Distributed and Secure ML with Self-tallying Multi-party Aggregation

Yunhui Long *
UIUC
Urbana, Illinois
ylong4@illinois.edu

Tanmay Gangwani *
UIUC
Urbana, Illinois
gangwan2@illinois.edu

Muhammad Haris Mughees
UIUC
Urbana, Illinois
mughees2@illinois.edu

Carl A. Gunter
UIUC
Urbana, Illinois
cgunter@illinois.edu

Abstract

Privacy preserving multi-party computation has many applications in areas such as medicine and online advertisements. In this work, we propose a framework for distributed, secure machine learning among untrusted individuals. The framework consists of two parts: a two-step training protocol based on homomorphic addition and a zero knowledge proof for data validity. By combining these two techniques, our framework provides privacy of per-user data, prevents against a malicious user contributing corrupted data to the shared pool, enables each user to self-compute the results of the algorithm without relying on external trusted third parties, and requires no private channels between groups of users. We show how different ML algorithms such as Latent Dirichlet Allocation, Naïve Bayes, Decision Trees etc. fit our framework for distributed, secure computing.

1 Introduction

Machine learning models are being increasingly deployed to harness useful information from raw data. Availability of large amounts of training data prevents over-fitting in the models and improves its generalization. However, there is an important tension between the need for large training datasets and the privacy concerns of owners of those datasets. This is best exemplified when considering ML for health and medicine. For instance, assume multiple hospitals, each with access to high-quality (albeit limited in quantity) data about patient medical records. Jointly training a Latent Dirichlet Allocation (LDA) topic model on the union of data would provide insightful information for all the hospitals [18]. However, there is a huge privacy concern for sharing this data as it may contain sensitive information. In this work, we propose a framework for distributed training of ML algorithms among untrusted parties. The framework is secure since the parties can collaboratively train models without revealing their data. Checks for data validity provide robustness against a malicious party contributing illegal data. Furthermore, model aggregation is performed without relying on any external trusted agents.

Related Work. The problem of distributed and secure machine learning falls under the broad regime of secure multi-party computation (SMPC) [8]. Gentry proposed Fully Homomorphic Encryption (FHE) [11] as a means to achieve SMPC. Current FHE schemes are inefficient and only work with small circuits [7]. Homomorphism under addition, however, has been extensively studied, and many robust implementations exist [5, 13]. Hao et al. [13] apply additive homomorphism to create an

*These two authors contributed equally to the work.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

anonymous voting application. Their construction enables self-tallying of votes, precluding the need for trusted third parties for counting. Corrigan et al. [6] propose a more scalable secure aggregation protocol and apply it to linear regression.

Contributions. Our protocol broadens the scope of the ideas presented in [6, 13]. Our contribution is three-fold. First, we examine various ML algorithms under the lens of SMPC through homomorphic addition; second, we incorporate input validity checks to dissuade users with malicious data; and third, we propose efficient constructions using basic cryptographic tools like zero-knowledge proofs and ElGamal encryption. We also implement the protocol and present some empirical analysis.

2 Distributed and Secure ML

In the following subsections, we first outline our protocol for secure aggregation of arbitrary integer data vectors from different users. Following that, we detail the reduction of various ML algorithms to generalized vector addition, thereby making them compatible with our framework and enabling secure, distributed training on aggregated data.

2.1 Threat Model and Notations

Suppose there are n users. Each user U_i owns an integer data vector T_i of size m. We then desire the output of the vector addition $T = \sum_{i=1}^{n} T_i$ with the following properties:

- *Privacy:* The contents of T_i should be kept a secret from users other than i. In our protocol, this secrecy is maintained unless all of the other users have been compromised.
- *Input validity:* A malicious user should not be able to corrupt T by providing *unexpected* values. Depending on the ML algorithm, this could mean preventing a large integer input which can disturb cumulative statistics, or a negative input for an always-positive variable.
- *Self-tallying:* Any user should be able to compute T without relying on external talliers.
- *No private channels:* We assume only the availability of a publicly verifiable ledger (e.g. blockchain) and no user-to-user private channels. This offers dispute-freeness.

The zero-knowledge proof-of-knowledge ($\texttt{ZKPoK}$) used in our protocol are expressed in Camenisch-Stadler notation [4]: $\texttt{ZKPoK}_x(w) : L(w, x)$. Here, x is the public statement, w is the secret witness and L represents the conditions that the statement and witness must satisfy.

2.2 Two-round Protocol for Homomorphic Vector Addition

Let C be a finite cyclic group of prime order q in which the discrete log problem is hard, and g be a generator in C. There are n users, each with a secret key sk_i, and they agree on (C, g). User U_i's contribution to the aggregate (T) is a m-dimensional vector (T_i).

First Round: Each user U_i selects m random values $(x_{i1}, x_{i2}, \ldots, x_{im}) \in_R \mathbb{Z}_q$, publishes to the public ledger the values $(g^{x_{i1}}, g^{x_{i2}}, \ldots, g^{x_{im}})$ and a $\texttt{ZKPoK}$ of discrete log $(\texttt{ZKPoK}_A(a) : g^a = A)$ for each x_{ij} $(1 \leq j \leq m)$. At the end of this round, each U_i checks the validity of the $\texttt{ZKPoK}$s on the ledger, and computes:

$$h_{ij} = g^{y_{ij}} = \frac{\prod_{k=1}^{i-1} g^{x_{kj}}}{\prod_{k=i+1}^{n} g^{x_{kj}}}, \quad \forall 1 \leq j \leq m.$$

Second Round: Each user U_i computes the ElGamal encryption of T_{ij} for $1 \leq j \leq m$ as

$$\mathrm{E}[T_{ij}] = (g^{x_{ij}}, g^{T_{ij}} h_{ij}^{x_{ij}}).$$

U_i then publishes the encrypted vector $(\mathrm{E}[T_{i1}], \mathrm{E}[T_{i2}], \ldots, \mathrm{E}[T_{im}])$. Our construction of the public keys (h_{ij}) is similar to that in the first round of anonymous voting in [13]. Hence, it follows that by multiplying the correct ciphertext values, any user can compute $g^{\sum_i T_{ij}}$ for $1 \leq j \leq m$. Although computing $\sum_i T_{ij}$ requires taking a discrete log, the range of $\sum_i T_{ij}$ is generally not large, and a baby-step/giant-step approach [16] is practical. At the end of the second round, each user can produce the vector summation T by self-tallying the values for each index (j) of the vector.

To discourage malicious users from submitting encryptions of corrupted (or disallowed) T_i in the second round, we augment the protocol with input validity checks. Specifically, along with the encrypted T_i, each user is required to submit another proof to the ledger which can be validated by others for compliance of the input data. We consider two such compliance conditions - L^2-norm and L^1-norm of T_i. In many algorithms, such as collaborative filtering (Appendix 5.3.4), imposing a bound on L^2-norm, i.e. $\|T_i\|_2$, serves as a reasonable precondition. In notation, we want the ZKPoK:

$$\texttt{ZKPoK}_{(\vec{x},\vec{y},B)}(\vec{a},\vec{r}) : (x_i, y_i) = (g^{r_i}, h^{r_i} \cdot g^{a_i}) \wedge \|\vec{a}\|_2 \leq B \tag{1}$$

Bounding the L^2-norm does not guarantee that all (or any) of the entries in the vector T_i are non-negative. Non-negative inputs are required in some algorithms like LDA and decision trees (Appendix 5.3.1). Moreover, it is more useful to bound the L^1-norm, i.e. $\|T_i\|_1$, than the L^2-norm:

$$\texttt{ZKPoK}_{(\vec{x},\vec{y},B)}(\vec{a},\vec{r}) : (x_i, y_i) = (g^{r_i}, h^{r_i} \cdot g^{a_i}) \wedge \|\vec{a}\|_1 \leq B \wedge a_i \geq 0 \tag{2}$$

The ZKPoKs in Equations (1) and (2) are constructed from other simpler ZKPoKs mentioned in Appendix 5.1. We deem this construction to be an important contribution of this work. It is detailed in Appendix 5.2, along with the complete steps run by the prover and the verifier to generate and validate the proofs. We also mention future work on optimizing these proofs.

2.3 Reduction of Algorithms to Vector Addition

We now discuss several algorithms which fit into our framework for distributed and secure computation. In each case, it can be shown that the algorithm decomposes into a simple addition of integer vectors (or matrices) created from disjoint data pieces. This enables the various untrusting parties to safely engage in joint training of ML models using the protocol from previous subsection. Table 1 summarizes the algorithms, along with a validity check (L^1, L^2-norm) for it, and the significance of the check. Note that the L^1-norm bound check (Eq. 2) also includes the non-negativity constraint. We explain one algorithm (LDA) in detail here; reduction of other algorithms is in Appendix 5.3.

Application	**Validity**	**Significance of Check**
Latent Dirichlet Allocation	L^1-norm	Limit number of times a word is assigned to a topic by each user; disallow negative values
Decision Trees	L^1-norm	Limit number of training samples per user; disallow negative values
Naïve Bayes	L^1-norm	Limit number of training samples per user; disallow negative values
Cumulative Voting	L^1-norm	Limit total number of votes by each voter; disallow negative values
Linear Regression	L^2-norm	Limit contribution to β, prevent over-fitting
Collaborative Filtering	L^2-norm	Limit contribution to the preference matrix

Table 1: Summary of Algorithms

Latent Dirichlet Allocation. LDA is a generative probabilistic modelling technique for collections of discrete data such a text documents [2]. For each document j, there is a multinomial distribution θ_j over K hidden topics. Also, the k^{th} topic is represented by a multinomial distribution ϕ_k over the word vocabulary. x_{ij}, which is the i^{th} word in document j, is associated with a latent topic assignment z_{ij}. Given all words in all documents $\mathbf{x} = \{x_{ij}\}$, the inference task in LDA is to compute the posterior over $\mathbf{z} = \{z_{ij}\}$, θ_j and ϕ_k.

We summarize the approximate distribued LDA algorithm proposed by Newman et al. [17] which uses collapsed Gibbs sampling to sample the posterior z_{ij} at each state of the Markov chain. The algorithm initially divides the document corpus among different processors. We consider different processors as different users. Each user does local Gibbs sampling for a few iterations before synchronizing with other users. We encourage interested readers to look at Algorithm 1. in [17]. The synchronization involves a matrix reduction operation and is the only medium through which the privacy of a user's data could be violated: $N_{wk} \leftarrow \sum_{u \in \text{users}} N_{wk}^{(u)}$.

Computing $\mathbf{N_{wk}}$: After local Gibbs sampling for few iterations, each user computes $N_{wk}^{(u)}$, which is a matrix containing counts of the number of times a particular word is assigned to a particular topic. The encrypted matrix from each user can be homomorphically added and the result N_{wk} can be obtained by each user independently by self-tallying. To prevent a malicious user from including large or negative values in $N_{wk}^{(u)}$, the parties can decide on a bound for the L^1-norm of the input, and require that each user provide the corresponding range proofs.

3 Implementation

In this section, we evaluate the homomorphic vector addition protocol through the application of cumulative voting, and summarize some observations. In cumulative voting, each voter is given $B-1$ number of votes, and can arbitrarily distribute these votes among the candidates. A voter's input is considered legal as long as the total number of votes given by her is less than B. The voters are allowed to vote for more than one candidate and to put more than one vote on preferred candidates. Suppose there are n voters and m candidates. Let the vector $T_i = (T_{i1}, T_{i2}, \ldots, T_{im}) \quad (1 \leq i \leq n)$ be the votes of voter i, where T_{ij} is the number of votes given by voter i to candidate j. The result of cumulative voting can be tallied by adding the vote vectors from all voters:

$$T_{\text{result}} = (\sum_{i=1}^{n} T_{i1}, \sum_{i=1}^{n} T_{i2}, \ldots, \sum_{i=1}^{n} T_{im}).$$

To guarantee the fairness of cumulatve voting, it is necessary for each user to provide a ZKPoK for L^1-norm bound on each voting vector T_i. This limits the total number of votes by each voter and disallows negative votes (Table 1).

Our implementation[2] consists of the following layers: an ElGamal Encryption library implemented over elliptic curves, ZKPoK libraries, an interfacing client, which we call **Zorro client**, and a cumulative voting application (Figure 1). To simulate the environment of the blockchain, we implement a public ledger class that stores the encrypted data and ZKPoKs. In practice, the public ledger can be replaced by a smart contract and deployed on the Ethereum block chain. Further details on the components of the implementation are in Appendix 5.4. Therein, we also include an analysis on the machine time taken to generate and verify the ZKPoKs. The computational cost for ZKPoKs depends on the vector length (total candidates) and the bound (maximum votes allowed per voter), with the former being the more dominant factor. We provide some discussion on the time-complexity of baby-step/giant-approach [16], showing that it speeds up the discrete-log step. We also measure the effects of using integer precision rather than floating point precision for a simple linear regression problem, concluding that the accuracy-loss can be controlled. A more extensive study is interesting future work.

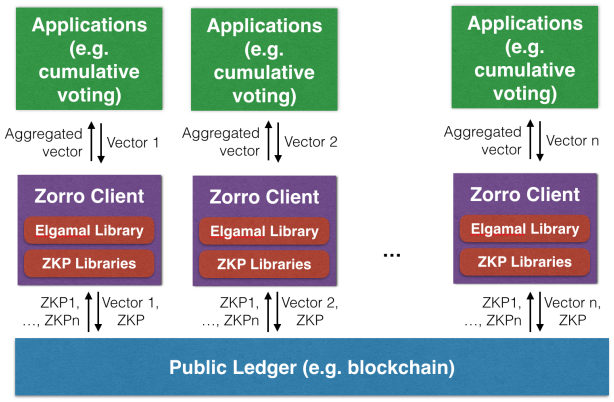


Figure 1: Structure of Implementation

4 Conclusion

In this paper, we outline a protocol for secure, distributed computing with multiple mutually distrusting parties. It includes input validity checks (bound on L^1-norm and L^2-norm) to guard against malicious users. It uses efficient constructions to prove information in zero-knowledge, uses a public-ledger to offer dispute-freeness, and is self-tallying, thus obviating presence of trusted third parties. We show how popular ML algorithms such as LDA, Naïve Bayes, Decision Trees etc. can be used with our framework. Furthermore, we implement our protocol on top of cryptographic constructs and open-source our Zorro client for multi-party cumulative voting.

[2]https://github.com/tgangwani/Zorro_SMPC

References

[1] libsnark. https://github.com/scipr-lab/libsnark.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[3] J. Camenisch, R. Chaabouni, et al. Efficient protocols for set membership and range proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 234–252. Springer, 2008.

[4] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*, pages 410–424. Springer, 1997.

[5] J. Canny. Collaborative filtering with privacy. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 45–57. IEEE, 2002.

[6] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282, 2017.

[7] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[8] W. Du and M. J. Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*, pages 13–22. ACM, 2001.

[9] W. Fang, C. Zhou, and B. Yang. Privacy preserving linear regression modeling of distributed databases. *Optimization Letters*, 7(4):807–818, 2013.

[10] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[11] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[12] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[13] F. Hao, P. Y. Ryan, and P. Zielinski. Anonymous voting by two-round public discussion. *IET Information Security*, 4(2):62–67, 2010.

[14] C. Hazay and Y. Lindell. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.

[15] J. Kun. Elliptic curves finite fields. https://github.com/j2kun/elliptic-curves-finite-fields, 2014.

[16] A. Lenstra and H. Lenstra Jr. Algorithms in number theory, handbook of theoretical computer science, vol. a, 673–715, 1990.

[17] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *Journal of Machine Learning Research*, 10(Aug):1801–1828, 2009.

[18] M. J. Paul, B. C. Wallace, and M. Dredze. What affects patient (dis) satisfaction? analyzing online doctor ratings with a joint topic-sentiment model. In *AAAI Workshop on Expanding the Boundaries of Health Informatics Using AI*, 2013.

[19] K. Peng and F. Bao. Batch range proof for practical small ranges. In *International Conference on Cryptology in Africa*, pages 114–130. Springer, 2010.

[20] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[21] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.

[22] P. Tüfekci. Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods. *International Journal of Electrical Power & Energy Systems*, 60:126–140, 2014.

5 Appendix

5.1 Zero Knowledge Proof-of-knowledge

We express the various zero-knowledge proof-of-knowledge (ZKPoK) used in our protocol in Camenisch-Stadler notation:

$$\texttt{ZKPoK}_x(w) : L(w, x),$$

where x is the public statement, w is the secret witness and L represents the conditions that the statement and witness must satisfy. We use the following ZKPoKs:

$$\texttt{ZKPoK}_A(a) : g^a = A \tag{3}$$
$$\texttt{ZKPoK}_{(g,h,u,v)}(w) : g^w = u \wedge h^w = v \tag{4}$$
$$\texttt{ZKPoK}_{(x,y)}(r) : (x, y) = (g^r, h^r) \vee (x, y) = (g^r, h^r \cdot g) \tag{5}$$
$$\texttt{ZKPoK}_{(x_a,y_a,x_b,y_b)}(a, b, r_a, r_b) : (x_a, y_a) = (g^{r_a}, h^{r_a} \cdot g^a) \wedge (x_b, y_b) = (g^{r_b}, h^{r_b} \cdot g^b) \wedge b = a^2 \tag{6}$$

In words, (1) is the ZKPoK of discrete log; (2) proves that (g, h, u, v) forms a Diffie-Hellman 4-tuple [14]; (3) is ZKPoK for ElGamal encryption of $m \in \{0, 1\}$; (4) proves the square relationship between pre-images of two ElGamal encryptions. In our implementation, we make them non-interactive by using Fiat-Shamir's heuristics [10].

Proof of Discrete Log, Eq. 3

Prover$(a, A = g^a)$		**Verifier**(A)
$k \leftarrow_\$ \mathbb{Z}_q$		
$K := g^k$	$\xrightarrow{K}$	$c \leftarrow_\$ \mathbb{Z}_q$
$s := k + ca$	$\xleftarrow{c}$	
	$\xrightarrow{s}$	$g^s \stackrel{?}{=} KA^c$

Proof of Diffie-Hellman Tuple, Eq. 4

Prover(G, q, g, h, u, v)		**Verifier**(G, q, g, h, u, v)
$w \mid u = g^w, v = h^w$		
$r \leftarrow_\$ Z_q \mid a = g^r, b = h^r$	$\xrightarrow{a, b}$	$e \leftarrow_\$ \{0, 1\}^t \mid 2^t < q$
$z = r + ew(\mod q)$	$\xleftarrow{e}$	
	$\xrightarrow{e}$	$g^z \stackrel{?}{=} au^e, h^z \stackrel{?}{=} bv^e$

Proof of encryption of $\mathbf{x_j} \in \{0, 1\}$, Eq. 5

Prover			**Verifier**
$w, r_1, d_1 \in_R Z_q \mid$	$w, r_2, d_2 \in_R Z_q$		
$x \leftarrow g^{x_j}$	$x \leftarrow g^{x_j}$		
$a_1 \leftarrow g^{r_1} x^{d_1}$	$a_1 \leftarrow g^{w}$		
$b_1 \leftarrow h^{r_1} y^{d_1}$	$b_1 \leftarrow h^{w}$		
$a_2 \leftarrow g^{w}$	$a_2 \leftarrow g^{r_2} x^{d_2}$		
$b_2 \leftarrow h^{w}$	$b_2 \leftarrow h^{r_2}(y/g)^{d_2}$	$\xrightarrow{x, y, a_1, b_1, a_2, b_2}$	$c \leftarrow_{\$} \mathbb{Z}_q$
$d_2 \leftarrow c - d_1$	$d_1 \leftarrow c - d_2$	$\xleftarrow{c}$	$c \stackrel{?}{=} d_1 + d_2$
$r_2 \leftarrow w - x_j d_2$	$r_1 \leftarrow w - x_j d_1$	$\xrightarrow{d_1, d_2, r_1, r_2}$	$a_1 \stackrel{?}{=} g^{r_1} x^{d_1}$
			$b_1 \stackrel{?}{=} h^{r_1} y^{d_1}$
			$a_2 \stackrel{?}{=} g^{r_2} x^{d_2}$
			$b_1 \stackrel{?}{=} h^{r_1} y^{d_1}$

Proof of square relation, Eq. 6

Prover$(s_a, s_b \in_R \mathbb{Z}_q)$		**Verifier**
$A \leftarrow (g^{s_a}, \gamma^a h^{s_a}) \mod p$		
$B \leftarrow (g^{s_b}, \gamma^b h^{s_b}) \mod p$		
$x, r_a, r_b \leftarrow_{\$} \mathbb{Z}_q$		
$C_a \leftarrow (g^{r_a}, \gamma^x h^{r_a}) C_b \leftarrow A^x (g^{r_b}, h^{r_b})$	$\xrightarrow{C_a, C_b}$	$c \leftarrow_{\$} \mathbb{Z}_q$
$v \leftarrow ca + x(\mod q)$	$\xleftarrow{c}$	
$z_a \leftarrow cs_a + r_a(\mod q)$		
$z_b = c(s_b a s_a) + r_b(\mod q)$	$\xrightarrow{v, z_a, z_b}$	$(g^{z_a}, \gamma^v h^{z_a}) \stackrel{?}{=} A^c C_a$
		$A^v (g, h)^{z_b} \stackrel{?}{=} B^c C_b$

5.2 Range Proofs

5.2.1 Range-proof for L^2-norm

In notation, we want the ZKPoK:

$$\texttt{ZKPoK}_{(\vec{x},\vec{y},B)}(\vec{a},\vec{r}) : (x_i, y_i) = (g^{r_i}, h^{r_i} \cdot g^{a_i}) \wedge \|\vec{a}\|_2 \leq B$$

We now show how we compose this ZKPoK using the basic ZKPoKs (Eq. 3- 6).

Step 1: Each user U_i generates an ElGamal public key (h_i) from its private key sk_i, and encrypt each T_{ij} as:

$$\mathrm{E}^*[T_{ij}] = (g^{x_{ij}}, g^{T_{ij}} h_i^{x_{ij}}). \tag{7}$$

Then, U_i proves that $\mathrm{E}[T_{ij}]$ and $\mathrm{E}^*[T_{ij}]$ encrypt the same plaintext. For this, it's sufficient to prove that $(g, \frac{h_{ij}}{h_i}, g^{x_{ij}}, \frac{\mathrm{E}[T_{ij}]}{\mathrm{E}^*[T_{ij}]})$ is a Diffie-Hellman 4-tuple using ZKPoK (Eq. 4). This is because of the following equation:

$$\frac{\mathrm{E}[T_{ij}]}{\mathrm{E}^*[T_{ij}]} = (1, (\frac{h_{ij}}{h_i})^{x_{ij}}).$$

Step 2: Each user U_i calculates the square vector ($\mathbf{w}_i$), encrypts it using the ephemeral key detailed below in Eq. 11, and publishes the encryption on the public ledger. It also provides a proof of the square relation (ZKPoK (Eq. 6))

$$\mathbf{w}_i = (w_{i1}, w_{i_2}, \ldots, w_{im}) = (T_{i1}^2, T_{i2}^2, \ldots, T_{im}^2).$$

Let B be the bound on $\|T_i\|_2$. U_i needs to prove the following:

$$s = \sum_{j=1}^{m} w_{ij} < B^2.$$

We provide a range-proof for s by decomposing s into binary representations [5]. Let $L = 2\log_2 B$. Then, s can be represented by an L-digit binary value, and expressed as a weighted sum of each digit:

$$s = \sum_{l=0}^{L-1} 2^l s_{il}.$$

To prove that $\|T_i\|_2^2 < B^2$, we need two sub-proofs. Firstly, we need to show that $s_{il} \in \{0, 1\}$ for all $0 \leq l \leq L-1$. This can be easily done by ZKPoK (Eq. 5). The second challenge is to prove that each s_{il} is indeed a digit in the binary representation of $\sum_{j=1}^{m} w_{ij}$. That is, the user should show the following, without revealing the values of w_{ij} and s_{ij}:

$$\sum_{j=1}^{m} w_{ij} = \sum_{l=0}^{L-1} 2^l s_{il}. \tag{8}$$

The protocol to validate Eq. 8 is as follows:
First, each user U_i selects L random values $x'_{i1}, x'_{i2}, \ldots, x'_{iL} \in_R \mathbb{Z}_q$, and encrypts each s_{il} $0 \leq l \leq L-1$ as:

$$\mathrm{E}[s_{il}] = (g^{x'_{i(l+1)}}, g^{s_{il}} h_i^{x'_{i(l+1)}}). \tag{9}$$

Then, each user U_i selects m random values $x^*_{i1}, x^*_{i2}, \ldots, x^*_{im} \in_R \mathbb{Z}_q$. For all $1 \leq j \leq m$, U_i calculates:

$$r_{ij} = (\sum_{k=1}^{j-1} x^*_{ik} - \sum_{k=j+1}^{m} x^*_{ik}) x^*_{ij}. \tag{10}$$

Assuming $m > L$, U_i encrypts each w_{ij} $(1 \leq j \leq m)$ as:

$$\mathrm{E}[w_{ij}] = \begin{cases} (g^{r_{ij}+x'_{ij}2^{(j-1)}}, g^{w_{ij}} h_i^{r_{ij}+x'_{ij}2^{(j-1)}}) & \text{if} \quad j \leq L \\ (g^{r_{ij}}, g^{w_{ij}} h_i^{r_{ij}}) & \text{if} \quad j > L \end{cases}. \tag{11}$$

To verify Eq. 8, a verifier needs to check that:

$$\prod_{j=1}^{m} \mathrm{E}[w_{ij}] = \prod_{l=0}^{L-1} \mathrm{E}[s_{il}]^{2^l}.$$

Or equivalently,

$$\begin{cases} \prod_{j=1}^{L} g^{r_{ij}+x'_{ij}2^{(j-1)}} \prod_{j=L+1}^{m} g^{r_{ij}} = \prod_{l=0}^{L-1} g^{x'_{i(l+1)}2^l} \\ \prod_{j=1}^{L} g^{w_{ij}} h_i^{r_{ij}+x'_{ij}2^{(j-1)}} \prod_{j=L+1}^{m} g^{w_{ij}} h_i^{r_{ij}} = \prod_{l=0}^{L-1} g^{s_{il}2^l} h_i^{x'_{i(l+1)}2^l} \end{cases} . \tag{12}$$

Since $\sum_{j=1}^{m} r_{ij} = 0$, the noise terms r_{ij} cancels out. Eq. 12 should hold if and only if $\sum_{j=1}^{m} w_{ij} = \sum_{l=0}^{L-1} 2^l s_{il}$, thereby completing the proof for Eq. 8. An alternative to using encryptions where the noise terms r_{ij} nullify each other is to use a Diffie-Hellman proof (ZKPoK (Eq. 4)) for Eq. 8. It achieves the same goal, albeit at the cost of an extra ZKPoK. Below we summarize the complete steps run by the prover and the verifier to generate and validate the range proof for L^2-norm, respectively.

As mentioned previously, U_i provides a proof that $w_{ij} = T_{ij}^2$, for all $1 \leq j \leq m$ (ZKPoK (Eq. 6)). We use the construction by Canny [5] for this ZKPoK. Canny's proof requires that w_{ij} and T_{ij} to be encrypted under exponential ElGamal encryption with the *same* public key. Therefore, in the proof, we use $\mathrm{E}[w_{ij}]$ and $\mathrm{E}^*[T_{ij}]$, which are both encrypted under the same key h_i.

Algorithm 1 Proof generation by user i

Require: $(T_{i1}, T_{i2}, \ldots, T_{im})$, $(\mathrm{E}[T_{i1}], \mathrm{E}[T_{i2}], \ldots, \mathrm{E}[T_{im}])$,
$(h_{i1}, h_{i2}, \ldots, h_{im})$, $(x_{i1}, x_{i2}, \ldots, x_{im})$,
ElGamal parameters (g, h_i), $L = 2\log_2 B$

1: Encrypt each T_{ij} as $\mathrm{E}^*[T_{ij}]$ (Eq. 7)
2: For each T_{ij}, generate proof for Diffie-Hellman 4-tuple $(g, \frac{h_{ij}}{h_i}, g^{x_{ij}}, \frac{\mathrm{E}[T_{ij}]}{\mathrm{E}^*[T_{ij}]})$
3: Calculate $w_{ij} = T_{ij}^2$, and $s = \sum_{j=1}^{m} w_{ij}$
4: Calculate $s_{i0}, s_{i1}, \ldots, s_{i(L-1)}$ such that $s = \sum_{l=0}^{L-1} s_{il} 2^l$
5: Generate L random values $x'_{i1}, x'_{i2}, \ldots, x'_{iL} \in_R \mathbb{Z}_q$
6: Encrypt each s_{il} as $\mathrm{E}[s_{il}]$ (Eq. 9)
7: Generate proof for $s_{il} \in \{0, 1\}$, for each s_{il}
8: Generate m random values $x^*_{i1}, x^*_{i2}, \ldots, x^*_{im} \in_R \mathbb{Z}_q$
9: Calculate r_{ij} (Eq. 10) and encrypt each w_{ij} as $\mathrm{E}[w_{ij}]$ (Eq. 11)
10: Generate proof for $(w_{ij} = T_{ij}^2)$, for each w_{ij}
11: Send the following messages to the verifier:
$(\mathrm{E}[T_{i1}], \mathrm{E}[T_{i2}], \ldots, \mathrm{E}[T_{im}])$, $(\mathrm{E}^*[T_{i1}], \mathrm{E}^*[T_{i2}], \ldots, \mathrm{E}^*[T_{im}])$,
$(h_{i1}, h_{i2}, \ldots, h_{im})$, h_i, ZKPoK$(g, \frac{h_{ij}}{h_i}, g^{x_{ij}}, \frac{\mathrm{E}[T_{ij}]}{\mathrm{E}^*[T_{ij}]})$ for each T_{ij},
$(\mathrm{E}[s_{i0}], \mathrm{E}[s_{i1}], \ldots, \mathrm{E}[s_{i(L-1)}])$, $(\mathrm{E}[w_{i1}], \mathrm{E}[w_{i2}], \ldots, \mathrm{E}[w_{im}])$,
ZKPoK$(s_{il} \in \{0, 1\})$ for each s_{il}, ZKPoK$(w_{ij} = T_{ij}^2)$ for each w_{ij}

Algorithm 2 Proof verification

Require: Messages received from user i in Algorithm 1

1: Verify ZKPoK$(g, \frac{h_{ij}}{h_i}, g^{x_{ij}}, \frac{\mathrm{E}[T_{ij}]}{\mathrm{E}^*[T_{ij}]})$ for each T_{ij}
2: Verify $\prod_{j=1}^{m} \mathrm{E}[w_{ij}] = \prod_{l=0}^{L-1} \mathrm{E}[s_{il}]^{2^l}$
3: Verify ZKPoK$(s_{il} \in \{0, 1\})$ for each s_{il}
4: Verify ZKPoK$(w_{ij} = T_{ij}^2)$ for each w_{ij}

5.2.2 Range-proof for L^1-norm

In notation, we want the ZKPoK:

$$\text{ZKPoK}_{(\vec{x}, \vec{y}, B)}(\vec{a}, \vec{r}) : (x_i, y_i) = (g^{r_i}, h^{r_i} \cdot g^{a_i}) \wedge \|\vec{a}\|_1 \leq B \wedge a_i \geq 0$$

With slight abuse of terminology, we'll call this proof as range-proof for L^1-norm, although it is much stronger and includes the additional proof for non-negativity of values. The proof proceeds in a manner very similar to

section 5.2.1, but we now require a range-proof for each element (T_{ij}) of the vector T_i. Like before, we do this by decomposing T_{ij} into binary representations [5].

5.2.3 Optimizations

The range-proof for L^1-norm of a vector requires range-proofs for all the elements of the vector. This leads to large time and space overheads in practice. There are a few approaches in literature which we can use to overcome this. Camenisch et al. [3] use a base $B, (B > 2)$ decomposition of a number s rather than base 2. This reduces the number of ciphertexts sent from the prover to the receiver. The authors use an elegant protocol to prove set membership $s_i \in \phi = \{0, \ldots, B-1\}$. The basic idea is to have the verifier provide a signature on each element of the set ϕ. The prover then proves in zero knowledge that it possesses a signature on the committed value s_i. The proof is sound because the prover can't fake a signature on a value outside the set ϕ. The efficiency of the protocol stems from the fact that the same set of signatures from the verifier can be used multiple times to commit to different s_i values.

Peng et al. [19] propose an approach called *batched* range proofs to improve computational efficiency. They also use a higher base decomposition and reduce the problem to proof of membership in a set of size k. Set membership is proved using a proof of knowledge of 1-out-of-k discrete logarithms. The novelty of their protocol is in batching (or combining) n such instances of 1-out-of-k discrete logarithms proof into one single proof, using generalized Pedersen commitments. This reduces the complexity of the overall protocol.

5.3 Reduction of ML Algorithms to Vector Addition

5.3.1 Decision Trees

Decision Trees are widely used for non-linear multi-class classification. The ID3 algorithm [20] for decision trees forms the tree by a recursive process. In each step of the recursion, a metric known as *entropy gain* is calculated for each feature in the feature-vector using the data-set available in the step. The feature with the highest entropy gain is selected as the root of the ensuing sub-tree. The recursion is usually terminated after a short depth to prevent over-fitting, with the leaves of the tree forming the class labels.

In the equations below, D is the complete dataset and q_j is the fraction of samples with label j in D. Let f be any feature which takes values $v \in F$. D_v is the set of samples from D where the feature f has a value v.

$$\text{entropy}(D) = -\sum_j q_j \log q_j$$

$$\text{gain}(f) = \text{entropy}(D) - \sum_{v \in F} \frac{|D_v|}{|D|} \text{entropy}(D_v)$$

Computing $\mathbf{entropy(D)}$: Let D_i be the fraction of the complete dataset in possession of user i. If the total number of labels is k, each user creates an encrypted vector $(c_1, \ldots, c_k)$, where c_j is the number of samples of label j in D_i. To prevent a malicious user from supplying large values for c_j which can corrupt the model parameters, range proofs for c_j and L^1-norm of the vector are required. Each user can then calculate q_j, and hence $\text{entropy}(D)$, by homomorphically adding all the vectors.

Computing $\mathbf{gain(f)}$: For ease of exposition, assume that $F = \{0, 1\}$, and there is only one feature f. User i creates two encrypted vectors $(p_1, \ldots, p_k)$ and $(q_1, \ldots, q_k)$, where p_j is the number of samples in D_i with $\{f = 0, label = j\}$, and q_j is the number of samples in D_i with $\{f = 1, label = j\}$. For input validity, a proof for $c_j = p_j + q_j$ is required. As before, using homomorphic addition, each user can compute $\text{entropy}(D_v)$, and hence $\text{gain}(f)$.

5.3.2 Naïve Bayes

Naïve Bayes classifiers are probabilistic classifiers which utilize the *naïve* assumption of conditional independence of the features, given the class label. Given a data sample $(x_1, \ldots, x_n)$, it uses Bayes' theorem to calculate the likelihood that the sample belongs to a particular class label:

$$\Pr(y|x_1, \ldots, x_n) = \frac{\Pr(y) \Pr(x_1, \ldots, x_n|y)}{\Pr(x_1, \ldots, x_n)}.$$

Using Naïve Bayes assumption and simplifying, the classification rules is given by-

$$\hat{y} = \underset{y}{\text{argmax}} \Pr(y) \prod_{i=1}^{n} \Pr(x_i|y).$$

The model parameters that are learned from the training data are $\Pr(y)$ and $\Pr(x_i|y)$. Although different assumptions can be made on the distribution of the parameters, we estimate them empirically using the counts from the training data:

$$\Pr(y = l) = \frac{|y = l|}{|D|},$$

$$\Pr(x_i = m|y = l) = \frac{|x_i = m, y = l|}{|y = l|}.$$

Computing $\mathbf{Pr(y = l)}$: Identical to the computation of q_j in ID3. Each user contributes a vector $(c_1, \ldots, c_k)$, along with range proofs.

Computing $\mathbf{Pr(x_i = m|y = l)}$: Identical to the computation of $\text{entropy}(D_v)$ in ID3. Each user creates as many vectors as the number of possible values for x_i, along with a proof that the vectors sum to $(c_1, \ldots, c_k)$.

5.3.3 Linear Regression

Given data samples of the form $(\vec{x}, y)$, linear regression models y, which is referred to as the dependent variable, as a linear combination of $\vec{x}$, which are called explanatory variables. More formally, the learning problem is the calculation of a vector β such that

$$y = \vec{x}^T \beta + \epsilon.$$

Least-squares method is a popular approach for estimating β. Let X be the design matrix with n data samples and Y be the corresponding vector of labels. The model parameters are then given by

$$\beta = (X^T X)^{-1} X^T Y. \tag{13}$$

Let X_i and Y_i be a horizontal partitioning of the design matrix and label vector, respectively. Each user i only has access to X_i and Y_i. As noted by the authors in [9], the following equations hold

$$X^T X = \sum_i X_i^T X_i,$$
$$X^T Y = \sum_i X_i^T Y_i.$$

Therefore,

$$\beta = (\sum_i X_i^T X_i)^{-1} \sum_i X_i^T Y_i. \tag{14}$$

Computing β: Let the dimension of the data $(\vec{x})$ be d. Each user independently computes a $d \times d$ matrix $(X_i^T X_i)^{-1}$ and a d dimensional vector $X_i^T Y_i$. The encrypted tensors are submitted along with range proofs on the L^2-norm to bound the influence of each user on the final model parameters. The tensors are homomorphically added to calculate β as per equation 13.

5.3.4 Collaborative Filtering

Collaborative Filtering (CF) is a technique most commonly used in recommender systems to predict the preferences of a user by accumulating preferences of multiple users. Among the various approaches that exist in literature for CF [5, 12, 21], we focus on the one used by Canny [5]. This work uses the ideas of secret sharing and threshold decryption to achieve CF with privacy. It relies on a majority vote among untrusted tallying authorities to get the result of the computation. In contrast, our approach gets rid of the tallying authorities by carefully designing the encryptions. We only mention the key computation steps of the algorithm by Canny; interested readers should refer to [5] for details.

Let there be n users providing integer ratings to m items. Let $P^{n \times m}$ be the user preference matrix such that P_{ij} is the rating given by user i to item j. P_{ij} is 0 if the item is unrated. The first step is the derivation of a low dimensional approximation to P. Let $A^{k \times m}$ (k is small) be such an approximation:

$$A = \sup_{U:UU^T=I} \text{tr}(PU^T UP^T).$$

Starting from a random matrix, A is computed iteratively using conjugate gradient. Let $A_{(t)}$ be value of the matrix at iteration t, and P_i denote the $1 \times m$ matrix of data from user i. The gradient for the current iteration can be calculated as

$$G_{(t)} = \sum_{i=1}^{n} A_{(t)} {P_i}^T P_i (I - A_{(t)}^T A_{(t)}).$$

After $A_{(t)}$ is updated using the gradient, the process is repeated (until convergence). Generating new recommendations from A entails more steps like partial SVD and probabilistic latent variable modeling [5].

Computing G in every iteration: Since $G = \sum_i G_i$, we can use homomorphic encryption to securely calculate the gradient in a distributed setting. Each user creates an encrypted matrix ${G_i}^{k \times m}$. To limit the effect of each G_i on the final gradient, a range proof on the L^2-norm of G_i is required.

5.4 Evaluations

ElGamal Encryption and ZKPoK libraries: To achieve higher efficiency, we write our own lightweight ZKPoK libraries instead of using existing general ZKPoK libraries such as zk-SNARK [1]. We implement ElGamal encryption over elliptic curve using Jeremy Kun's elliptic curve library [15] in Python. For each proof mentioned in Appendix 5.1, we implement a ZKPoK library to generate and verify the proof based on ElGamal encryption.

Zorro Client: This is an interfacing client that takes an input vector (T_i) from the application, and returns a vector summation $(\sum_i T_i)$ computed over all the parties involved in the protocol. Developers who want to implement applications in Table 1 can use the Zorro client as a black-box and do not need to be aware of the underlying ZKPoKs or the interactions with the public ledger. Specifically, the Zorro client handles the following 3 tasks for the higher level application:

- Generate ZKPoKs for input validity and commit them to the ledger;
- Verify ZKPoKs commited by other users;
- Calculate vector summation by homomorphic vector addition over encrypted inputs of all the users.

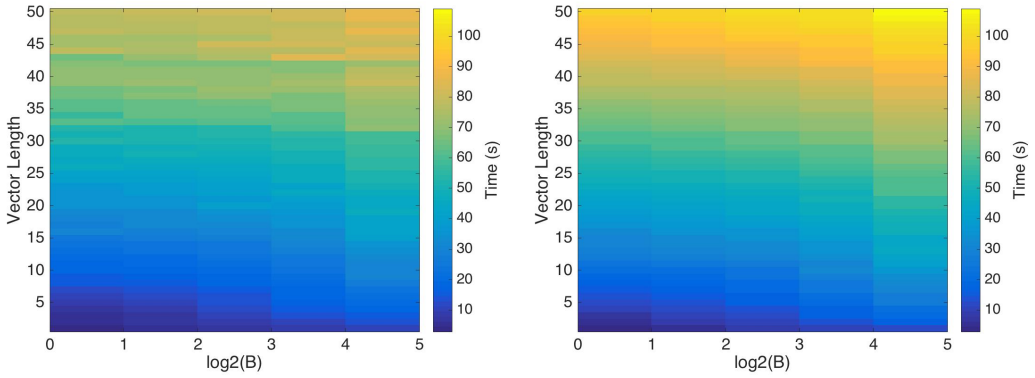
To evaluate the efficiency of Zorro client, we measure the ZKPoK generation and verification time for each user for the application of cumulative voting. Considering that users of the application (i.e., voters) would not have access to specialized hardware, the evaluations are done on a regular laptop with 2.7 GHz Intel Core i5. The Zorro client implements the two-round homomorphic vector addition protocol introduced in Section 2.2. The first round of the protocol consists of one ZKPoK of discrete log (Eq. 3) for each element in the input vector. The time complexities of ZKPoK generations and verifications for the first round increase linearly with the length of vector, and do not depend on any L^1-norm constrains on the input. Therefore, we focus our evaluations on the second round of the protocol.

ZKPoK Generation Time The ZKPoK generation time in round 2 depends on two factors: vector length (m) and maximum bound (B) on the L^1-norm. For cumulative voting, the vector length corresponds to the number of candidates, and the maximum bound corresponds to the number of votes per voter. The former determines the number of range proofs one client needs to generate, while the latter determines the complexity of each range proof. Figure 2a shows the variation of ZKPoK generation time per client for vector length $1 \leq m \leq 50$ and maximum bound $2^1 \leq B \leq 2^5$. The figure reveals positive correlations between ZKPoK generation time and vector length, and between ZKPoK generation time and maximum bound. Out of vector length and maximum bound, we observe the impact of the former to be higher. For example, when $m = 1$, $B = 2^5 = 32$, it takes only 9.9 seconds to generate ZKPoKs. However, when $m = 32$, $B = 2^1$, the generation time takes around 53.2 seconds. Therefore, Zorro can handle cumulative voting with relatively large number of votes per user, but is more suitable for a small number of candidates. When the number of candidates exceeds 35, it takes more than one minute to generate the ZKPoKs even when only one vote is allowed per voter.

ZKPoK Verification Time The ZKPoK verification time depends on three factors: vector length (m), maximum bound (B), and total number of users (n). Since each client needs to verify ZKPoKs of *all* the users, ZKPoK verification time per client increases linearly with the number of users. Figure 3 shows the increase in average per-client verification time (when $m = 1$ and $B = 2$) as the number of total users increases from 1 to 10. On average, it takes around 5 seconds to verify the ZKPoKs of each user. Therefore, when there are thousands of users, the verification phase can take hours. However, since the verification for different users is independent, the overall time can be greatly reduced by using multi-core parallelism. Furthermore, the optimization techniques discussed in section 5.2.3 can also be applied to improve efficiency. Similar to ZKPoK generation time, ZKPoK verification time is also influenced by vector length and maximum bound. Figure 2b shows ZKPoK verification time with $1 \leq m \leq 50$, $2^1 \leq B \leq 2^5$, and $n = 1$. On average, the time it takes to verify ZKPoKs is slightly higher than the time it takes to generate them.

Taking the Discrete Log As mentioned in Section 2.2, each user calculates $g^{\sum_i T_{ij}}$ by multiplying the correct ciphertexts, and uses the baby-step/giant-step algorithm [16] to obtain the discrete log. The algorithm has a time complexity of $O(\sqrt{N})$, for a search space of N numbers. In Figure 4a, we plot the time to compute the discrete log as a function of the bound on the input from each user. We simulate 1000 users, each with an integer input T_{ij} in the range $[0, B]$, generated using a uniform distribution. It then follows that the sum $\sum_i T_{ij}$ is a value in the range $[0, 1000 \times B]$, distributed according to a Irwin-Hall distribution. We record the time taken to compute the discrete log of the sum, average it over 10 observations and plot. Figure 4a shows that the algorithm has sub-linear time complexity. Moreover, the discrete log can be calculated in less than a second even with B=32. Hence, this step is very fast compared to the ZKPoK generation and verification steps mentioned above.

Impact on Accuracy Our elliptic curve cryptography system uses a finite field of integers modulo p, $\mathbb{Z}_p$. Therefore, the input vectors to our homomorphic vector addition algorithms can only be integers from this field.



(a) ZKPoK generation time per user.

(b) ZKPoK verification time per user. ($n = 1$)

Figure 2: Time Complexity Analysis under Varying Vector Length and Maximum Bound

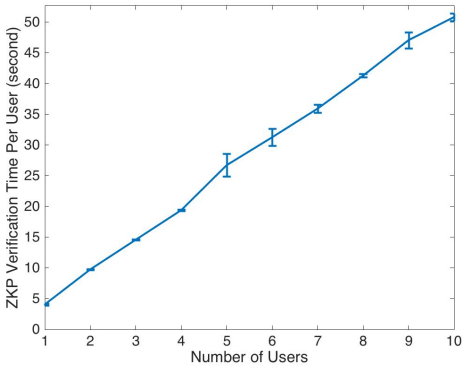
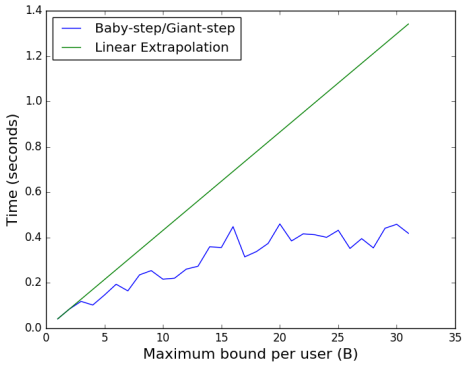
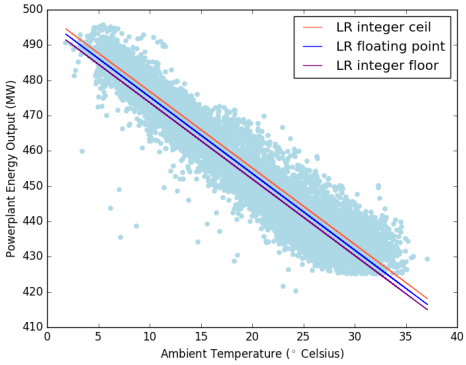


Figure 3: ZKPoK Verification Time Per User against Increasing Number of Users ($m = 1$, $B = 2$)



(a) Time to compute discrete log using baby-step/giant-step



(b) Power output prediction using linear regression

Figure 4: (a) Analysis on discrete log computation and (b) Accuracy loss with linear regression

Although sufficient for cumulative voting, this may be restrictive for some machine learning applications which are sensitive to floating point (FP) precision. We evaluate uni-variate linear regression (Section 5.3.3) on a real data-set and quantify the loss. Figure 4b plots the variation of the electrical power output from a power plant with ambient temperature [22]. The input and output values have FP precision. We fit a linear regression model to the data in three ways, first by using the original values, and then by using `floor` and `ceil` on the FP data in two separate experiments. We observe that `floor` and `ceil` models have 8.3% and 8.4% higher mean square

error than the FP model, respectively. This shows that the loss in accuracy due to FP rounding-off errors can be small. Furthermore, we can use FP quantization methods to improve precision, if needed.