MPC extension to Prac2PC

Siddhant Mago 2017CS50419, Ashish R Nair 2017CS50521,

Rahul Yadav 2017CS50602

May 2021

1 Introduction

Air pollution is one of the main problems faced by Delhi today. Cab companies have a growing vehicle fleet which they are willing to instrument for measurements of traffic, air pollution etc. While the data from these companies can be very beneficial to analyse, this data can also give away the vehicle location and the number of vehicles in their fleet. To overcome this, past work (Prac2PC) looked at using secure 2-party computation like Yao's garbled circuits in order to preserve sensitive information as well as answer many practical queries. It allows a server to answer queries from users without providing exact data. In this project we extended this to multi party computation, so that different available servers can answer a query together and also jointly train on their data.

2 Problem Description

As stated earlier, the problem involves responding to client queries based on pollution data collected by various cab and delivery companies for a particular city. The client is interested in knowing the level of pollution on their route along with how reliable the information is. At the same time, the client doesn't want to share their exact location. Similarly, the companies want their sensitive data to be secure as it is possible to estimate their fleet location using confidence scores. The city is modelled as a grid with pollution values available for some grid points. Each company has a different set of pollution values for these points which are updated periodically. For the points where pollution values are not available we perform interpolation. There are two ways to do this — either each company can interpolate independently on its own data and the results from all companies can be combined or the interpolation can be performed jointly over the combined data all companies collectively have. The next step is to answer client queries using these interpolated values. The queries could be average or maximum pollution value on a path etc. Note that all joint computation must happen securely.

2.1 Prac2PC

In Prac2PC, a client could ask queries from only a single server. The client did so by engaging in a 2 party computation with the server by using Yao's garbled circuits. The server only

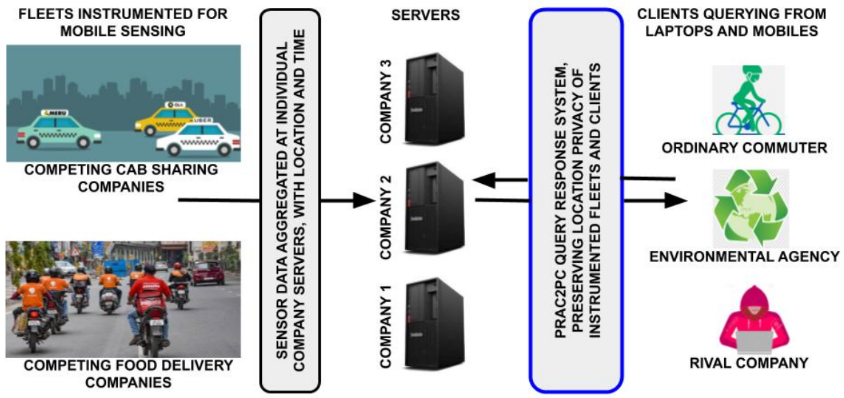


Figure 1: Overall Architecture

performed interpolation on its own data which meant that these 2-party queries tend to be accurate only in areas where that particular cab company's fleet is located.

2.2 Extending Prac2PC to PracMPC

2-party secure communication protocols used in Prac2PC do not work for the multi-party setting and we needed to explore and use MPC protocols here. In the next section we will talk about the protocols used. We proceed with the project in two steps:

- We first only perform secure aggregation and query answering, where each server does interpolation independently on its own data. Here, we first need to merge pollution predictions from different servers for the same points and then answer client pollution queries. 1. independent interpolation 2. secure aggregation

- Then we extend this to secure interpolation across servers, where servers collectively interpolate their data and then securely answer queries. This problem was absent in the 2PC setting as there was a single server which did the interpolation. Also, joint interpolation may lead to better predictions for unknown locations. 1. joint interpolation

3 Secure MPC Background

Secret sharing based protocols are popular techniques to achieve secure multi-party computation. In our project use two libraries, namely `MPyC` and `Crypten` which are based on secret sharing protocols. In this section we talk about the various protocols used by these libraries.

Shamir's Secret Sharing

Let $x \in \mathbb{F}$ be a secret. Shamir's secret sharing ensures that any t out of n parties can recreate the original secret but $t-1$ parties cannot. The way this is done is by picking a random $t-1$ degree polynomial f such that $f(0) = x$. Then the secret share that is given to party p is $[x]_p = f(p)$, $p \in \{1, 2, \ldots, n\}$.

If t parties combine their shares then they can interpolate their shares and recreate x as

$$f(0) = x = \sum_{i=1}^{t} \lambda_{p_i} [x]_{p_i}$$

where λ_{p_i} are coefficients given by

$$\lambda_{p_i} = \frac{\prod_{j=1, j\neq i}^{t} (0 - p_j)}{\prod_{j=1, j\neq i}^{t} (p_i - p_j)}$$

BGW Protocol (MPyC)

`MPyC` uses the BGW protocol to enable n parties to jointly evaluate a publicly known function without revealing their respective inputs in a way that is secure against an honest majority Let the function to be evaluated be $g(x_1, x_2, \ldots, x_n)$ where x_p is ithe input of party p and the arithmetic circuit that evaluates g is known to all the parties. Shamir's secret sharing is used with $t = \lfloor (n-1)/2 \rfloor$ to create shares for encrypted values.

Each party p picks a random t degree polynomial f_p, where $f_p(0) = x_p$ and sends the shares $[x_p]_q = f_p(q)$ to all other parties q. Each party now has a vector of shares. The protocol defines how sharing works for addition gates, multiplication gates and multiplication-by-constant gates.

Addition Gates For an addition gate $c = a + b$, the arguments a and b are first shared using Shamir's scheme in a $(t+1)$-out-of-n fashion. Thus, each party p has the shares $[a]_p$ and $[b]_p$ corresponding to polynomials f_a and f_b such that $[a]_p = f_a(p)$ and $[b]_p = f_b(p)$. Then each party individually calculates their share $[c]_p = [a]_p + [b]_p$. Each party now holds a point on the polynomial $f(x) = f_a(x) + f_b(x)$. Since this polynomial also has degree at most t and $f(0) = f_a(0) + f_b(0) = a + b = c$, these are valid shares of c. The multiplication-by-constant gate can be implemented similarly where each party locally multiplies their share by the constant.

Multiplication Gates Consider a multiplication gate $c = ab$ where the shares $\{[a]\}_{p=1}^{n}$ and $\{[b]\}_{p=1}^{n}$ are shared to all parties. The parties first locally multiply their shares of a and b to each get a point on the polynomial $q(x) = f_a(x) \cdot f_b(x)$. The resulting polynomial can be of degree atmost $2t$ so some extra steps are required to obtain a sharing of c with degree atmost q. To do this, it is important to note that $c = q(0)$ can be written as a linear combination of the party's shares as

$$q(0) = \sum_{i=1}^{2t+1} \lambda_{p_i} q(p_i)$$

where λ_{p_i} are the appropriate Lagrange coefficients.

The protocol then dictates that $2t+1$ parties generate a sharing of their values of $q(p)$. Each party can then locally compute their share $[q(0)]_p = \sum_{i=1}^{2t+1} \lambda_{p_i} [q(p_i)]_p$

Arithmetic Secret Sharing (Crypten)

`Crypten` uses pseudo-random zero share (PRZS) in order to share values. Let the value to be shared be x, such that $x \in \mathbb{Z}/Q\mathbb{Z}$ where $\mathbb{Z}/Q\mathbb{Z}$ is a ring with Q elements. Suppose we have to share the value x with $p \in \mathcal{P}$ parties. Let the shares of x be denoted by $\{[x]_p\}_{p\in\mathcal{P}}$, where $[x]_p$

denotes the share of party p.
So to share the value x, the parties first generate a pseudo-random zero share (PRZS), having $|\mathcal{P}|$ random numbers, by the property of PRZS these shares sum to zero. The party which owns x adds x to its share of PRZS and then discards x.
Now, to recover x, it is clear that $x = \sum_{p \in \mathcal{P}} [x]_p \mod Q$.

Beaver Triples (Crypten)

Beaver Triples is a secret sharing based method of multiplying two integers.
Let us assume we have multiply shares of two integers x and y. Let [x] denote the secret share of integer x. We have to compute [z] such that $z = xy$.
Suppose parties have a secret sharing of a random product: [a], [b], [c] such that $c = ab$.
So, now each party also has [x] and [y], the protocol is as follows:

- Each party computes $[x - a] = [x] - [a]$ and $[y - b] = [y] - [b]$ and publishes this result. This all the parties know $x - a$ and $y - b$ but as a and b are random integers they don't know x or y
- All parties compute $[z]$ as:

$$
\begin{gathered}
[z] = [c] + [x](y-b) + [y](x-a) - (x-a)(y-b) \\
\text{So, } z = c + x(y-b) + y(x-a) - (x-a)(y-b) \\
z = c + xy - xb + xy - ay - xy + ay + xb - ab = xy
\end{gathered}
$$

So using a random multiplicative triple, we can secretly multiply two numbers. This random multiplicative triple can be generated by a Trusted third Party(TTP) or by Oblivious Transfer(OT). `Crypten` uses a TTP to do this currently but support for using OT instead is expected to arrive soon.

4 Answering queries by merging interpolated grids

Let $p_i(x, y)$ be the i^{th} server's predicted pollution value at grid location (x, y) and $\sigma_i(x, y)$ be its confidence at the same point. Before answering the client's query, the n servers must combine their predictions. Let $\hat{p}(x, y)$ and $\hat{\sigma}(x, y)$ be these aggregated values. We use inverse-variance weighting to estimate $\hat{p}$ and $\hat{\sigma}$ since this gives the least variance estimate among all weighted averages. These are evaluated as below.

$$
\boxed{\hat{p}(x, y) = \frac{\sum_i^n p_i(x, y)/\sigma_i(x, y)^2}{\sum_i^n 1/\sigma_i(x, y)^2} \qquad \hat{\sigma}^2(x, y) = \frac{1}{\sum_i^n 1/\sigma_i(x, y)^2}}
$$

To perform the above computation, each server distributes a secure share of their values to the other servers and then performs a sequence of secure arithmetic operations to obtain shares of the aggregated values, $\hat{p}$ and $\hat{\sigma}$. This ensures that the servers don't reveal any sensitive information.

After we have these merged grids, we can perform query operations on the same with the client as another party. The client first specifies a sub-grid which includes its query route. The client then securely provides a binary mask on this sub-grid which describes its route. This mask is the client's input in the query computation and the sub-grid with pollution values is the input of the servers. The queries may be finding the average and maximum pollution in query path or counting instances when pollution is greater than a threshold. The entire pipeline is shown in figure 2. How their queries are replied to is described in algorithm 4.1. The queries *min* and *range* can also be dealt with similarly. All values here are shared secrets with computation happening in parallel in all parties involved. As described in the protocols described earlier, some computations may also require communication between the parties. A restriction we face is that we cannot use conditionals as intermediate results are stored as shared secrets and cannot be decrypted by the parties to decide which branch to take.

Algorithm 4.1: REPLY$(query, \hat{p}, \hat{\sigma})$

$(sum, pathLength, sumVariance, max, count) \leftarrow (0, 0, 0, 0, 0)$

comment: average pollution along path

for each $(x, y) \in query.subgrid$

$$\textbf{do} \begin{cases} onQueryPath \leftarrow query.mask(x, y) \\ sum \leftarrow sum + onQueryPath * \hat{p}(x, y) \\ sumVariance \leftarrow sumVariance + onQueryPath * \hat{\sigma}(x, y)^2 \\ pathLength \leftarrow pathLength + onQueryPath \end{cases}$$

$avg \leftarrow sum/pathLength$

$avgConfidence \leftarrow \text{SQRT}(sumVariance)/pathLength$

comment: maximum pollution along path

for each $(x, y) \in query.subgrid$

$$\textbf{do} \begin{cases} onQueryPath \leftarrow query.mask(x, y) \\ isGreaterThanMax \leftarrow \hat{p}(x, y) > max \\ max \leftarrow isGreaterThanMax * \hat{p}(x, y) + (1 - isGreaterThanMax) * max \end{cases}$$

comment: count of points with pollution greater than threshold

for each $(x, y) \in query.subgrid$

$$\textbf{do} \begin{cases} onQueryPath \leftarrow query.mask(x, y) \\ isGreaterThanThreshold \leftarrow \hat{p}(x, y) > query.threshold \\ count \leftarrow count + isGreaterThanThreshold \end{cases}$$

return $(avg, avgConfidence, max, count)$

We try the two secure MPC libraries, `MPyC` and `Crypten` for the above setup. The different parties are simulated on a single machine using inter-process communication. The following sub-sections discuss the key-points and results using libraries.

4.1 MPyC

Some important points about the `MPyC` library:

- It implements the BGW protocol.

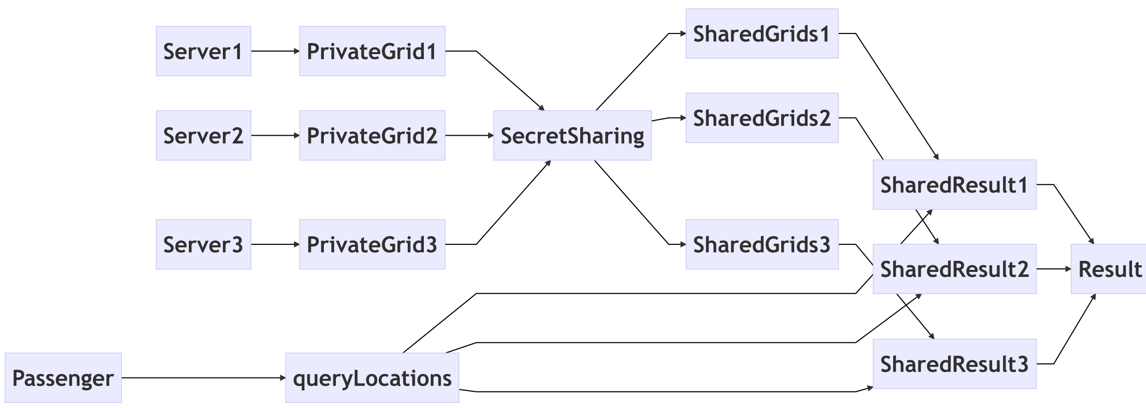


Figure 2: Query answering pipeline by merging interpolated grids

- It is not suited for ML and gradient based learning.
- It is slow as the implementation is entirely in python and doesn't use any optimizations.

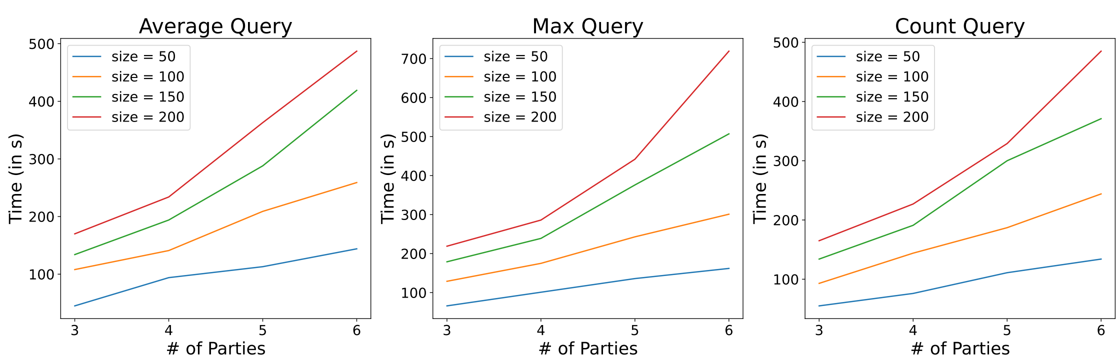


Figure 3: Timings of different queries with `MPyC`

Timing Results

As is expected, one can see the running time increase as the number of parties and the number of sub-grid points increase. Answering each query takes an order of a few minutes. We see that these running times are much slower that what we would need for real deployment as when run on multiple servers the timings would get much worse. The error in results generated by `MPyC` compared to a single process implementation were negligible.

4.2 CrypTen

`Crypten` uses secret sharing protocols as discussed above.

- Addition is carried out by simply asking the parties to sum their secret shares.
- It uses Beaver Triples for multiplication.
- All linear functions are represented as a combination of addition and multiplication.

- Non linear functions are implemented using standard approximations which use addition and multiplication.

Crypten builds on pytorch and thus has a C++ implementation under the hood. As a result, Crypten's performance is promising.

Timing Results

We observe that as the number of parties increase, the time taken also increases which is expected as communication costs increase. However, timing doesn't change a lot with increasing size. Also, Crypten provides a range of numbers on which some of the operators work accurately; out of that range we usually get junk values.

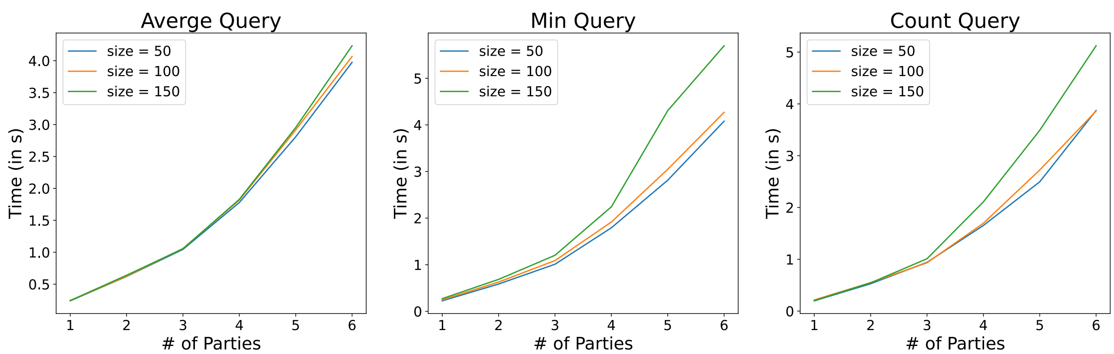


Figure 4: Timings of different queries with Crypten

The errors increase very slightly as we increase the grid size. On average the relative error across different queries observed was:

Average: $10^{-3} - 10^{-4}$ %
Min/Max/Count/Range: $10^{-5} - 10^{-6}$ %

5 Secure Interpolation across Servers

The pipeline discussed above where interpolation from different servers was used for answering queries is complete and can be used on it's own. However, there may be reasons to instead perform the interpolation together using data from all the servers. This can lead to better interpolation and take care of insufficient data at individual servers.

We explore two ways to achieve this — using Graph Convolution Networks and using Gaussian Process Interpolation. We discuss the results and issues with these in the following subsections. As before, note that interpolation needs to be done securely and the inputs and final interpolated values are all secure and shared. Also note that here the objective here isn't to get the best model to solve the problem but rather to understand how interpolation and learning can happen jointly across parties using available protocols and libraries.

5.1 Graph Convolution Networks

The input to a GCN is a graph and features of nodes of the graph. The output is the values of nodes which have to be predicted. This prediction is based on features of nodes and that of their neighbourhood. In our case, we take both the inputs and outputs to be pollution values with values of some location being unknown. We also take masks denoting whether values are known for grid points to be another set of input. This way, the unknown points can be predicted by using values of their neighbours. The Adjacency matrix is such that there are outgoing edges only from nodes whose values are known.

5.1.1 Experiment

The city is divided into grids as earlier. The servers have their respective pollution values at different grid points. They secretly share these values and averaging is done on all known points. The above steps may be referred to as pre-processing. For the unknown points, we perform interpolation. We will like to point out that the data was relatively sparse with only values for $\sim 2\%$ grid points known in an hour.

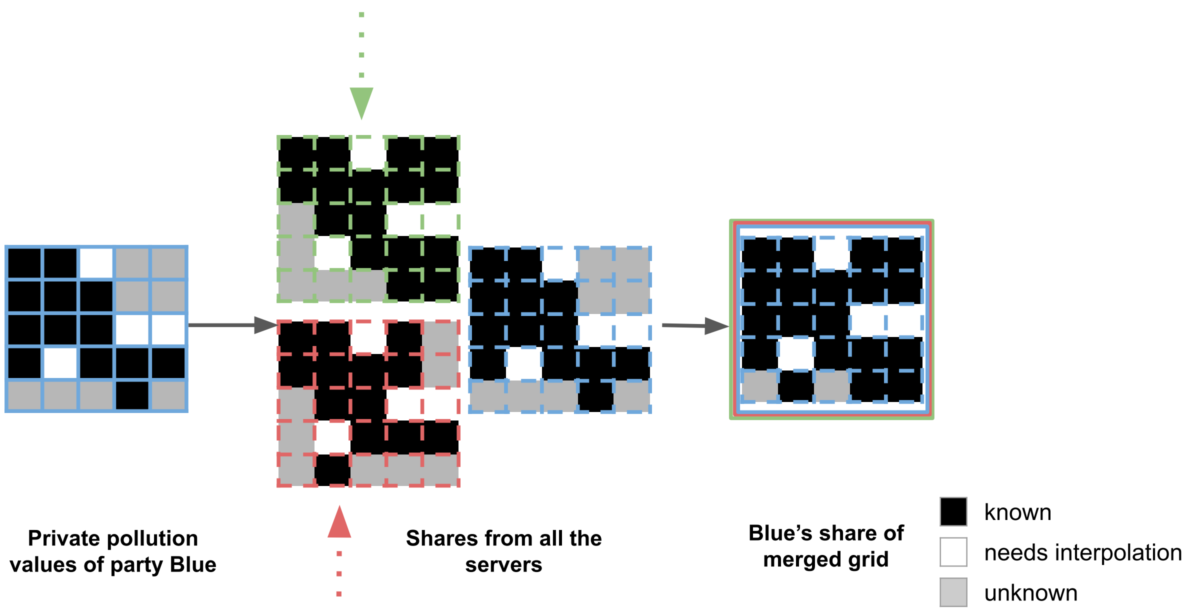


Figure 5: Preprocessing for Interpolation

To perform interpolation we use the GCN model. The GCN architecture involves two graph convolution layers followed by a fully connected layer. The input graph has edges between adjacent grid points. The model is trained on data accumulated in an hour. The weights learned however can be reused as pre-trained weights in later time windows. The input is split to test and train sets, with 1/20 of the locations being test locations. Some of the train points are randomly masked while training so that the model learns to predict values for unknown locations.

The training happens securely between parties so that the model learned is shared between them. Finally, predictions are made on the test points with the result for the entire grid now stored securely in a shared way between all the parties.

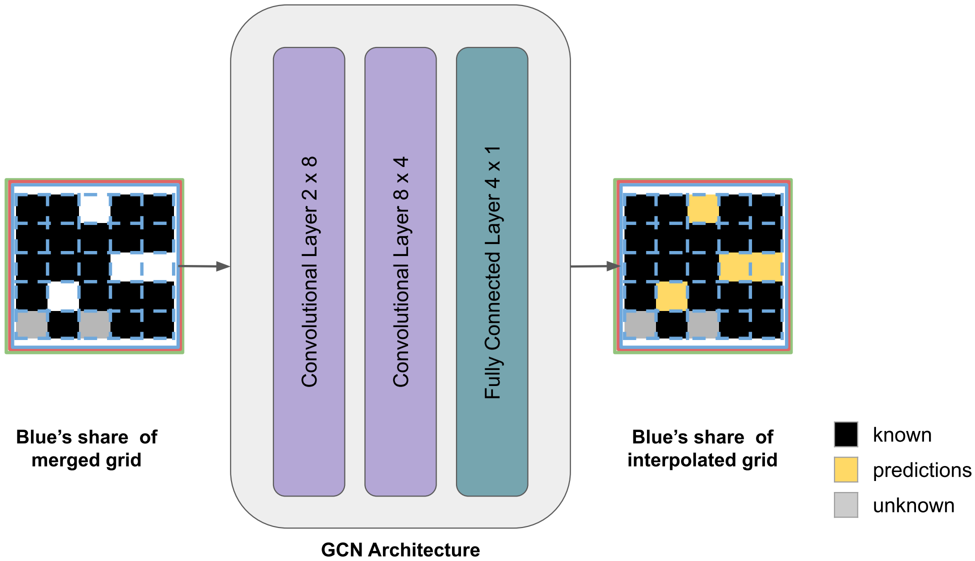
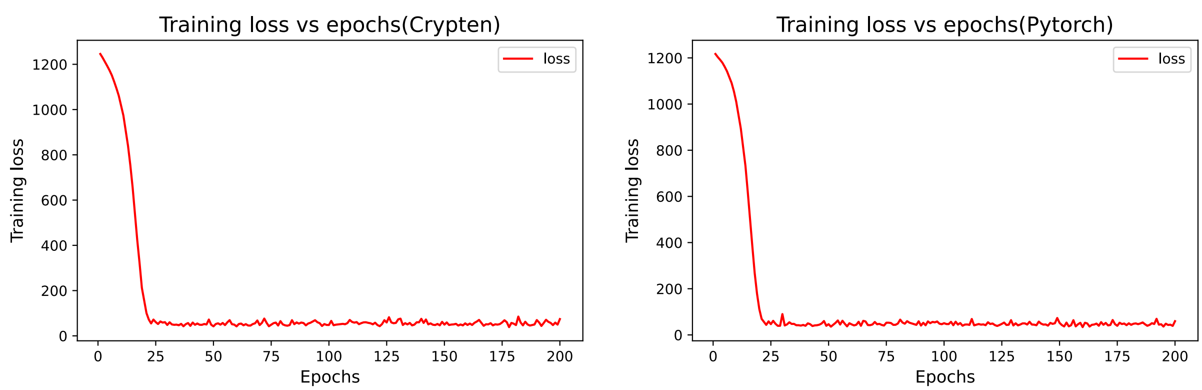


Figure 6: GCN Interpolation

Note that the experiments here were run on a single machine with all inter-party communications happening as inter-process communications. We use the `Crypten` library to achieve secure computation. We also implement the same pipeline (modulo shared secrets) using `Pytorch` and compare the results obtained.

5.1.2 Results

Training While training, `Crypten` performed as good as the standard `Pytorch` implementation when using SGD (Stochastic Gradient Descent), as evident from Figure 7.

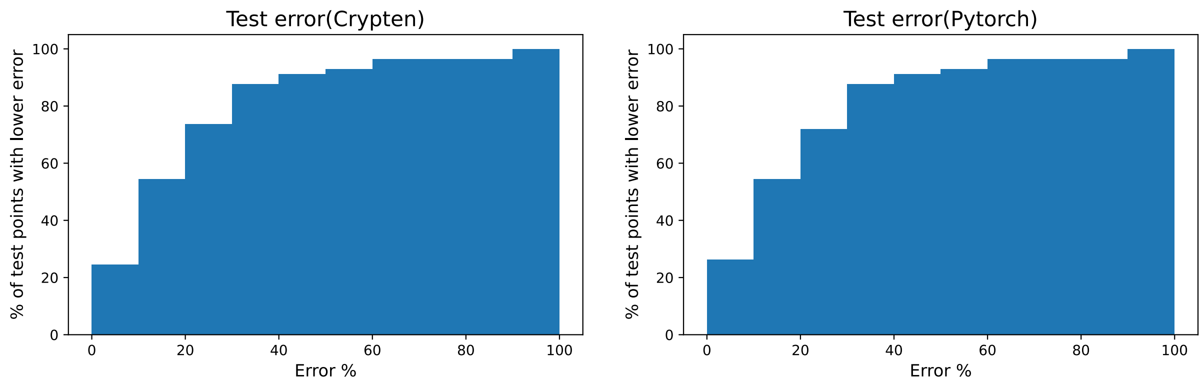


(a) Crypten (b) Pytorch

Figure 7: Training loss vs epoch

Accuracy The histograms in Figure 8 show the what % of test points have how much error. In this histogram, as well we can see performance of `Crypten` and vanilla `Pytorch` is similar.

This error % is for 4 different hours of data, for each hour model was first trained for 200 epochs and then tested. For both, `Crypten` and vanilla `Pytorch`, around 75% of the points have error less than 30%. Also, the points on which both of them performed poorly were usually the ones which had no known neighbors or differed significantly from their neighbors.



(a) Crypten

(b) Pytorch

Figure 8: Error %

Timing and Communication In terms of time, on the same machine `Crypten` is almost 100 times slower than `Pytorch`. For communication v/s computation breakup of `Crypten` we will analyse two stages of training separately, the two stages are:

1. **Pre-processing**: In this part the parties transfer the secret share of their data to each other and aggregate all the shares available to them before training.

   It is evident from the figure 9 that as we increase the number of parties, the communication time starts to dominate which is as expected. However, we see that the computation cost also increases slightly, this is because as the number of parties increases more grids have to be aggregated.

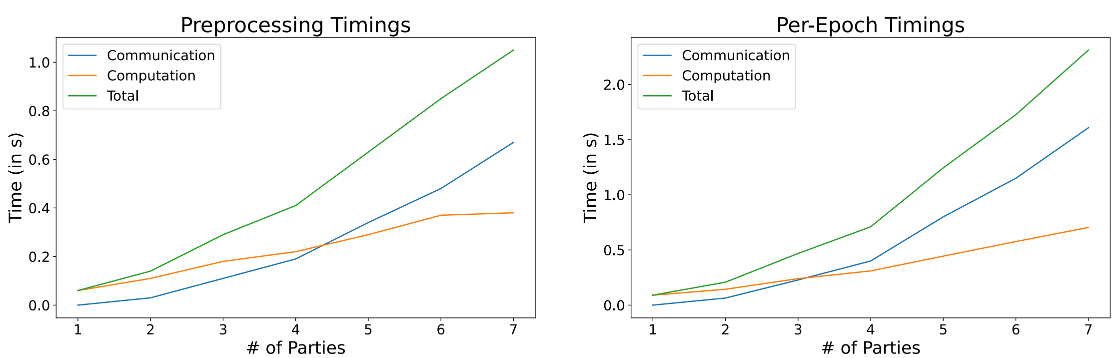


Figure 9: Timings during training the model in `Crypten`

2. **Training**: In the training phase, each party trains their own Graph Convolution Network for interpolation.

Here also, from Figure 9, we observe as the number of parties increase, the communication time starts to dominate. In per epoch timing, this effect is seen earlier as compared to in pre-processing, this is because in pre-processing there is significant CPU computation at each server as well.

5.1.3 Issues

Some issues we faced by using GCN for interpolation were:

- GCN doesn't return a confidence score.
- The test points need to be neighbours of known points.
- `CrypTen` doesn't support Adam optimizer which gave better results when the model was implemented in `PyTorch`.
- There were very few train points after we divide the city up into grids.

5.2 Gaussian Process Interpolation

We also explored using Variational Gaussian Process for interpolation. Variation Gaussian Processes have the advantage of providing a confidence value to their predictions. They can also make predictions on continuous input locations unlike GCN. For this we explored different libraries and analysed how their code base can be ported to `Crypten`. To be easily able to do such porting the first filter is to consider libraries implemented using `PyTorch`. Some of these libraries were `gpytorch`, `gptorch` and `pyro`. We found that all libraries implemented Cholesky factorizarion to compute matrix inverse. Some libraries were unable to perform the Cholesky factorization of a non-symmetric positive definite matrix and gave an unstable matrix warning even before attempting to port them to `CrypTen`. Also, large libraries like `gpytorch` have a very large number of lines of code with specific classes defined on top of `PyTorch` which are not easy to port. One such class we faced difficulty with was `lazyTensor`. Other smaller libraries like `gptorch` and `pyro` did not give very accurate and reliable results.