ISSN No:-2456-2165

# Anonymous Spam Detection Service Based on Somewhat Homomorphic Encryption

1st Ion Badoi
*Department of Applied Computer Science*
*Military Technical Academy "Ferdinand I"*
Bucharest, Romania

2nd Mihail-Iulian Plesa
*Department of Applied Computer Science*
*Military Technical Academy "Ferdinand I"*
Bucharest, Romania

**Abstract:- Because cloud-based services are becoming more and more used in the field of machine learning the issue of data confidentiality arises. In this paper, we address the problem of privacy-preserving spam classification. One of the most used algorithms for solving this problem is logistic regression. In this work, we suppose that a remote service has a pre-trained logistic regression model about which it does not want to leak any information. On the other hand, a user wants to use the pre-trained model without revealing anything about his mail. To solve this problem, we propose a system that uses somewhat homomorphic encryption to encrypt the user data and at the same time allows the service to apply the model without finding out any information about the user mail. The main contribution of this paper is a practical tutorial on how to implement the inference of a logistic regression model over encrypted data using the EVA compiler.**

***Keywords:-*** *Logistic Regression, Homomorphic Encryption, Privacy-Preserving, Spam Classification.*

## I. INTRODUCTION

Given the rapid development of cloud computing, we need to raise concerns about data privacy. Many companies offer machine learning services directly into the cloud. All the user has to do is upload his data to the cloud [1][2][3]. The model will be run over the uploaded data and a result will be returned to the user almost instantly. The problem of data privacy is a two-way problem. The company that provides the cloud services does not want to leak any information about the model to the user. On the other hand, the user does not want to give all its data to the company.

Currently, there are two major solutions for privacy-preserving machine learning. The first one proposes the use of secure multi-party computation. There are two parties involved: the user and the cloud service. The input of the user is the data, and the input of the service is the model. The goal is to use secure multi-party computations algorithms to compute the result of the machine learning algorithm without the user revealing his data or the service revealing the model. The second one proposes the use of homomorphic encryption. The main idea is that the user will encrypt his data using a homomorphic encryption scheme before sending it to the cloud. The service will apply the model over the encrypted data and return the encrypted result to the user. The user will decrypt and use the result. In this way the company will not leak any information about the model and the user will not reveal anything about the data.

In this work, we focus on privacy-preserving spam classification. The proposed solution is based on somewhat homomorphic encryption. We emulate the case in which the cloud service owns a pre-trained logistic regression model for spam classification and a user wants to use the service to classy his mail as spam or ham.

### A. *The system arhitecture*

Figure 1 shows the proposed architecture. The user performs the following series of steps:

1) *Text encoding:* Since the classification algorithm does not work directly on text documents we need to encode the mail as an array of integers.
2) *Encryption:* To ensure the confidentially of the mail, the user will encrypt the encoded text using his public key.
3) *Classification:* The user will send the encrypted mail to a remote server from the cloud. The server runs the classification algorithm over the encrypted data and will return to the user the encrypted result.
4) *Decryption:* The user will decrypt the result received from the server using his private key.

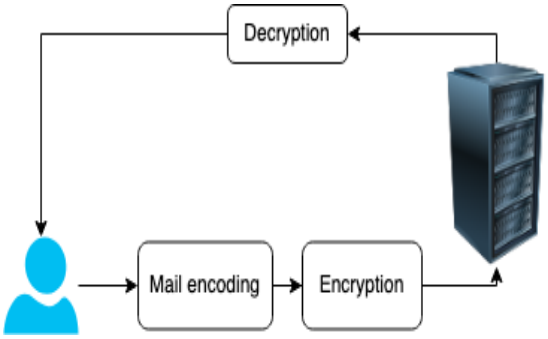


*Figure 1: The system architecture*

### B. *Related work and our contribution*

The classification algorithm used in this work is logistic regression. Due to its efficiency and simplicity, this algorithm is one of the first applications of homomorphic encryption in machine learning. In [4] the authors construct three major classification algorithms: hyperplane decision, Naïve Byes and decision trees from a privacy-preserving manner. The protocols that the authors proposed are low level and although they are efficient, they are not accessible to a programmer

without very thorough mathematical training. In [5] the authors show a new privacy-preserving protocol for logistic regression training and inference based on homomorphic encryption and secure multi-party computation. [6] shows a new approach that uses the Residue Number System to ensure the confidentiality of the data for the logistic regression algorithm. In [7] the authors proposed the use of fully homomorphic encryption to encrypt the data from the iDASH 2017 secure genome analysis competition. In [8] it is proposed a privacy-preserving spam filtering algorithm that uses functional encryption.

Although there are many solutions for privacy-preserving logistic regression, none of these shows how these solutions can be implemented in real software. We do not claim any new scientific novelty, but we contribute through an open-source system that emulates a cloud service that provides privacy-preserving spam classification. We describe a practical tutorial that shows step by step how to implement such a system and what are the latest technologies to do this.

## II. THE ENCRYPTION SCHEME

### A. The CKKS encryption scheme

The CKKS scheme was first introduced in 2017 during ASIACRYPT 2017 [9]. It is the first somewhat homomorphic encryption (SWHE) scheme designed especially for working with real numbers. Unlike fully homomorphic encryption (FHE) schemes, CKKS does not support an arbitrary number of computations. Another distinctive property of CKKS is that it is an approximate scheme. In general, an FHE or SWHE respects (1) and (2):

$$Enc(m_1) + Enc(m_2) = Enc(m_1 + m_1) \quad (1)$$

$$Enc(m_1) * Enc(m_2) = Enc(m_1 * m_1) \quad (2)$$

CKKS respects an approximative version of (1) and (2):

$$Enc(m_1) + Enc(m_2) = Enc(m_1 + m_1) + \varepsilon \quad (3)$$

$$Enc(m_1) * Enc(m_2) = Enc(m_1 * m_1) + \varepsilon \quad (4)$$

Although (3) and (4) approximates the requirements of an SWHE scheme, $\varepsilon$ is a small error that enables the scheme to be much more efficient than any other FHE or SWHE for computations on real numbers.

The scheme is based on the RLWE problem [10]. The plaintext space is the set $\mathbb{C}^{N/2}$. Each plaintext will be encoded into a polynomial belonging to the factor ring $\mathbb{Z}_q[X]/\phi_M(X)$ where $\phi_M(X)$ is the $M^{th}$ cyclotomic polynomial defined in (5), where $M = 2N$ and $N$ is a power of 2:

$$\phi_M(X) = X^N + 1 \quad (5)$$

As expected from an RLWE based scheme, the ciphertext space is $\mathbb{Z}_q[X]/\phi_M(X)^2$. After decryption, each plaintext will be decoded into $\mathbb{C}^{N/2}$. The secret key is a polynomial $s \in \mathbb{Z}_q[X]/\phi_M(X)$. The public key, $p$ is generated in (6) where $a \in \mathbb{Z}[X]/\phi_M(X)$ is a sampled uniformly and $e \in \mathbb{Z}_q[X]/\phi_M(X)$ is a small noisy polynomial:

$$p = (-as + e, a) \quad (6)$$

The encryption of the plaintext $m \in \mathbb{Z}_q[X]/\phi_M(X)$ generates the ciphertext $c = (c_0, c_1) \in \mathbb{Z}_q[X]/\phi_M(X)^2$ as in (7) where $b = -as + e$:

$$c = (m + b, a) \quad (7)$$

Similarly, the decryption of the ciphertext $c = (c_0, c_1) \in \mathbb{Z}_q[X]/\phi_M(X)^2$ using the secret key $s$ produces the plaintext $m \in \mathbb{Z}_q[X]/\phi_M(X)$ as in (8):

$$m = c_0 + sc_1 \quad (8)$$

Let $c = (c_0, c_1)$ and $c' = (c'_0, c'_1)$ be the ciphertexts of $m$ and $m'$. Let $c_{add}$ be the sum of $c$ and $c'$ defined in (9):

$$c_{add} = (c_0 + c'_0, c_1 + c'_1) \quad (9)$$

According to the decryption procedure defined in (8), $c_{add}$ is the encryption of the sum of the two plaintexts $m$ and $m'$ as it can be observed from (10):

$$(c_0 + c'_0) + s(c_1 + c'_1) = (c_0 + sc_1) + (c'_0 + sc'_1) = m + m' \quad (10)$$

Let $c_{const\_mul}$ be the ciphertext $c$ multiplied by the plaintext $m'$. This operation is defined in (11):

$$c_{const\_mul} = (c_0 m', c_1 m') \quad (11)$$

Similar to the addition of two ciphertexts, $c_{const\_mul}$ is the encryption of the product between $m$ and $m'$ as proved in (12):

$$c_0 m' + sc_1 m' = m'(c_0 + sc_1) = m'm \quad (12)$$

So far, we have proved that the scheme is homomorphic with respect to addition and constant multiplication. CKKS is also homomorphic with respect to ciphertext multiplication. Let $c_{mul}$ be the multiplication of $c$ and $c'$ defined in (13):

$$c_{mul} = (d_0, d_1, d_2) = (c_0 c'_0, c_0 c'_1 + c'_0 c_1, c_1 c'_1) \quad (13)$$

Obvious, $c_{mul}$ cannot be decrypted using (8). Intuitive, the decryption of $c_{mul}$ must be the product between the decryption of $c$ and the decryption of $c'$, that is the product of the plaintexts $m$ and $m'$. This product is defined in (14).

$$mm' = (c_0 + sc_1)(c'_0 + sc'_1) = c_0 c'_0 + s(c_0 c'_1 + c'_0 c_1) + s^2 c_1 c'_1 = d_0 + sd_1 + s^2 d_2 \quad (14)$$

The key idea to be able to decrypt $c_{mul}$ using (8) is the process of relinearization. The scheme defines an evaluation key, $\varepsilon$ as in (15) where $p$ is a big integer and $a_0, e_0 \in \mathbb{Z}_{pq}[X]/\phi_M(X)$:

$$\varepsilon = (-a_0 s + e_0 + ps^2, a_0) \; mod \; pq \tag{15}$$

The relinearization rewrites $c_{mul}$ as the sum of two valid ciphertexts i.e., ciphertexts that can be decrypted using (8), using the evaluation key as in (16) where the pair $(f_0, f_1)$ is defined in (17):

$$c_{relinearization} = (d_0, d_1) + (f_0, f_1) \tag{16}$$

$$(f_0, f_1) = (p^{-1} d_2(-a_0 s + e_0 + s^2), p^{-1} d_2 a_0) \; mod \; q \tag{17}$$

Decrypting the ciphertext $c_{relinearization}$, results in the product of the plaintext $m$ and $m'$ as proved in (18). Since $f_0 + sf_1$ is reduced modulo $q$ and $p$ is a big integer we can consider the error term $p^{-1} d_2 e_0$ small enough to be ignored.

$$\begin{aligned} d_0 + sd_1 + f_0 + sf_1 &= d_0 + sd_1 \\ &+ p^{-1} d_2(-a_0 s + e_0 + s^2) \\ &+ sp^{-1} d_2 a_0 \\ &= d_0 + sd_1 + s^2 d_2 + p^{-1} d_2 e_0 \\ &= d_0 + sd_1 + s^2 d_2 = mm' \end{aligned} \tag{18}$$

So, the scheme is homomorphic with respect to the operations of additions, constant multiplication and ciphertext multiplication. All these proprieties enable us to implement any kind of computation without revealing the actual data that is being computed on.

### *B. EVA Compiler*

The encryption scheme described in Section A, can be used in practice through the EVA compiler [11]. It is the first compiler for homomorphic encryption developed by Microsoft. At the time of writing, the compiler supports only the CKKS scheme. Although the CKKS encryption scheme is efficient due to the high degree of parallelism, the main practical disadvantage consists in the fact that the programmer must know many mathematical details which requires a long study. Due to this fact, only a limited number of people can create efficient software that implements homomorphic encryption. The EVA compiler solves this problem by allowing the programmer to write code in Python that processes encrypted data without directly selecting any cryptographic parameters. The programmer must select only two parameters: the output ranges and the input scales. Since the CKKS is an approximative scheme, these two parameters control the approximation error. There are several methods frequently used while programming with EVA.

The *compile* method receives as input an EVA program and returns the compiled program, the encryption parameters required by the scheme and a signature that specifies how inputs and outputs will be encoded and decoded. The compiled program can be visualized as a graph using a library such as Graphviz.

The *generate_key* method receives as input the encryption parameters returned by the *compile*. It returns the public key and secret key that will be used for data encryption and decryption.

The *encrypt* and *decrypt* methods receive as inputs the data to be encrypted or decrypted and the signature generated by the *compile*. The methods return either a ciphertext or a plaintext.

The *execute* method receives as input the compiled program and the encrypted inputs return by *encrypt*. The method run the homomorphic circuit over the encrypted data and return the encrypted outputs of that circuit.

The *decrypt* method receives as input the encrypted outputs produced by the *execute*. The method returns the decrypted output. Following the definition of a homomorphic encryption scheme, the decrypted outputs should be the same as the outputs of the circuit run over the plaintext data.

The *evaluate* method receives as inputs the compiled program and the plaintext inputs. The method returns the outputs of the circuit run directly over the plaintext data. This method is used together with the *valuation_mse* method that receives as inputs the decrypted outputs and the outputs obtained by running the circuit over the plaintext data. The method returns the MSE between the two outputs.

## III. IMPLEMENTATION

In this section, we provide a practical tutorial on how to implement privacy-preserving logistic regression classification using the EVA compiler. As far as we know this is the first paper proving a concrete implementation in Python of a machine learning algorithm over encrypted data. In the code we used the following global variables:

1) $num\_samples$: The number of samples used to test the model.
2) $num\_features$: The number of features of each sample
3) $X\_test$ : A matrix with $num\_samples$ rows and $num\_features + 1$ columns containing all testing samples
4) $y\_test$ : A binary raw vector of dimension $num\_samples$ containing the ground truth labels of each test sample
5) $X\_train$ : A matrix with $num\_samples$ rows and $num\_features$ columns containing all training samples
6) $y\_train$ : A binary raw vector of dimension $num\_samples$ containing the ground truth labels of each training sample
7) $y\_prob\_predict$ : A raw vector of dimension $num\_samples$ containing the predicted probability of each test sample to be spam

8) $y\_pred$ : A raw vector of dimension $num\_samples$ containing the predicted class of each test sample
9) $weights:$ A raw vector of dimension $num\_samples$ containing the weights produces by training the logistic regression model

*A. Data preprocessing*

The first step is to train a logistic regression model to classify emails as ham or spam. To do this, we use the sklearn library. The dataset used is Spambase Data Set from UCI Machine Learning Repository [12]. The dataset is split into training and testing:

```
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size = 0.2, random_state = 10)
```

To extract features, we use the bag of words representation of a text document. The first step is to tokenize the text and assign an integer identifier to each token. The second step is to count the frequency of each token. We consider a feature to be the frequency of a token and a sample to be the vector of the frequencies of all tokens that characterize a text. In the code presented in this paper, the number of tokens is retained in the variable $num\_features$. The dataset is represented by a matrix where each row is a vector of frequencies. In practice, an email uses very few unique tokens resulting in a sparse matrix. We also standardize the data by extracting the mean and diving by the standard deviation. There are two distinct parameters of the logistic regression model resulted from the training process: the intercept and the coefficients. Given the fact that both types of parameters will be encrypted, the function *logistic_regression_train_plaintext* returns the concatenation between the coefficient and the intercept:

```
def logistic_regression_train_plaintext(X_train, y_train):
    model = LogisticRegression(random_state=0)
    model.fit(X_train,y_train)
    return list(model.intercept_) + list(model.coef_[0])
```

Symbolically, the result returned by the function is retained in the $weights$ variable:
`weights = logistic_regression_train_plaintext(X_train, y_train).`

Formally, applying the model over the test data involves the multiplication between the $X\_test$ matrix and the $weights$ vector. Initially, after splitting the data, $X\_test$ matrix has $num\_features$ columns. The $weights$ vector returned by the training function has a length of $num\_features + 1$ due to the concatenation between the $num\_features$ coefficients and the intercept. To include the intercept when applying the model, we prepend a column of ones to the $X\_test$ so that the matrix has $num\_features + 1$ columns. To test the model by classifying both spam and ham emails, we select a balanced dataset of $num\_samples$ from the $X\_test$. The function *select_balanced_subset* implements the selection of $num\_samples$ balanced samples:

```
def select_balanced_subset(X, y):
    X_tmp = np.zeros((num_samples, num_features+1))
    y_tmp = np.zeros((num_samples, 1)).astype(int)
    positive_cnt = 0
    negative_cnt = 0
    cnt = 0
    for i in range(len(X)):
        if y[i] == 1 and positive_cnt < num_samples//2:
            for j in range(num_features+1):
                X_tmp[cnt][j] = X[i][j]
            y_tmp[cnt] = y[i]
            positive_cnt+=1
            cnt += 1
        if y[i] == 0 and negative_cnt < num_samples//2:
            for j in range(num_features + 1):
                X_tmp[cnt][j] = X[i][j]
            y_tmp[cnt] = y[i]
            negative_cnt+=1
            cnt += 1
        if negative_cnt+positive_cnt == num_samples:
            break
    return X_tmp, y_tmp
```

The function *select_balanced_subset* is called inside the *prepare_test_data* function right after we append a column of ones:

```
def prepare_test_data(X_test, y_test):
    X_test = np.hstack((np.ones((len(X_test), 1)), X_test))
    X_test, y_test = select_balanced_subset(X_test, y_test)
    return X_test, y_test
```

After all the above procedures, the $X\_test$ dataset used for testing will have $num\_samples$ samples each with $num\_features + 1$ features:
`X_test, y_test = prepare_test_data(X_test, y_test)`

*B. EVA programming*

The first step taken when using the compiler is to create a dictionary with all inputs that must be encrypted. In our system, we only encrypted the test data i.e., the $X\_test$ matrix so we create a dictionary with one key, “data” and one value, the test data:

```
def make_eva_dictionary(X):
    data = X.flatten()
    return {'data': data}
```

The data to be encrypted is retained in the $inputs$ variable:
`inputs = make_eva_dictionary(X_test)`

The multiplication between the encrypted $X\_test$ matrix and the $weights$ vector involves the calculation of $num\_samples$ dot products between each encrypted row of the $X\_test$ matrix and the $weights$ vector. When multiplying an encrypted list of data with a plaintext, EVA multiplies each element of the list with the plaintext. In other words, we cannot control what elements of the list are multiplied with the plaintext. To address this problem, we proposed the following algorithm for the dot product between an encrypted vector, $vector1$ and a plaintext vector, $vector2$:

```
def dot_product(vector1, vector2, vector_len):
    const_zeros = [0] * vector_len
    const_zeros[0] = 1
```

ISSN No:-2456-2165

```
for i in range(vector_len):
    rotated = vector1 << i
    partial = rotated * vector2[i]
    if i == 0 :
        result = partial
    else:
        result += partial
result = result * const_zeros
return result
```

After the rotation with $i$ positions to the left, the element that is in the position $i$ in the encrypted vector reaches the first positions. We then multiply the whole vector with $vector2[i]$ and add the result to the variable $result$. In this way, we multiply $vector1[i]$ with $vector2[i]$ and add the partial results thus calculating the dot product. The $result$ will be a vector of $vector\_len$ elements in which on the first position we find the dot product between $vector1$ and $vector2$. Although the EVA compiler does not allow us to extract the first element from the $result$, we can make all elements except the first equal to zero by multiplying $result$ with a const vector, $const\_zeros$ in which the first element is one and the rest are zero.

While using EVA, we cannot store a matrix as a bidimensional array but as a raw vector. When multiplying a matrix by a raw vector, we must use the same mechanism of rotations to calculate the dot products. Suppose the matrix has $num\_rows$ rows and $num\_columns$ columns. The elements belonging to the row number $i$ from the matrix will be found from position $i * num\_columns$ to position $(i+1) * num\_columns$. Since we cannot extract separately each row, we will shift by $i * num\_columns$ positions to the left the vector that represents the matrix and multiply this vector to a constant vector which has the first $num\_columns$ equal to one and the rest of $(num\_rows - 1) * num\_columns$ elements equal to zero. In this way, we make each row in the matrix to be found one by one at the beginning of the vector that represents the matrix. We then calculate the dot product between this vector of encrypted entries and the plaintext vector. The resulted dot product will be shifted $i$ positions to the right and then will be added to the variable $result$. In this way, at the position $i$ on the vector $result$ we found the dot product between the row number $i$ of the original encrypted matrix and the plaintext vector. Since the vector $result$ will be as long as the vector that represents the matrix i.e. $num\_rows * num\_columns$, only the first $num\_rows$ will be occupied while the rest will be equal to the encryption of zero.

```
def matrix_vector_multiplication(matrix, num_rows, num_columns, vector):
    const_zeros = np.zeros(num_rows * num_columns).astype(int)
    const_zeros[:num_columns] = 1
    const_zeros = list(const_zeros)
    for i in range(num_rows):
        row = matrix << i*num_columns
        row = row * const_zeros
        dot = dot_product(row, vector, num_columns)
        if i == 0:
            result = dot
        else:
            result += dot>>i
    return result
```

With the above function, we can compute the product between a matrix and a vector thus we can multiply the matrix $X\_test$ with the vector $weights$. According to the logistic regression algorithm, to obtain the probability that a given sample email is spam we must apply the sigmoid function to the dot product between the feature vector that characterizes an email and the vector $weights$. In our case, we must apply the sigmoid function to each element of the vector resulting from the multiplication between $X\_test$ and $weights$. Since the data from $X\_test$ are encrypted so will the result of multiplication with $weights$. EVA compiler does not allow any operations other than multiplications and additions thus we must use the Taylor approximation of the sigmoid function which is given in (19):

$$\sigma(x) \approx \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^3}{480} \tag{19}$$

Equation (19) is implemented by the function *apply_aprox_sigmoid* which uses the default Python operators for exponentiation, multiplication, and addition:

```
def apply_aprox_sigmoid(x):
    return 1/2+x*(1/4)-(x**3)*(1/48)+(x**5)*(1/480)
```

All processing over the encrypted data is done inside an EVA program. When instantiating an EVA program, we need to specify its name and the size of the encrypted input. In our case, the name of the program is "encrypted_logistic_regression". Since the encrypted data consist of the matrix of features $X\_test$, the size of the input is the total number of elements from this matrix i.e., $num\_samples * (num\_features + 1)$. Inside the program, we identify the encrypted input by the dictionary we have created. Given the encrypted features matrix, all the EVA program does is to first call the *matrix_vector_multiplication* function to multiply each sample with the vector $weights$ and then call *apply_aprox_sigmoid* function to apply the approximate sigmoid function to each element of vector resulted from multiplication:

```
encrypted_logistic_regression = EvaProgram('encrypted_logistic_regression', vec_size=num_samples*(num_features+1))
with encrypted_logistic_regression:
    data = Input('data')
    data = matrix_vector_multiplication(data, num_samples, num_features+1, weights)
    data = apply_aprox_sigmoid(data)
    Output('data', data)
```

To use the program stated above, we need to compile it:

```
compiler = CKKSCompiler()
compiled, params, signature = compiler.compile(encrypted_logistic_regression)
```

Following the compilation process results the compiled program that will be run, the parameters of the encryption scheme and the signature of the program. Based on the parameters, we use *generate_keys* to generate the public and the private key:

```
public_ctx, secret_ctx = generate_keys(params)
```

Given the public key and the program signature we encrypt the inputs:

```
enc_inputs = public_ctx.encrypt(inputs, signature)
```

Once we have the encrypted inputs, we can run the EVA program:

```
enc_outputs = public_ctx.execute(compiled, enc_inputs)
```

The EVA program calculates the probability that each test mail to be spam. Since the program runs over encrypted data, the outputs will also be encrypted. To get the plaintext results, we need to decrypt the *enc_outputs* using the private key:

```
outputs = secret_ctx.decrypt(enc_outputs, signature)
```

To see the error due to the approximations made in the encryption scheme, we run the functions *evaluate* and *valuation_mse*:

```
reference = evaluate(compiled, inputs)
print('MSE', valuation_mse(outputs, reference))
```

The outputs variable represents a vector in which each element is the probability that a test mail to be spam. To classify the mail as spam or ham we compare each probability with the standard threshold of 0.5. We use the sklearn function *accuracy_score* to calculate the accuracy of classification made over the encrypted data:

```
y_pred = []
for i in range(num_samples):
  if outputs['data'][i] < 0.5:
    y_pred.append(0)
  else:
    y_pred.append(1)
print('Model acc decrypted: ', accuracy_score(y_pred, y_test))
```

## IV. EXPERIMENTS

In this section, we made a series of experiments using the implementation described in Section III.

Figure 2 shows the squared difference between the decrypted probabilities obtained using EVA programming and the probabilities given by the logistic regression algorithm applied directly over the plain data. As it can be seen, the difference is almost 0 except for a few test samples. This means that the error due to the approximations inside the encryption scheme does not affect the practical results.

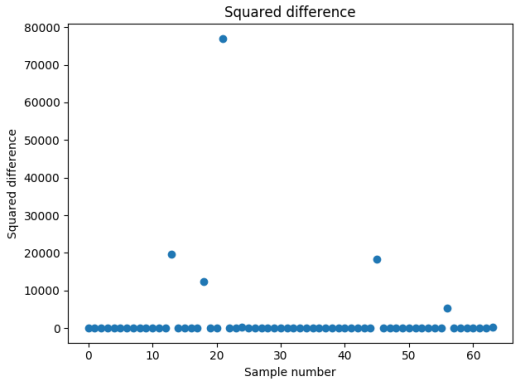


*Figure 2: Squared difference for each test sample*

The most important errors come from the approximation of the sigmoid function. Figure 3 present the approximation of the sigmoid function. The approximation does not work for points that are not in a close neighborhood of 0.

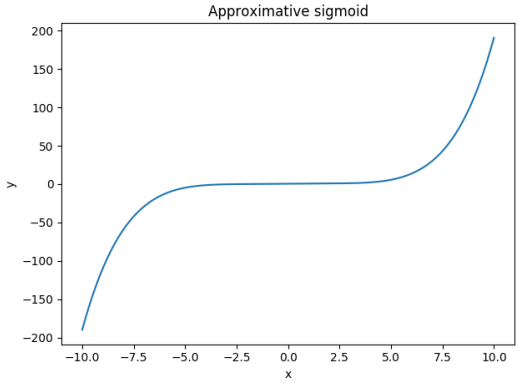


*Figure 3: The approximation of the sigmoid function*

To solve this problem, we can use more terms in the Taylor expansion, but this strategy implies a much deeper circuit that needs to be homomorphically evaluated. Although we cannot recover the exact probabilities returned by the logistic regression algorithm over the encrypted data, this does not mean that the accuracy of the model is affected. Over 64 test samples, the algorithm run over plain data has an accuracy of 73% which is the same as the accuracy obtained for the decrypted results.

## V. CONCLUSIONS AND FURTHER DIRECTIONS OF RESEARCH

In this paper, we presented the architecture of a cloud-based service that provides privacy-preserving spam detection. The service owns a pre-trained logistic regression model which it wants to expose without leaking any information about the model parameters. The user, on the other hand, has a set of emails that he wants to keep confidential. To meet the conditions of both the service and the user we have implemented a system based on somewhat homomorphic encryption in which the user encrypts his mail, and the server processes it in the encrypted form. For encryption, we used the CKKS scheme. All computations made over encrypted data were implemented using the EVA compiler.

The main contribution of this work is a practical tutorial on how to implement the inference for logistic regression over encrypted data. The core of applying a plain logistic regression model over a set of encrypted samples is the multiplication between an encrypted matrix and a plain vector. Although the EVA compiler allows the implementation of operations over encrypted data, they consist only of additions and multiplications. Due to this fact, one challenge was to compute the sigmoid function by using its Taylor expansion. The experiments showed that computing over encrypted data does not affect the accuracy of the model.

A further direction of research is to implement the training of a logistic regression model over encrypted data. This involves implementing the gradient descent algorithm over encrypted data which can be done with homomorphic encryption since every training iteration consists of an addition and a multiplication.

## REFERENCES

[1]. "Azure Machine Learning - ML as a Service: Microsoft Azure," *ML as a Service | Microsoft Azure*. [Online]. Available: https://azure.microsoft.com/en-us/services/machine-learning/. [Accessed: 01-Mar-2021].
[2]. "AI Platform | Google Cloud," *Google*. [Online]. Available: https://cloud.google.com/ai-platform. [Accessed: 01-Mar-2021].
[3]. T. M. Mitchell, "Machine learning," *Amazon*, 2017. [Online]. Available: https://aws.amazon.com/machine-learning/. [Accessed: 01-Mar-2021].
[4]. R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine Learning Classification over Encrypted Data," *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.
[5]. H. Kikuchi, H. Yasunaga, H. Matsui, and C.-I. Fan, "Efficient Privacy-Preserving Logistic Regression with Iteratively Re-weighted Least Squares," *2016 11th Asia Joint Conference on Information Security (AsiaJCIS)*, 2016.
[6]. J. M. Cortés-Mendoza, A. Tchernykh, M. Babenko, L. B. Pulido-Gaytán, G. Radchenko, F. Leprevost, X. Wang, and A. Avetisyan, "Privacy-Preserving Logistic Regression as a Cloud Service Based on Residue Number System," *Communications in Computer and Information Science*, pp. 598–610, 2020.
[7]. H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter, "Logistic regression over encrypted data from fully homomorphic encryption," *BMC Medical Genomics*, vol. 11, no. S4, 2018.
[8]. S. Jaiswal, S. C. Patel, and R. S. Singh, "Privacy Preserving Spam Email Filtering Based on Somewhat Homomorphic Using Functional Encryption," *Advances in Intelligent Systems and Computing*, pp. 579–585, 2015.
[9]. J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," *Advances in Cryptology – ASIACRYPT 2017*, pp. 409–437, 2017.
[10]. V. Lyubashevsky, C. Peikert, and O. Regev, "A Toolkit for Ring-LWE Cryptography," *Advances in Cryptology – EUROCRYPT 2013*, pp. 35–54, 2013.
[11]. R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
[12]. *UCI Machine Learning Repository: Spambase Data Set*. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Spambase. [Accessed: 20-Feb-2021].