Scalable Oblivious Data Analytics

D4.4 Proof-of-concept for results of tasks 1.3 and 2.3: Implementations of key MPC technologies

Paul Koster (PHI), Jonas Lindstrøm (ALX), Mark Abspoel (PHI), Berry Schoenmakers (TUE), Niek Bouman (TUE), Frank Blom (TUE)

The project SODA has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731583.

Project Information

Scalable Oblivious Data Analytics

Project number:	731583
Strategic objective:	H2020-ICT-2016-1
Starting date:	2017-01-01
Ending data:	2019-12-31
Website:	https://soda-project.eu/

Document Information

Title:	Proof-of-concept for results of tasks 1.3 and 2.3: implementations of key MPC technologies				
ID:	D4.4	Type:	OTHER	Dissemination level:	PU
		Month:	M36	Release date:	22-12-2019

Contributors, Editor & Reviewer Information

Contributors (person(partner): sections)	Paul Koster (PHI): 5, 6, 7 Jonas Lindstrøm (ALX): 2, 3 Mark Abspoel (PHI): 7 Berry Schoenmakers (TUE): 4, 8, 9 Niek Bouman (TUE): 8, 9 Frank Blom (TUE): 8
Editor (person/partner)	Jonas Lindstrøm (ALX)
Reviewer (person/partner)	Tore Kasper Frederiksen (ALX)

Release History

Release num-ber	**Date issued**	**Release description / changes made**
1.0	22-12-2019	Initial release

SODA Consortium

Full Name	Abbreviated Name	Country
Philips Electronics Nederland B.V.	PHI	Netherlands
Alexandra Institute	ALX	Denmark
Aarhus University	AU	Denmark
Göttingen University	GU	Germany
Eindhoven University of Technology	TUE	Netherlands

Table 1: Consortium Members

Executive Summary

This report summarizes the deliverable D4.4 proof-of-concept implementations developed in the SODA project, particularly related to tasks 1.3, 2.3 and 3.1. These proof-of-concepts includes new releases of the MPC frameworks, FRESCO and MPyC, including the implementation of general differential privacy functionality in both. The proof-of-concepts also include privacy-preserving implementations of data analytics algorithms, including logistic regression, decision tress and random forests, ridge regression and binarized neural networks.

About this Document

Role of the deliverable

This deliverable consists of proof-of-concept implementations developed in task T4.1 in the SODA project so far. The implementations are based on the results of tasks T1.3, T2.3 and T3.1. The proof-of-concept deliverables are available as open-source solutions online, and this report gives a brief description of each prototype and links to where the source code can be obtained.

Relationship to other SODA deliverables

This deliverable describes the proof-of-concept implementations created as part of task T4.1 which are based on tasks 1.3 and 2.3 (special-purpose protocols and use-case specific algorithms). Deliverable 4.2 consisted of proof-of-concept implementations created as part of task T4.1 based on tasks 1.2 and 2.2 (general purpose protocols and algorithms), and some the proof-of-concepts presented here are extensions of the proof-of-concepts presented in D4.2 deliverable. Some of the proof-of-concept implementations provide a basis for the demonstrators of D4.5 in task T4.2.

Relationship to other versions of this deliverable

N/A

Acknowledgements

Besides the authors this work reflects implementation efforts from: Stefan van Oord, Mark Spanbroek, Nikolaj Volgushev and Kimberley Thissen.

1 Table of Contents

2 Introduction

This deliverable describes the status of proof-of-concept implementations developed in the SODA project as part of task T4.1. The main part of the deliverable are the actual proof-of-concept implementations, most of which are open-source with source code available online. This report gives a brief overview of the features and status of the different implementations.

The MPC frameworks developed by the partners in SODA have been extended in the reporting period. The FRESCO framework has been improved and hardened, and libraries for statistics, differential privacy, machine learning and for outsourced MPC have been developed. These efforts are described in chapter 3. The MPyC framework has also been extended with new functionality for statistics and new demo applications, and three new versions have been released in the reporting period. This is described in chapter 4.

The SODA deliverable D3.4 on leakage control introduced “Achieving Differential Privacy in Secure Multi-Party Computation”, and the accompanying proof-of-concept implementation is presented in chapter 5.

The SODA project aims at providing privacy-preserving data analytics tools and has developed several prototypes of common data analytics and machine learning algorithms based on MPC and differential privacy. In particular, a prototype of the commonly used logistic regression using both MPC and differential privacy has been developed and the functionality and features as well as performance benchmarks are presented in chapter 6. Another popular concept in machine learning is decisions trees, and an MPC based algorithm for training decision trees was presented in D2.3, and an implementation of the algorithm is presented in chapter 7. An implementation of ridge regression, which is a regularized version of linear regression, is presented in chapter 8. An implementation of a secure multilayer perceptron (MLP) for recognizing MNIST handwritten digits is presented in chapter 9.

Partners shared the code of the proof-of-concept implementations with the project and the general public, e.g. on GitHub under open source or research licenses. The location is provided for each of the prototypes at the end of their corresponding chapters.

3 FRESCO

FRESCO is a Java framework for efficient secure computation aiming at making the development of prototype applications based on secure computation easy.

As part of this work package we have made substantial improvements and expansions to the FRESCO framework with focus on enhancing stability, usability and security. Extensions in functionality focus on data analysis, machine learning and statistics.

Philips used FRESCO as base for one of their PoC's and the feedback from their developers has been an important driver in the development.

3.1 General Improvements

There have been three releases of FRESCO since the last reporting in D4.2, versions 1.1.3, 1.2.0 and 1.2.1. The main improvements have been:

- Remove the dependency on the SCAPI library since SCAPI for Java is no longer being maintained,
- Add an implementation of MiMC with reduced encryption rounds for use as a PRF as described in Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart (2016): *MPC-Friendly Symmetric Key Primitives*. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16),
- Docker files for working with FRESCO in a Docker container and a demo on how to use it,
- Abstraction of the underlying field used in arithmetic protocols and two implementations: Modulo a prime using Java's BigInteger class and modulo a pseudo Mersenne prime modulus with faster modular arithmetic,
- Security bug fix in SPDZ mac checking protocol (reported during the hacking challenge, but the reporter did not want to sign up for any prizes),
- Reimplementation of fixed-point functions based on master thesis by Kimberley Thissen (from TU/e and Philips) to give more predictable results (on development branch),
- Faster modular reduction using Barrett reduction with precomputed values (on development branch).

3.2 FRESCO libraries

We aim at keeping the core of FRESCO as small as possible and are working towards an approach where almost anything except the framework and arithmetic functionality are kept in separate modules. For now, we have defined the following modules: fresco-stat, fresco-ml, fresco-dp and fresco-outsourcing:

- **fresco-ml:** This library is for machine learning including evaluation of neural networks, federated learning, decision trees, SVM evaluation and logistic regression.
- **fresco-dp:** Differential privacy functionality for FRESCO. Current functionality includes the Laplace and Exponential mechanisms, noisy descriptive statistics and a noisy χ^2-test based on Gaboardi et.al, Differentially Private Chi-Squared Hypothesis Testing: Goodness of Fit and Independence Testing.
- **fresco-outsourcing:** Tools to work with FRESCO in the outsourced MPC setting. I.e., the MPC setting where not all parties actually run the MPC protocol. Instead, only a limited number of parties, called *servers* will run the MPC protocol, while a larger number of parties called *clients* will supply the inputs and receive outputs from the computation performed by the servers. This should be done so that the in/outputs of the clients should be protected similarly to if they were directly participating in the MPC protocol. Such a setup is often more scalable than having all parties participate directly in the computation when

a large number of parties are involved. The implementation is based Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt and Tomas Toft (2015): “Confidential Benchmarking based on Multiparty Computation.” IACR Cryptology ePrint Archive 2015.

- **fresco-stat:** This library provides functions for descriptive statistics (sample mean, std deviation, variance, histograms and correlation) using fixed point numbers as well as statistical tests (student’s t-test, χ2-test), simple linear regression and sampling over continuous distributions (Bernoulli, categorical, exponential, Irwin-Hall, Laplace, Gaussian, Rademacher and uniform distributions).

3.3 Resources

FRESCO is available at GitHub at https://github.com/aicis/fresco, and the documentation can be found at Read the Docs at https://fresco.readthedocs.io/en/latest/.

The public channel where we receive questions, bug reports and suggestions on how to improve FRESCO can be found at https://gitter.im/FRESCO-MPC/Lobby

The statistics library built on FRESCO can be found at https://github.com/aicis/fresco-stat.

The library for differential privacy built on FRESCO can be found at https://github.com/aicis/fresco-dp.

Tools for making outsourced applications with FRESCO can be found at https://github.com/aicis/fresco-outsourcing.

4 MPyC

4.1 Introduction

MPyC is a Python package for secure multiparty computation. Development of the package is ongoing and several larger demo applications are available. The MPyC package is written in pure Python and can be used in combination with existing Python packages and tools. In particular, MPyC programs can be developed in Jupyter notebooks (in Jupyter lab). Usability is a major goal, with full and direct access to the framework for developers, while at the same time offering reasonable or simply adequate performance for non-trivial tasks.

As part of this work package we have extended the functionality of the MPyC framework, in particular, to support the demo applications for SODA.

4.2 Functionality and features

Since the launch of MPyC at TPMPC 2018 (on May 30th), the framework has been extended significantly. The following tagged releases appeared after July 1, 2018 on the Python Package Index at pypi.org: release 0.4 in October 2018; release 0.5 in March 2019; release 0.6 in December 2019.

Some major additions are:

- support for SSL/TLS (MPC parties communicate via point-to-point TLS connections)
- switch to dynamic setup for PRSS (Pseudo Random Secret Sharing)
- if_else() for oblivious branching
- secure integer division for arbitrary (secret) dividend and arbitrary (public) divisor
- SecFld() supporting secure extension fields (in particular, efficient binary extension fields)
- secure conversions between SecInt(), SecFxp(), SecFld() types
- help messages for command line interface
- support for up to m=256 parties on Windows and Linux

Addition of the following core modules:

- **mpyc.random**: secure versions of several functions for generating pseudorandom numbers, cf. the random module in the Python standard library
- **mpyc.mpctools**: currently provides alternative implementations for two functions in Python's itertools and functools modules, respectively. The round complexity is reduced from linear to logarithmic
- **mpyc.statistics**: secure versions of common mathematical statistics functions, modelled after the statistics module in the Python standard library
- **mpyc.seclists**: secure (or, oblivious) alternative to Python lists, with secret-shared list items and oblivious access to list items

4.3 Applications

The following applications are included as major demos in the MPyC repository:

- bnnmnist.py, cnnmnist.py: neural networks for classifying MNIST dataset of hand-written digits, respectively, using a binarized multilayer perceptron and a convolutional neural network
- id3gini.py: ID3 decision trees
- kmsurvival.py: Kaplan-Meier survival curves and logrank tests
- ridgeregression.py: ridge regression for datasets with up to 11 million samples
- lpsolver.py, lpsolverfxp.py: linear programming using secure integers and secure fixed-point numbers, respectively

- aes.py, onewayhashchains.py: AES with secret-shared plaintext, key, ciphertext, and secret-shared one-way hash chain reversal (Lamport’s identification scheme) using an AES-based one-way function

In addition, several Jupyter notebooks are included to give further explanation.

4.4 Resources

MPyC homepage: https://www.win.tue.nl/~berry/mpyc/

GitHub repository at https://github.com/lschoe/mpyc

Python package index entry: https://pypi.org/project/mpyc/

5 Differential Privacy

5.1 Introduction

SODA deliverable D3.4 introduces “Achieving Differential Privacy in Secure Multi-Party Computation”. It discusses how differential privacy can complement MPC. Specifically, how it can provide privacy guarantees on outputted data such as intermediate or the final results. This work was accompanied by a proof-of-concept implementation made in the MPyC framework.

5.2 Functionality and features

Differential privacy requires several functions that are performance critical in MPC context. The proof-of-concept code comprises the building blocks and mechanisms to create differentially private noise. Specifically, the following was implemented:

- functions: 2x, ex, ln(x), sqrt(x), cos(x), sin(x)
- noise generating mechanisms: Laplace, staircase, geometric, Gaussian

The implementation achieved the targeted performance and accuracy objectives. The mathematical functions are not limited to differential privacy, but are generally usable.

5.3 Applications

Differential privacy was applied to logistic regression as described in D3.4 chapter “Differentially Private Logistic Regression” and the next section. It is also this application that defines and implements the functionality to calculate e.g. the applicable differential privacy sensitivity parameter.

As reported in D4.2, the logistic regression implementation is made using the FRESCO framework. This means that the MPyC implementation of the differential privacy functions could not be used directly, but that the approach was adopted by ALX in the FRESCO framework. Similarly, FRESCO also improved some of its mathematical functions to achieve the necessary accuracy and performance. The same holds for Laplacian noise. On the other end, the logistic regression code implements the Laplacian and normal noise generation mechanisms, which provides the basis for a gamma noise distribution.

5.4 Resources

In the future it is expected that the functions and mechanisms will be part of MPyC: https://github.com/lschoe/mpyc

For FRESCO, the new function and mechanism implementations can be found at: https://github.com/aicis/fresco and https://github.com/aicis/fresco-stat

Resources for the application in logistic regression are referenced in the next section.

6 Logistic Regression

6.1 Introduction

Logistic regression is one of the machine learning algorithms to play a role in the project demos. The approach for differentially private logistic regression is described in D3.4 chapter “Differentially Private Logistic Regression”. An early version of a logistic regression implementation without differential privacy was presented in D4.2 chapter 8. The final logistic regression implementation described here forms the basis of the predictive modelling demo with logistic regression for chronic heart failure risk reported in deliverable D4.5.

6.2 Functionality and features

The logistic regression implementation is created in Java using the FRESCO 2.0 framework. The logistic regression algorithm exposes the following main features:

```
java -jar logistic-regression-jar-with-dependencies.jar

Usage: LogisticRegression [-hV] [--debug] [--dummy] [--dummy-data-supplier]
                          [--trace] [--iterations=<iterations>]
                          [--lambda=<lambda>] [--max-bit-length=<maxBitLength>]
                          [--mod-bit-length=<modBitLength>]
                          [-b=<privacyBudget>] -i=<myId> -p=<parties>[:
                          <parties>...] [-p=<parties>[:<parties>...]]...
Secure Multi-Party Logistic Regression
      --iterations=<iterations>
                          The number of iterations performed by the fitter. If
                            omitted, the default value is 5.
      --lambda=<lambda>   Lambda for fitting the logistic model. If omitted, the
                            default value is 1.0.
      --max-bit-length=<maxBitLength>
                          For experimenting; default is 200
      --mod-bit-length=<modBitLength>
                          For experimenting; default is 512
  -b, --privacy-budget=<privacyBudget>
                          Enables using differential privacy for updating the
                            weights given this budget. If omitted, differential
                            privacy is not used.
```

The input data uses a json format over standard input (stdin) when used from the command line (or the applicable Java interface when used from Java). The current data loading implementation assumes a horizontally segmented data set, but in principle the algorithm is generic.

It outputs the coefficients beta and the intercept as a vector to standard output. Intermediate outputs are optionally logged.

The logistic regression parameters are the number of iterations and the regularization parameter lambda.

The differential privacy parameters are the privacy budget epsilon. When activated, differentially private noise is added to all (intermediate) coefficients. Use of this option requires normalization of input data before being given as input.

The bit length parameters allow for faster computation. To prevent overflows normalization of input data is recommended.

6.3 Performance

Figure 1 depicts performance numbers for 3 data sets of different sizes (cars 32x3, breast 588x9, heart 2476x12). It measures wall clock hours needed to learn the model (5 iterations) in a two-party active secure setting, with both parties on the same host in a Linux compute cluster. The time needed includes all processing; including preprocessing and the online phase. The cluster is a shared resource, so all timings are indicative only.

The improved implementation achieves a ~40x speedup amongst others due to use of a 128 modulo length rather than 512 in combination with other improvements and bug fixes in both FRESCO and the logistic regression implementation. Smaller moduli for the larger data sets also require the input data to be normalized in order to prevent overflows. The larger heart failure data set runs in about 4 hours.

The algorithm has also been run on different hosts in the Linux compute cluster connected with a high performance LAN, which lead to comparable result (differences less than variance by other causes).

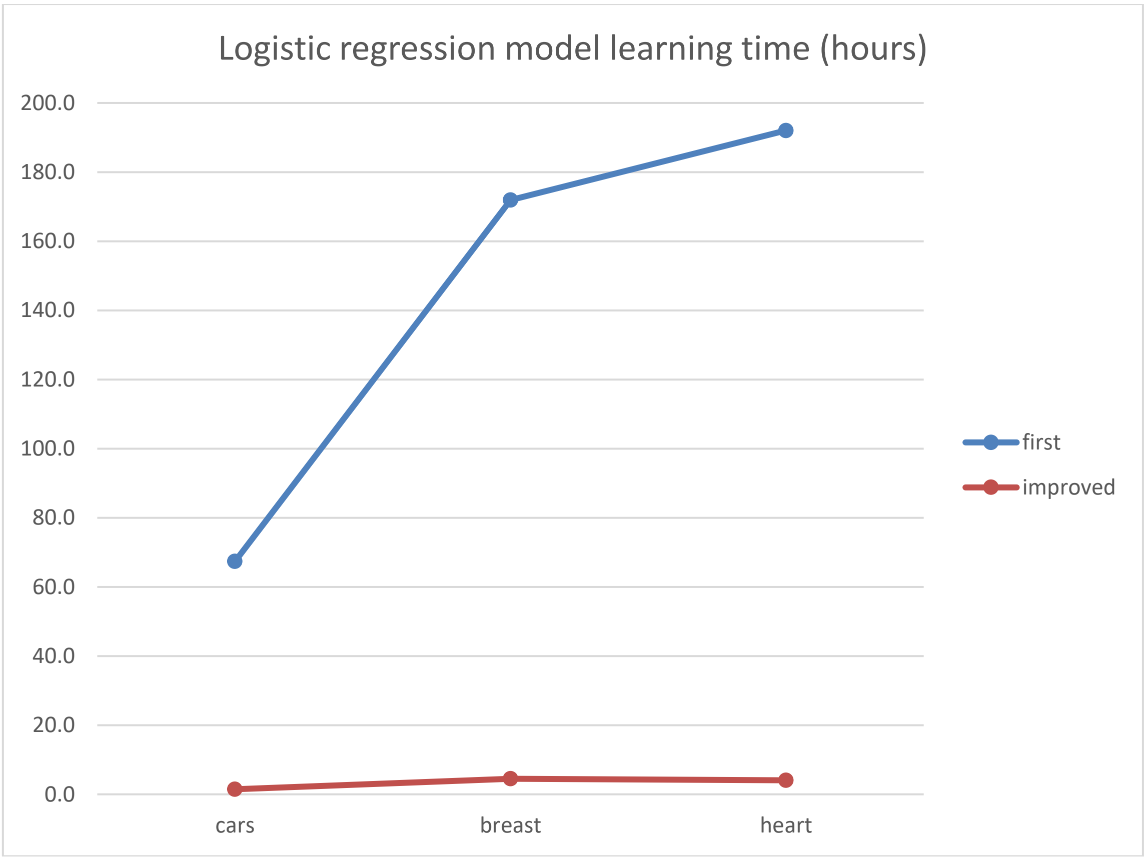

Figure 1: performance of logistic regression implementation

Table 2 presents a breakdown of computation time per function of the algorithm as measured for a run for the different datasets. Note that Hessian and Cholesky are computed once and forward substitution, backward substitution and generation of noise for differential privacy per iteration, i.e. 5x here.

The table also identifies opportunities for optimizations. Most obvious for larger data sets is to leave out differential privacy. Already for the heart feature dataset the noise added becomes relatively small, and could be left out, ensuring privacy simply by aggregation. The other optimization is to generate differential private noise only once per iteration. The current implementation generates noise per party, which can be optimized. This also provides an opportunity to generate smaller total noise.

Table 2: Breakdown of time spent per function

	dataset	**Hessian**	**Cholesky**	**forwardsub**	**backwardsub**	**gennoise1**	**gennoise2**
1	cars	1%	8%	21%	19%	27%	25%
2	breast	5%	13%	21%	22%	19%	20%
3	heart	10%	10%	21%	22%	18%	18%

From the time needed, more than 99% is spent on preprocessing. The online MPC phase takes less than 1%. For example, in 4 hours this is less than 1 minute. This also identifies a significant opportunity for optimization, because this version of FRESCO is not optimized for preprocessing. It does preprocessing on the fly with the computations, but e.g. does not fully exploit parallelization opportunities. It also does not support offline preprocessing.

When simulating offline preprocessing (--dummy-data-provider) we can also test other scenarios. For example, jointly learning the small “cars” data set with an average business laptop and a host in aforementioned compute cluster connected via a residential internet connection and VPN (14ms round-trip time) can be done in 8 minutes. The larger “heart failure” data set gets trained in 57 minutes vs. 39 seconds (88x) on the same host in the compute cluster or 70 seconds (49x) on the same laptop. Extrapolation of these results indicate that the case with preprocessing under aforementioned pessimistic network conditions could result in 100+ hours, but in realistic real-world settings of high performance cloud environments likely closer to the aforementioned 4 hours.

6.4 Applications

The logistic regression algorithm is applied in a demo setting in D4.5. This demo uses the algorithm to learn a risk predictive model for chronic heart failure. It does this by following a typical workflow of a data scientist with Jupyter Notebook and scikit-learn and a real-world data sets originating from multiple parties.

6.5 Other remarks

A side effect of developing the logistic regression machine learning on the FRESCO framework by an outside non-expert team has been that it used and tested various aspects of the framework and its documentation and examples in new ways. This has led to several improvements, bug fixes, added features and plans to make it easier to use.

6.6 Resources

The first version is available at https://github.com/philips-software/fresco-logistic-regression-2 . The improved version is available at https://github.com/jonas-lj/fresco-logistic-regression-2 .

7 Decision trees

7.1 Introduction

We developed a novel algorithm for securely training a decision tree with continuous attributes, as discussed in deliverable D2.3. As input for the algorithm, we assume a database that is secret-shared among multiple mutually distrustful parties. This situation is very general and encompasses for instance horizontally or vertically partitioned datasets. The database consists of a number of samples (rows), each containing values for a set number of attributes (either discrete or continuous), together with a binary output value.

Using the database, the algorithm securely trains a classification tree, i.e. a model that aims to predict the output value given the attribute values. Our secure algorithm produces this tree in secret-shared form — we train the tree up to a constant depth, so that the shape of the tree is publicly known and give as output the secret-shared data corresponding to each node in the tree.

We have implemented this algorithm in the MP-SPDZ framework, which allows us to benchmark the algorithm with various underlying protocols. We evaluated the protocols with 3 parties, and arithmetic modulo 2^{64}, both with passive and active security.

7.2 Functionality and features

The implementation uses a fixed (but configurable) domain size for secret-shared values. For our benchmarks we used 2^{64}. Continuous attributes are supported with value within this domain. The domain size should be at least 5*(log(N) - 1) bits, where N is the number of samples in the dataset, since it needs to represent a Gini index, representing the "goodness of fit" of a node.

We support:

- an arbitrary number of parties, with both passive and active security, as supported by the MP-SPDZ framework
- an arbitrary number of continuous attributes
- discrete attributes are not specifically implemented, but can be seen as a special case of continuous attributes
- a binary output variable

The MP-SPDZ framework uses a Python-based domain specific language, which is compiled to bytecode. This is a local computation (the same for each party), and needs to be done only once for a set of parameters (tree depth, number of attributes, number of samples). The bytecode is then executed by the MPC engine.

One problem with this approach is that the time and memory needed for the compilation step far exceeds the actual execution for reasonable parameters. Practically, we did not compile beyond 8192 samples, a depth 4 tree, and 2 continuous attributes.

7.3 Applications

We apply our methods to a large-scale medical dataset of ~290,000 samples and 128 attributes, with a binary output variable. This dataset was previously used to build a predictive model using gradient boosted trees to predict the risk of emergency hospital transport of elderly patients within the next 30 days. We demonstrate that using a random forest of 200 trees of depth-4 with 11 attributes, we can obtain accuracies that are only slightly worse than the gradient boosted tree model. To obtain these numbers, we trained our decision trees in the clear. By extrapolating numbers (which we show to be a reasonable method), we obtain that training a single one of these trees can be done in approximately 1 hour (with 3 parties on a LAN setting and passive security). Since this can be done in parallel, we conclude that our implementation allows practical training of decision trees for large scale datasets.

7.4 Resources

Abovementioned implementation is currently held in a private repository and has not been published.

An implementation in the MPyC framework inspired by above approach is available at https://github.com/Charterhouse/random_forest which is used in the demo implementation of above-mentioned application reported on in SODA deliverable D4.5.

8 Ridge regression

8.1 Introduction

Ridge regression is a regularized version of linear regression commonly used in machine learning nowadays. We have developed a new MPC protocol for privacy-preserving ridge regression. Our protocol does not assume that any of the input data is held privately by any of the parties running the MPC protocol. Instead it is assumed that the entire dataset is unknown to any of the parties and is available only in secret-shared (or, encrypted) form.

When run in the MPyC setting (honest majority, with Shamir secret sharing), full advantage is taken of the fact that dot products x . y can be evaluated quickly (constant communication complexity, independent of the length of the vectors x and y). The closed form solution for ridge regression is used, which boils down to solving a particular system of linear equations. The Gaussian elimination step for solving the linear system is done entirely over the integers, carefully avoiding rational reconstruction. Compared to previous approaches the size of the prime field needed for Shamir secret sharing is halved, which is a critical improvement, as the sizes of the primes used run into thousands of bits.

8.2 Functionality and features

Given a dataset viewed as a matrix X and a column vector y, where each row of X contains a sample of feature values, and the corresponding entry of y is the observed outcome, a model is computed to predict the outcome for other samples. Due to regularization, the resulting model is robust to outliers.

The implementation is privacy-preserving, and the model is the only information "leaked" about the given dataset.

The performance of the model (root mean square error) is comparable to the performance of Scikit-learn ridge regression.

8.3 Applications

The MPyC demo can be run with synthetic datasets (generated using Scikit-learn), and on a range of datasets from the UCI machine learning repository. A new record is set by processing the HIGSS dataset, which contains 11,000,000 samples in 35 seconds with m=3 parties.

8.4 Other remarks

We have also implemented the protocol of Chapter 1 from Deliverable D2.3: "A Practical Approach to the Secure Computation of the Moore–Penrose Pseudoinverse over the Rationals" in MPyC. This protocol will be made available on the MPyC GitHub repository later.

8.5 Resources

See Deliverable 1.3, Chapter 4 "Efficient Secure Ridge Regression from Randomized Gaussian Elimination, " which is based on https://eprint.iacr.org/2019/773.

MPyC demo ridgeregression.py.

9 Binarized neural networks

9.1 Introduction

Deliverable D2.2 reported on a secure Convolutional Neural Network for recognizing MNIST hand-written digits. To improve upon this solution we have developed an alternative secure Multilayer Perceptron (MLP) for the same task. To take advantage of the fast secure-comparisons based on the Legendre-symbol, a binarized MLP is used. In the binarized MLP the weights and activation levels are all +1 or -1, to ensure that values used in secure comparisons are relatively small. For the comparisons in the first layer, however, we still need to use full-range comparisons, as the values compared are too large for the Legendre-based comparisons.

9.2 Functionality and features

The MNIST dataset of hand-written digits is used to compare the performance of several variants of the binarized MLP. The type of secure comparison can be selected. As a further enhancement, the option of using a vectorised secure comparisons is also provided (all comparisons at the same depth are evaluated jointly).

Hand-written digits can be processed one at a time, or in batches. The average time per digit drops for larger batches.

9.3 Applications

We have run our experiments on a 3PC-LAN setup and measured a 5-fold speedup for the Legendre-based comparisons over (standard) full-range comparisons. Similar speedups may be expected with other MPC frameworks for applications with comparisons restricted to medium-sized integers. The error rates do not vary significantly for the various approaches.

9.4 Resources

See Deliverable 2.2, Chapter 2 “Fast Secure Comparison for Medium-Sized Integers and Its Application in Binarized Neural Networks” which is based on https://eprint.iacr.org/2018/1236.

MPyC demo: bnnmnist.py.