Roseman Labs

Machine Learning on Encrypted Data

Niek J. Bouman

April, 2024

1 Introduction

The ability to extract insights from disconnected confidential-data silos has lots of business value. The traditional approach of centrally combining data sources into a joint data set, and then performing data analysis, typically suffers from a lack of trust from the data owners, and, in case of personal data, is often not compliant with data privacy laws.

The advent of Privacy Enhancing Technologies (PETs) is a game changer for exactly this type of multi-owner data analyses: encrypted computing technologies like Secure Multi-Party Computation (MPC) and Fully Homomorphic Encryption (FHE) provide novel, trustworthy and regulatory-safe solutions. Gartner predicts that 60% of large organizations will use one or more privacy-enhancing computation techniques in analytics, business intelligence or cloud computing by 2025. [G22]

At Roseman Labs we have built a groundbreaking solution to train and use AI/ML on data that is too sensitive to be shared. Our solution is used by 150+ organizations across Healthcare, Public Sector and Financial Services to solve real world problems.

The Roseman Labs platform enables you to encrypt, link and analyze multiple data sets, while safeguarding the privacy and commercial sensitivity of the underlying data. You can combine information from several organizations, run your analyses on records at a granular level, and generate new insights – all without ever being able to view other participants' input. You get the insights you need, while the data stays protected.

1.1 Making Encrypted Computing Mainstream

Roseman Labs is on a mission to make encrypted computing mainstream, by which we concretely mean that quality, performance, scalability, availability and affordability surpass a joint threshold at which encrypted computation becomes the technology of choice for most confidential data analyses:

- Quality: the quality (numerical accuracy, etc.) of the results of privacy-preserving training and inference is comparable to their cleartext baselines (results obtained using cleartext computations);
- Performance: the solution meets its latency and throughput requirements of common real-world applications;
- Scalability: the solution is capable of processing data volumes that are required in common real-world applications;
- Availability: the system runs on commercial off-the-shelf hardware, instead of relying on non-ubiquitous hardware accelerators;

- Affordability: achieve cost-effectiveness through a combination of choosing the right set of trade-offs, leveraging efficient algorithms, making highly optimized software implementations and performing application-specific resource provisioning during deployment.

In this document, we will demonstrate by means of various benchmarks that our MPC engine can realize these goals. For example, we will see that the quality of a secure logistic regression model trained in MPC by our software is as good as a solution found by scikit-learn, a popular Python package for (cleartext) machine learning. Also, we prove by means of several benchmarks that high-performance data processing is not only feasible with MPC, but that our engine excels in nearly all benchmarks.

Also, we illustrate our ability to scale vertically, which demonstrates our focus on exploiting hardware parallelism. Our software runs on smaller as well as larger multi-core servers, which are ubiquitous in today's clouds. This is a strong benefit over Fully Homomorphic Encryption; essentially all FHE-based solutions rely on future specialized hardware acceleration technology for their performance promise.

In the following sections, we will describe the steps involved to train a machine learning model on encrypted data, thus without ever revealing that data. We will use logistic regression as a running example, and explain the basics of logistic regression, and how a model is trained. Also, we will demonstrate data preparation in the context of encrypted computing. Finally, we will run various benchmarks to demonstrate the state of the art.

1.2 Data Preparation in the Context of Collaborative Encrypted Computing

In a collaborative setting, participants each provide part of the data. When we want to train a machine learning model in this setting, we need to join those separately provided parts into one data set. Let us assume that the data is provided in tabular form, which is a natural assumption given the ubiquity of relational databases.

The main challenge here is that we cannot use an ordinary database to perform this join operation, because then all data would have to be revealed to that database. Hence, to avoid such unwanted data leakage, the entire join operation must be performed as an encrypted computation on the encrypted tables.

We distinguish between "horizontally partitioned" and "vertically partitioned" tabular data; see also Figure 1. Let us explain these notions for the case of two data providers.

Horizontally partitioned means that both providers have different subjects in their data tables, but both have information about the same features. For example, data provider 1 has information about Alice and Fred, while data provider 2 has information about Mary and Josh. But both providers know their first name, last name and favorite color.

Vertically partitioned means that provider 1 and 2 each know different features about the same subjects. For example, provider 1 knows the first and last name of each subject, while provider 2 knows the favorite color of those subjects.

When we would like to join these tables exactly, we need a unique identifier for each subject. For persons, think of a unique social security number. Then, to join horizontally partitioned data we would perform, in SQL terminology, an `OUTER JOIN` (followed by some minor post processing), while to join vertically partitioned data we would perform an `INNER JOIN`.

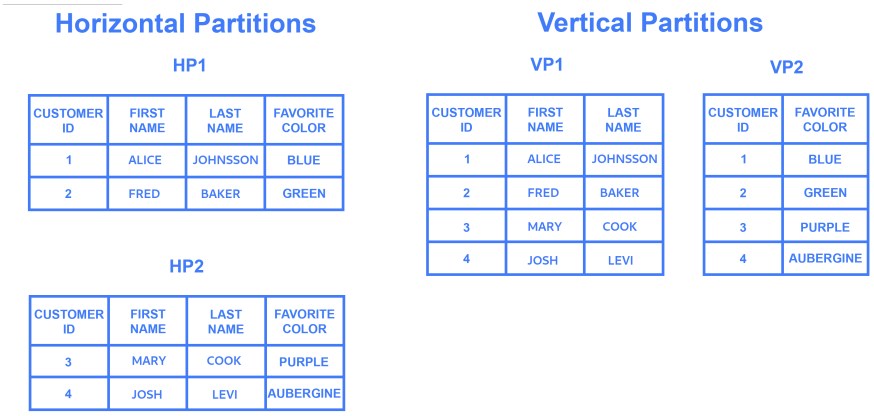

Figure 1: Horizontally vs. vertically partitioned data.

Later, we will present some benchmarks of elementary tabular operations, like joining tables of various sizes. Before presenting those, we will give some more background information on logistic regression. Readers who wish to jump to the benchmarks immediately can follow this link.

Note that two other important data preparation steps are data re-scaling (normalization) and splitting the data set into a training set and a test set. Typically, these steps have to be carried out using encrypted computations, because by the time these steps are performed, the input data has already been encrypted.

2 Preliminaries

2.1 Binary Logistic Regression

Binary (or binomial) logistic regression (usually attributed to Berkson, 1944) is a statistical method for predicting a two-outcome variable (0 or 1, "yes" or "no", etc.) based on a number of features. For example, think of a doctor who wants to predict whether a cancer tumor is malignant or benign, based on various patient characteristics (like age, gender) and various characteristics of the cancer tissue.

Sometimes, we want to predict the probabilities of the possible outcomes, rather than the most likely outcome itself. Note that this is more general; when given a probability p we can always apply some decision rule to predict the outcome, e.g., by assigning "no" if p lies in the interval $[0, \frac{1}{2}]$ and "yes" otherwise. The benefit of having a prediction in the form of a probability is that it also expresses the certainty of a prediction: although the probabilities .51 and .99 will both lead to a prediction of "yes" under the exemplary decision rule from above, the latter probability is much farther away from the decision boundary, $\frac{1}{2}$, hence is a much more confident prediction.

2.2 Other Forms of Logistic Regression

A generalization of binary logistic regression is multi-class logistic regression (also called multinomial logistic regression), in which there exist more than two outcome classes, say, "apple", "pear", "banana" and "strawberry". Another generalization is ordinal logistic regression, in which the (possibly more than two) outcome classes have a total ordering; think for example of a risk score that can be "low", "medium" of "high". Although multi-class logistic regression could in principle also be used for this risk-score example, the ordinal regression method will generally give better predictions because it exploits the extra information provided by the ordering relation.

2.3 What is a Model?

We define a (binary) model as the function f that takes as input a vector $x = (x_1, \ldots, x_k) \in \mathbb{R}^k$ of k real-valued feature values, and outputs a probability distribution over the outcome $Y \in \{0, 1\}$. Because the probabilities that comprise a probability distribution by definition add up to one, this probability distribution is simply $(1 - p, p)$, and can be represented by p alone, which is the conditional probability of obtaining 1, conditioned on the features taking on the values that were given as input:

$$f(x) := p = \Pr[Y = 1 \mid X = x]$$

2.4 Mathematical Structure of the Logistic Regression Model

The logistic regression model is a member of the family of generalized linear models, which have the following structure:

$$f(x) = \tau(\beta + w \cdot x)$$

where β is a parameter called the intercept, $w = (w_1, \ldots, w_k)$ is a vector of model parameters, $x \cdot w$ denotes the dot product (or inner product) between x and w, and $\tau : \mathbb{R} \to \mathbb{R}$ is some non-linear map. In case of logistic regression, τ is taken to be the logistic function or sigmoid function (plotted in Figure 2):

$$\lambda(z) = \frac{1}{1 + \exp(-z)}.$$

Note that the logistic function ensures that $f(x)$ will always be in the interval $[0, 1]$, hence can always be interpreted as a probability.

A logistic regression model for k features thus has $k + 1$ parameters; one parameter per feature, plus the intercept parameter.

2.5 Model Training

Informally speaking, the model can be viewed as a box with multiple knobs (like in Figure 3), one per model parameter. Each knob can be adjusted, and the combination of all knob-settings (and the model architecture) determines how the model maps an input (the vector of feature values x), to the output (the probability p). Training is the process of cleverly adjusting the knobs based on given input-output pairs (training examples).

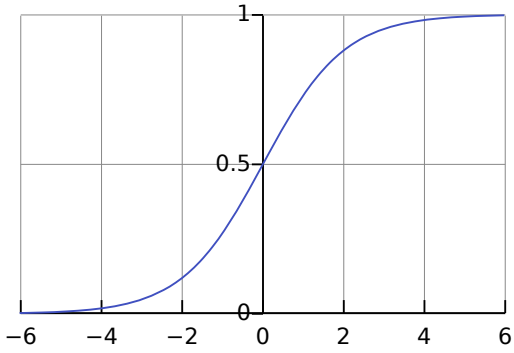

Figure 2: The logistic (or sigmoid) function.

Figure 3: Metaphor of a machine learning model with tunable model parameters.

Mathematically, model training is essentially optimizing a loss function that penalizes wrong predictions on the training data set. When defined appropriately, the loss function of logistic regression (the cross-entropy loss, which we will not define here) is convex, hence the training problem can be solved easily using a standard optimization method.

Gradient Descent (GD) is an iterative first-order (as in first-order Taylor approximation) optimization method that in each iteration takes a step into the direction of the (negative) gradient of the loss function. Stochastic Gradient Descent (SGD) approximates gradient descent by calculating an estimate of the gradient of the loss function from a random subset of the training data set (instead of using the entire training data set).

Newton's method is a second-order optimization method, hence uses not only the gradient but also the curvature (the Hessian matrix) of the loss function. The advantage of taking the curvature into account is to get better asymptotic (namely, quadratic) convergence. Because it is typically computationally expensive to compute the Hessian matrix (in fact, its inverse is needed) in each iteration, so called quasi-Newton methods have been devised to estimate the inverse Hessian matrix without too much computational work from the gradient. An example of a quasi-Newton method is the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm.

Another important topic in the context of optimization, but which we will not discuss here in detail, is how to choose the step size: how big of a step should be taken towards the computed direction in each optimization iteration? And should this step be constant over all iterations, or be determined in each iteration? And what is the impact of the chosen step-size strategy on the stability and the convergence behavior?

If we ignore — for the sake of the argument — the influence of the choice of the step size strategy, then we can expect that a second-order optimization method will require less iterations to train a model up to a given quality level (see also the upcoming section about model quality) when compared to a first-order method, or, alternatively, when keeping the number of iterations fixed, the second-order method is expected to produce a better model fit. (The latter is under the assumption that the training procedure has not yet converged; otherwise there is no point to iterate further.)

3 Assessing the Quality of a Fitted Model

Various methods and goodness-of-fit metrics exist to assess the quality of a fitted model. Especially if the model parameters are to remain secret (i.e., encrypted), evaluating the quality metric should itself be performed as an encrypted computation. Also, the quality metric should be concise, ideally a single number, because it typically has to be revealed for some data analyst to judge it. If needed, though, the information leakage could be further reduced by comparing (again, as an encrypted computation) the quality metric to some lower bound, and revealing the (binary) result of this comparison. (I.e., declaring "success" if the metric exceeded the lower bound, and declaring "failure, retraining needed with more iterations" otherwise.)

In our experiments, however, we use a non-sensitive benchmark data set, hence for frameworks that do currently not support evaluating the goodness-of-fit metrics as an encrypted computation, we will simply reveal the trained model parameters and evaluate the metrics using "cleartext computations".

A suitable goodness-of-fit metric for logistic regression is McFadden's ρ^2 (also known as McFadden's pseudo-R^2 metric). McFadden's ρ^2 metric [MF77] is defined as:

$$\rho^2 := 1 - \frac{\mathcal{L}(\beta, w)}{\mathcal{L}(\beta_0, 0)},$$

where $\mathcal{L}(\beta, w)$ is the log-likelihood function:

$$\mathcal{L}(\beta, w) = \sum_{(x,y)\in(\mathcal{X},y)} \big[-\log(1 + \exp(\beta + w \cdot x)) + y(\beta + w \cdot x) \big],$$

and the notation $(\mathcal{X}, y)$ represents a data set: $\mathcal{X}$ represents a matrix of which each row is a feature vector, and y is a (column) vector that contains the corresponding binary (0 or 1) labels. Furthermore, 0 denotes the zero vector of length equal to that of w, and β_0 is the maximum likelihood estimate

$$\beta_0 = \log \frac{\bar{y}}{1 - \bar{y}}$$

where $\bar{y}$ is the mean of the labels y.

The McFadden ρ^2 score is computed from the same likelihood function used to fit the model, hence the McFadden score is evaluated on the training data set.

It holds that $\rho^2 \leq 1$, and ρ^2 could become negative which indicates a poor fit. The interpretation according to McFadden is:

> "[...] values of .2 to .4 for [ρ^2] represent an excellent fit." [MF75]

Note that because the McFadden score depends on the number of model parameters, it should not be used to compare models having different numbers of parameters.

3.1 Predictive Power

To assess the model's predictive power (based on evaluation on the test data set), we could compute the mean squared error (MSE) or Brier score:

$$\text{MSE} := \frac{1}{N} \sum_{(x,y)\in(\mathcal{X},y)} (y - f(x))^2.$$

where N is the size of the (test) data set. It is easy to see that $0 \leq \text{MSE} \leq 1$, and a model that would predict perfectly would have $MSE = 0$.

Another relevant metric is Tjur's coefficient of discrimination D [Tj09], which is defined as:

$$D := \frac{1}{N_1} \sum_{\substack{(x,y)\in(\mathcal{X},y) \\ \text{s.t. } y=1}} f(x) - \frac{1}{N_0} \sum_{\substack{(x,y)\in(\mathcal{X},y) \\ \text{s.t. } y=0}} f(x),$$

where $N_b = \sum_{y\in y} [y = b]$, with $[\cdot]$ denoting the Iverson bracket. Tjur proves that $D \geq 0$ with equality if all predicted probabilities are equal to each other (which would mean that the model has no predictive power), and $D \leq 1$ with equality if all predicted probabilities are equal to the binary labels (meaning: the model makes perfect predictions).

4 Benchmarks for Elementary Tabular Operations in Roseman Labs' Platform

In this section we benchmark the performance of elementary tabular operations in Roseman Labs' MPC platform, like filtering rows of a table based on a predicate, joining tables and performing a group-by.

To perform the benchmarks we make use of encrypted demo tables (`crandas.demo_table`), which are tables of configurable dimensions that look as follows:

Table 1: Output of the `crandas.demo_table` command.

col1	col2	col3	col4	...
1	1	1	1	...
2	2	2	2	...
3	3	3	3	...
⋮	⋮	⋮	⋮	⋱

It might seem that using these demo tables, with their simple structure, results in particularly "easy" problem instances for the filter/join/groupby protocols. This is not the case because the protocols are fully oblivious; they cannot "see" the data and take some kind of shortcut based on the structure.

One subtle issue that we need to take into account to get a fair benchmark is related to the column bounds feature: each numeric column maintains a (public) lower and upper bound on its elements (automatically updated throughout a computation), to be able to warn the user about potential overflow. Moreover, certain operations adjust their behavior based on these bounds for efficiency. Because a demo table has tight column bounds by default, we manually re-define those bounds (by means of the `astype` method) for certain operations (i.e., filter and groupby) to ensure that the tables used in those benchmarks have the same bounds, regardless of the number of rows.

We benchmark the filter operation by creating a demo table having 10 columns and a varying number of rows, which we filter with a predicate that includes the given row if the (secret) value in its first column exceeds the constant value 100:

```
import crandas as cd
df = cd.demo_table(num_rows, 10)
filter_result = df.filter(df["col1"].astype("uint24") > 100)
```

We benchmark the join operation by creating two demo tables, each having 10 columns, and join them on the first column:

```
join_result = cd.merge(cd.demo_table(num_rows, 10), \
  cd.demo_table(num_rows, 10), on="col1")
```

Finally, we benchmark the group-by operation on a 10-column demo table with a varying number of rows:

```
groupby_result = cd.demo_table(num_rows, 10).\
  assign(col1=lambda x: x.col1.astype("uint24")).groupby("col1")
```

Below, we first show single-CPU-core benchmark results in Table 2 as well as in plots in Figure 4. We then show how hardware parallelism can be used to speed up those operations in Table 3 and Figure 5.

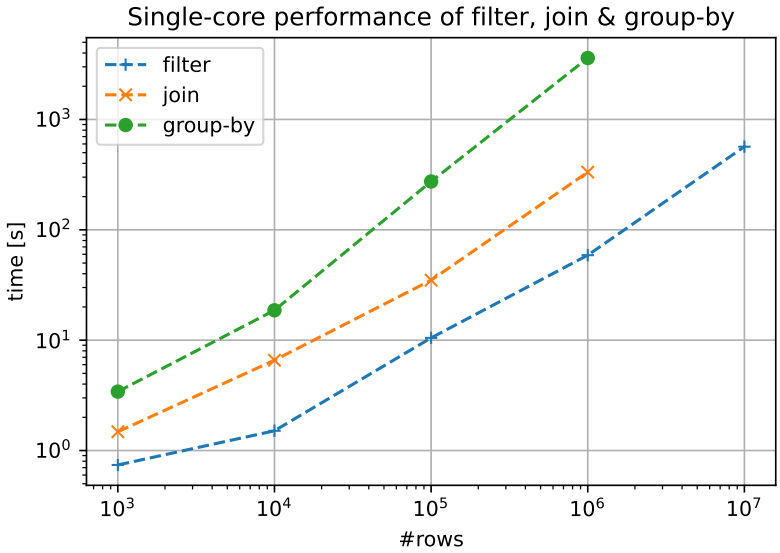

Figure 4: Single-core performance of filter, join and groupby operations. For each benchmarked protocol, the computation time scales approximately linearly in the number of rows. Theoretically, the filter and join indeed have linear complexity in the number of rows, while groupby has $O(n \log^2 n)$ asymptotic complexity (where n denotes the number of rows) because it uses a sorting subroutine with this complexity.

Table 2: Single-core performance of filter, join and group-by.

# rows	compute time (filter)	compute time (join)	compute time (groupby)
1,000	0.74s	1.48s	2.13s
10,000	1.51s	6.58s	18.47s
100,000	10.49s	34.96s	211.59s
1,000,000	58.92s	333.17s	3328.09s
10,000,000	566.84s		

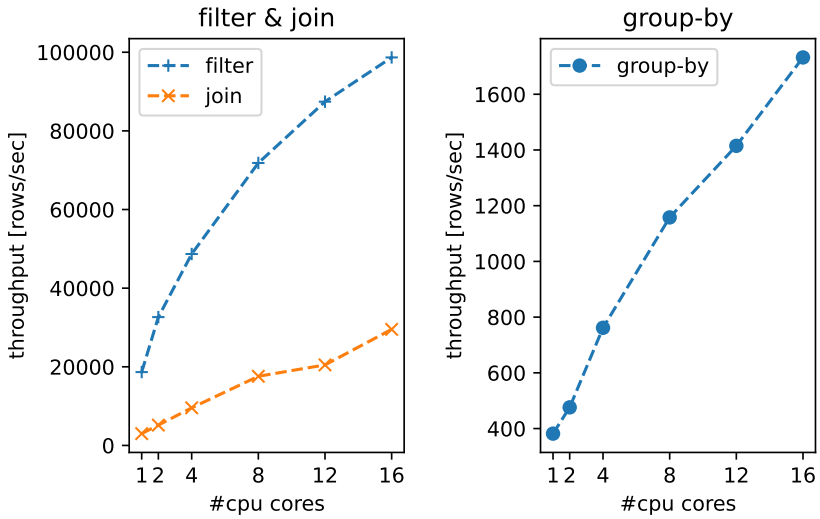

Figure 5: Scaling of throughput (rows/sec) when increasing the number of cores.

Table 3: Multi-core performance of filter, join and group-by. We denote the results as pairs consisting of the number of rows and the computation time. For example, the pair (1e7, 205.40s) means that processing 10,000,000 rows took 205.40 seconds. A reason for adjusting the number of rows depending on the number of cores, is to ensure that the experiments run "long enough" (in the order of hundreds of seconds), to minimize the impact of parts of the computations whose compute time is independent of the data volume.

#cores	(#rows, compute time) (filter)	(#rows, compute time) (join)	(#rows, compute time) (groupby)
1	(1e7, 536.08s)	(1e6, 333.17s)	(1e5, 261.93s)
2	(1e7, 306.39s)	(1e6, 193.48s)	(1e5, 210.04s)
4	(1e7, 205.40s)	(1e6, 104.54s)	(1e5, 131.29s)
8	(1e7, 139.20s)	(5e6, 284.37s)	(5e5, 431.79s)
12	(1e7, 114.38s)	(5e6, 244.12s)	(5e5, 353.40s)
16	(1e7, 101.34s)	(5e6, 169.39s)	(5e5, 288.66s)

5 Benchmarking Model Training

We perform our logistic regression experiments with the UC Irvine "Covertype" data set, which is a geological data set for classifying the forest cover type of a piece of land based on various attributes such as elevation, aspect, slope, hill shade and soil-type. The original data set consists of slightly more than half a million examples, each having 54 features,

and 7 class labels. We can easily manipulate this data set for testing purposes, i.e., by reducing the number of classes, examples, and/or features. Because of benchmarking binary logistic regression, we will always restrict the dataset to the classes 1 and 2. Note that there is some imbalance between those class labels, in that there are slightly more than twice as many examples having label 2 than there are examples with label 1.

We performed experiments by training a logistic regression model, using the algorithm provided by each of the libraries under comparison (either GD, SGD or L-BFGS). To get a baseline for the model quality, we have used the `LogisticRegression` (for L-BFGS) and `SGDClassifier` (for SGD) functions from the Python scikit-learn library. For all MPC frameworks, we set the statistical security parameter to 30 bits. We ran the experiments on Intel x86 64-bit, 32-core Xeon 8468 servers with 64GB memory. Those servers reside in the same data center with 1.5ms ping latency; in the MPC literature this is commonly referred to as the LAN setting. Access to the server hardware was kindly provided by Intel Corporation, via their Intel Liftoff program.

5.1 Frameworks/Libraries Compared

Roseman Labs Platform (crandas). crandas is the Python interface of our MPC platform, which uses three-server (3-party) Shamir-sharing-based MPC in the honest-majority passive security setting. Version used: v10.1.

Data61 MP-SPDZ. MP-SPDZ is Data61's open-source MPC benchmarking framework. Data61 is the data and digital specialist arm of Australia's national science agency. MP-SPDZ is also a circuit-compiler-based framework, and supports various MPC protocols; we have run benchmarks with the 3-party replicated Mod-2^k variant (`ring.sh`). This variant should be among the fastest protocols implemented in MP-SPDZ. The program `covtype.mpc` program which we used for the benchmark can be found in the appendix. The `SGDLogistic` function takes the `program` object, with which we can optionally provide the `approx` compilation option, which will use a three-piece approximate sigmoid. We included this option in our benchmark, but we did barely notice a performance difference. Version used: v0.3.8.

TNO Secure Learning is TNO's open-source MPC machine learning package. This package is built on top of MPyC, an open-source Python MPC framework developed by Berry Schoenmakers from the Technical University of Eindhoven. TNO is an independent not-for-profit research organization, roughly half-funded by the Dutch government. The logistic regression training functionality of TNO Secure Learning uses gradient descent and has two accuracy modes: a faster but coarse approximation to the exponential function ("APPROX"), and a more accurate – albeit slower – one ("EXACT"). The inference part of the model has not been published by TNO; to be able to compute the Tjur metric and Brier score we have performed the inference part using a scikit-learn model in which we inserted the model parameters trained by TNO's method(s). Version used: v1.1.1.

5.2 Definitions: Epochs and Mini-Batches

To be able to make reasonable comparisons between the various methods and implementations, it is important to understand how each library defines a unit of work. A training epoch is typically defined as one pass over the training data set. The `max_iter` parameter of the scikit-learn functions `LogisticRegression` and `SGDClassifier` is to be interpreted as

the number of training epochs. In a single epoch, the `fit` method of `SGDClassifier` updates the parameters n times, where n is the number of examples in the training data set. A single such update is based on the gradient estimated from a single training sample.

MP-SPDZ also uses mini-batch SGD, but here a single epoch seems to (based on our observations during experiments) make a pass over the entire training data set.

5.3 Experiment: scikit-learn Logistic Regression Training Convergence Behavior

In this and the following experiments, we reduce the Covtype data set to 10,000 examples, which we split into 7000 training examples and 3000 test examples.

We start by running an experiment with scikit-learn, to get some feeling for the convergence behavior of the SGD and L-BFGS algorithms (based on the implementations of those algorithms available in scikit-learn) on our data set. More precisely, we run both algorithms repeatedly, for an increasing number of training iterations, and each time compute the McFadden ρ^2 score of the model, as well as the mean squared error of the model's probability predictions on the test set. In other words, we run both algorithms for 1 iteration, compute the McFadden and Brier scores, then refresh the algorithms' states and re-run the algorithms for two iterations, compute the McFadden and Brier score, etcetera, up to 100 iterations. For SGD, because it is a stochastic method, we repeat the above experiment 10 times, and ensure that within each of those runs, we fix the random coins that the SGD algorithm uses (via the `random_state` option of `SGDClassifier`). Figure 6 shows the result of this experiment. We see that the L-BFGS converges monotonically, whereas SGD has a much more noisy convergence behavior.

5.4 Experiment: Training LR on 7000 Examples, 54 Features, Varying Number of Iterations/Epochs

We now turn our attention to the encrypted-computing training methods. Note that because we compare not only between different algorithms (GD vs SGD vs L-BFGS), but also between different implementations of those algorithms, it is not fully clear to what extent one "training iteration" (or epoch) is comparable across those implementations (see also Section 5.2). Hence, we experiment with varying epochs and focus on comparing the training time versus the model-quality metrics.

For each method, we let it perform the training procedure (for a given number of iterations or epochs) and then compute the McFadden ρ^2 score, and Tjur's D and Brier (MSE) score (the latter two based on the 3000 test examples). Note that the "training time" as reported in the table below excludes the inference step (probability prediction) required for evaluating the Tjur and Brier metrics (see next section for inference benchmarks).

We first compare crandas MPC L-BFGS-based logistic regression training method with scikit-learn as a cleartext-training baseline for various epochs. We see that the methods behave similarly in terms of model quality vs. the number of training epochs.

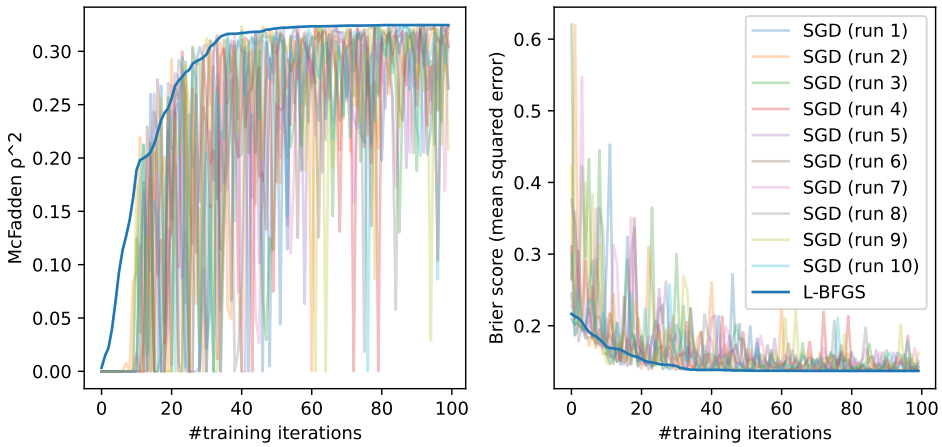

Figure 6: Comparison of the convergence behavior of scikit-learn's L-BFGS-based `LogisticRegression` function and SGD-based `SGDClassifier`. Due its non-stochastic nature, the L-BFGS converges monotonically, whereas SGD has a much more noisy convergence behavior. Note that although the McFadden ρ^2 score can become negative if the model fits poorly, in the left plot we replace negative McFadden scores by zero for the sake of readability.

Table 4: Comparison of crandas' L-BFGS-based logistic regression training to the scikit-learn baseline. We see that the methods behave similarly in terms of model quality vs. the number of training epochs.

Library	Epochs	Optim. method	McF ρ^2	Tjur D	Brier	Training time
scikit-learn	1	L-BFGS	0.0033	0.0077	0.2165	< 1s
crandas MPC	1	L-BFGS	-0.7979	0.0039	0.3008	6.09s
scikit-learn	10	L-BFGS	0.1624	0.1920	0.1764	< 1s
crandas MPC	10	L-BFGS	0.1410	0.1285	0.1811	40.54s
scikit-learn	20	L-BFGS	0.2465	0.2867	0.1562	< 1s
crandas MPC	20	L-BFGS	0.2504	0.2802	0.1545	76.75s
scikit-learn	30	L-BFGS	0.2858	0.3398	0.1446	< 1s
crandas MPC	30	L-BFGS	0.3135	0.3184	0.1454	104.98s

We now show some benchmarks of MP-SPDZ (runtime: Rep3-Mod2^k) on the same dataset. We use mini-batches of size 10. A single training epoch with MP-SPDZ's SGD-based training algorithm already gives a reasonable fit (unlike scikit-learn's SGD implementation, i.e., see Figure 6) and takes roughly 220 seconds (averaged over 3 runs). If we compare these results to Table 4, we see that Crandas finds a superior fit (judging from the higher McFadden score) in about 105 seconds (twice as fast) using 30 L-BFGS iterations.

Table 5: Experiments with MP-SPDZ SGD-based logistic regression training. (Note that we use the three-party passively-secure replicated-secret-sharing-based runtime, Rep3-Mod2^k.) The second last entry uses the piecewise sigmoid approximation, which seems to have only minor impact on the model quality as well as on the running time. The last line shows the result of running 30 epochs with scikit-learn's (non-mini-batch) SGD implementation.

Library	Epochs	Optim. method	McF ρ^2	Tjur D	Brier	Training time
MP-SPDZ	1	SGD	0.2172	0.2526	0.1610	213.81s
MP-SPDZ	1	SGD	0.2263	0.2282	0.1589	221.46s
MP-SPDZ	1	SGD	0.2336	0.2611	0.1562	228.78s
MP-SPDZ	10	SGD	0.3065	0.3410	0.1408	2,429.38s
MP-SPDZ	20	SGD	0.2099	0.3748	0.1601	4,906.12s
MP-SPDZ	30	SGD	0.3120	0.3915	0.1375	7,375.35s
MP-SPDZ ("approx")	30	SGD	0.3180	0.3855	0.1372	7,210.15s
scikit-learn	30	SGD	0.2832	0.3923	0.1436	

For TNO's Secure Learning library we have to manually choose a step size for the optimizer (chosen to be a fixed step size 0.1). The results are shown in Table 6. Both the faster 'approx' and slower 'exact' variant fail to produce good fits for the explored parameter space.

Table 6: Experiments with TNO Secure Learning GD-based logistic regression training. The (fixed) step size is chosen to be 0.1. The method fails to find a good model fit, even after 30 training epochs. The lack of multi-threading partly explains the long running times.

Library	Epochs	Optim. method	McF ρ^2	Tjur D	Brier	Training time
TNO-approx	10	GD	-0.0482	0.0124	0.2301	857.33s
TNO-approx	20	GD	-0.0325	0.0251	0.2254	1,581.22s
TNO-approx	30	GD	-0.0227	0.0327	0.2225	2,307.12s
TNO-exact	10	GD	0.0228	0.0155	0.2109	17,240.49s
TNO-exact	20	GD	0.0490	0.0370	0.2035	34,617.72s
TNO-exact	30	GD	0.0720	0.0605	0.1973	50,061.19s

5.4.1 Experiment: LR Inference on 100,000 Examples, 54 Features

Here we compare the performance of evaluating the logistic regression model (inference), namely to let the model predict probabilities (the `predict_proba` method in scikit-learn). In the table below, we have measured the inference time, and computed from this the inference rate. We do not show inference benchmarks for TNO's Secure Learning library, as its inference procedure is not part of the publicly available software library.

Table 7: Inference benchmarks

Library	Inference time	Inference rate (examples/sec)
crandas MPC	39.34s	2542
MP-SPDZ	32.40s	3086
MP-SPDZ (approx)	32.24s	3102

6 References

[G22] Gartner, “Gartner Identifies Top Five Trends in Privacy Through 2024”, 2022.

[MF75] Daniel McFadden, “Urban Travel Demand: A Behavioral Analysis”, Chapter 5, 1975.

[MF77] Daniel McFadden, “Quantitative Methods for Analyzing Travel Behavior of Individuals: Some Recent Developments”, 1977, page 35.

[Tj09] Tue Tjur, Coefficients of Determination in Logistic Regression Models—A New Proposal: The Coefficient of Discrimination, 2009.

7 Appendix

7.1 Computing McFadden’s ρ^2 in Python

```
import numpy as np
def mcfadden_pseudo_r2(X, y, w, intercept):
    """
    Compute McFadden's pseudo-R^2 metric, which is a metric for
    judging the quality of a fitted logistic regression model.

    The interpretation according to McFadden is:
    "[...] values of .2 to .4 for [rho_2] represent an excellent
    fit." (See "Quantitative Methods for Analyzing Travel
    Behaviour of Individuals: Some Recent Developments", 1977,
    page 35.)

    Parameters
    ----------
    X : 2d numpy array of numbers
        m-by-n data matrix with m examples each having n features
    y : length-m numpy array, each value \in {0,1}
        labels corresponding to the examples in X
    w : length-n numpy array
        model parameters
    intercept : scalar
        intercept value of the model
    """
    X_dot_w = np.dot(X,w.transpose())
```

```
    yy = y.reshape(X_dot_w.shape)
    log_likelihood_func = lambda x: \
        np.sum(-np.log(1 + np.exp(x)) + yy * x)
    ll_model = log_likelihood_func(intercept + X_dot_w)
    y_mean = np.mean(y)
    beta0 = np.log(y_mean / (1 - y_mean))
    ll_null = log_likelihood_func(beta0)
    return 1 -  ll_model / ll_null
```

7.2 Computing Tjur's D and Brier score in Python

```
import numpy as np
def TjurD(labels, predictions):
    """
    Compute Tjur's D metric, which is a metric for judging the
    quality of a fitted logistic regression model.

    It holds that D \in [0, 1], where D=0 means that the model
    has no predictive power and D=1 would say that the model
    is a perfect predictor.

    Parameters
    ----------
    labels      : length-m numpy array, with each value in {0,1}
                  class labels of the test set
    predictions : m-by-2 matrix, with probability predictions
                  made by the model on the test set
                  (each row has the form (p, 1-p) with
                  p \in [0,1])
    """
    return np.mean(predictions[labels == 0,0]) \
           - np.mean(predictions[labels == 1, 0])

def Brier(labels, predictions):
    return np.sum((predictions[:,1] - labels)**2)/len(labels)
```

7.3 MP-SPDZ Script for Log.Reg. Experiment

On each server, the MP-SPDZ compiler was invoked with the following options:

```
./compile.py -E ring covtype
```

and, to enable the approximate sigmoid function, with:

```
./compile.py -E ring covtype approx
```

Then, to run the program interactively over the network, the program was invoked on each server with:

```
./replicated-ring-party.x <party id> covtype -S 30 -h <party 0 ip>
```

The MP-SPDZ program is given below.

```
# Programs/Source/covtype.mpc
program.use_trunc_pr = True
program.use_edabit(True)

DATASET_CUTOFF_SIZE = 10000
DATASET_CUTOFF_SIZE2= 100000
FEATURES = 54
TEST_SPLIT = 0.3
TRAINING_ITERATIONS = 30
MINI_BATCH_SIZE = 10

import pickle, pathlib
from sklearn.datasets import fetch_covtype
from Compiler import ml
ml.set_n_threads(32)

path = 'dataset.bin'
if pathlib.Path(path).is_file():
    with open(path, 'rb') as infile:
        dataset = pickle.load(infile)
else:
    dataset = fetch_covtype()
    with open(path, 'wb') as outfile:
        pickle.dump(dataset, outfile, 5)

# The covtype dataset has labels 1 up to 7.
# We create a dataset with only two labels (1, 2) out of this
# suited for binary logistic regression

binary_filter = (dataset['target'] == 1) | (dataset['target'] == 2)

X = dataset['data'][binary_filter]
y = dataset['target'][binary_filter]

print('Dataset size after restricting to two classes:', len(X))

data_range = range(0, DATASET_CUTOFF_SIZE)

from sklearn.preprocessing import MinMaxScaler
Xnorm = MinMaxScaler(feature_range=(-1, 1)).\
  fit_transform(X[data_range, :FEATURES])

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(Xnorm, \
  y[data_range], test_size=TEST_SPLIT, random_state=42)

X_train_mpc = sfix.input_tensor_via(0, X_train)
```

```
y_train_mpc = sint.input_tensor_via(0, y_train-1)
X_test_mpc = sfix.input_tensor_via(0, X_test)
y_test_mpc = sint.input_tensor_via(0, y_test-1)

from Compiler import library

# Benchmark training on 7,000 rows
library.start_timer(1)
log = ml.SGDLogistic(TRAINING_ITERATIONS, MINI_BATCH_SIZE, program)
log.fit(X_train_mpc, y_train_mpc)
library.stop_timer(1)

# Benchmark inference on 100,000 rows
data_range = range(0, DATASET_CUTOFF_SIZE2)
Xnorm2 = MinMaxScaler(feature_range=(-1, 1))\
    .fit_transform(X[data_range, :feature_range])
X_infbench_mpc = sfix.input_tensor_via(0, Xnorm2)
library.start_timer(2)
pred = log.predict_proba(X_infbench_mpc).reveal()
library.stop_timer(2)

# Reveal predictions on 3,000 rows for quality estimation
print_ln('%s', (log.predict_proba(X_test_mpc)).reveal())
print_ln('%s', (log.opt.layers[0].W).reveal())
print_ln('%s', (log.opt.layers[0].b).reveal())
```

7.4 TNO Secure Learning Script for Log.Reg. Experiment

The script has been adapted from a demo script from the TNO Secure Learning distribution.

```
import numpy as np
from mpyc.runtime import mpc
import time
from tno.mpc.mpyc.secure_learning import (
    ExponentiationTypes,
    Logistic,
    PenaltyTypes,
    SolverTypes,
)

# Fixed random state for train-test-split reproducibility
STATE = 42

# tolerance inactive in practice; max_iter is reached earlier
tolerance = 1e-4

secnum = mpc.SecFxp(l=64, f=32)

def load_data():
```

```
    DATASET_CUTOFF_SIZE = 10000
    FEATURES = 54
    TEST_SPLIT = 0.3

    import pickle, pathlib
    from sklearn.datasets import fetch_covtype
    path = 'dataset.bin'
    if pathlib.Path(path).is_file():
        with open(path, 'rb') as infile:
            dataset = pickle.load(infile)
    else:
        dataset = fetch_covtype()
        with open(path, 'wb') as outfile:
            pickle.dump(dataset, outfile, 5)

    # The covtype dataset has labels 1 up to 7.
    # We create a dataset with only two labels (1, 2) out of this
    # suited for binary logistic regression
    binary_filter = (dataset['target'] == 1) | \
                    (dataset['target'] == 2)
    X = dataset['data'][binary_filter]
    y = dataset['target'][binary_filter]
    print('Dataset size after restricting to two classes:', len(X))
    data_range = range(0, DATASET_CUTOFF_SIZE)

    from sklearn.preprocessing import MinMaxScaler
    Xnorm = MinMaxScaler(feature_range=(-1, 1)).\
        fit_transform(X[data_range, :FEATURES])

    from sklearn.model_selection import train_test_split
    return train_test_split(Xnorm, y[data_range], \
         test_size=TEST_SPLIT, random_state=STATE)

def get_mpc_data(X, y):
    X_mpc = [[secnum(x, integral=False) for x in row] \
            for row in X.tolist()]
    y_mpc = [secnum(y, integral=False) for y in y.tolist()]
    return X_mpc, y_mpc

def distribute_data_over_players(X_mpc, y_mpc):
    X_shared = [mpc.input(row, senders=0) for row in X_mpc]
    y_shared = mpc.input(y_mpc, senders=0)
    return X_shared, y_shared

async def logistic_regression_example():
    print(
        "Logistic regression training with l2 penalty, \
         with gradient descent method"
    )
```

```
alpha = 0.1

X_train, X_test, y_train, y_test = load_data()
X, y = X_train, y_train-1
# Transform labels from {0, 1} to {-1, +1}.
y = [-1 if x == 0 else 1 for x in y]
X = np.array(X)
y = np.array(y)
X_mpc, y_mpc = get_mpc_data(X, y)

async with mpc:
    X_shared, y_shared = \
      distribute_data_over_players(X_mpc, y_mpc)

# Train secure model with approximation of logistic function
# (faster, less accurate)
model = Logistic(
    solver_type=SolverTypes.GD,
    exponentiation=ExponentiationTypes.APPROX,
    penalty=PenaltyTypes.L2,
    alpha=alpha
)

start = time.time()
async with mpc:
    coef_approx = await model.compute_coef_mpc(
        X_shared, y_shared, tolerance=tolerance, \
        nr_maxiters = 30
)
stop = time.time()

print("Training approx took :", stop - start)
print("Coefficients (approximated exponentiation):", coef_approx)

# Train secure model with exact logistic function (slower, more accurate)
model2 = Logistic(
    solver_type=SolverTypes.GD,
    exponentiation=ExponentiationTypes.EXACT,
    penalty=PenaltyTypes.L2,
    alpha=alpha
)

start2 = time.time()
async with mpc:
    coef_exact = await model2.compute_coef_mpc(
        X_shared, y_shared, tolerance=tolerance, nr_maxiters = 30
    )
stop2 = time.time()
```

```
        print("Training exact took :", stop2 - start2)
        print("Coefficients (exact exponentiation):", coef_exact)

if __name__ == "__main__":
    mpc.run(logistic_regression_example())
```