bits along with "xor shares" of the carry-in bit (which is initialized to zero for the first bit position).

- **Input** Party 1 has $a_1, \hat{a}_1, b_1, \hat{b}_1$ likewise for party 2.
- **Setup** Party 1 generates the key and shares the public key with party 2.
- **Step 1** Party 1 computes $\text{Enc}(a_1)$, $\text{Enc}(b_1)$, $\text{Enc}(a_1 b_1)$, $\text{Enc}(a_1 \hat{b}_1)$, $\text{Enc}(b_1 \hat{a}_1)$, $\text{Enc}(\hat{a}_1 \hat{b}_1)$ and sends these to party 2.
- **Step 2** Party 2 may then compute $\text{Enc}(p^2[a_1 b_1 + a_1 b_2 + a_2 b_1 + a_2 b_2])$, $\text{Enc}(p(a_1 + a_2)(\hat{b}_1 + \widehat{b_2} - 2\hat{b}_1 \hat{b}_2))$, $\text{Enc}(p(b_1 + b_2)(\hat{a}_1 + \widehat{a_2} - 2\hat{a}_1 \hat{a}_2))$ and $\text{Enc}(m^2(\hat{a}_1 + \hat{a}_2 - 2\hat{a}_1 \hat{a}_2)(\hat{b}_1 + \hat{b}_2 - 2\hat{b}_1 \hat{b}_2))$. From this he may compute $\text{Enc}(p^2 ab)$, by subtracting the last 3 messages from the first using the homomorphic properties of the cryptosystem.
- **Step 3** Party 2 draws s_1 uniformly at random from $\{m^2, \cdots, n\}$ and computes $\text{Enc}(p^2 ab - s_1)$ as in 1. He sends this value to party 1.
- **Step 4** Party 1 decrypts the message to obtain his share $s_2 = p^2 ab - s_1 \bmod mp$. He then re-encrypts his share and sends $\text{Enc}(s_2)$ back to party 2.
- **Step 5** Party 2 computes $\text{Enc}(s_1 + s_2) = \text{Enc}(p^2 ab + \hat{s} m)$ where $\hat{s}$ is the unknown binary flag indicating whether the shares sum to more than m or not. Subtracting from the previous encryption of the product from step 2, we have $\text{Enc}(\hat{s} m)$. If m is a power of two, then it has a multiplicative inverse in $\mathbb{Z}_n$ since n is odd, so we may compute $\text{Enc}(\hat{s})$. Drawing $\tilde{s}_1$ uniformly on $\{2, 3 \cdots n-1\}$, he computes $\text{Enc}(\hat{s} + \tilde{s}_1)$ and sends this to party 1. Then we have $\hat{s}_1$ is the remainder of $\tilde{s}_1$ when dividing by two.
- **Step 6** Party 1 decrypts and obtains $\hat{s}_2$ as the remainder of the value when dividing by two.

Figure 2: A protocol for computing shares of a product with the fixed point arithmetic scheme.

- **Input** Party 1 has a, c_1 likewise party 2 holds b, c_2.
- **Step 1** Each party computes his share of the "sum bit" as $a+c_1$ and $b+c_2$ where $+$ means xor. Note that then $(a+c_1)+(b+c_2) = a+b+(c_1+c_2) = a+b+c$ as required.
- **Step 2** Using the cryptography, the parties compute xor-shares of $f = c_1 b$ using **??** for $l = 2$.
- **Step 3** Using the cryptography, the parties compute xor-shares of $g = a(c2+b)$ where $+$ is xor using **??** for $l = 2$.
- **Step 4** The parties compute respectively: $d_1 = c_1 a + f_1 + g_1$, $d_2 = c_2 b + f_1 + g_1$.
- **Output** The bits d_i are xor shares of the carry out bit, and the bits computed in step 1 are xor shares of the sum bit.

Figure 3: A secure implementation of a full adder.