Technische Universität München
Fakultät für Mathematik

TUM Data Innovation Lab
together with d-fine GmbH

Federated Learning:
Collaborative Machine Learning
without Centralized Training Data

Project Documentation

Robin Fritsch
Shayoni Halder
Valentin Hartmann
Dmitrii Petukhov

Supervision: Dr. Patrick Biermann, Dr. Markus Seifert, Dr. Ferdinand Graf,
M.Sc. Todor Dobrikov, Dr. Ricardo Acevedo
Examination: Prof. Dr. Massimo Fornasier

Contents

1 Introduction

The Subprime Mortgage Crisis of 2007-2010 [7], which resulted in a sharp increase of high-risk mortgages that went into default, was the most severe recession in the last decade. While the reasons for this are numerous, the primary was inappropriate assessment of credit risk. Surprisingly, mortgages were extended to those who could not otherwise qualify for traditional loans because of a weak credit history or other disqualifying credit measures. It even ushered in ninja loan [19], meaning loan was extended to borrowers with no income or assets. All of these contributed to a long-term economic downturn.

The crisis illustrated how ignorance in default assessment can pave the way for economy-wide difficulties, pointing to the potential benefits of sophisticated risk management practices. In particular, it brought to light the need for efficient assessment of risk factors, credit scoring and user rating, among others. However, traditional tools may not be ideal to deal with the growing complexity, volume and pace of transactions. Technological innovations stemming from machine learning enable new risk management techniques that allow for better risk decisions. However, they may also expose institutions to risks arising from unaddressed data privacy and protection regulations. This limits the availability of data, which is usually the key in building good models for risk assessment and thereby poses serious challenges in their development.

In particular, it would be desirable for financial institutions to be able to share their data to build better models, since modern machine learning models such as neural networks require huge amounts of data that a single institution might not be able to provide. For example, the different Sparkassen in Germany could collaborate to build a risk assessment model with higher performance than any individual branch would be able to. This would also reduce development and maintenance cost for individual institutions, since it removes the necessity to have the machine learning knowledge for building a model at every branch.

The idea of this project was to develop a framework for doing this kind of model training on distributed data, but without breaking privacy in the process, i.e., protecting the individual institutions' data.

2 Problem Statement

The goal of supervised machine learning is to use (observed) pairs $\{(x_i, y_i)\}_{i=1}^n$ of feature vectors x_i and label vectors y_i — both consisting of numerical and/or categorical entries — to build a function f that, given the feature vector x of an unknown sample, outputs the corresponding label y. In our case, the feature vector's entries are, among others, the average account balance of a bank's customer, their maximal limit utilization and their revenue growth rate, and y is whether the customer will default or not.
Typically the function f to be learned is parametrized by a vector w. Its performance with respect to an individual training sample (x_i, y_i) is measured by a loss function L, $L(x_i, y_i, w)$. The training then consists of minimizing the cost function J, that is, the normalized sum over the losses of all training samples:

$$\text{Find } w^* \in \underset{w \in \mathbb{R}^d}{\arg\min}\, J(w) \quad \text{where} \quad J(w) = \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, w). \tag{1}$$

Often a regularization term is added to the cost function in order to penalize large model parameters. Using L2-regularization this would be $\alpha||w||_2$ where $\alpha > 0$ is the regularization strength.
Our setting, however, differs from the one usually encountered in machine learning in a way that isn't captured by the above formulation: The data isn't present as one dataset; instead, there are multiple parties, "clients", each of which owns a distinct share of the dataset, which is stored on their local hardware. We would, however, like to leverage the information contained in the joint dataset consisting of all individual datasets to train a machine learning model. Since this joint dataset is a lot larger than each of the local datasets it consists of, one would expect a much better performance of a model trained on the joint than just on one of the local datasets. More precisely, there are K clients, each possessing a share $\{(x_{i_j}, y_{i_j})\}_{j=1}^{n_k}$ of the entire dataset. So the overall number of training samples can be partitioned into $n = \sum_{k=1}^{K} n_k$. We are now able to refine the problem formulation (1):

$$\text{Find } w^* \in \underset{w \in \mathbb{R}^d}{\arg\min}\, J(w) \quad \text{where} \quad J(w) = \frac{1}{n} \sum_{k=1}^{K} \sum_{j=1}^{n_k} L(x_{i_j}, y_{i_j}, w). \tag{2}$$

The naïve approach to deal with the partition of the data would be to send all data to a central server and apply a well-established machine learning algorithm to it. But often this is not desirable or even possible. Collecting all data in one place creates an additional, much more valuable attack surface. And in many cases clients are not willing or even able to share their data due to its sensitive nature. Therefore the general problem we are addressing can be summarized in the following way:

> We want to build a model that predicts the probability that a bank's customer will default within one year given a set of this customer's features by solving a minimization problem of the form (2). The training data is spread over multiple clients, and the training algorithm only has access to information derived from each client's dataset that reveals no sensitive information about the underlying data.

In our specific case, there are two additional challenges:

1. We cannot assume that the data is identically distributed, i.e., that the datasets of all clients follow the same underlying distribution. An example would be clients that are located in different countries. Let's assume we have one client in Germany and one in Saudi Arabia. Germany's most important export products are vehicles and machinery, while Saudi Arabia's main source of income is the oil industry. Since those are economic sectors that underlie completely different dynamics, customers from them will be hardly comparable.

2. In most cases, the data is unbalanced, i.e., the sizes of the individual datasets vary, sometimes heavily, since not each client has the same amount of customers, whose data could be collected.

3 Literature Review

3.1 Optimization Algorithms

Common Optimization Algorithms. A detailed survey of algorithms for solving the optimization problem (1) is given in [14]. We give a brief overview here.
The most basic of these algorithms is *Gradient Descent* (GD). Starting from a random point the algorithm moves in the direction of the negative gradient,

i.e., of the steepest descent, in every iteration in order to arrive at a local minimum. However, calculating the gradient in every iteration is impractical for large numbers of samples. *Stochastic gradient descent* (SGD) addresses this problem by randomly choosing a subset of functions to estimate the gradient. This yields cheaper iterations, but adds noise to the gradient which slows down convergence.

There are many methods that aim to improve the convergence speed of GD and SGD. The main idea of *Momentum* [20] is to accelerate GD in the relevant direction towards the minimum by dampening oscillating movements. This is achieved by using a weighted average of the gradients from previous iterations. *RMSProp* addresses the issue that the magnitude of the gradient can be very different for different weights and can change during learning. Since this is hard to handle with a global learning rate, it computes adaptive learning rates for each parameter [22]. *Adam* [13] combines Momentum and RMSProp into a single algorithm.

Recent randomized methods try to, roughly speaking, combine the benefits of cheap iterations of SGD with fast convergence of GD. One of these is *randomized coordinate descent* (RCD) [18]. Here a random coordinate is chosen in each iteration and GD is performed in that direction, which can be done efficiently for sparse data. Another possibility is applying RCD to the dual problem of (1), which is called *stochastic dual coordinate descent* [26].

Distributed Optimization Algorithms. In a distributed setting in which the data is distributed across several machines (e.g., because it does not fit on a single one) the communication costs become an additional factor that needs to be considered. The CoCoA framework [12] is designed for this setting. In each training iteration, each client solves an optimization problem on their local dataset up to a predefined accuracy. The clients' local updates are then accumulated on the server. What makes this framework distinct from other solutions is that clients can use any method they want to solve their local subproblem. Further distributed optimization methods are DANE [25], its accelerated variant AIDE [21], and DiSCO [27].

According to [14], however, these algorithms are not applicable when the data is not identically distributed across the nodes, or in the case of CoCoA do not perform well. McMahan et al. [16] suggest *FederatedAveraging* (FedAvg) for this case. It makes use of the fact that GD and SGD in a distributed setting are equivalent to doing a gradient step locally and averaging the results on the server. In order to reduce the necessary number of rounds of communication FedAvg performs multiple local gradient steps before sending the result to the server for averaging.

3.2 Privacy

Earlier work on privacy in machine learning settings is mostly concerned with modifying sensitive datasets such that they can be published without revealing information that is supposed to be protected. Machine learning can then be done on those modified datasets in a classical offline fashion. The problem with this approach, that doesn't exploit the distributed setting, is that privacy usually comes at a much higher price, since adversaries can perform arbitrary queries on the data and arbitrarily many of them. When, e.g., noise is added, each individual record must be perturbed and not only an aggregate value like the gradient computed over several records. However, this doesn't necessarily have to come with a loss in model performance; see below for an example.

The idea of adding noise to the attributes — specifically Gaussian or uniformly distributed noise — is investigated in [3], where a mathematical notion of privacy is given. The authors propose a method to approximate the original data distribution from the perturbed dataset and apply it to training a decision tree classifier.

k-anonymity is a different notion of privacy: A database is k-anonymous when for each entity in the data and each of the entries in the database corresponding to it, it holds that this entry is indistinguishable from at least $k - 1$ other entries. This can be achieved by suppressing attributes, i.e., removing them from the database, or by generalizing them, i.e., putting an entry together with similarly-valued entries into a common bin. The first algorithm to achieve k-anonymity was proposed in [23].

Another idea is to rotate the datapoints in space by a random rotation matrix [6]. Classifiers that rely on certain geometric properties of the data aren't affected by this, i.e., the accuracy achieved when training them on the rotated dataset is the same as when training them on the original dataset. Those classifiers include support vector machines, the perceptron and the k nearest neighbor classifier.

Note that all of these approaches require the clients to share some kind of information without being able to rely on a trusted server. See Section 5.3.2

for a discussion of possible ways to do this.
A relatively recent branch of privacy research is that of differential privacy, introduced in 2006 [9], which we will investigate in more detail in 5.3.1. The general idea is, as opposed to the first three approaches, that the owner does not share their database; it stays on their device. Instead, the owner accepts queries to their database and executes them, but returns only noisy results. Differential privacy gives mathematical guarantees how much those results reveal about the content of the database. It has already been applied to the distributed setting, but with a slightly different goal: One does not try to hide the individual training samples, but whether a given client has participated in the training of the machine learning model or not [17, 11].
Secure aggregation [5] is a privacy scheme that was designed with a setting similar to ours already in mind. It allows for computing the sum of different vectors. Each client contributes one of those vectors, which could, e.g., be gradients with respect to the clients' datasets in the case of a training with gradient descent. In the end, only the sum is known by the clients and the server, but not the individual summands. The whole process works without the addition of noise, so no accuracy is lost. Secure aggregation has some fundamental limitations, though: (1) It only works with integer-valued vector entries. (2) It needs a peer-to-peer connection between the clients or trusted server that handles the key agreement. (3) The sum of gradients could still reveal information about samples on which it is based.

4 Data

In our project we are given a dataset of bank customers and are modelling the probability of their default. The target variable is a boolean variable whether or not the customer defaulted within one year. The dataset consists of 767,431 records with 42 features each.
The features are divided into three groups, so called modules. The behavioral module consists of 28 features describing the previous behavior of the customer at the bank. Some examples are the maximum number of days past due, the current average balance and the number of negative balance moments.
The financial module with seven features contains some key financial figures about the customer such as the revenue growth rate and the ratio of EBITDA

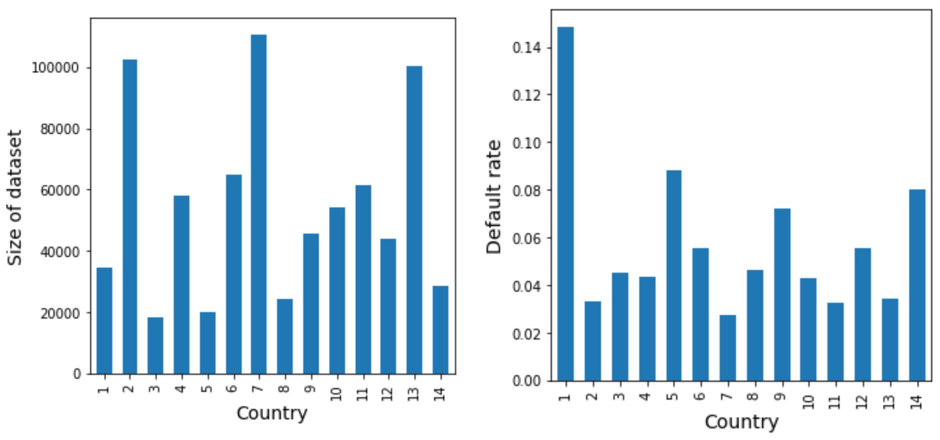


Figure 1: Dataset size (left) and default rate (right) for each country.

to equity. Furthermore there is a module containing eight non-financial features such as the age of the company.

Each feature of the behavioral module is missing in between 46% and 96% of all cases. The other features are present in more than 80% of all observations. Behavioral data is obviously missing for all new bank customers while it is present for existing customers. Using a single model for predicting the default of these very different groups of customers leads to poor model performance. For this reason it is common practice to have separate risk models for new and existing customers.

We therefore split our data into two parts based on whether or not a record has behavioral data. This gives us two separate datasets with 415,819 and 351,612 and records, respectively.

The data contains observations from 14 different countries. The size of the dataset and the default rate strongly vary for different countries as can be seen in Figure 1. Furthermore, Figure 2 shows that some risk factors such as the number of negative balance moments have very different distributions across countries. Others, like the revenue growth rate, however, are for the most part equally distributed.

When looking at the correlation matrix of the dataset in Figure 3, we can clearly see that many risk factors within the behavioral module are correlated. The same is true for some features within the non-financial module. However, the cross module correlation is very low.

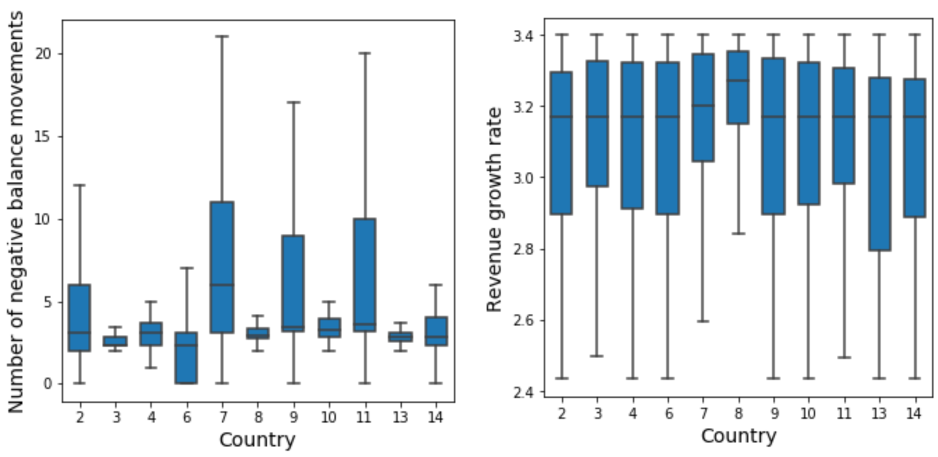


Figure 2: Distribution of two selected risk factors by country.

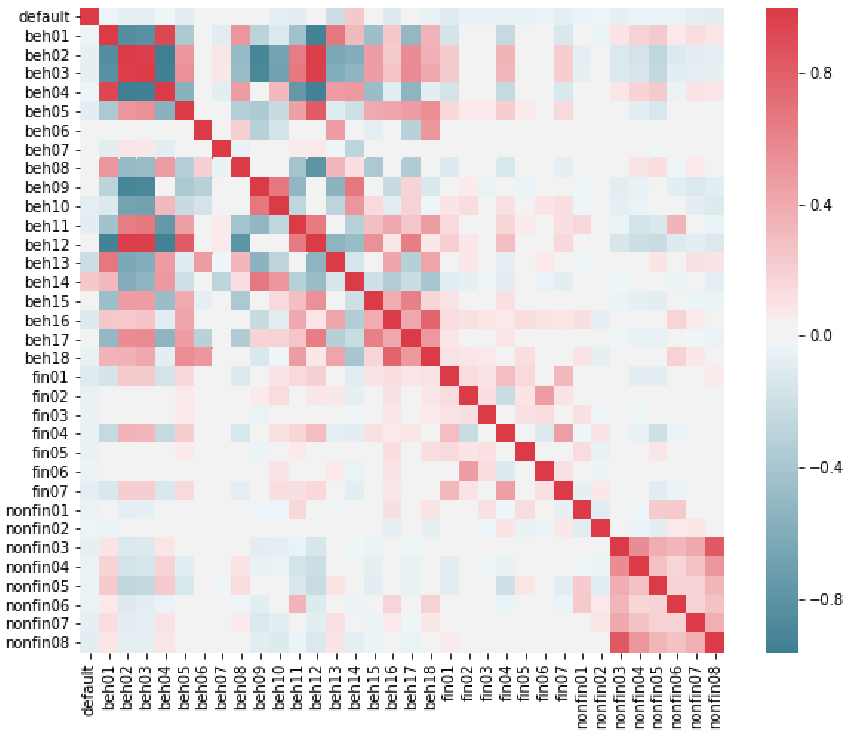


Figure 3: Correlation matrix of the risk factors.

4.1 Outliers and Data Normalization

During data preprocessing we found some outliers in the dataset. These would drastically reduce the use of their respective features. We chose to clip all features whose maximum values is more than ten times their 99th percentile at the latter. This is the case for six behavioral features in the dataset. When the data is distributed across clients, each needs to do their own outlier handling. This would most likely lead to a very similar result, since outliers in the complete dataset will probably also be outliers in their local dataset.
We also normalize the data such that all features have zero mean and unit variance. This is common practice in machine learning and often improves model performance, especially for neural networks. Furthermore it improves the convergence speed of optimization algorithms like GD and Adam. In a federated setting each client needs to scale their own dataset and this needs to be done with the same parameters for centralizing and scaling. Therefore the mean and variance of the data across all clients need to be calculated in a privacy preserving fashion. How this can be done is described in 5.3.2.

4.2 Splitting the Data

When applied in the real world, each client will only have access to their own dataset and split it into training, validation and test data. We also split our data in this way. For training the joint baseline model we combine these local datasets into a global training, validation and test sets. This ensures comparability of the models, since the same data was used for training, validating and testing.
For each client we split off 10% of the samples as a test set and again 10% of the remaining samples as a validation set. We do this in a stratified fashion in order to guarantee an equal proportion of defaults in each of the sets, which is important since the proportion of defaults in the dataset is very small.
Our dataset contains records from 14 different countries, which we used as a natural split of the dataset into client datasets. In order to test smaller sizes of client datasets, we randomly split each of those into multiple datasets. In order to still have a reasonable number of samples, especially of defaults, we do not split the validation and test data any further than country level.

As seen previously, the country datasets differ significantly in the distributions of some risk factor. In order to test federated learning for identically distributed datasets, we also split our data *randomly* into 14 local datasets.

5 Applied Methods

5.1 Models

5.1.1 Logistic Regression

Logistic regression is a model that is used when the label y one tries to predict is binary, that is, it can only take the values 0 and 1. The output of the model is an estimate of $\Pr(y = 1|x)$, i.e., the probability that feature vector x belongs to class 1. Note that in reality you never observe $\Pr(y = 1|x)$, but only the actual outcome of the random variable that specifies the label.
The formula for logistic regression is derived using Bayes formula

$$\Pr(y = 1|x) \propto \Pr(x|y = 1) \Pr(y = 1).$$

When assuming a Bernoulli distribution of y and Gaussian distribution of $x|y$, one arrives at

$$\Pr(y = 1|x) = \sigma(w^T x + b) := \frac{1}{1 + \exp(-(w^T x + b))}$$

for appropriate parameters w and b and the sigmoid function σ. These parameters are typically estimated using maximum likelihood estimation, i.e., ones tries to best fit the training data $\{(x_i, y_i)\}_{i=1}^n$ by maximizing

$$\prod_{i=1}^{n} \Pr(y_i = 1|x_i)^{y_i} + (1 - \Pr(y_i = 1|x_i))^{1-y_i}, \tag{3}$$

which is equivalent to minimizing

$$\sum_{i=1}^{n} y_i \log \Pr(y_i = 1|x_i) + (1 - y_i) \log(1 - \Pr(y_i = 1|x_i)),$$

which in turn we identify as a problem of the type (1).

5.1.2 Neural Networks

Neural networks (NNs) can be seen as generalization of logistic regression. The function that a logistic regression classifier computes is a one-dimensional affinely linear mapping, followed by a non-linear function. That is the building block of NNs. An NN consists of several concatenated layers, where each layer comprises a (multi-dimensional) affinely linear function and a non-linear function that is applied to each dimension of the previous function's output, a so called activation function. More precisely, an NN computes the function

$$f(x, \mathbf{W}, b) = \alpha_k(\mathbf{W}_k \dots (\alpha_1(\mathbf{W}_1 \alpha_0(\mathbf{W}_0 x + b_0) + b_1) \dots) + b_k),$$

where the $\mathbf{W}_i$ and b_i are weight matrices and bias vectors, respectively, and the α_i are a non-linear activation functions, nowadays typically the ReLU (rectified linear unit) $\max(0, x)$. However, for classification problem, the very last non-linearity is usually the softmax function

$$\sigma(z)_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)},$$

which is the generalization of the sigmoid function defined above. The k-th entry can be interpreted as the probability that the given input belongs to class k. One then trains the NN by minimizing the cross-entropy loss

$$\prod_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{y_i = k\} \sigma(x_i)_k$$

using a variant of gradient descent. As opposed to (3), this function is non-convex, thus one can only hope to reach a local but not a global optimum.

5.2 Training Methods

5.2.1 Gradient Descent

Gradient descent is a first order iterative optimization algorithm for finding the minimum of a function. It is based on the observation that if a multi-variable function $F(x)$ is defined and differentiable in a neighborhood of a point x, then the negative gradient of F in x, $-\nabla F(x)$, points in the direction of the steepest descent. It can be shown that, if

$$\theta_{t+1} = \theta_t - \gamma \nabla F(\theta_t),$$

then $F(\theta_t) \geq F(\theta_{t+1})$, for $\gamma > 0$ small enough.
The value of γ, known as the learning rate, can change at every iteration. With certain assumptions on the function F and for certain choices of γ, convergence to a local minimum can be guaranteed. If F is convex, all local minima are global minima and hence gradient descent converges to a global solution.
Gradient descent can easily be applied in a setting where the data is distributed among multiple clients as described in (2). The gradient of the cost function can be written as

$$\nabla J(w) = \sum_{k=1}^{K} \frac{n_k}{n} g_k \quad \text{where} \quad g_k := \frac{1}{n_k} \sum_{j=1}^{n_k} \nabla L(x_{i_j}, y_{i_j}, w). \tag{4}$$

Hence the gradient can be computed by averaging the local gradients computed on the clients' datasets.

5.2.2 Adaptive Moment Estimation (Adam)

This method, introduced in [13], aims at speeding up gradient descent by using a weighted average of gradients from the previous iterations instead of only the latest one. It also keeps a weighted average of the second moment of the gradients and uses this to scale the learning rate.
In each iteration, the following steps are performed. All operations on vectors in the following equations are to be understood component-wise.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla F(\theta_t) \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \left[\nabla F(\theta_t)\right]^2 \end{aligned}$$

Often in order to correct bias towards zero (due to 0 initialization), bias corrected versions are used:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

Then the current solution is updated as follows:

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t.$$

As defaults for the introduced parameters the authors of [13] suggest $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.
The necessary gradient can be computed in a distributed setting in the same way as for gradient descent.

5.2.3 Federated Averaging

Federated Averaging, introduced in [16], aims at reducing the necessary number of rounds of communication when training a federated model. It is based on the observation that the update step

$$w_{t+1} = w_t - \gamma \sum_{k=1}^{K} \frac{n_k}{n} g_k$$

of gradient descent in a federated setting, as described in (4), can be equivalently formulated as

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k \quad \text{where} \quad w_{t+1}^k := w_t - \gamma g_k.$$

Formulated this way, each client makes a local update step before the updated models are averaged, as opposed to the gradients being averaged in (4). Now it is possible for each client to do multiple local updates before the resulting models are averaged. This can lead to a reduced number of rounds of communication between the clients and the server, at the expense of more local computational cost. This is often a good trade-off in a distributed setting, since latency and bandwidth as well as the privacy aspect limit the number of rounds of communications, while local computation power is not a bottleneck. The procedure of Federated Averaging is given in Algorithm 1.

Algorithm 1 FederatedAveraging. The K clients are indexed by k, E is the number of local epochs, and γ is the learning rate.

```
Server executes:
  Initialize $w_0$
  for each round $t$ = 1,2,.. do
    for each client $k \in \{1, \ldots, K\}$ in parallel do
      $w^k_{t+1} \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \Sigma_{k=1}^{K} \frac{n_k}{n} w^k_{t+1}$
ClientUpdate($k, w$): //run on client $k$
  for local epoch $i$ in 1 to $E$ do
    $w \leftarrow w - \gamma \frac{1}{n_k} \sum_{j=1}^{n_k} \nabla L(x_{i_j}, y_{i_j}, w)$
  return $w$ to server
```

5.3 Privacy

5.3.1 Differential Privacy

Not sending the entire dataset to the server but only model updates already means a huge gain in privacy. We would, however, like to quantify this to give the clients privacy guarantees. Unfortunately, it is very hard to determine what information a complex function of a dataset such as the gradient reveals. We thus resort to worst-case guarantees that we obtain by adding noise and that we formulate in terms of differential privacy. In this framework, a client computes a query on their dataset, returns the result with added noise and can deduce from the amount of noise added how much information it reveals in the worst-case by answering the query. This amount of information is described by the ability of the party receiving the query result to determine whether the query was executed on a dataset in which a certain record was present or not. Formally:

Definition 1. A randomized algorithm $\mathcal{M} : \mathcal{D} \to \mathbb{R}^k$ with domain $\mathcal{D}$ (the set of all possible databases) is (ε, δ)-differentially private, for $\varepsilon, \delta > 0$, if for all $\mathcal{S} \subset \text{Range}(\mathcal{M})$ and for all databases D_1, D_2 that differ in at most one entry:

$$\Pr(\mathcal{M}(D_1) \in \mathcal{S}) \leq e^{\varepsilon} \Pr(\mathcal{M}(D_2) \in \mathcal{S}) + \delta.$$

Clearly, one aims for a small ε and δ, since then the confidence with which the party receiving the query result can say which entries are contained in

the database is very low.
But how does one design an (ε, δ)-differentially private algorithm? One way is to perturb the exact query result with Gaussian noise. For determining the appropriate level of noise, possible query results must be bounded by a known constant. The intuition behind this is as follows: Suppose a function f has values in $[0, 1]$ and we add $\mathcal{N}(0, 1)$ to its output. Scaling f to $[0, 100]$, adding the same amount of noise and scaling down to $[0, 1]$ again, would effectively reduce the amount of noise to $1/100\mathcal{N}(0, 1) = \mathcal{N}(0, 1/10, 000)$ without changing anything about f. Therefore we define the sensitivity of a function f as follows:

Definition 2. The l_2-sensitivity of a function $f : \mathcal{D} \to \mathbb{R}^k$ with values in $\mathbb{R}^k$ is defined as

$$\nabla_2(f) = \max_{D_1, D_2 \text{ differ in exactly one record}} \|f(D_1) - f(D_2)\|_2 .$$

We are now ready to state the main theorem for adding Gaussian noise [8, Thm. 3.22]:

Theorem 1. *Let $\varepsilon \in (0, 1)$. For $c > \sqrt{2 \ln(1.25/\delta)}$, the Gaussian mechanism with parameter $\sigma \geq c\nabla_2(f)/\varepsilon$, that is, adding $\mathcal{N}(0, \sigma^2 I)$ to the output of the function f, is (ε, δ)-differentially private.*

Since it not only suffices to compute one model update on each client's dataset, but we need to do this many times, we need a way to combine several function evaluations. The advanced composition theorem [8, Thm. 3.20] is a tool for this. We state a slightly simplified version here.

Theorem 2. *For all $\varepsilon, \delta, \delta' > 0$, k executions of (ε, δ)-differentially private algorithms satisfy $(\tilde{\varepsilon}, \tilde{\delta}) := (\varepsilon', k\delta + \delta')$-differential privacy for*

$$\varepsilon' = \sqrt{2k \ln(1/\delta')}\varepsilon + k\varepsilon(e^{\varepsilon} - 1).$$

δ' allows to make the final $\tilde{\varepsilon}$ smaller in exchange for a larger $\tilde{\delta}$ and the other way around.
There are two ways in which this theorem can be applied for training a machine learning model. In the first one, a certain privacy requirement has to be fulfilled. Then the target privacy $(\tilde{\epsilon}, \tilde{\delta})$ is fixed and one is limited in the number of iterations k. Or one is mainly interested in achieving a good model

performance and trains until the model has achieved the desired performance. Theorem 2 then gives a post-hoc guarantee for how much information has been revealed during the training process.

5.3.2 Data Scaling

Many optimization algorithms, in particular gradient descent, converge significantly faster if the attributes are scaled. If, e.g., the values of a feature i are several orders of magnitude larger than those of a feature j, then the gradient will be a lot bigger in direction i than in direction j. This leads to larger steps in direction i than in direction j, since the learning rate is the same for all parameter vector entries. Hence the learning rate will either be too big for i or too small for j or both.
The standard solution for this problem is to scale the features values. Typically, one of the following two methods is applied:

1. Scale the features to a fixed interval, e.g., $[0, 1]$.
2. Subtract the mean of each feature and divide by its empirical standard deviation, so that it has mean 0 and empirical standard deviation 1.

Method 1 needs the maximum and minimum value for each feature, method 2 needs the mean and the empirical standard deviation. Computing those quantities requires a pass over the entire dataset — and in our implementation, where we use method 2, we compute them in this way for simplicity. But in the real setting, an initial communication round is needed during which the scaling parameters are determined. For method 1 each client would share the minimum and maximum of each feature with respect to their local dataset; the server could then compute the global minimum and maximum. For method 2 each client would share $\sum_{j=1}^{n_k} x_{i_j m}$ and $\sum_{j=1}^{n_k} x_{i_j m}^2$ for each feature m with the server, in addition to their number of samples n_k. The global mean of feature m would then be given by

$$\mu_m = \frac{1}{n} \sum_{j=1}^{n_k} x_{i_j m}$$

and the empirical standard deviation by

$$\sigma_m = \sqrt{\frac{1}{n} \sum_{j=1}^{n_k} x_{i_j m}^2 - \mu_m^2}.$$

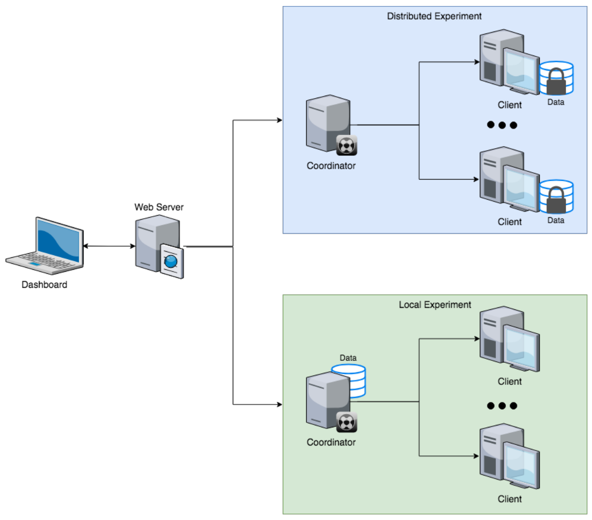


Figure 4: Overview of the Implemented Framework

In the case that the clients are not even willing to share those few pieces of information about their data, a secure multiparty computation protocol such as [4] or the already mentioned [5] can be used to compute μ_m and σ_m. Secure multiparty computation protocols allow for computing functions on the shared data of multiple parties without any party learning anything about the data of a different party and without the need for a trusted aggregator. For most practical applications such protocols are too slow, too limited and/or too complicated (that is why we don't use it in our setting for training the logistic regression model), but for such a simple function as summing K scalars, it is practical.

6 Implementation

6.1 Architecture

The whole system comprises three major components (see Figure 5 for a schematic overview):

1. *Dashboard*

   The dashboard is a browser interface that allows to conduct experiments (see Figure 5). Each experiment can be either local or distributed. The experiment description contains all information sufficient to perform it, such as the machine learning model (logistic regression or NN), learning parameters (optimizer, regularization strength, etc.), data source (in case of a local experiment), clients (in case of a distributed experiment).

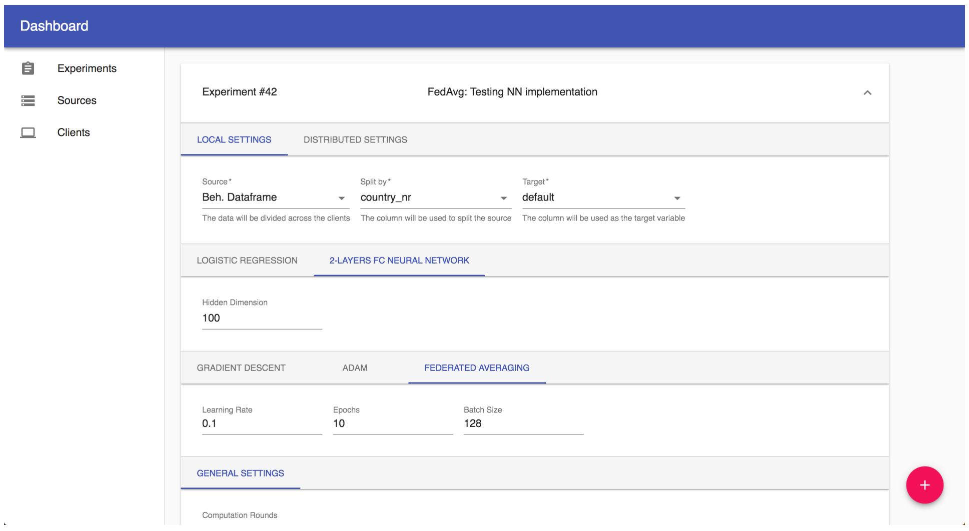
2. *Coordinator*

   The coordinator is responsible for performing and coordinating experiments. It holds the current global model, creates the local clients or connects to distributed ones, orchestrates the computation rounds and distributes new models to the clients. During the training it saves experiment data to a MongoDB database server and sends the status updates to the dashboard.

3. *Clients*

   Each client computes updates to the current model using their local portion of data and sends the update back to the server. If an experiment is set to be local, then the clients reside in the RAM of the web server and the coordinator populates them with appropriate portions of data, otherwise they connect over the network and neither the web server nor the coordinator has access to their datasets. Since they are completely decoupled from everything else, we can switch between different methods to compute the actual updates in any way, be it Scikit-learn, PyTorch or our own implementations using NumPy.

The entire system, except for the dashboard, is written in Python 3.6. The dashboard is written in TypeScript using React.js library to develop the user interface.
All of the system's components communicate using Google's Protocol Buffers. This allows us to describe the communication messages format only once and then use it across all system's components.
Moreover, we use the gRPC framework for the network communication, which gives us higher performance due to binary serialization and the use of HTTP/2 protocol. The client/server architecture of the coordinator and the clients components allows us to switch computation frameworks on clients easily.
Support for the local experiments was added with the intention of debugging the algorithms faster and easier.
The typical workflow in our framework looks as follows:


Dashboard

Experiments
Sources
Clients

Experiment #42 FedAvg: Testing NN implementation

LOCAL SETTINGS | DISTRIBUTED SETTINGS

Source* Beh. Dataframe — The data will be divided across the clients
Split by* country_nr — The column will be used to split the source
Target* default — The column will be used as the target variable

LOGISTIC REGRESSION | 2-LAYERS FC NEURAL NETWORK

Hidden Dimension 100

GRADIENT DESCENT | ADAM | FEDERATED AVERAGING

Learning Rate 0.1
Epochs 10
Batch Size 128

GENERAL SETTINGS

Computation Rounds

Figure 5: Part of the dashboard's UI.

1. The user specifies the experiment parameters. For a local experiment, a data source available on the web server has to be chosen, otherwise available distributed clients, connected to the server, has to be selected.

2. The web server receives the request to perform a new experiment from the dashboard and creates an appropriate coordinator.

3. In the case that the distributed setting is selected, the coordinator connects to the chosen clients and asks them to compute the updates on their data using the current global model. In the local setting, the coordinator receives the data source from the web server and creates the clients in memory with data portions distributed across them.

4. The coordinator runs the experiment for a specified number of computation rounds. At the end of each round it sends back the status to the dashboard and saves it to the MongoDB database.

6.2 Learning Algorithms

For implementing and testing the learning algorithms described in Section 5.2, we simulated the distributed setting offline without actual communication over a network. This allowed for faster and easier debugging and testing. First we implemented logistic regression, then a neural network.

Libraries that support training logistic regression models typically assume access to the entire dataset at once and are thus not suited for the distributed setting. Therefore we needed our own implementation of a function that minimizes the logistic regression cost function. Our idea was to take an established library function and come as close to its results as possible with our own implementation. As our reference we chose scikit-learn's "LogisticRegression" [24]. They use a Newton method for optimization [10], while we relied on the first order methods of Section 5.2, which have the advantage of being also applicable to neural networks.
Our expectation was that our implementation should converge to the same weights and bias as computed by scikit-learn if using same regularization type and strength. This, however, was not the case at first. There were two main reasons for this. The first one was that scikit-learn claims to not apply regularization to the bias term of the parameter vector [1]. But the documentation is wrong about this point, see [15]. The other reason was the very slow convergence was of the gradient descent, which didn't allow us to see whether our weights and bias converge to those of scikit-learn. That's why we implemented momentum, a simpler version of Adam, and then Adam itself. That already gave improvements, but the major gain in speed was achieved when scaling the features to zero mean and unit variance in a preprocessing step, see Section 5.3.2.
But so far, the classification results themselves weren't very good: recall and precision was very high for the non-default class and very low for the default class. The cause that we identified was that there are a lot more non-default than default observations in our dataset and thus the influence of the default observations on the loss function was comparatively small. We overcame this problem by weighting the samples' contribution to the cost function based on their frequency in the dataset. Let c_0 and c_1 be the number of samples of the non-default and the default class, respectively. Then the samples of the non-default class are weighted with $\frac{c_0+c_1}{2}\frac{1}{c_0}$, those of the default class with $\frac{c_0+c_1}{2}\frac{1}{c_1}$. The normalization $\frac{c_0+c_1}{2}$ ensures that the regularization strength remains comparable if the ratio c_0/c_1 changes.
As with the data scaling, counting the frequency in the distributed setting again requires the clients to share the frequencies in their datasets, or use one of the techniques described in Section 5.3.2.
After having a working implementation of logistic regression, expanding it

to a neural network was relatively easy. We used a fully connected network with one hidden layer consisting of 100 neurons. The gradient computation is performed using PyTorch.

6.3 Differential Privacy

Applying Theorem 1 from Section 5.3.1 to our setting is not straightforward, since we typically do not have a bound on the model updates, e.g., the gradient, computed on the clients' datasets. We therefore enforce a bound C ourselves by projecting the model updates to the L_2-ball of radius C, i.e.,

$$v \mapsto \min\left(1, \frac{C}{\|v\|_2}\right) v.$$

The main effect of this projection is that it can slow down the training, making it require more iterations, which in turn requires the addition of more noise due to the composition theorem 2. Theorem 1 shows that one has a linear trade-off between C and ε, and an exponential one between C and δ. In our implementation we give the user the freedom to experiment with this trade-off by specifying C and the amount of noise added. The latter one not in terms of ε and δ, but in terms of σ. The reason is that, given σ, C and the number of iterations, there are infinitely many (ε, δ)-pairs corresponding to these parameters. And one does not force the user to use Theorems 1 and 2, for example in the case that stronger theorems are proven later.

7 Experiments

7.1 Potential of Federated Learning

The promise of federated learning is that combining multiple clients' datasets and learning a joint model on them will lead to a better model performance than learning multiple local models on the local datasets. This, of course, depends on the type of model used and the distribution of data between the clients. We try two different types of models: logistic regression and a simple fully connected neural network with one hidden layer and a varying number of neurons. We also try different sizes of local datasets and different ways of splitting the data, as described in Section 4.2.
We want to argue that clients participating in federated learning profit from

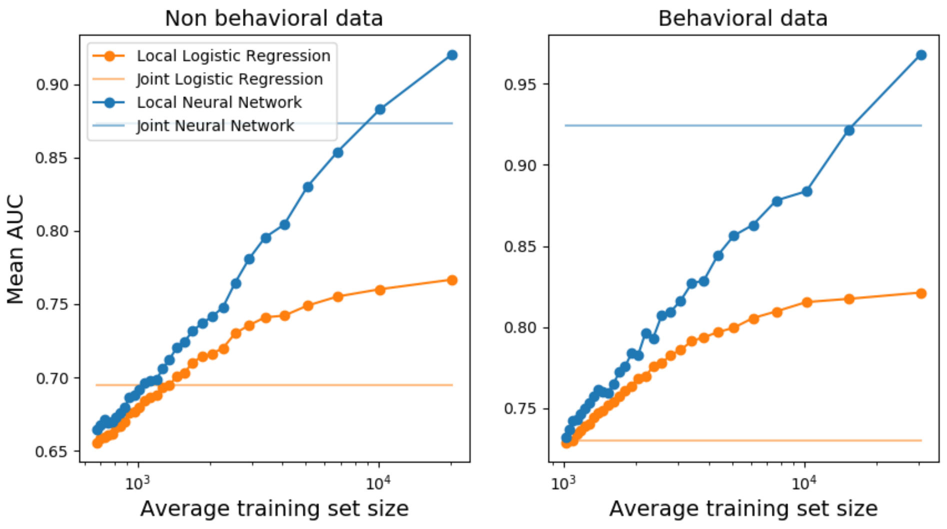


Figure 6: Model performance for varying local training set size where the dataset was initially split by country.

it, since the produced joint model performs better on their local data than a locally trained model would do. We therefore compare the models in the following way. We build a local model with each client's local training data using the validation data for hyperparameter tuning. In case of the neural network, we stop training as soon as the loss on the validation set increases more than one percent from the previous minimum. Then the performance of each local model is evaluated on the respective local test set using the ROC AUC as the performance metric. We in addition build a joint model using the combined training data of all clients. This model, too, is evaluated on each local test set. This gives us a pair of performance measures for each client, one from the local and one from the joint model.
The effect of different dataset sizes on the potential of federated learning can be seen in Figure 6. The NN used here has one hidden layer with 100 units. We see that the joint model performs better than the local ones for small local training set sizes. However, when the local datasets are larger, the local models actually perform better than the joint one, which means that using federated learning in these situations would not be beneficial. This effect is a lot more pronounced for logistic regression than for NNs.
The advantage of the local models is that they have greater expressive power than the single joint model, since there are more of them. Given they have a

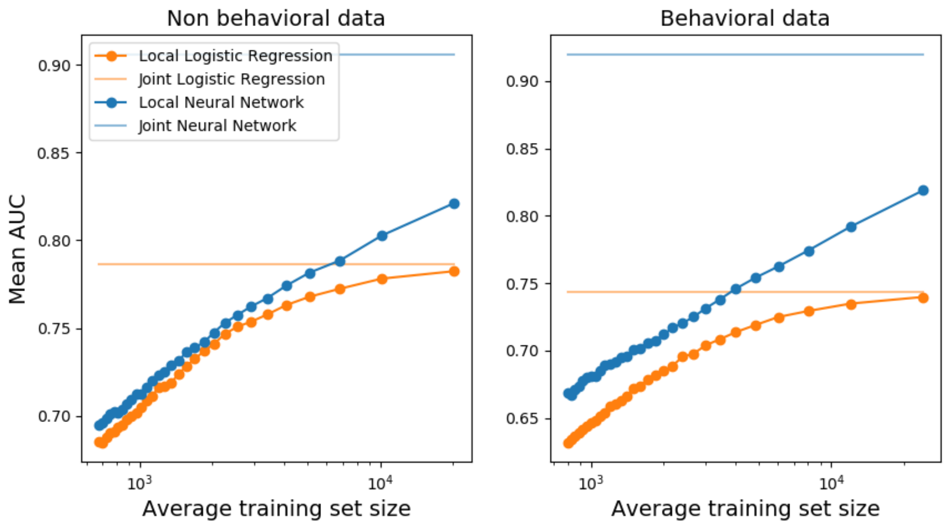


Figure 7: Same setup as in Figure 6, but using a random split into local datasets.

sufficient amount of local data to train on, the local models can adapt to the specifics of each local dataset, which have significant differences as we saw in Section 4. This makes them perform better than the joint model.

For logistic regression, the simpler of the two tested models, the amount of local training data needed to sufficiently train it, is, of course, smaller than for a neural network. The latter has a lot more expressive power and can benefit enormously from the larger amount of data in the joint training set. This explains the greater benefit of federated learning for this model.

In Figure 7 we see the situation where the dataset was split randomly into local datasets. The local datasets are not differently distributed as in the previous case and the local models cannot specialize to them. Here the joint model performs better in all cases.

The second point we investigate is how the complexity of the model affects the performance difference between local and joint model. We used the split of the data by country for this. As we see in Figure 8, the joint model has no advantage over the local ones for a very simple neural network with a small number of hidden units. With increasing complexity, meaning growing number of model parameters, the joint model strongly increases its performance while the local models only improve slightly. This makes sense since the very simple networks do not have enough expressive power to profit

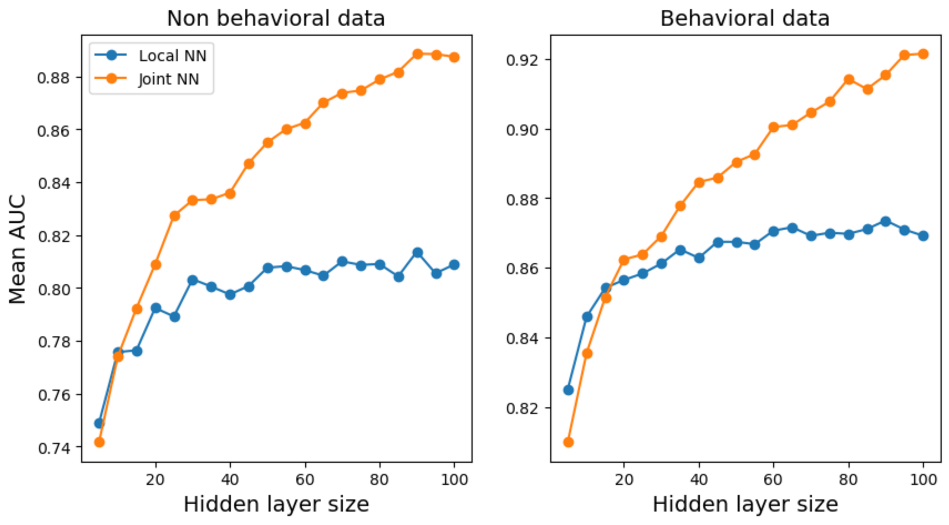


Figure 8: Performance of joint and local models for NNs with varying number of hidden units.

from a larger amount of training data and overcome the advantage the local models have from specializing on the local datasets. However, more complex networks can make use of the larger training set and give the joint model a huge advantage over the local ones.
To summarize, we can say that a better model performance can be achieved by using federated learning in many situations. It has more potential when the local training sets are smaller, the used model is more complex and the local datasets do not have significantly different distributions.

7.2 Learning Algorithms

We implemented vanilla gradient descent and Adam and combined both with Federated Averaging.
As best-working parameters for logistic regression we found a decaying learning rate $0.5/\sqrt{t}$, where t is the number of the current iteration, and a regularization strength of 1. For the NN, a constant learning rate of 0.001 and a regularization strength of 0.00001 performed best. We didn't do extensive experiments on finding the optimal number of local iterations E for Federated Averaging, but fixed $E = 50$. Figure 9 shows that both Adam, and Federated Averaging combined with Adam significantly outperform the variants of gradient descent. However, there is no major advantage of using

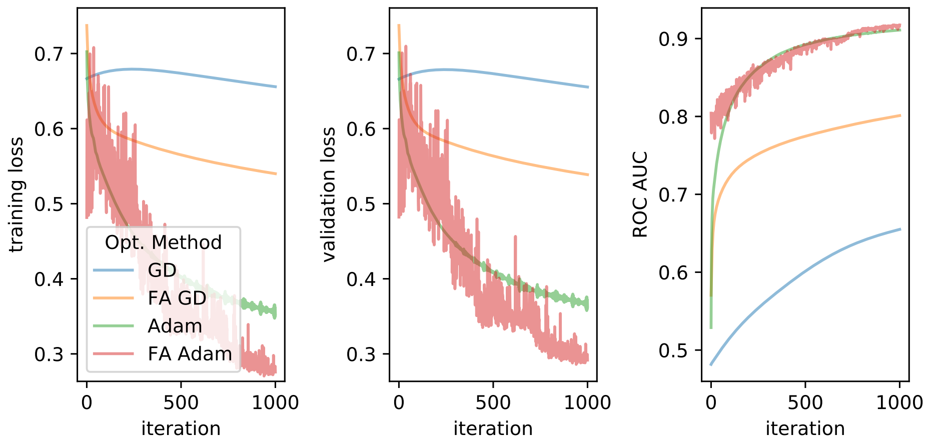


Figure 9: Comparison of gradient descent (GD), Adam and Federated Averaging (FA) combined with the two when training the NN on the behavioral dataset. For FA we used 50 local iterations per global iteration, i.e., $E = 50$ in Algorithm 1.

Federated Averaging with Adam over standard Adam, hence we stuck with the latter for the differential privacy experiments in Section 7.3 because it allowed us to conduct more experiments since it is computational less expensive.

7.3 Impact of Differential Privacy

To gauge the effect that making the training differentially private has on the model performance, we tried multiple values for the standard deviation σ of the Gaussian noise added and different values for the sensitivity C. $\sigma = 0.5$ gave the best trade-off between ROC AUC and privacy, so we fixed this value and varied C. Figures 10 and 11 show how the noise addition affects the training with Adam and the final model performance. We see that the performance of the NN suffers a lot more than that of the logistic regression model. We explain this by the fact that the NN has a lot more parameters and represents a non-convex function, which makes the training more difficult, and sensitive to perturbations. In the case of logistic regression, sensitivity values of 0.2 and 0.4 still give relatively good results, for smaller values the resulting model is not useful anymore.
The (ε, δ) guarantee of the training procedure also depends on the number of training iterations. More iterations mean that more information gets

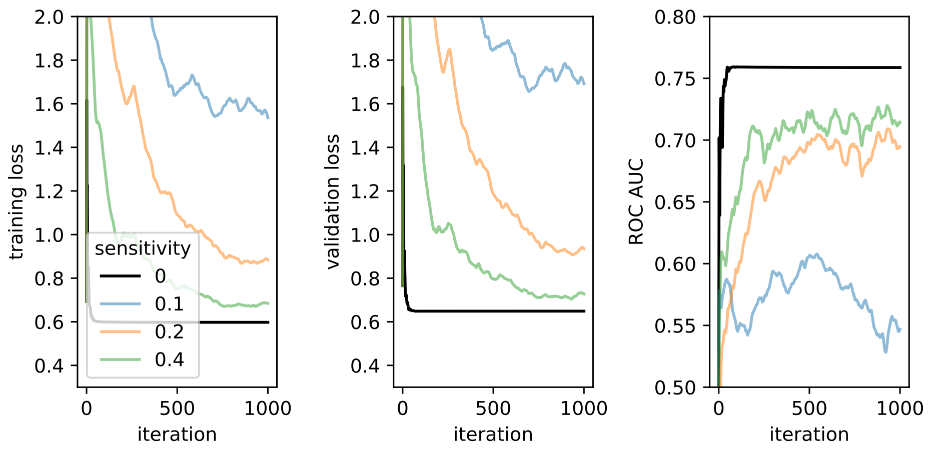


Figure 10: Training of the logistic regression model with Adam on the behavioral dataset without noise (black) and with Gaussian noise with standard deviation 0.5 and varying sensitivity C.

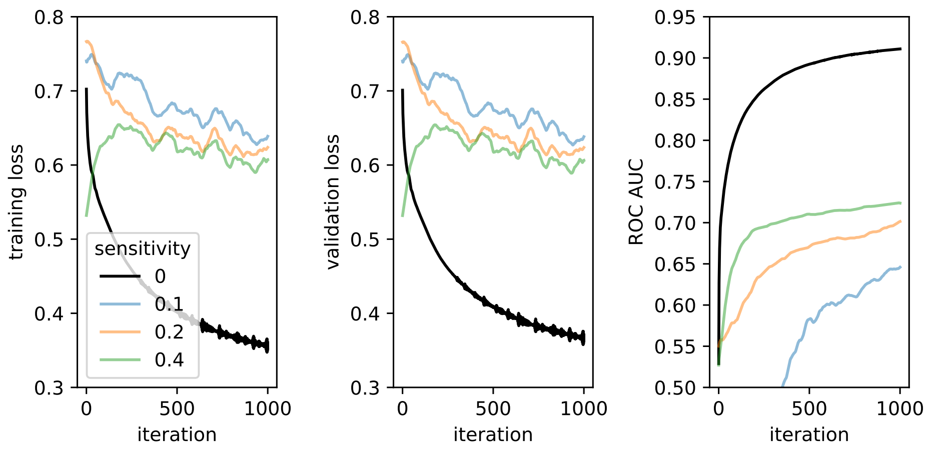
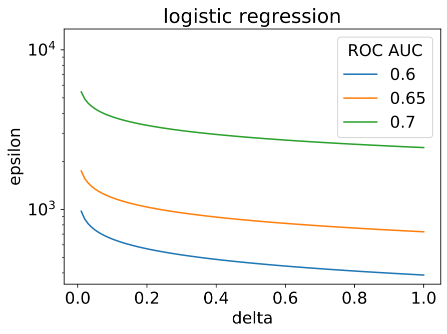
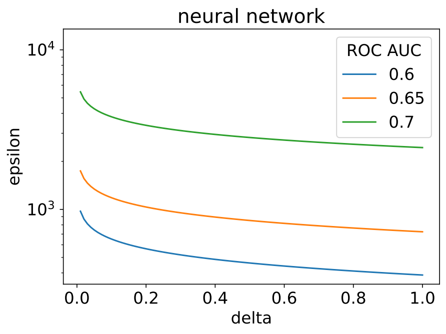


Figure 11: Training of the neural network with Adam on the behavioral dataset without noise (black) and with Gaussian noise with standard deviation 0.5 and varying sensitivity C.



(a) Corresponding iteration numbers: 112 (green), 187 (orange), 512 (blue).



(b) Corresponding iteration numbers: 132 (green), 314 (orange), 989 (blue).

Figure 12: (ε, δ)-pairs that correspond to the noise level $\sigma = 0.5$, sensitivity $C = 0.2$ and minimal iteration number such that the desired ROC AUC value is reached on the behavioral dataset.

revealed, thus ε and δ become larger. But even if the number of iterations is fixed, there are still infinitely many (ε, δ) pairs that correspond to the level of added noise: One value can be made larger in favor of the other one, which gets smaller. In Figure 12 we have plotted the possible values for (ε, δ) for the same experiments as in Figures 10 and 11. We chose the minimal number of iterations such that given ROC AUC value is reached. One sees that for very small values of δ one has to pay a huge price in terms of ε, while after that a larger δ gives almost no benefits in terms of ε. Thus a δ value of around 0.1 seems like a reasonable choice.

We would like to note that the (ε, δ) values provided by the composition theorem 2 are not tight, since the theorem does not take into account the shape of the added noise (in our case Gaussian). Therefore the actually achieved privacy might even be a lot better. Possible improvements may be achieved using more advanced methods from [8] or a recent method called moments accountant introduced in [2] which specifically focuses on Gaussian noise. However, for this project our main focus was to determine applicable values for the variance of the noise and the sensitivity that still allow to train a reasonable model.

8 Conclusion

In this project we investigated the potential of using distributed datasets to build a joint model in a privacy-preserving fashion. We also developed a framework that facilitates this kind of collaboration between different data-owning parties. Training in our setting is done by exchanging model updates instead of whole datasets.
Our experiments were conducted using data of bank customers with the goal of predicting whether they will default within one year. We showed that there can be a huge gain in model performance from collaborating and using multiple datasets instead of just training a model on a single local dataset. This is the case for sufficiently small individual datasets or sufficiently complex models such as neural networks.
In return for model performance, we are also able to give certain — however, not very strong — upper bounds on how much information from the local datasets is leaked during the training process.
Our fully-functional implementation shows that such a distributed, privacy-preserving training environment can be realized in the short time frame of a single semester.

References

[1] *1.1. Generalized Linear Models — scikit-learn 0.19.1 documentation.* `http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression`. Accessed: 2018-07-15.

[2] Martin Abadi et al. *Deep Learning with Differential Privacy.* 2016. URL: `https://arxiv.org/pdf/1607.00133.pdf`.

[3] Rakesh Agrawal and Ramakrishnan Srikant. "Privacy-preserving data mining". In: *ACM Sigmod Record.* Vol. 29. 2. ACM. 2000, pp. 439–450.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computation". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing.* ACM. 1988, pp. 1–10.

[5] Keith Bonawitz et al. "Practical Secure Aggregation for Federated Learning on User-Held Data". In: *30th Conference on Neural Information Processing Systems (NIPS)* (2016). URL: `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45808.pdf`.

[6] Keke Chen and Ling Liu. *A random rotation perturbation approach to privacy preserving data classification.* 2005.

[7] John V. Duca. *Subprime Mortgage Crisis.* Accessed: 2018-07-15. Nov. 2013. URL: `https://www.federalreservehistory.org/essays/subprime_mortgage_crisis`.

[8] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407. ISSN: 1551-305X. DOI: `10.1561/0400000042`. URL: `http://dx.doi.org/10.1561/0400000042`.

[9] Cynthia Dwork et al. "Calibrating noise to sensitivity in private data analysis". In: *Theory of cryptography conference.* Springer. 2006, pp. 265–284.

[10] Rong-En Fan et al. "LIBLINEAR: A library for large linear classification". In: *Journal of machine learning research* 9.Aug (2008), pp. 1871–1874.

[11] Robin C. Geyer, Tassilo Klein, and Moin Nabi. *Differential Private Federate Learning: A Client Level Perspective.* 2018. URL: https://arxiv.org/pdf/1712.07557.pdf.

[12] Martin Jaggi et al. "Communication-efficient distributed dual coordinate ascent". In: *Advances in Neural Information Processing Systems.* 2014, pp. 3068–3076.

[13] D.P. Kingma and J.L. Ba. "Adam: a Method for Stochastic Optimization". In: *International Conference on Learning Representations* (2015), pp. 1–13.

[14] Jakub Konecny et al. *Federated Optimization: Distributed Machine Learning for On-Device Intelligence.* 2016. URL: https://arxiv.org/pdf/1610.02527.pdf.

[15] *Logistic regression regularization penalization is applied to intercept — Issue 10626 — scikit-learn/scikit-learn.* https://github.com/scikit-learn/scikit-learn/issues/10626. Accessed: 2018-07-15.

[16] H. Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data". In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)* 54 (2017). URL: https://arxiv.org/pdf/1602.05629.pdf.

[17] H. Brendan McMahan et al. "Learning Differential Private Recurrent Language Models". In: *ICLR* (2018). URL: https://arxiv.org/pdf/1710.06963.pdf.

[18] Yu Nesterov. "Efficiency of coordinate descent methods on huge-scale optimization problems". In: *SIAM Journal on Optimization* 22.2 (2012), pp. 341–362.

[19] *NINJA Loan.* Accessed: 2018-07-15. URL: https://www.investopedia.com/terms/n/ninja-loan.asp.

[20] N. Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks : The Official Journal of the International Neural Network Society* 12.1 (1999), pp. 145–151. URL: http://doi.org/10.1016/S0893-6080(98)00116-6.

[21] Sashank J. Reddi et al. *AIDE: Fast and communication efficient distributed optimization.* 2016. URL: https://arxiv.org/pdf/1608.06879.pdf.

[22] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747 (2016). arXiv: `1609.04747`. URL: `http://arxiv.org/abs/1609.04747`.

[23] Pierangela Samarati. “Protecting respondents identities in microdata release”. In: *IEEE transactions on Knowledge and Data Engineering* 13.6 (2001), pp. 1010–1027.

[24] *Scikit-learn Logistic Regression Documentation.* URL: `http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression`.

[25] Ohad Shamir, Nati Srebro, and Tong Zhang. “Communication-efficient distributed optimization using an approximate newton-type method”. In: *Proceedings of the 31st International Conference on Machine Learning.* 2014, pp. 439–450.

[26] “Stochastic dual coordinate ascent methods for regularized loss”. In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 567–599.

[27] Yuchen Zhang and Xiao Lin. “DiSCO: Distributed optimization for self-concordant empirical loss”. In: *Proceedings of the 32th International Conference on Machine Learning.* 2015, pp. 362–370.