Implementing ML Algorithms with HE

Yilun Du yilundu@mit.edu
Laura Gustafson lgustaf@mit.edu
David Huang huangd@mit.edu
Kelly Peterson kellypet@mit.edu

Abstract

An increase in cloud-based computing leads to an increased worry in the security of user data. Typically, data is sent to a third-party server which performs analytics or machine learning on the data. However, in most of these scenarios, the data involved is sensitive and should remain private. Homomorphic encryption, a form of encryption that allows functions to be performed on encrypted ciphertext, allows privacy-preserving data analysis of existing large private, sensitive data sets. We implement and analyze the performance of linear regression and K-means clustering using the homomorphic encryption library SEAL and provide an extension of the SEAL library to matrix operations.

1. Introduction

Cloud-based machine learning platforms provide scalable machine learning resources with reduced hardware cost and maintenance overhead for end users. Similarly, cloud computing continues to see increasing adoption, as it offers many benefits including a minimal need for interactions between data providers and the cloud, and the removal of most of the computational power required by individual analysts. However, using third-party cloud servers for computation raises privacy concerns, especially when sensitive data such as medical records, financial, or federal data is involved. Thus, it is imperative that cloud computing does not violate the privacy of sensitive data. Today, there exist solutions to this problem in the form of off-the-shelf tools for Secure Multiparty Computation (SMC) protocols. Unfortunately, many of these tools are costly and inefficient to deploy in real-world cloud computing systems and hinder the development of privacy-preserving applications in the cloud.

One method of preserving the privacy and security of data when outsourcing computation is to encrypt the data before uploading it to the cloud. However, the utility of the data becomes severely limited if encryption prevents useful computations from being performed. Fortunately, this challenge can be solved by using a homomorphic encryption scheme, which allows one to perform operations on encrypted data without decrypting it. Homomorphic encryption was originally proposed by Rivest et al. (1978) as a way to encrypt data such that certain operations can be performed on it without decrypting it first. Moreover, recent advancements in cryptography have led to the development of fully homomorphic encryption (FHE) schemes, which support the evaluation of arbitrary functions on encrypted data. The first FHE scheme was presented in 2009 by Craig Gentry; this scheme was the first to support an arbitrary number of additions and multiplications on encrypted data. However, such FHE schemes rely on a bootstrapping procedure in order to maintain usable levels of noise, which is extremely computationally expensive.

Since the first FHE scheme was constructed by Gentry in 2009, subsequent schemes [10, 8] have become increasingly practical, with improved performance and parameters. Then, in 2011, Gentry and Shai Halevi presented the first working FHE implementation. Following this, several other HE schemes have been implemented, including the Fan-Vercauteren (FV) scheme implemented in Microsofts *Simple Encrypted Arithmetic Library (SEAL)*[9].

Although there exist several working HE implementations today, most are impractical or infeasible for use on large datasets. Indeed, very few implementations have been proven efficient for large-scale applications. In addition, there exist even fewer machine learning algorithms implemented using HE today. Thus, in this project, we wanted to explore implementing a working machine learning algorithm using HE.

This work proposes a protocol for performing linear regression and k-means clustering over a dataset that is distributed over multiple parties.

Our goal was to build something to demonstrate some of the many capabilities made possible by using homomorphic encryption. We also wanted to provide building blocks that others can use to implement additional FHE machine learning algorithms. In addition, we hoped to not only build a proof-of-concept demonstrating the capabilities of modern-day FHE software libraries, but hoped to also provide a working, open-source prototype which could be immediately deployed and tested by researchers, organizations, and other interested parties. We hope that the result of our implementation lowers the barrier to entry for

others to begin using and developing FHE systems, making FHE-based ML algorithms more accessible to novice engineers, researchers, and others in the cryptography and machine learning communities.

Moreover, this project aimed to evaluate the current level of maturity of FHE Machine Learning implementations and the *SEAL* cryptographic library. Conducting this evaluation not only highlights the great potential of *SEAL* and FHE-based ML algorithms but also underscores a need for the greater community to continue researching, developing, and extending the *SEAL* library and FHE applications.

Based on the work of Nikolaenko, et. al., [12], we were inspired to implement a privacy-preserving variant of linear regression using homomorphic encryption. In their paper, *Privacy-Preserving Ridge Regression on Hundreds of Millions of Records*, Nikolaenko et. al present a system for privacy-preserving ridge regression using both homomorphic encryption and garbled circuits. Like ridge regression, linear regression is a fundamental building block for many machine learning operations. Thus, it seemed logical to extend the work of Nikolaenko by exploring the possibility of implementing linear regression. However, unlike Nikolaenko, our implementation relies only upon homomorphic encryption, not garbled circuits.

Similarly, we were inspired by the work of Bost et.al. [4] to implement several crucial classification algorithms for machine learning. Given that there appeared to already exist a handful of implementations of classification algorithms, it seemed logical to implement a different class of machine algorithm. Since we had already decided to implement a regression algorithm (linear regression), we chose to implement a clustering algorithm. In particular, we chose to pursue k-means clustering due to its breadth of useful applications, especially in the health and medical field.

2. Related Work

2.1. Homomorphic Encryption

An operation f is homomorphic in an encryption scheme if $\text{Dec}(f(\text{Enc}(x))) = f(x)$. A Somewhat Homomorphic Encryption (SHE) scheme allows only a limited subset of functions to be evaluated homomorphically, while a Fully Homomorphic Encryption (FHE) scheme allows arbitrary functions. Some SHE schemes, such as FV and YASHE [10], allow homomorphic addition and multiplication, which would allow for arbitrary computation, but are limited in the depth of a function evaluation circuit by noise growth. Such SHE schemes can be extended to be fully homomorphic by performing a bootstrapping procedure to reduce the noise. To do so involves evaluating the decryption circuit homomorphically. The need to perform this procedure repeatedly causes such FHE schemes to be extremely computationally expensive in both time and memory.

2.2. HE Linear Regression

Regression algorithms are important mechanisms used to solve machine learning problems. For this reason, regression is one of the first machine learning algorithms to have been implemented using HE. Previous studies have worked to produce implementations of different regression models using HE. This includes implementations of Ridge Regression, Linear Regression, and Multiple Linear Regression. Below, we expand upon related approaches to Liner Regression in HE.

Various previous studies have presented protocols used to obtain a linear regression model from FHE encrypted data. Several studies used Cramers rule or matrix inversion to obtain a linear regression model. However, their proposed method is only practical for data with small dimensions (i.e. less than 6). [16]

Almost none of the studies we found had used an existing, off-the-shelf implementation of an HE scheme or a public library implementing HE. Most studies implemented linear regression in one of the following ways, which included either **i)** implementing their own, modified variants of proposed HE schemes, or **ii)** providing their own HE schemes, or **iii)** borrowing mathematical elements from existing HE libraries (such as HELib) to construct a new implementation. [12, 2, 3, 1]. To our knowledge, there exists no off-the-shelf implementation of linear regression using a preexisting HE implementation or HE library. Thus, we chose to explore the viability of using an existing HE library (i.e. *SEAL*) and its underlying HE scheme as a means of building a linear regression model, and sought to test the quality of the results we obtained.

Linear Regression The goal in linear regression is to create a line of best fit for a set of data. The input data X is a series of vectors transposed (so that each data point is a row). Each input data point has a corresponding value. The values are represented as a vector y such that the value of the vector of row i is y_i. We can solve this problem using the Least Square's method. This method involves solving the following equation: $\theta = (X * X^T)^{-1} * X^T * y$. θ is a vector of degree d. To obtain the predicted value for a new data point v, one simply needs to calculate θv. As comparison is generally not possible in Homomorphic Encryption, many of the standard techniques to solve a system of linear equations are not possible. There were four main approaches to solving this equation that we found that been experimented with for Linear Regression on Encrypted Data: Cholesky decomposition, Division-Free Matrix Inversion, Cramer's Rule, and Gradient Descent.

Cholesky Decomposition The technique of Cholesky decomposition is a data-agnostic method (i.e. its execution

path does not depend on the input) which can be used to solve linear systems, such solving the Linear Regression equation. Cholesky decomposition decomposes a matrix A into two lower triangular matrices B and B^T such that $B * B^T = A$. Using $A = X * X^T$, we can get two lower triangular matrices that making solving $\theta = (X * X^T)^{-1} * X^T * y$ easy using back-propagation.

For computation on unencrypted matrices, Cholesky decomposition has been proven to be roughly twice as efficient as LU decomposition. Moreover, it is often advantageous to use Cholesky decomposition when efficient, exact solutions to matrix inversion are desired. Cramer's rule is exact another method for performing matrix inversion, but is far less efficient.

Despite the benefits of Cholesky decomposition, its use Cholesky is far more limited because it can only be applied to symmetric, positive, semidefinite matrices. Thus, it seems impractical to rely on Cholesky decomposition in real-world settings as we cannot feasibly assume that all input data are in the form of symmetric, positive, definite matrices. In addition, Cholesky decomposition, requires the ability to take square roots and perform division [12]; unfortunately, these operations are not yet built into *SEAL*. As a result, linear regression using Cholesky decomposition is not readily implementable using *SEAL*.

Division-Free Matrix Inversion In the paper, *Using Fully Homomorphic Encryption for Statistical, Ordinal, and Numerical Data*, the authors propose an alternative approach to building a linear regression model with high dimensional data from FHE ciphertexts [11]. Unlike alternative approaches, the computational complexity of their method does not blow up factorially with increased data dimensions. Central to their approach is the use of a division-free variant of iterative matrix inversion, which allows them to more efficiently compute the matrix inversion on FHE encrypted matrices.

With this approach, the authors are able to perform linear regression on large-scale datasets. For example, letting N denote the number of data points and d denote the dimensionality of the input data, they are able to run linear regression on one dataset with as many as $N = 32,561$ and $d = 6$, as well as on another dataset with $N = 1,994$ and $d = 20$.

Despite the benefits of increased scalability, their approach is not practical without the use of several key assumptions. The method they use for division-free matrix inversion of a matrix M, relies on the assumption that the largest eigenvalue of M is known beforehand. It is necessary for the largest eigenvalue to be known in order for their method to converge quadratically to a close approximation of the inverse of matrix M. However, this assumption is not practical because, in typical real-world settings, the largest eigenvalue of an FHE encrypted matrix would not be known beforehand. To obtain the largest eigenvalue of a matrix M, the authors apply PCA, which returns the eigenvalues of the covariance matrix, Σ. Then, they use an iterative algorithm (the PowerMethod), to evaluate the k-th eigenvalue λ_k, and the corresponding principal component u_k with some specified T number of iterations. It follows that evaluating the k-th eigenvalue for $k = 1$ gives them the value of the largest eigenvalue of M. Performing PCA before running division-free matrix inversion adds significant complexity to the authors' proposed method. Experimental results showed that evaluation time of PCA increases linearly with input dimension, and evaluation time of linear regression increases quadratically with input dimension. In addition, division is required to perform the variants of PCA and the PowerMethod leverage in their protocol, which means their approach is not immediately implementable in *SEAL* (due to *SEAL's* lack of division and square root operations).

For the above reasons, we did not choose to implement this approach for linear regression in FHE. We also note that we are only aware of HE implementations of division-free matrix inversion using HElib; to our knowledge, no such implementation exists using *SEAL*.

Cramer's Rule Cramer's rule is an exact method used to solve a system of linear equations. Cramer's rule states that $A^{-1} = \frac{1}{det(A)} * adj(A)$. Using $A = X * X^T$, we can easily compute A^{-1} given that we calculate the determinant and adjugate matrix. Using Laplace's formula and cofactors, we can solve for both of these without using division, square root, or comparison operations. Like Cholesky decomposition, Cramer's rule is often advantageous when data-agnostic, exact methods are desired. For small dimensions, it is feasible to simply hard-code the computations necessary to compute the result. For larger dimensions, however, the problem needs to be solved recursively.

Gradient Descent Another relatively efficient method for solving linear systems is gradient descent. Using a fixed number of iterations of gradient descent, one can approximate a reasonable solution to $\|A\beta - b\|_2^2$. When the matrix, A, is sparse, gradient descent is often preferred over exact methods. An obvious drawback of using gradient descent for linear regression is that it only approximates solutions. Moreover, the accuracy of the resulting approximation is often depend on a the number of fixed iterations performed; choosing the optimal number of iterations can require extra work upfront. In addition, due to the iterative nature of the gradient descent method, any errors in the estimation of B become amplified with each iteration, which can then result in severely inaccurate estimations of the solution to a linear system.

However, in some scenarios, such as when the matrix A is sparse, gradient descent is ideal. One study, *Fast and Secure Linear Regression and Biometric Authentication with Security Update*, [1] implemented a secure, HE system with linear regression using gradient descent. With this approach, the authors were able to able to successfully perform linear regression over a simulated dataset of 10^8 records each with dimensionality, $d = 20$ in about 10 minutes. However, there solution is not error-free, as gradient descent provides only approximate solutions.

2.3. Privacy Preserving K-means

Several past publications have explored privacy preserving K-means algorithm. However, to our knowledge, our result is the first that has explored the use of a single server to compute clusters through a homomorphic encryption library. Previous attempts [13, 14] involve multiple cross client communication to achieve collection operations on encrypted data.

2.4. Other Privacy-Preserving Machine Learning Algorithms

Exploring the application of HE to machine learning problems is not a new phenomenon. Several previous studies have demonstrated that it is possible to implement privacy-preserving machine learning algorithms by using homomorphic encryption schemes. The available research can be divided into several groups on the basis of what type of algorithm or machine learning problem each paper explores. At a high level, research into other privacy preserving machine learn algorithms can roughly be divided into the following groups:

- *Regression:* (Linear Regression, Ridge Regression) [12, 6, 16, 1]
- *Classification:* (Linear Means Classifier, Naive Bayes, Decision Trees, SVM) [7, 4]
- *Clustering:* (K-Means, K-NN)[13, 14], and
- *Neural networks:* (Crypto-Nets) [17]

3. Problem Description & Threat Model

In this section, we provide a brief overview of the secure multi-party computation problem we seek to solve using FHE Machine Learning algorithms. We give two practical, real-world use cases. Finally, we describe the problem's Threat Model.

The union of cloud computing and machine learning has fundamentally transformed the value of data and enabled the development of myriad complex technological innovations. However, a dichotomy seemingly exists between the increasing need to make more and more data available, and the necessity to protect the confidentiality of this data. This problem can be formalized as a scenario in which a single entity (i.e. a Cloud Service Provider) performs machine learning on a multi-party dataset aggregated from various, distinct sources (i.e. Data Owners). In this scenario, each Data Owner should only be allowed access to the data he/she owns, and should learn nothing about the data provided by other Data Owners.

In the scenarios we will consider, each party has some of the data points which comprise the training set. In other words, we assume a *horizontally partitioned* dataset. We focus on the following challenge: training a machine learning model (e.g. linear regression or k-means clustering) on data points that must be kept confidential and are owned by multiple parties.

3.1. Example Use Cases

Below, we provide concrete examples of real-world scenarios in which it would be advantageous to use FHE-based machine learning algorithms. Specifically, we illustrate two practical scenarios in which it would be advantageous to apply FHE implementations of K-Means clustering and Linear regression, respectively. These scenarios are easily generalizable and demonstrate the importance of similar, efficient FHE implementations.

3.1.1 Example A: K-Means Clustering

K-Means clustering has many applications, especially in the medical field. A common approach is for medical researchers to use k-means clustering to investigate how patients with similar attributes might be related. For example, medical researchers or doctors located across several hospitals might apply k-means clustering to categorize patients as having high or low risk of developing heart disease on the basis of high blood pressure and cholesterol level. Due to privacy agreements such as HIPAA, doctors and medical researchers are prohibited from sharing sensitive patient information with each other or with third parties. However, using a variant of K-Means clustering with HE, the researchers could run k-means on their aggregated dataset without revealing any sensitive information about individual patients.

3.1.2 Example B: Linear Regression

Consider a doctor who would like to use a given linear regression method in order to predict the likelihood of a patient developing breast cancer on the basis of health measurements (e.g. age, height, weight, ethnicity, BMI, blood type) and test results (e.g. mammograms, MRI scans). In order to avoid computing a biased model, it is ideal to run the selected linear regression model on data points collected

in different hospitals in various countries across the world. On the other hand, each hospital cannot legally share unencrypted patients sensitive data (i.e. the measurements and test results) with other hospitals or with a third party (e.g. a cloud-computing server).

3.2. Threat Model

The adversaries in this model can be divided into three categories:

1. The server
2. Malicious users
3. External adversaries (e.g. attackers listening in on client-server communication)

Our goal is to ensure that the server learns nothing about input data received from users, and that all parties involved in providing input data to an aggregated dataset are unable to learn anything about each other's data.

We do not assume that the server is honest and assume that there may exist adversaries trying to eavesdrop on communication between users and the server. Our assumptions about the honesty of users is dependent on which algorithm is in consideration; in linear regression, we assume that there exist some malicious users; in k-means clustering, we assume that all users are honest.

4. Protocol Description

4.1. HE Linear Regression Protocol

There are users, who own data that they want to run LR on. There is an untrusted server which will perform the computations. One of the users is known to be trusted, and is called the secure user. The secure user is in charge of homomorphic Key generation and distribution.

4.1.1 Input

Each user has a unique shared secret key with the server ($K_{Server-User}$). This key is generated using Diffie-Hellman key exchange. Each user also has a unique shared secret key with the secure user ($K_{Secure_User-User}$). This key is generated using Diffie-Hellman key exchange. Each user calculates the number of data points (n) and sends $Enc(K_{Server-User}, n)$ to the server Each user calculates the number of data points (d) and sends $Enc(K_{Server-User}, d)$ to the server. If a users d does not match the other users d, the server will send the user and error and ask for a new value. The server then decrypts all of the ns and sums them together to get the total dataset size. The server generates a random dataset ($dataset - random$) which approximately the same number of data points and same dimension as the desired dataset. The server runs the linear regression model on $dataset - random$. The server then creates a set of homomorphic keys and encrypts $dataset - random$ using them. The server then runs the linear regression algorithm on $Enc(dataset - random)$. The server will tweak the parameters of the homomorphic key generation until the results from linear regression on encrypted data matches the results from the non-encrypted data.

The server encrypts the required values for the parameters of key generation and sends back $Enc(K_{Server-Secure-User}, parameter - values)$ to the secure user. The secure user uses these parameters to generate a public/private key pair for Homomorphic Encryption. The secure user sends $Enc(K_{Secure-User-User}, K_{ML})$ (the private key for Homomorphic Encryption) to each of the users using their unique shared key. The users now all know the secret key. This is the secret key used for homomorphic encryption. Each user encrypts his data with SK_{ML}. He then encrypts his data using $K_{Server-User}$ and sends it to the server.

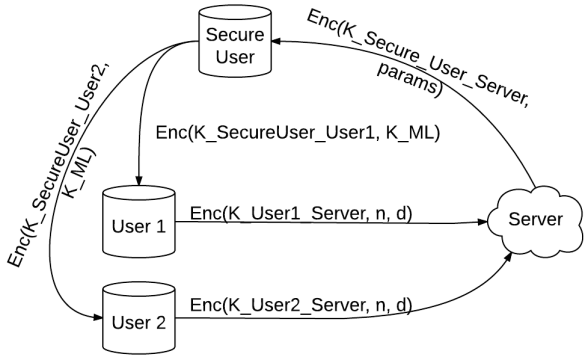

Figure 1. Diagram of how information flows through our system to generate the private homomorphic key.

4.1.2 Linear Regression Model

The server receives all of the data from the users. The server decrypts each batch of data using $K_{Server-User}$ for each user. The resulting data is still encrypted using SK_{ML}, which the server does not know. The server then runs the Linear Regression algorithm on the data-set. After this algorithm has completed, the server has computed the regression Matrix and determinant. As this was done on encrypted data, the resulting matrix ($Enc(SK_{ML}, M)$) and determinant ($Enc(SK_{ML}, d)$) are encrypted. The server can not decrypt either of these.

4.1.3 Output (Version 1)

The server uses $K_{Server-User}$ to encrypt and send back $Enc(K_{Server-User}, Enc(K_{ML}, M))$ and $Enc(K_{Server-User}, Enc(K_{ML}, d))$ for each user. Each

user then uses $K_{Server-User}$ to decrypt the messages into $Enc(K_{ML}, M)$ and $Enc(K_{ML}, d)$. As each user knows the homomorphic encryption secret key, they can decrypt the results into M (the vector) and d. Whenever a user wants to get predict using the regression for a new data point x, they only need to compute $\frac{1}{d} * (Mx)$.

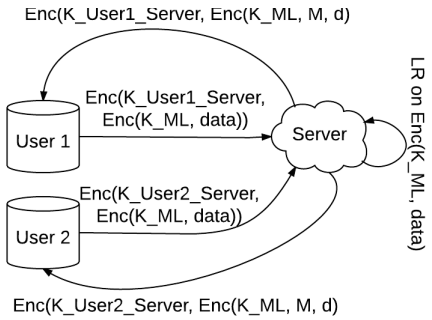

Figure 2. Diagram of how the data and results would flow through our system for output version 1.

4.1.4 Output (Version 2)

The server now has a Linear Regression model that it can not decrypt. When a user wants to compute the regression on a new data point x, they send $Enc(K_{Server-User}, Enc(SK_{ML}, x))$ to the server. The server decrypts it into $Enc(SK_{ML}, x)$. The server then computes $Enc(SK_{ML}, M) * Enc(SK_{ML}, x) = Enc(SK_{ML}, y)$. The server uses $K_{Server-User}$ to encrypt and send back $Enc(K_{Server-User}, Enc(SK_{ML}, y))$ and $Enc(K_{Server-User}, Enc(SK_{ML}, d))$ to the user. The user then decrypts $Enc(SK_{ML}, y)$ and $Enc(SK_{ML}, d)$ using SK_{ML} into y and d. The resulting y-value (or regression result) for the vector x is y/d. As the data is decrypted at this point the user can easily compute this.

4.2. HE Linear Regression Safety

4.2.1 Server

As the server does not know SK_{ML}, it is unable to decrypt any of the data it is given. Even when the server computes the resulting linear regression matrix, it can not decrypt it to know what the model is. The only knowledge the server has about the data and resulting model is the number of data points and dimension of the each data point.

4.2.2 Malicious Users

Malicious Users are unable to determine any of the data that the model was trained on except for the data that they uploaded. As they only know their specific $K_{Server-User}$, they will not be able to decrypt any of the messages between other users and the server. Therefore they will not be able to determine any data from any other user.

4.2.3 External Adversaries

Any adversary listening to communication between the server and a user will not be able to gain any knowledge about the data or model, as they do not know $K_{Server-User}$ for any of the users. Even if the adversary was able to trick the server into thinking it was a valid user (and gaining their own $K_{Server-User}$), they would not be able to learn any valuable information about the either the data or the model. In Output Version 1 the user could ask the server for the model and receive it. The model that they received, however, is encrypted using SK_{ML}, so the adversary would not be able to decrypt it. The adversary might be able, however, to see the dimension of the matrix. (As the resulting matrix is just a vector of encrypted points (i.e. the coefficients)). If this is considered a security flaw, the server could then generated a series of random numbers (and encrypt them using PK_{ML}) and append them to the bottom of the vector. As the users know the dimension (d) of the data-set, they can just use the first d dimensions of the resulting vector for their calculations.

4.3. Client Communication Protocol

Certain operations are difficult to simulate in the SEAL library such as division, as most approximation algorithms require an order of magnitude estimate of the reciprocal in order to converge. Therefore, to allow computation of reciprocals on the server side, we propose the following protocol for computing the reciprocal of a cipher-text C.

1. Given a ciphertext C, the server chooses some random noise e, and gets $y = Ce$ and sends the result to a client.

2. The client, upon receiving y, decrypts y using the secret key to get y'. The client then calculates $z' = 1/y'$ and encrypts z' to z and sends the result to the server.

3. The server, upon receiving z, computes $1/C = ze$ homomorphically.

Unfortunately, despite obscuring data with noise, the client may still have order of magnitude estimation of various values in our K-means system. To remove information the user may receive about intermediate computations, we also send random values for the client to decrypt.

4.4. Division

One limitation of the seal library is the lack of a homomorphic division operation. One approach to resolve this is to implement an approximate division algorithm using only homomorphic addition and multiplication. The Goldschmidt method of division is as follows: in order to approximate $\frac{1}{b}$, let Y be an initial guess, and let $e = 1 - bY$.

Then

$$\frac{1}{b} \approx Y(1+e)(1+e^2)(1+e^4)\dots.$$

However, this method requires an initial guess that is reasonable close e.g. within an order of magnitude of the actual value in order for the value to converge quickly. If no information is known about the value to be inverted, then this method is unsuitable. In our code, we instead used the above client communication protocol to calculate division.

4.5. K-Means Clustering Protocol

4.5.1 Input

Each user generates a shared private key for encryption and sends their encrypted data as well as their corresponding public key to the server. The server acknowledges receipt of the public key if all public keys are matching and sends an ack message to each user. Each user then sends their encrypted dataset to the server.

4.5.2 Computation

The server aggregates all the encrypted data together and generates a set of proposals for cluster centers c_i. We currently randomly generate the center proposals, though its initial centers can be initialized to a weighted mean of the inputs.

We then run the following loop for a user specified number of iterations

1. For each point x_i and center c_j, we compute the inverse square distance $d_{ij} = 1/(x_i - c_j)^2$
2. For each point x_i and center c_j, we assign x_i to c_j with weight $w_{ij} = \frac{d_{ij}}{\sum_j d_{ij}}$
3. We update each cluster $c_j = \sum_i w_{ij} * x_i$

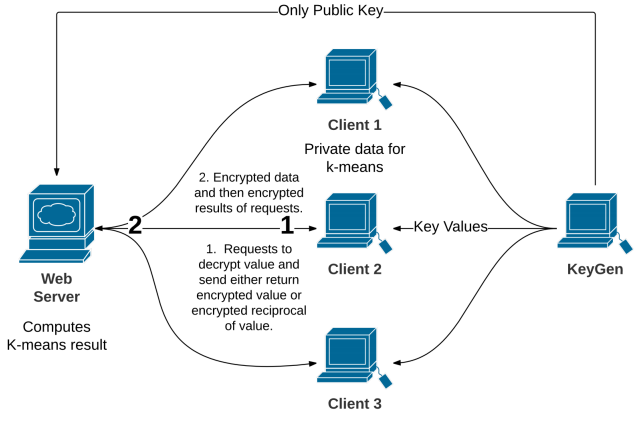

Figure 3. Schematic of Communication Between Client and Server

Each of the above operations can be done homomorphically using existing operations in SEAL expect for the division tasks through which we invoke the client communication protocol. In addition, in order to reduce exponential noise increase of the newly generate centers in the third step of the algorithm, we ask clients to decrypt and encrypt each of our centers multiplied by random noise. An overview of our system can be found in 3

4.6. K-Means Safety

Our protocols for K-Means and for computing reciprocals of numbers requires that both clients and servers be honest and return correct results. Under the assumption that certain clients may return fake results, we could potentially send requests to multiple clients and take the value that is agreed on by a majority or clients. A passive adversary listening to our communication will not gain any information as all communication with encrypted cipher-text so an active adversary able to modify our communication information will impact our computations.

5. System Description

Our implementation is available on Github at here. We used the *Simple Encrypted Arithmetic Library (SEAL)*, which uses the FV HE scheme, to implement two machine learning algorithms: linear regression and k-means clustering.

5.1. SEAL Crypto library

When planning how to implement machine learning algorithms using homomorphic encryption schemes, we discovered that there exist several software libraries implementing homomorphic encryption. These include HElib, $A\ \lambda$, NFLib, and *SEAL*[5, 8]. The most viable options appeared to be the SEAL and HElib libraries. Both libraries implement fully homomorphic encryption schemes; *SEAL* uses a more generalized version of the Fan-Vercauteren (FV) scheme. The textbook presentation of the FV encryption scheme is a leveled, fully homomorphic, and supports an arbitrary number of additions and multiplications on encrypted data.

Fortunately, the most recent version of the *SEAL* library (v2.1) is accompanied by detailed documentation explaining *SEAL's* core functionality, as well as how to select the optimal set of parameters for good performance for *SEAL*[5, 9].

6. Additions to the SEAL Library

6.1. Matrix Library

In order to make implementing machine learning algorithms easier, the we first created a Matrix library. This

matrix was represented as a 2-D vector in C++ of matrix entries, each which was a encrypted value (BigPolyArray). We then implemented methods for the following common matrix operations, adding two matrices, multiplying two matrices, multiplying a matrix by a constant, determining the determinant of a matrix, calculating the transpose of a matrix, and calculating the adjugate of a matrix. For determining the determinant and adjugate of a matrix, we hard-coded in the cases for square matrices of dimension 1,2,3. This helped increase the performance of the linear regression. For matrices of dimension 4 or larger, we used a recursive algorithm to calculate the determinant and adjugate matrix.

Determinant Calculating the determinant efficiently is hard without division. We chose to hard-code the algorithms (in terms of matrix indices) for dimensions 1,2,3. This allowed for the fastest result, as we were not doing any unnecessary computations to calculate the determinant. For dimensions 4 and higher we chose to calculate the determinant recursively, by using Laplace's formula and co-factors. This lead to a huge increase in time taken for dimensions 4 or larger. We could possibly achieve some speed up by using $CITE$ the division-free algorithm for these dimensions. However, our results for d = 3 (where determinant formula was hard-coded) were unconvincing that more efficient computation would fix the problem of overflow.

Adjugate Matrix Similar to the determinant, we hard-coded the results (in terms of matrix indices) for dimensions 1,2,3. Also similar to the determinant, we found the adjugate matrix recursively for dimensions 4 and higher using cofactors.

6.2. Baseline model

For our baseline model we chose to use the same algorithm for calculating the Least Square's regression. We used our existing C++ code for Linear Regression on encrypted data and adapted it to work with non-encrypted data. This allowed for a good timing comparison, as any inefficiencies due to our choice of algorithm, or speed-ups by using C++ over a different language would show in the results for our baseline. We also used our baseline model to verify the results that we received from our HE LR.

7. Linear Regression

7.1. Approach

The matrix class helped make implementing Linear Regression much easier. Least-squares linear regression is the vector θ in the following equation: $\theta = (X*X^T)^{-1}*X^T*y$ where X is the matrix of each of the data points transposed (such that each data point is a row). y is a vector of the corresponding y values for each data point. The difficult part of linear regression is computing the inverse. We chose to use Cramer's Rule ($A^{-1} = \frac{1}{det(A)} * adj(A)$) where is $adj(A)$ is the adjugate matrix of A. As we can efficiently do division in SEAL, we chose to output $det = det(X * X^T)$ and $M = adj(X * X^T) * *X^T * y$. The resulting M is a d dimensional vector, where d is the dimension of each of the input vectors. If a user has an unencrypted M and det, to run the regression on a new point v all the need to do is calculate $\frac{1}{det} * M * v$.

7.2. Result Validation

First, we tested our Matrix class in order to make sure that all of the operations worked as they were supposed to. We created a series of test matrices for the functions, and validated the results against Wolfram Alpha for the computation. For linear regression, we validated the results that we got from the both models by checking the coefficients of the resulting model to see if they matched the line generated from the data.

7.3. Parameters

In order to get accurate results from the HE LR, we needed to carefully choose the encryption parameters in order to make sure that the large number of operations that we needed to perform did not create overflow.

Polynomial Modulus This parameter represented the polynomial modulus for the ring R [5]. For most of the time (degrees 1 and 2, and 100 or less data-points in degree 3), the polynomial $1x^{8192} + 1$ was sufficient. This made the model tolerant to 202 bits of noise. This was the second to largest polynomial recommended by SEAL. For degree 4, and 100-200 points in degree 3, the result were too noisy and we needed a larger polynomial. We used $1x^{16384} + 1$, the largest recommended polynomial. In this case the system was tolerant to 404 bits of noise. Because we made the polynomial modulus so large the encryption and arithmetic operations were very slow. Also, a larger polynomial modulus is generally less secure, which is why we tried to use a smaller one when possible.

Plaintext Modulus This parameter represented the maximum coefficient in the plaintext space. All of the decrypted results were $/modplaintext - modulus$. For most of the linear regressions, 2^{23} was a reasonable value for this. It's maximum value was 2^{30}. In the cases of the 100+ data points for degree 4 or 500+ datapoints for degree 3, this value (2^{30}) was not large enough. There were too many additions/multiplications done that the value overflowed the plaintext modulus, leading to an inaccurate result. This is

because the the resulting vector was supposed to be scaled down by the determinant, but because there was no division we were unable to scale down the result.

Decomposition Bit Count The decomposition bit count affected the speed of relinearization and noise growth from relinearization. A low decomposition bit count meant slower noise growth but also slower relinearization. A higher decomposition bit count meant faster noise growth and faster relinearization. The decomposition bit count was supposed to be between 1/10 and 1/2 of the significant bit count of the coefficient modulus As we were more concerned with noise growth than time taken, we chose the minimum decomposition bit count for our current polynomial modulus. For every case except degree 4, and degree 3 with 200 data points, that meant decomposition bit count was 24. When the polynomial was $1x^{16384} + 1$ the decomposition bit count was 44.

8. Performance

8.1. Overview of Experiments

One motivation for performing these experiments was to show that applying basic machine learning algorithms to compute on encrypted data is in fact possible using open-source software and modern-day cryptographic schemes. Moreover, we performed these experiments in order to assess whether or not these implementations were not just possible but also practical.

In order to demonstrate the functionality of our privacy-preserving machine learning algorithms, we conducted several rounds of experiments in which we varied the number of features (i.e. the datas dimensionality, d) and the number of data points (i.e. the datas volume, or number of data points, n) used as input to train our machine learning algorithms. Our experimental data is comprised of several synthetically generated, numerical datasets of varying number of points, n, and dimensionality, d.

We then compared how these parameters affected each algorithms performance and accuracy. We note here that our experiments confirmed the feasibility of performing machine learning on encrypted data, but revealed that current real-world implementations are still far less practical than similar, less privacy-focused implementations used to compute on unencrypted data.

Furthermore, we compare the runtime required for these two algorithms (linear regression and k-means clustering) to compute on encrypted and unencrypted data to illustrate the difference in computational costs between computing on encrypted and unencrypted data. One of the main goals of our research was to assess the feasibility of deploying machine learning algorithms implemented using FHE schemes to be used in real-world applications. In such real-world settings, these machine learning algorithms would likely be implemented as part of larger, performance-optimized systems. For this reason, we chose to *fix* the security parameters used by the system, as this most closely simulates how real-world cloud systems would operate.

Degree	Num. Points	Time	Noise
1	10	5008	83
1	25	43733	87
1	50	25172	86
1	100	46344	87
1	200	195172	88
1	500	246219	88
1	1000	558470	89
2	10	18920	93.5
2	25	39638	94.5
2	50	74042	96
2	100	148536	96
2	200	301047	97
2	500	780306	97.5
2	1000	2258306	98
3	10	71218	134
3	25	114733	135
3	50	182402	135
3	100	331933	136
3	200	655707	158.667
4	10	1608986	245.25
4	25	1943827	248
4	50	2504500	245

Table 1. Table 1. Results from LR on HE data

In the following sections, we provide a more detailed performance analysis of linear regression and K-means clustering based on the results of our running our experiments.

8.2. Linear Regression Performance Analysis

The performance of the linear regression model depended on the number of points in the data-set and the dimension, or number of features,of each data point. We will evaluate the performance of the models based on time taken to generate the model and the amount of noise in average amount of noise in the resulting model (vector).

Table 1 shows the results for LR on HE data for many trials varying the degree of the regression, and number of data points we were regressing on.

Figure 4 shows the comparison in time taken for LR on encrypted vs non-encrypted data. Here we fixed the degree of the data to be 2. For the non-encrypted data, we ran the sample least square's regression algorithm as for the encrypted data. For every size data-set, LR on non-encrypted data took under 1 ms to complete. For encrypted data, as you can see from the graph, time grows somewhat linearly with number of data points. It takes a lot longer than non-

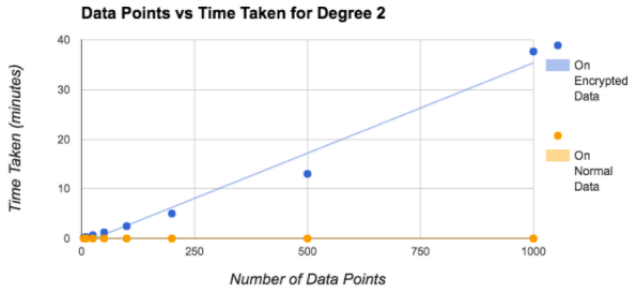

Figure 4. Graph of the amount of time taken for LR on encrypted and non-encrypted data.

encrypted data, 1000 data points here for degree 2 takes 40 minutes.

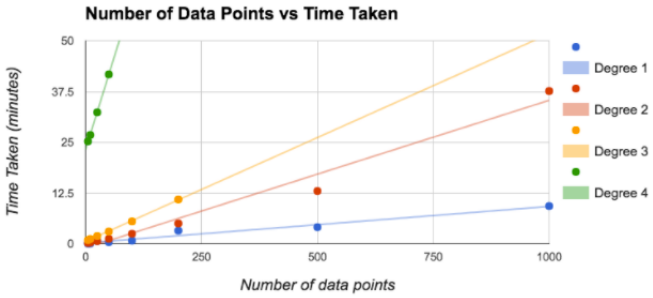

Figure 5. Graph of the amount of time taken for LR on encrypted data for vary degrees.

Figure 5 shows the comparison in time taken for LR on encrypted data for varying degrees. Low degrees take less time for Linear Regression. This is because calculating the determinant or adjugate matrix requires no or close to no computations. For degree 4, the determinant and adjugate matrix were calculated recursively, leading to such a significant decrease in performance. For degrees 3,4 after a certain number of data points (200 for degree 3, 50 for degree 4), the resulting LR vector became non-decryptable. This is because there were so many additions required in the calculation that lead to the coefficient overflowing the maximum coefficient, leading to the incorrect result in decryption.

Figure 6 shows the comparison in the amount of noise in the result from LR on encrypted data for varying degrees. Low degrees yield less noisy results. As you can see, once we fix the degree, the amount of noise stays close to constant. The growth of noise for larger datasets is very minimal. For each of the trials in the graph, the encryption parameters were set up such that they had a noise tolerance up to 202 bits. As you can see for the degree 3 case, the noise is still within a reasonable threshold by the time the LR is no longer able to compute the correct result. The reason that this is happening is that there are so many small multiplications and lots of additions such that the coefficients are growing past the maximum coefficient size. Since the coefficients are $coeff \mod max - coeff$, this leads them to get a new value not equal to their old. When we decrypt the new value, it is different than the old one, giving us the wrong result. Since we are unable to do division in *SEAL*, we can not scale the numbers down at any point to prevent this.

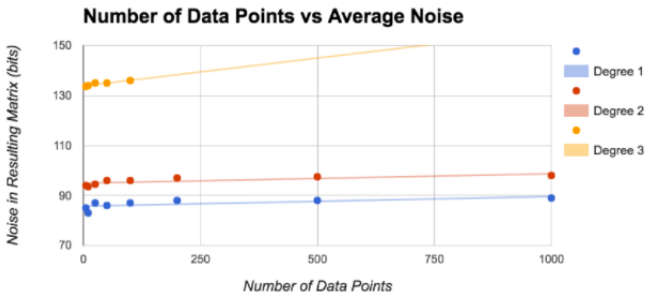

Figure 6. Graph of the amount of noise in the result of LR on encrypted data for vary degrees.

8.3. K-Means Performance Analysis

In this section, we examine four variables independently: the total number of data points used, the number of independent computing entities communicating, and the count of clusters (means) found in the data. We only consider 2 dimensional data, though we believe the run-time on our data will scale linearly with increased number of dimensions. Each of these is evaluated independently, with all other variables held constant, and compared to run-time.

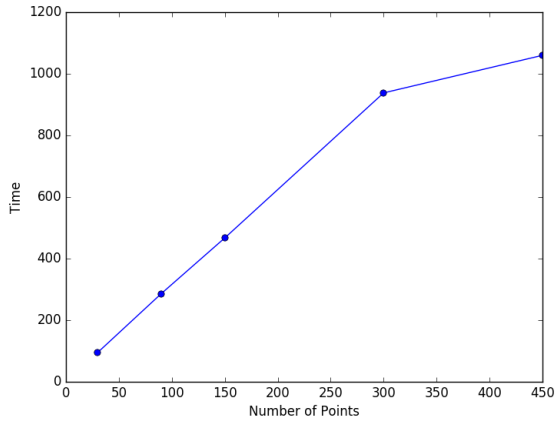

Figure 7. Graph of the time per loop of K-means with 3 clusters(in seconds)

We first analyzed the increase in run-time as we increased the number of points being clustered in Figure 7. We found that time increased approximately linearly with the increased number of points. Note that the graph shows

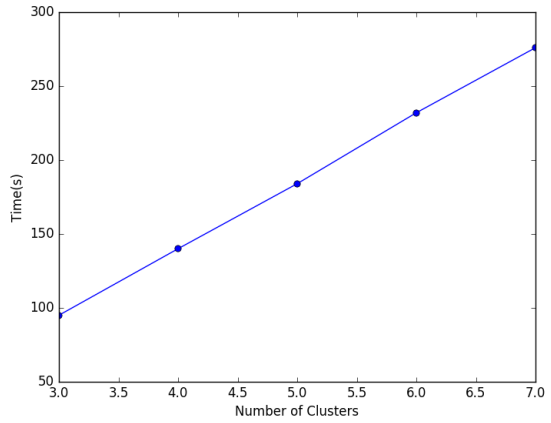

Figure 8. Graph of the time per loop of K-means with 30 points

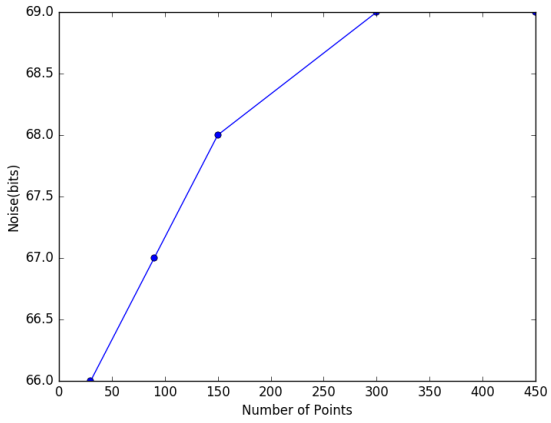

Figure 9. Graph of the noise per loop of K-means with 3 clusters(in bits)

the amount of takes it takes to update the centers of the cluster once. In a typical K-means application, we would need to updated centers iteratively at least 10 times.

Next, we analyzed the increase of run-time as we increased the number of clusters we were clustering our points on as found in Figure 8. It appears that time is also linearly dependent on the number of clusters we are trying to find.

We also analyzed the increase in noise as we increased the number of points being clusters as found in Figure 9. It appears that noise remains relatively constant despite the large increase of points we are clustering. It appears that with other parameters, the amount of noise of centers still remains relatively constant. This may mean that as long as time is disregarded, we can compute clusters for a large number of coordinates.

For all of our analysis with used as our polynomial modulus, $x^{8192} + 1$ with default parameters. As our noise levels were not near the noise cap of 198, we may have been able to achieve significantly faster results with a smaller polynomial modulus.

We include additional images of our algorithm running in practice at the end of the paper.

9. System Improvements

Ideally, it would be useful and practical to improve the scalability of our algorithm implementations. Specifically, it would be useful for our implementation to support computations on datasets with greater volume and higher dimensionality than the datasets used in our experimental tests. Moreover, it would be useful to support computation on larger datasets while maintaining high performance, accuracy, and efficiency.

Another way in which we might improve our system is to implement faster algorithms by exploiting GPUs. Several recent studies have explored the potential of using GPUs as a means of accelerating fully homomorphic encryption. These studies have shown that when GPUs are used to accelerate vector operations, the resulting implementations run significantly faster than CPU-based implementations. For example, one such study demonstrated that a GPU-based implementation ran up to 273.6 times faster than on CPU [15].

Alternatively, we could accelerate the runtime of our algorithms by implementing a different, faster algorithm used to calculate the determinant of a matrix. This would especially help to accelerate our linear regression algorithm, as well as many other machine learning algorithms which could be implemented in the future. In particular, we would like to implement a division-free algorithm for computing determinants, such as the one proposed by Richard Bird [3]. This would have decreased the time to train for matrices with dimension greater than three.

A possible other speedup would be parallelize many of the algorithms we wrote for both K-means and linear regression. Both have many repetitive operations that are independent of each other that can be done parallel.

In addition, we would like to widen the applications with which our system might be used. One way to achieve this would be to augment our system such that it supports not only numeric data, but categorical and ordinal data, too. This would allow our system to perform a wider range of statistical procedures (e.g. histograms and k-percentile) and would better enable our system to implement categorical classification via machine learning algorithms.

On a different note, we would also like to improve the usability and accessibility of the SEAL library and our algorithm implementations. To do this, we would ideally add support in SEAL for other languages to be used. This would make the SEAL library more accessible to users unfamiliar with C/C++. Our first step might be to develop tools to make C/C++ functions/methods accessible from Python

by generating binding (Python extension or module) from header files. Similar tools for C/C++ binding generators for other languages could also be developed. However, we observe that algorithm performance using other languages may be suboptimal compared to using C++.

10. Conclusion

In this project, we sought to explore secure, machine learning algorithm implementations using a fully homomorphic encryption library, allowing efficient data analysis on private, aggregated datasets. Specifically, we wanted to implement useful Machine Learning (ML) algorithms in HE to show whether or not applications of HE are actually possible and potentially even practical, compared to non-HE algorithms. We chose to implement and explore two essential machine learning algorithms: linear regression and a horizontally-partitioned k-means clustering algorithm. Our implementation was coded in C++ using Microsofts open-source encryption library, *SEAL*, as a building block.

Our evaluation of our implementation suggests that, although we were successful in implementing working HE implementations for both linear regression and k-means clustering, they proved infeasible for large-scale datasets and relatively impractical to use when solving real-world machine learning problems. Thus, there is still substantial development needed in order to implement machine learning algorithms which are practical, and efficient for even small datasets. Additionally, there is a need for significant improvements to be made before today's working HE implementations are feasible on datasets with large dimensionality and high volume. Despite the gap in immediate, real-world applicability of HE-based machine learning algorithms and their less privacy-preserving, non-HE counterparts, it is promising that there exist easily accessible tools such as *SEAL* which enabled us to build working machine learning models. We hope that our work inspires future research in advancing the efficiency and practicality of HE implementations, and ignites interest in furthering the development of even more practical implementations of machine learning algorithms in HE.

Acknowledgements

We would like to thank our 6.857 professors Ron Rivest and Yael Kalai for their guidance, feedback, and support of our project. We would also like to thank our 6.857 TAs Cheng Chen for his useful feedback and discussion about project direction. Finally, we would also like to thank Shai Halevi and Kristin Lauter for providing valuable advice and insight.

References

[1] Y. Aono, T. Hayashi, L. T. Phong, and L. Wang. Fast and secure linear regression and biometric authentication with security update. *IACR Cryptology ePrint Archive*, 2015:692, 2015.

[2] R. S. Bird. A simple division-free algorithm for computing determinants. *Information Processing Letters*, 111(21):1072–1074, 2011.

[3] R. S. Bird. A simple division-free algorithm for computing determinants. *Information Processing Letters*, 111(21):1072 – 1074, 2011.

[4] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[5] H. Chen, K. Laine, and R. Player. Simple encrypted arithmetic library-seal v2.

[6] I. Giacomelli, S. Jha, and C. D. Page. Privacy-preserving linear regression on distributed data. 2017.

[7] T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer, 2012.

[8] S. Halevi and V. Shoup. Algorithms in helib. In *International Cryptology Conference*, pages 554–571. Springer, 2014.

[9] H. C. Kim Laine and R. Player. Simple encrypted arithmetic library-seal (v2. 1). Technical report, Technical report, Microsoft Research, 2016.

[10] T. Lepoint and M. Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *International Conference on Cryptology in Africa*, pages 318–335. Springer, 2014.

[11] W.-j. Lu, S. Kawasaki, and J. Sakuma. Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data (full version). 2017.

[12] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. pages 334–348, May 2013.

[13] S. Samet, A. Miri, and L. Orozco-Barbosa. Privacy preserving k-means clustering in multi-party environment. In *SECRYPT*, pages 381–385, 2007.

[14] J. Vaidya and C. Clifton. Privacy-preserving k-means clustering over vertically partitioned data. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 206–215. ACM, 2003.

[15] W. Wang, Z. Chen, and X. Huang. Accelerating leveled fully homomorphic encryption using gpu. pages 2800–2803, June 2014.

[16] D. Wu and J. Haven. Using homomorphic encryption for large scale statistical analysis, 2012.

[17] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. Lauter, and M. Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

11. Appendix

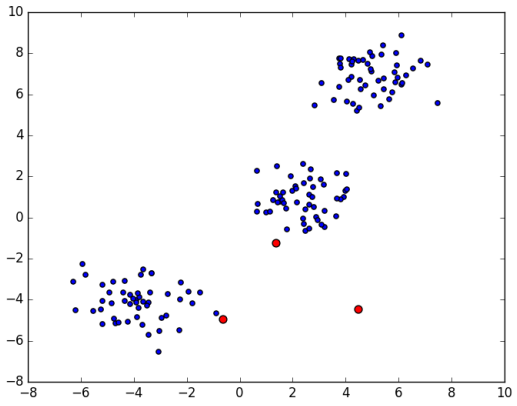

Figure 10. K-means(Default center initializations in red)

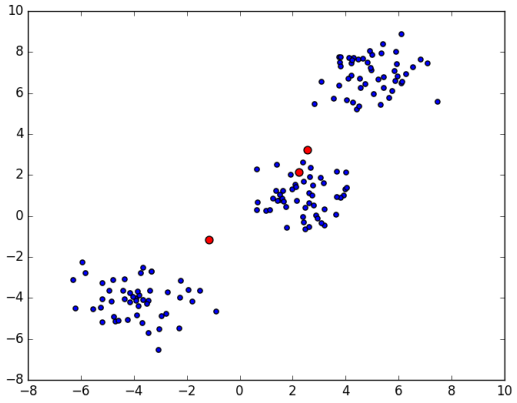

Figure 11. K-means(After 1 Iteration)

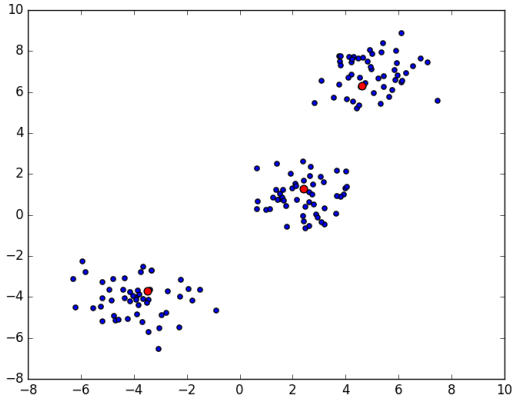

Figure 12. K-means(After 5 Iterations)

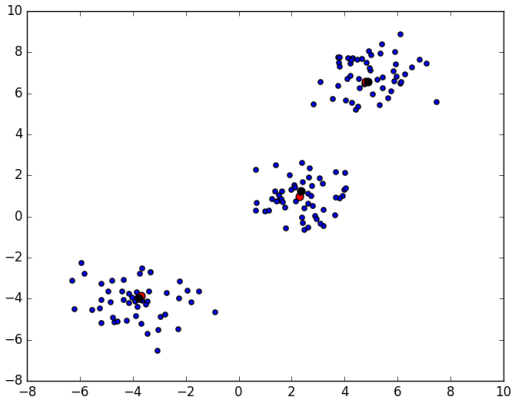

Figure 13. K-means(After 10 Iterations, black dots indicate python computed centers)

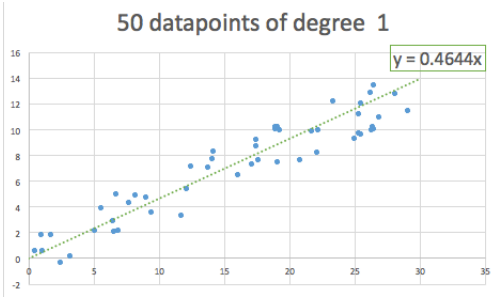

Figure 14. Linear Regression result on encrypted data(50 data points of degree 1generated noisily around $y = 0.45x$)

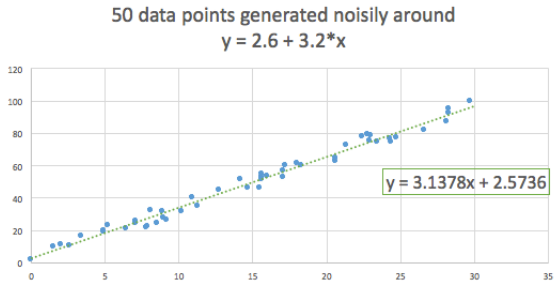

Figure 15. Linear Regression result on encrypted data(50 data points of degree 2 generated noisily around $y = 2.63.2x$)

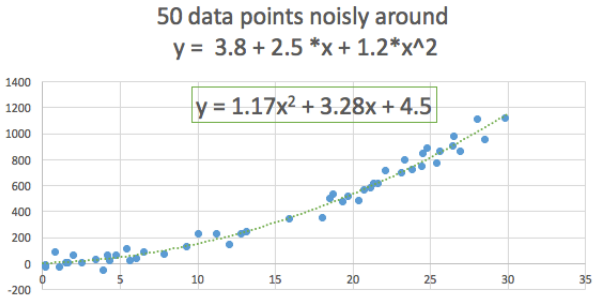

Figure 16. Linear Regression result on encrypted data(50 data points of degree 3 generated noisily around $y = 1.2x^2 + 2.5x + 3.8$)