**PySyft – evaluation**

* **How does it make sure privacy is preserved?**

It is ensured by the modules included in the PySyft package and the code implementations. Every time when we request code execution via

dl\_training\_project = sy.Project(...)

dl\_training\_project.create\_code\_request(syft\_fl\_experiment, datasite)

project = dl\_training\_project.send()

we do not access the raw data! Assuming that the submitted code was verified and approved before being submitted to the client, it should not transfer any information about local database outside. The code is executed locally on the clients’ dataset and only the result is sent back. It can be done only via

datasite.code.dl\_experiment(data, model\_params,training\_epochs).get()

Therefore, privacy is preserved for the whole time of the machine learning experiment. If we try to access raw data, it will return an error (unless you are logged in as a client or did not apply create\_code\_request correctly). When testing your code to check for privacy preserving features, you should always log out from clients’ accounts (if you are uploading the data yourself to clients in experimental setting). Otherwise, you will be able to access the private data not because you did not apply privacy features correctly, but because you are recognized as the client.

If you are logged out and applied built-in PySyft functions correctly, it automatically ensures data privacy, you do not have to do anything.

In practice, this is not an issue: you can never be logged in as a user while running your experiments. Therefore, the access to the raw data will be refused. It also does not leak in any way, only the result is passed. If you want to ensure that you also cannot trace back where from the result is coming, you can add extra layers of privacy by using other PETs. However, the mere application of PySyft, by default provides data privacy.

* **Ease of use**

PySyft does not have as detailed documentation as other Python packages, but the community around it is growing and the package is systematically updated. Once you understand how each of the functions work, it is as comfortable to use as any other Python package. There are some issues while debugging since it is not the most commonly used package and there are not as many solutions published online. Overall, it is a great package especially for experimentation and learning about FL. It is more important to understand how FL works and what are the principles of working on data ‘that is not there’, rather than difficulties with applying the package.

Shortly: If you are aiming for quick deployment tool, it is probably not the best choice. If you are looking for a tool to quickly apply the whole model, it is also not the best choice. But if you are looking for a tool that supports understanding of FL and allows for experimentation and individual applications, it is perfect (assuming you have a good command of Python).

* **What works well**
* It is compatible with other Python packages.
* Once you understand the principles of FL, it is straightforward to apply
* Flexible (allows for many modifications; to apply a full FL loop, it makes use of other Python packages and does not restrict any functionality; therefore, you can apply any kind of model you want and request any kind of output if it aligns with privacy regulations; it also allows for adding specific input and output policies)
* Safe (it does not allow access to raw data unauthorized parties)
* Preserving privacy is built-in into the functions provided
* **What could be improved**
* No one function implementing FL -> you have to create your own machine learning model, evaluation, and training loop.
* Despite all functions provided, implementation is still manual
* No functions answering the issue of averaging parameters of different shapes in the federated learning loop -> it has to be applied manually
* The package is still being developed -> some available tutorials have outdated solutions and does not utilize full potential of PySyft
* Knowledge of machine learning required to be able to apply federated learning with PySyft (you need to create the model and loop manually)
* Useful for academic experimentations but not the best choice for projects ready for deployment
* **Things to keep in mind**

Depending on the used data, some issues might occur. While defining the machine learning model, it is necessary to define the architecture of the used network. In that case, the most intuitive approach is to adjust it dynamically to each dataset we are using. It is also the most efficient method. However, in case of federated learning, this approach often fails. Different architecture across different clients produces mismatch in parameters. There are two ways to approach this problem. Firstly, set a fixed architecture for all clients; the unused spaces will be filled with zeros and will not affect the whole learning procedure. The other approach is to apply advanced averaging function that can account for different shapes of incoming parameters.

I encountered this issue while working with categorical data. My neural network was defining its architecture based on the unique features present in the dataset. As mentioned, there are a few ways to work around this issue. It is also possible to ‘skip’ the problem by ignoring the mismatching layers, however, this will not return the most satisfying results.

* **Advanced federated averaging methods**
* FedMA

This approach matches and averages hidden elements (such as hidden states, neurons in fully connected layer) between layers to construct a general model. This way it does not matter what are the global parameters, because this technique looks deeper into the model training and takes functional similarity into account. Authors of this implementations utilized PyTorch. Mainly useful when clients have models with different numbers of neurons.

Repository: <https://github.com/IBM/FedMA/tree/master>

* Partial Model Averaging (PMA)

In this approach not all of the parameters are considered while averaging. This way, each client can contribute to the global model and keep its own unique features that can be used locally. This addresses an issue of individual differences and model discrepancies. It also has potential of making FL more efficient.

Article introducing the method: <https://arxiv.org/pdf/2201.03789>

* Combination of the two -> FedMPT

This approach addresses real-world scenario: devices have different computational capabilities so some of them might work on partial models. Therefore, they can also contribute only partially to the global model. This model is able to compute and average parameters of different dimensions. It also prioritizes deep layers which usually hold more meaning.

Article introducing the method: <https://arxiv.org/pdf/2311.10002>

Generally, there are not as many advanced methods. It is more common to try to keep the architecture of the model the same throughout the whole experiment. It might not be the most efficient solution, but it works well and is already used in applications.

* **Requirements**

All files and experiments were done with 3.10.17 version of Python and 0.9.5 version of PySyft. It is possible that some of the issues listed in this file will have an easier solution in a near future.

* **Sources**

Official repository: <https://github.com/OpenMined/PySyft>

Official website with tutorial on how to use PySyft: <https://docs.openmined.org/en/latest/getting-started/introduction.html>

Official article on how to use PyTorch with PySyft (required to apply any kind of model in practice): <https://openmined.org/blog/upgrade-to-federated-learning-in-10-lines/>

Official repository with the newest tutorials: <https://github.com/OpenMined/syft-heart-disease-tutorial/tree/main>

The latest PySyft updates: <https://openmined.org/blog/announcing-pysyft-09/>

New approach on parameter averaging: <https://openmined.org/blog/fl-in-10-lines-of-code-with-pysyft/>