Training a Logistic Regression Model Using Homomorphic Encryption

CS 1657 Privacy in Electronic Society

Project I Report

By: Mackenzie McIntyre

I. BACKGROUND

According to a blog post by Damien Desfontaies, Latanya Sweeney, a PhD student, heard about a health insurance company that was compiling a database of hospital visits by state employees. However, there were concerns about allowing researchers to look at other citizens' health records which felt like a major privacy violation. Even after removing the columns with patient identifiers (name, ssn, address, etc.), Sweeney was able to conduct a successful reidentification attack using the demographic information left in the database. This situation exemplifies how conducting research on sensitive information can lead to the privacy of individuals being compromised even when select patient identifiers are removed. Privacy is needed to preserve the integrity, confidentiality, and availability of information system resources.

The idea of fully homomorphic encryption was first described by Rivest, Alderman, and Dertouzos in 1978 (Rivest and Alderman being two co-authors of the RSA scheme). Using privacy homomorphisms, they articulated the problem by saying, "It remains to be seen whether it is possible to have a privacy homomorphism with a large set of operations which is highly secure" [10].

This introduces the motivation behind using homomorphic encryption to protect patient's privacy when research is being conducted. This project will explore this concept in relation to the training of machine learning algorithms.

More specifically, I will be experimenting with using a homomorphic encryption scheme, Cheon-Kim-Kim-Song (CKKS), to encrypt the patient data used to train a logistic regression machine learning model for binary classification of whether a patient has diabetes

A. Homomorphic Schemes

Homomorphic schemes have the capability of performing operations on the encrypted data and still preserve the result of the plaintext. Mathematically speaking, if E represents encryption, m represents the plaintext message, and D represents decryption, the following equation shows how the concept of homomorphism operates to recover the plaintext without operating on it directly:

$$E(m1) + E(m2) = E(m1+m2) = D(E(m1+m2)) = m1 + m2$$

This concept can be applied to training data for machine learning algorithms where the operations within the logistic regression model will be performed on encrypted patient data, preserving the privacy of such patients. This allows analysts to train models without interacting with the raw data that could reveal sensitive information about the patients.

II. ABSTRACT

Using this concept, I will be focusing on a data set from the National Institute of Diabetes and Digestive Kidney Diseases that was published by the University of California, School of Information and Computer Science to train a

logistic regression model. The dataset contains information from female patients over age 21 of Pima Indian heritage that include several medical predictor variables that will act as features of the training data set. Such features include sensitive information about the patients including, number of pregnancies, plasma glucose levels, diastolic blood pressure, serum insulin, triceps thickness, BMI, age, and diabetes pedigree. I will be using three different methods to train the model. The first method will be training the model using plaintext data (Algorithm 0), the second method with use the CKKS homomorphic encryption scheme to protect the data (Algorithm A), and lastly a modified CKKS algorithm that uses bootstrapping (Algorithm B). Bootstrapping is a technique used to iteratively reduce the noise in the ciphertext, allowing for a greater number of homomorphic operations on the data while still allowing it to be decryptable in the end.

With these three different methodologies I will be analyzing and comparing the accuracy and quantifying the privacy levels of the models. At the end of this experiment, I would like to be able to answer questions such as, how does the encryption of training data enhance privacy? How does the encryption of training data affect the accuracy of machine learning models? Can attackers gain information about the data from the model's outputs? If and how bootstrapping can aid in homomorphic encryption applications to machine learning?

Data analysts need a secure way to train models with sensitive data without compromising patient's privacy and I would like to explore if homomorphic encryption is a suitable solution for this growing issue. I will be quantifying the levels of privacy provided by measuring the entropy of the inputs of the different methods amount of privacy as well as analyzing the tuning parameters of the CKKS context variable required to create CKKSVectors when using the homomorphic encryption scheme with logistic regression machine learning models.

III. EXPERIMENT

A. Preprocessing data

When training any type of machine learning model, I have learned that the most important step to take first is to preprocess the data. I created a pipeline in my Machine Learning Studio on Microsoft Azure to convert the data set from the National Institute of Diabetes and Digestive Kidney Diseases to a proper csv for handling in python. The two data sets are the actual numerical values and the column labels being combined for handling.

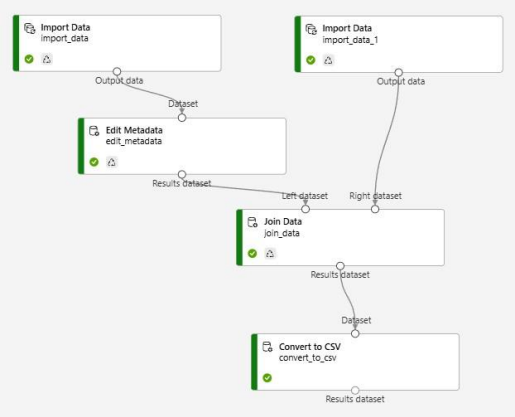

Figure [1] Pipeline of converting data to a CSV file for download

The next part of preprocessing the data can be seen in the python file preprocessData.py where I load in the csv, "patient_data.csv". From here, I extract the feature columns without their headings (i.e. Pregnancies, BMI, etc.) and exclude the column "PatientID" since it has no relevance to training the model. I also extracted the labels column that consists of 1s and 0s as a binary classifier for diabetes. Next, I appended an array of ones to the feature matrix as the bias term and then normalized the data using Min-

Max Normalization. Normalizing the data ensures all the features are on the same scale which helps increase the accuracy of the model.

However, the bulk of the preprocessing step comes from splitting the data into training and test sets for the model. I accomplished this by randomly grabbing 90% of the feature matrix and using those indices to also grab 90% of the labels array to be my x_train and y_train while the remaining 10% are used as my test data, x_test and y_test. I finished the preprocessing returning x_train, y_train, x_test, and y_test as the plaintext NumPy arrays setting up my inputs for Algorithm 0.

B. Algorithm 0

This part of the experiment consists of simply training a basic logistic regression machine learning model on the plaintext inputs discussed above. In the file logisticRegression.py I constructed a class called, LogisticRegression, that accepts the plaintext inputs, a learning rate (alpha), and an epoch count (iterations). This is a pretty standard implementation using the sigmoid function to compute the hypothesis and a cost function to compute the differences between the hypothesized values and the actual y_train values. The cost function also updates and tunes the gradients for the theta and b parameters for each iteration. The heart of the algorithm is in the gradient descent function, grad_descent(), where I call the cost function to perform forward and backward propagation and update theta and b.

Additional functionality was added to print out the iteration number because I needed a large number to train such a large data set and there were points when I was wondering if my program was still iterating of if it got stuck somewhere. This was just a "nice to have" for me to track the progress of the program while the model was training. I also include functionality to create a plot of iterations vs cost. I found this helpful not only for evaluating the model (which will come later) but also for tuning parameters alpha and iterations. I was able to use the chart to see after how many iterations the cost began to plateau and how alpha affected the rate. Plotting the cost was a tool for me to adjust the parameters to be what I deemed "optimal". Meaning, fewest iterations and biggest alpha I could get without detracting from the accuracy.

Below are a few trials I went through to eventually get to the optimal alpha and epoch values of 0.001 and 4500, respectively. These values make sense to me because the dataset is quite large and complex with many features and samples, so a small learning rate is necessary, so the model doesn't converge at a local minima due to overstepping. I chose 4500 because any higher and the model was just plateauing and any less the model still had room to reduce the cost more.

Figure [2] 1000 iterations, alpha = 0.001 Cost has not plateaued yet, so I need more iterations

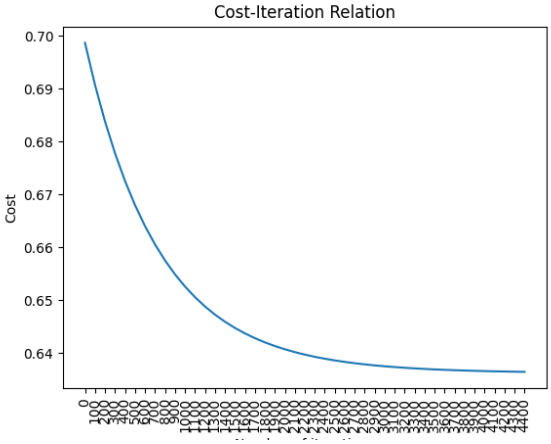

Figure [3] 4000 iterations, alpha = 0.001
I would like to see if I can get the cost to plateau a bit more

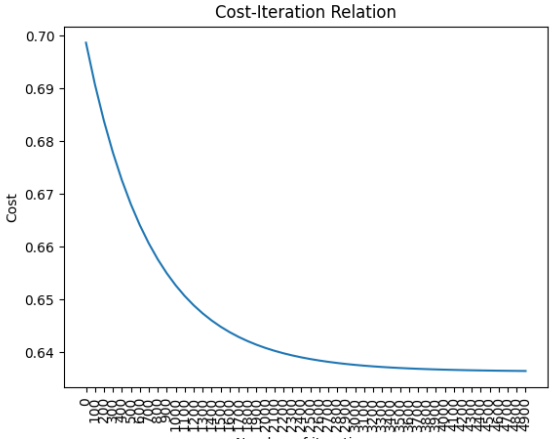

Figure [4] 5000 iterations, alpha = 0.001
Cost has plateaued but I want to see if changing the learning rate can give me a better accuracy

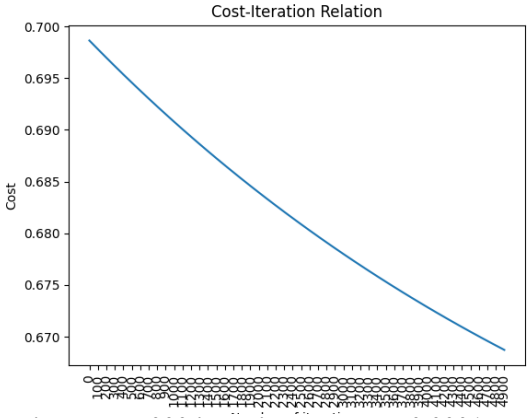

Figure [5] 5000 iterations, alpha = 0.0001
Learning rate too small

This process of tuning alpha and the number of iterations went on for a while, I believe it is a bit redundant to show every iteration given that there would then be many pages full of graphs showing the cost. These figures are to show the overall process of how I reached my ideal values for training the model with plaintext.

Once the model was trained and tuned, I ended with an accuracy of 68.467% and an error rate of 31.533%. In terms of machine learning algorithms, this doesn't seem like a good accuracy however given the size of the data and the complexity, it's clear that the model may need some feature engineering, or the data should be trained with a model that can handle such a complex dataset like SVM, but that is out of the scope of this project so I was satisfied with my results.

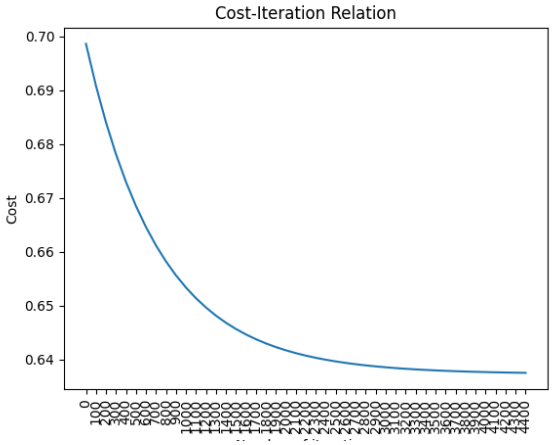

Figure [6] 4500 iterations, alpha = 0.001
Final Result

C. Algorithm A

Algorithm A is the real meat and potatoes of this experiment. For Algorithm A, I developed a logistic regression model that performs operations on data encrypted using the CKKS homomorphic encryption scheme. This means I had to adjust all my functions from my LogisticRegression class from Algorithm 0 to be able to operate on CKKSVectors from the

TenSEAL python library. CKKSVectors have their own built-in methods to perform operations on them without needing to be decrypted and they are meant for homomorphic encryption to preserve privacy which was my main motivation for choosing them over PyTorch tensors. Even though tensors are great for matrix operations and machine learning, my primary focus was preserving the privacy of the data, so it made sense to me to use just CKKSVectors.

I originally had the intention of using the Microsoft SEAL library, however I had an extremely difficult time trying to get it install and since the TenSEAL documentation on GitHub states that is is, "…built on top of Microsoft SEAL", I thought this would be a good alternative [2].

D. Preprocessing data pt. II

To preprocess the patient data, I wanted to use a separate file to simulate the idea of training a machine learning model without access to the actual data. Therefore, I made a file, EncryptedData.py, to process the output from the preprocessed plaintext (i.e. x_train, y_train, x_test, and y_test already extracted). I worked on the preprocessing for Algorithm A in tandem with the development of the actual model because I often found that I had to adjust certain parameters throughout the development process (more on this later).

The first part of preprocessing for the new model was to create a TenSEAL context which essentially defines the encryption scheme and its parameters. The parameters of the context include a scheme type, polynomial modulus degree, coefficient modulus bit size, degree of optimization, and a global scale.

a. Cyclotomic Polynomial

CKKS executes homomorphic computations by leveraging polynomial rings modulo a polynomial of a specific degree. That polynomial is known as the cyclotomic polynomial or the polynomial modulus degree. During the encoding process of CKKS, vectors are transformed into polynomials to make the encryption scheme more effictient [1]. The parameter, poly_modulus_degree (polynomial modulus degree) in my EncryptedData.py file is equal to 8192, so all CKKS operations are executed in the ring of polynomials modulo a polynomial of degree 8192.

However, this parameter comes with tradeoffs. A higher degree allows for larger ciphertexts and noise budget, allowing more computations. This grants a higher level of security by increasing the difficulty of Ring-LWE (Learning with Error), the mathematical principle behind CKKS, but also a higher computational overhead. Using a scheme that operates on an LWE principle creates a high amount of privacy over the data due to that fact that the LWE problem is notoriously as hard to crack as worst-case lattice problems that are secure against quantum computer attacks.[1]. Since 8192 is a very high polynomial degree, the encrypted data has a high level of security, but that means my logistic regression model has to be robust to complex computations and large datasets which will eventually require me to sample the large dataset into a smaller set that my machine is able to handle. This is the reasoning behind the sampling functionality in EncryptedData.py.

b. Degree of Optimization

The next parameter that I had to tune was the degree of optimization. I chose (-1) simply because that is standard practice. However, this parameter is also involved in balancing the utility vs privacy dilemma. The lower the

degree, the stronger the security, but greater overhead due to the more computationally insensitive operations. A high optimization degree can simplify the operations; however, the privacy of the data is reduced [2]. For users who prioritize efficiency, they may want to use a higher value, but considering I am trying to maximize the privacy levels of the patient data I stuck with (-1).

c. Coefficient Modulus Bits

The coefficient modulus bits simply define the bit sizes of the coefficient moduli to determine the precision and scale of the computations. Larger bit sizes give room for more accurate computations since more data can be encoded, but they limit the number of homomorphic operations since the noise budget would be reduced. Pretty much every operation in CKKS adds noise, so a large noise budget is crucial for being able to decrypt the outputs in the end.

I chose a list of values for my coefficient modulus bits because this allows the modulus to be distributed across the different values, mitigating attacks that would target a single modulus scheme. While training my updated logistic regression model, I even received an error that the encrypted data couldn't be manipulated further due to the depletion of modulus levels. Originally, my coeff_modulus_bits variable was defined using only three values, (i.e. [60, 40, 60]) and then I had to continuously update it until finally settling on [40, 21, 21, 21, 21, 21, 21, 40], giving my model the ability to run more iterations and perform more computations.

d. Global Scale

The global scale parameter of the CKKS context defines the scaling factor used in encryption and decryption and helps balance precision and efficiency. A large scale enables precise operations on the data which is why I chose 2**21 to scale the plaintext data before encryption. When encoding a vector into a polynomial for CKKS, the scaling factor is what defines the encoding precision for the binary representation of the number. Essentially, it is a way to represent real numbers as integers.

However, after looking more closely at my plaintext patient data, I really don't have any crazy large numbers that need to be scale by a number as large as 2**21. Looking back now, I could see this as a potential issue as to why I couldn't process my encrypted data at first on my machine since the values were too large. I basically took a very large dataset that was difficult for me to use even before encryption and made it unmanageably large. I definitely wish I could have spent more time tuning this parameter instead of having tunnel vision on the other parameters that I felt needed tuning to enhance the privacy of the data.

e. Galois Keys

The Galois keys I added to the context are what enable the deserialization of the stored encrypted data. This was useful for when I was testing the model initially and needed a way to store the encrypted features and labels without having to perform the encryption every time. I serialized the data and stored it locally on my machine to load in whenever I wanted to test a new operation or fix a bug in the model. The keys also allow encrypted vector rotation operations on ciphertexts, but after training the model with a smaller sample of the data I decided I don't actually need to serialize and store the encryptions because that was no longer the most time-consuming task.

E. Preprocessing data pt. III

a. Sampling

After selecting values for all context parameters, I loaded in the patient data from my preprocessData.py file. I originally tried to encrypt all the data and work with the full dataset directly, however, after encryption the data was simply way too large for my machine to repeatedly encrypt and was taking quite a long time. I then decided to save the encrypted data to local files on my machine, so I wouldn't have to encrypt the data again every time I wanted to test my model. However, my machine, once again, was not able to load in the encrypted files because they were so large. I ran out of RAM and virtual memory, so I decided the data needed to be sampled for me to train this new model. Using a sample size of 100, I grabbed the first 100 values from x_train, y_train, but maintained continuity by using a 9:1 ratio between testing and training data.

Sampling allowed me to fail fast when working out bugs in my model and tune my alpha and iterations parameters faster. However, it is worth noting that the difference between sample sizes for Algorithm 0 and Algorithm A could have contributed to the difference in the accuracies between the models.

b. Encrypting the data

Using the TenSEAL library and the context defined above, I used the build in ckks_vector function to encrypt x_train, y_train, x_test, and y_test for use in the model. (Might be worth noting, I attempted to create my own CKKS encryption class, however, I quickly realized that was not something that was feasible for me at this time and that there is definitely a library for CKKS for a reason.) For my EncryptedLogRegression class I passed my context, and encrypted x and y values to the constructor to begin training the model.

F. Developing Algorithm A

a. Constructor

Starting at the top, I created a class constructor that would initialize a few important values that I would be using throughout the model. Some functionality in the constructor worth noting would be defining number of features and samples before too much noise is added from the different homomorphic operations in the rest of the class. Here, is also where I initialize values of theta and b to act as the weights for each feature and the bias term added in later. I also initialized the gradients for theta and b as CKKSVectors for compatibility throughout the rest of the program.

b. Initializing Weights and Bias

This function is simply for initializing theta and b as CKKSVectors with the appropriate starting values of zero.

c. Sigmoid

For the sigmoid function that is typically used within logistic regression models, an exponential is used to compute the hypothesis. However, CKKS operates using a polynomial ring, therefore the sigmoid function needed to be converted to a polynomial approximation. I decided to use the Chebyshev series class because I have utilized Chebyshev for big data operations before and it is known for minimizing the maximum error and converging quickly, making it great for approximating the s-curve [5]. The Chebyshev import uses least square to fit the sampled y data at x [7].

I created a plot shown below to visualize the approximation, because during the debugging process I was skeptical if the approximation was creating errors that were propagating through the descent. However, Figure [7] shows the plot

of both the sigmoid and the approximation to be accurate.

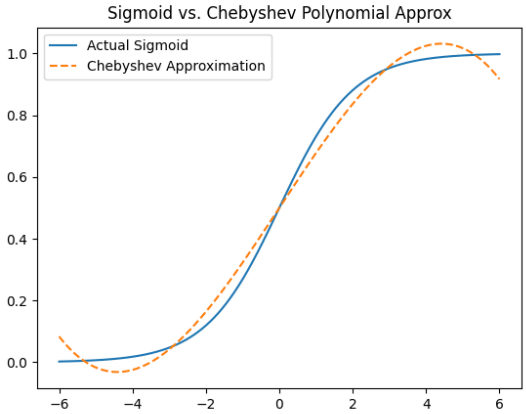

Figure [7] Plot displaying sigmoid and polynomial approximation used for homomorphic operations on dataset

d. Cost Function

This function uses the previously described sigmoid function to compute the hypothesis and compare it to the ground truth labels, calculating the prediction error. The gradients are updated and scaled here to be applied to theta and the bias term during the descent process.

Throughout this function there is minimal decryption but built in TenSEAL functions are used to encrypt the values that were decrypted for sub-operation functionality. Additional debugging would be necessary to work out all the kinks on the encrypted data, but for operations on CKKSVectors where I did not have the time, the values are temporarily decrypted.

f. Gradient Descent

The model performs gradient descent by updating the weights and bias using the gradients computed in the cost function and the learning rate, alpha. I plotted the cost value vs the iteration count to ensure the model was learning and the cost was decreasing. This allowed me to see where the cost started to plateau, similar to in Algorithm 0. Theta and b are updated using homomorphic operations of multiplication and subtraction on the encrypted values, preserving the integrity of the model's purpose.

I generated multiple plots throughout the tuning process of alpha and the iteration count to determine if the model was converging at a local minima and how I could get the smallest amount of iterations while preserving the accuracy.

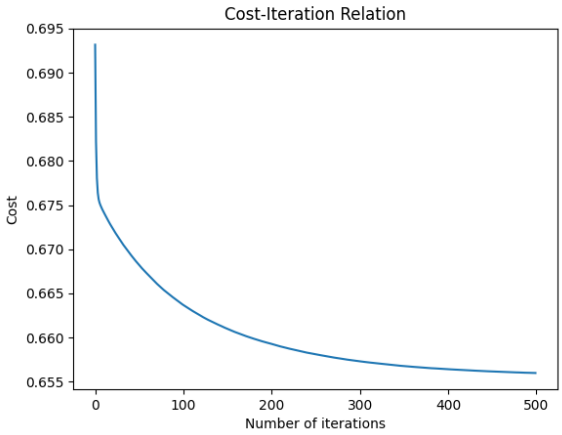

Figure [8] Iterations = 500, alpha = 0.5
Cost starting to plateau, but would like to see a few more iterations

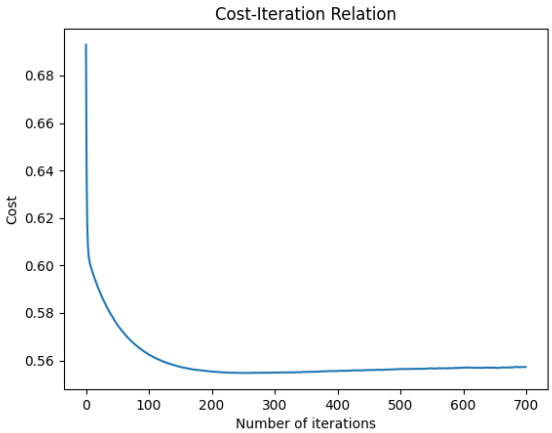

Figure [9] Iterations = 700, alpha = 0.5
Cost plateaus, want to try a smaller step size to see if the accuracy changes (FINAL)

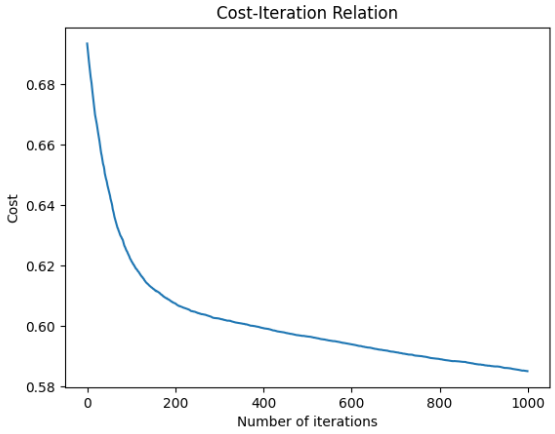

Figure [10] Iterations = 1000, alpha = 0.01
Cost still decreasing, want to try more iterations

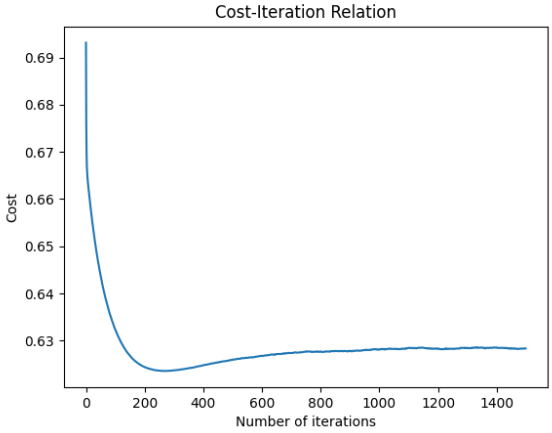

Figure [11] Iterations = 1500, alpha = 0.5
Cost starts to increase after too many iterations

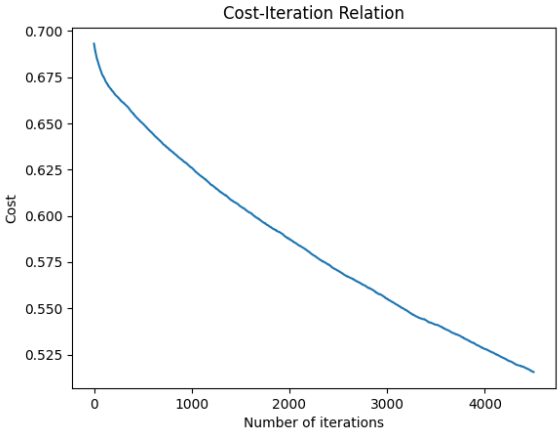

Figure [12] Iterations = 4500, alpha = 0.01
Tried same parameters as Algorithm 0

Figure [12] shows an attempt to use the same parameter values that tuned Algorithm 0, However, the plot clearly shows, the learning rate is way too small for the encrypted data.

I determined a larger learning rate of 0.5, helped the model converge faster and with only 700 iterations to achieve a cost lower than the cost in Algorithm 0. I was honestly quite surprised the encrypted data was able to have a faster learning rate and less iterations compared to the plaintext model. However, I believe this could be due to the sampling of x_train and y_train which decreases the complexity of the feature space. It may be fair to deduce that the plaintext model might have experienced some overfitting due to the complexity and size of the original data set.

F. Developing Algorithm B

Algorithm B is an inherited class from Algorithm A that implements bootstrapping in an attempt to improve accuracy and enhance privacy.

While developing Algorithm A, I ran into the problem of running out of available modulus levels which prevented any further manipulation of the encrypted data ("ValueError: end of modulus switching chain reached"). To combat this, I had to provide additional modulus coefficients, sample the data, and reduce the number of iterations. These modifications prevent the model from using all the samples to train on and also prevents the model from reaching a lower cost with the full-sized data because I was unable to perform additional epochs.

Bootstrapping enables the model to refresh the ciphertexts that have gone through complex computations. This is important because these computations add additional noise every time so refreshing the ciphertext is a way to decrease the amount of noise and make the data recoverable.

In the context of homomorphic encryption, each ciphertext can have an associated "level" and a value of "noise" added. During multiplication and addition operations, the level value associated with the ciphertext is decreased by one and we also increase the noise. After so many operations, the levels can no longer be decreased or the ciphertext is too noisy [9].

The mechanism behind bootstrapping is to return the ciphertext to a higher level. For CKKS, bootstrapping does not reduce the amount of noise because the plaintext already has noise added to it from encoding to approximate it as a real number. In classical encryption, the ciphertext is decrypted with a secret key as input. In fully homomorphic encryption, we use an encrypted secret key (bootstrapping key) and a ciphertext to create an "equivalent" ciphertext that can be operated on further. The figure below shows how bootstrapping isn't decryption (which would compromise the integrity of the patient data), it simply produces a ciphertext with a larger computational budget.

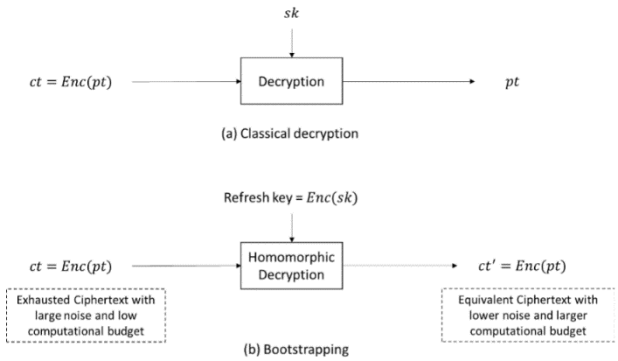

Figure [13] Classical decryption vs Homomorphic bootstrapping

Since CKKS is approximate and uses polynomial approximations to implement nonlinear functions, the bootstrapping for CKKS is also approximate. Meaning, the bootstrapped ciphertext is simply similar to the message before bootstrapping, but not equivalent [11]. Bootstrapping enhances the privacy of the data by allowing more computations on the encrypted data while remaining secure.

a. BootstrappedLogisticRegression class

The BootstrappedLogisticRegression class is an inherited class from EncryptedLogRegression() because it uses the same data and functions for consistency. The bootstrapped class proved just two additional functions from simulating the effect of bootstrapping. As previously stated, bootstrapping re-encrypts and operates on encrypted data to execute computationally expensive operations without exposing the plaintext data. The library I have been using TenSEAL, does not support full bootstrapping operations such as rotation and scaling. Therefore, I had to simulate approximate bootstrapping effects by sampling the data again and performing descent on those samples to iteratively approximate the statistical variability in training of the model.

This is not the way I wanted to bootstrap originally but due to the constraints on CKKSVectors and their supported operations, I just wanted to achieve some level of iterative refinement without compromising privacy.

b. Generate Bootstrap Samples

This method is used to generate seven different bootstrap samples using a random choice of indices with replacement. By using seven different samples from the original set, each sample provides the model with a different perspective on the data, increasing the variability. This aids in data privacy by using statistical estimates that don't reveal patterns in the original dataset or expose any data points.

Operating on these random samples reduces the risk of re-identification of patients in the set.

c. Training with Bootstrapping

This function is simply for passing the bootstrap samples to the inherited functions from the parent class to perform logistic regression.

The figures below show the cost function after a few bootstrap samples through the iterations

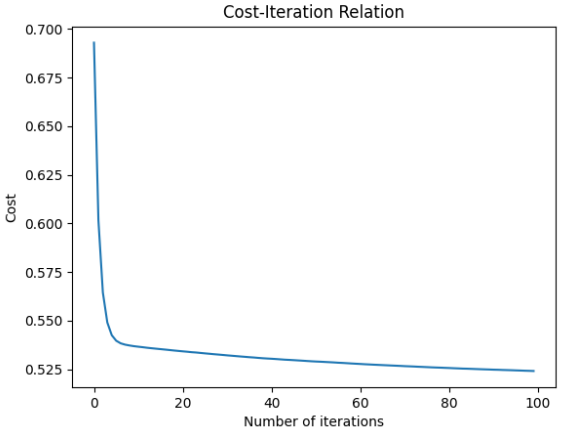

Figure [14] Cost function for sample 1. 100 iterations, alpha = 0.5

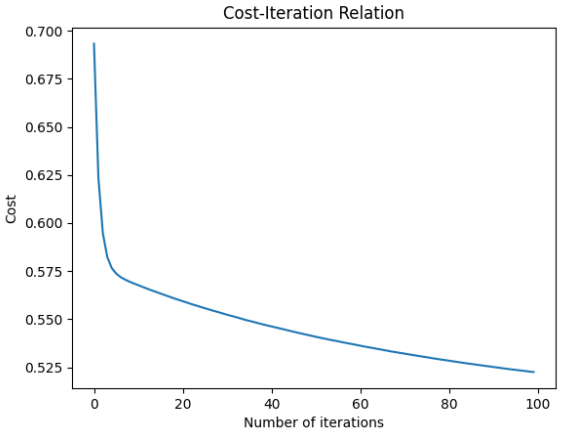

Figure [15] Cost function for sample 4. 100 iterations, alpha = 0.5

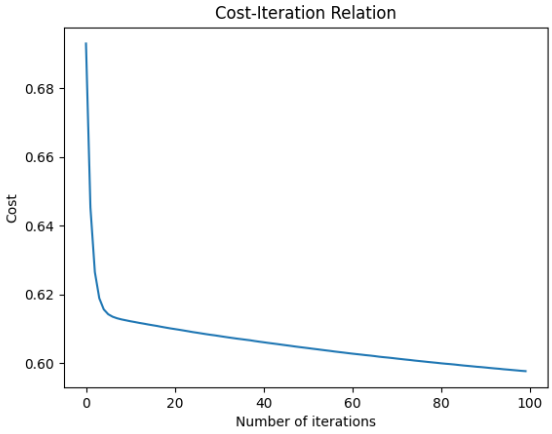

Figure [16] Cost function for sample 6. 100 iterations, alpha = 0.5

I turned the model to use the same parameters as the parent class, 700 iterations and alpha = 0.5. These figures only show 100 iterations due to timing constraints during the training process.

IV. EVALUATING ACURACY

I evaluated the accuracies of Algorithm 0, Algorithm A, and Algorithm B based off the final values of the tuning parameters that optimized each model. Algorithm 0 having the highest accuracy makes sense since CKKS is based off of polynomial approximations and since the logistic regression models also need to use polynomial approximations for computations that involved exponentials. Due to an accumulation of approximate values, the other algorithms were not able to exceed the accuracy of using the plaintext however, the difference with Algorithm B is not significant enough to dismiss the possibility of using homomorphic encryption to train machine learning models effectively.

Model	Alpha	Iteration Count	Accuracy	Error Rate	Sample
Algorithm 0	0.001	4500	0.68467	0.31533	13500
Algorithm A	0.5	700	0.64892	0.35	100
Algorithm b	0.5	700	0.663475	0.34	100

Figure [17] Highest accuracy outputs from tuned models

V. Evaluating Privacy

To evaluate the privacy beyond what has been discussed above, I have chosen to evaluate the entropy of the inputs to Algorithm 0 and Algorithm A. This will be a measurement of how unpredictable the data is and therefore more resistant to attacks. The figure below shows entropy values for the different inputs.

x_train0	y_train0	x_test0	y_test0	x_train_A	y_train_A	x_test_A	y_test_A
13.86	3.17	6.91	5.94	6.64	6.64	3.32	3.32

Figure [18] Entropy of inputs; plaintext[1-4], ciphertext[5-8]

Clearly, the table shows that the plaintext values had greater levels of entropy, however I believe this could be due to the sampling that was necessary for handling the encrypted data.

However, it is worth noting that even though the encrypted data training values were a fraction of the size of the original test set, they still maintained greater levels of entropy compared to the testing set for the plaintext data. Given that the encrypted test set is only 10% of the encrypted training set, a low entropy value makes relative sense here.

VII. Citations

[1] "CKKS Explained: Part 3 - Encryption and Decryption." *OpenMined Blog*, https://blog.openmined.org/ckks-explained-part-3-encryption-and-decryption/.

[2] "TenSEAL." *GitHub*, https://github.com/OpenMined/TenSEAL/tree/main.

[3] "A New Efficient Method for Bootstrapping in Homomorphic Encryption." *ePrint Archive*, https://eprint.iacr.org/2023/1788.pdf.

[4] "Tutorial 2 - Working with Approximate Numbers." *GitHub*, https://github.com/OpenMined/TenSEAL/blob/main/tutorials/Tutorial%202%20-%20Working%20with%20Approximate%20Numbers.ipynb.

[5] "On the Approximation of Functions by Polynomials." *Neural Network World*, http://www.nnw.cz/doi/2012/NNW.2012.22.023.pdf.

[6] "Chebyshev Polynomials." *NumPy Documentation*, https://numpy.org/doc/stable/reference/routines.polynomials.chebyshev.html.

[7] "pycanon." *PyPI*, https://pypi.org/project/pycanon/.

[8] "A Data Privacy Measure Based on Information Theory." *Nature*, https://www.nature.com/articles/s41597-022-01894-2.

[9] "What is Bootstrapping in Homomorphic Encryption?" *Zama*, https://www.zama.ai/post/what-is-bootstrapping-homomorphic-encryption.

[10] "Bootstrapping in Fully Homomorphic Encryption (FHE)." *DualityTech*, https://dualitytech.com/blog/bootstrapping-in-fully-homomorphic-encryption-fhe/.

[11] "Demystifying Bootstrapping in Fully Homomorphic Encryption." *Duality Tech*, https://www.dualitytech.com/blog/bootstrapping-in-fully-homomorphic-encryption-fhe/.

[12] Niu, Bo, et al. "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption." *ResearchGate*, https://www.researchgate.net/publication/350707182_TenSEAL_A_Library_for_Encrypted_Tensor_Operations_Using_Homomorphic_Encryption.

[13] "pycanon.anonymity.k_anonymity." *pycanon Documentation*, https://pycanon.readthedocs.io/en/latest/pycanon.anonymity.html#pycanon.anonymity.k_anonymity.