Federated Traditional Machine Learning Algorithms: From Formulation to Implementation

White Paper

Version 1.0 | April 2023

Document History

Version 1.0

Version	Date	Authors	Description of Change
1.0	April 19, 2023	Peter G.	Initial release

Table of Contents

Overview

Initially, the major advantage of federated learning is decoupling model training from direct access to the raw training data. When training a *deep neural network model* collaboratively, the communication cost is the principal constraint due to the significant number of required optimization steps, which makes conventional distributed training infeasible for decentralized data across devices/silos.

To address this issue, federated learning is proposed with the goal of being "communication-efficient": instead of collecting information needed at each step for distributed optimization, the model updates are aggregated at the server to produce a global model every few local epochs. This significantly reduces the communication burden, making collaborative deep learning possible.

Beyond deep learning, federated learning can also be applied to *traditional machine learning methods*: for example, linear regression, SVM, k-means clustering, and tree-based methods like random forest and boosting. Algorithm-wise, each of these models have distinct training schemes and model representations; implementation-wise, popular libraries providing these functionalities (e.g. scikit-learn, XGBoost) have different APIs and inner logics.

Hence, when developing a federated learning variant of a particular traditional machine learning method, considerations at these two levels need to be made:

1. At algorithm-level, how do you formulate the method under a federated/distributed setting? Specifically, you need to have answers to three major questions:
    a. What information should clients share with the server?
    b. How should the server aggregate the collected information from clients?
    c. What should clients do with the global aggregated information received from the server?
2. At implementation-level, what APIs are available and how can you utilize them to implement a federated pipeline to achieve the above algorithm formulation?

Another important point is that, for traditional machine-learning methods, the boundary between "federated" and "distributed", or even "ensemble", can be much more vague than for deep learning. According to the characteristics of a given algorithm under certain settings, they can be equivalent to each other.

Also note that, as the simplest form of federation, model ensembles can always be created as a “two-step” solution (rather than iterative optimization): clients build models based on the local data, and a global model can be built as the ensemble of all local models. Therefore, the following section will focus on iterative optimization and will not discuss ensembles unless they are expected by the algorithm itself.

Federated Learning Examples

The following examples illustrate how to formulate and implement a given federated traditional machine learning algorithm. These examples use the “scikit-learn” and “xgboost” libraries for implementation.

Linear Model

Linear model can be regarded as the most basic among all machine learning algorithms, featuring the following:

- What clients can share with the server: The model consists of a set of linear coefficients (and intercepts if fitted).
- How the server aggregates: The global model can be constructed as the (weighted) numeric average of all submitted models.
- What clients do with the global model: It is solvable with SGD optimization; hence, clients can perform further optimization steps from the received global model.
- Library support: scikit-learn provides the SGD optimizer for the linear model with “warm_start” capability.

  Note: For an iterative optimization process, the “warm_start” or “re-initialization” capability is important for achieving federated learning. Only with this capability can a client perform further optimization. This may not always be available from a particular library.

K-Means Clustering

- What clients share with server: The model consists of a set of centroids and corresponding statistics
- How the server aggregates: Numeric averaging may not be ideal for this application. Instead, the global model can be constructed by regarding the submitted models as mini-batches and syncing them according to the mini-batch K-Means algorithm.
- What clients do with the global model: Following the mini-batch K-Means sync, clients can perform further optimization steps from the received global centroids.

- Library support: scikit-learn provides a method for the mini-batch K-Means sync, but unlike the linear model, it does not come with a “warm_start” capability. Hence, the clients need to create a new model instance with “init=” specified to accomplish this goal every round.

Non-Linear SVM

- What clients share with the server: The model consists of a set of support vectors.
- How the server aggregates: A global round of SVM can be performed over the collected local support vectors to generate the global model.
- What clients do with the global model: Theoretically, SVM can be formulated as an iterative optimization process.
- Library support: scikit-learn provides a method for SVM training, but it supports neither “warm_start” nor iterative optimization.

Hence, federated SVM with scikit-learn can only be formulated as a two-step process: Local SVM rounds on each client’s data and global SVM rounds over clients’ support vectors.

Random Forest

- What clients share with the server: The model consists of a set of trees–a “sub-forest”.
- How the server aggregates: Bagging the sub-forests together to generate the global random forest.
- What clients do with the global model: By definition, random forest is not an iterative process, so no further optimization is needed.
- Library support: xgboost provides a method for random forest building.

In the case of random forest, by definition, federated learning from clients is equivalent to model ensembles.

XGBoost

Unlike random forest, where trees are trained in parallel, boosting is a sequential process, each step optimizing against the residual error produced by previous models. Since the communication frequency/cost is not as significant as deep learning, we can formulate federated XGBoost in two ways:

- Histogram-based:

    - What clients share with the server: Local data statistics (“histogram”) needed for building the tree
    - How the server aggregates: Finish the tree-building process with the collected data statistics from all clients.
    - What clients do with the global model: Compute new data statistics with the received model.
    - Library support: In this case, federated learning is equivalent to distributed training, which XGBoost supports.
- Tree-based:
    - What clients share with the server: A tree trained on local data
    - How the server aggregates: Bagging the local trees received from all clients to form a “forest” to be added to the boosting model.
    - What clients do with the global model: Train a new tree based on the global model of the boosted forest.
    - Library support: XGBoost supports separated statistics computation based on a model and tree construction with the results. Hence, although not supported in a simple API, this is achievable with some effort.

The distinctions between histogram-based and tree-based method are two-fold: first, histogram-based clients send the intermediate results to the server, while tree-based clients send the trained tree model; second, the global model for the histogram-based method is a boosted tree sequence, while for tree-based method it is a boosted forest sequence--each forest consists of “num_client” trees.

Conclusion

In conclusion, for federated traditional machine learning algorithms (unlike deep learning), the communication cost may no longer be the principal constraint, which gives you more flexibility in how to formulate the problem (for example, with XGBoost). On the other hand, each algorithm can have different forms of model and different optimization methods, so you need to specifically design what to share, how to aggregate, and how to proceed. Finally, APIs from a particular library may or may not support some formulations, limiting the options for final pipeline implementation.