An Asynchronous Federated Learning Approach for a Security Source Code Scanner

Sabrina Kall[1] and Slim Trabelsi[2]
[1]*EPFL, Lausanne, Switzerland*
[2]*SAP Labs France, Mougins, France*

Keywords: Federated Learning, Machine Learning, Cyber Threat Intelligence, Password Detection, Privacy, Security, Threat Awareness, Personalization.

Abstract: Hard-coded tokens and secrets leaked through source code published on open-source platforms such as Github are a pervasive security threat and a time-consuming problem to mitigate. Prevention and damage control can be sped up with the aid of scanners to identify leaks, however such tools tend to have low precision, and attempts to improve them through the use of machine learning have been hampered by the lack of training data, as the information the models need to learn from is by nature meant to be kept secret by its owners. This problem can be addressed with federated learning, a machine learning paradigm allowing models to be trained on local data without the need for its owners to share it. After local training, the personal models can be merged into a combined model which has learned from all available data for use by the scanner. In order to optimize local machine learning models to better identify leaks in code, we propose an asynchronous federated learning system combining personalization techniques for local models with merging and benchmarking algorithms for the global model. We propose to test this new approach on leaks collected from the code-sharing platform Github. This use case demonstrates the impact on the accuracy of the local models employed by the code scanners when we apply our new proposed approach, balancing federation and personalization to handle often highly diverse and unique datasets.

1 INTRODUCTION

The recent explosion of data resources has opened up a world of possibilities for machine learning, which relies on large amounts of data to improve its accuracy. However, gathering the data for training remains a challenge, especially as data owners become more aware of the security and privacy risks involved in sharing potentially sensitive information. Cleaning and labelling the data is also a bottleneck, often requiring human intervention.

Federated learning provides a solution to these problems. Instead of pooling data, machine learning models are brought to the local machines of the data owners. The owners can personally label their smaller, to them better-known datasets, and train the models on their machines, then share the resulting weights with the central node. The central node merges the weights to create a final global model which has learned from all the available data (Yang et al., 2019), (Li et al., 2020), (Kairouz et al., 2019), (McMahan et al., 2017). This process has already been shown to work efficiently in real-world systems such as Google's next-word prediction learning task for smartphone keyboards (Yang et al., 2018). However, federated learning also presents its own set of challenges, in particular the lack of control offered to clients, whose behavior is often dictated by the central node, and the heterogeneity of the data between edge node datasets.

We begin by addressing the issue of control. In many federated learning processes, scheduling and participation are dictated by the central node rather than the clients. Of course in certain contexts, such as extremely large systems with weak clients, this makes sense. It would be prohibitively inconvenient, for example, to ask smartphone users to decide whether they want to share their keyboard data every single time Google decides to retrain its next-word prediction model. However, in a process with a smaller set of more powerful data owners, these clients might want to decide on a case-by-case basis when and whether to contribute to learning. Motivations for this decision can range from having a particularly vulner-

Kall, S. and Trabelsi, S.
An Asynchronous Federated Learning Approach for a Security Source Code Scanner.
DOI: 10.5220/0010300305720579
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 572-579
ISBN: 978-989-758-491-6

able dataset at a given iteration to not having enough data or hardware capability at the time chosen by the central server.

We also address the issue of data heterogeneity across edge nodes. Federated learning clients tend to have non-identically distributed data between each other, with each participant having their own unique patterns and particularities in its data, which can make a generic global model a lose-lose solution. Our final goal is therefore to create a set of local models that work best for their respective clients, rather than the optimal global model which federated learning systems usually aim for.

In this paper, we introduce an asynchronous federated learning process in which control of the learning process has been shifted from the central server to the edge nodes, reflecting a power dynamic more reminiscent of a decentralized network than of the typical master-slave system. Clients can join or leave the training process and share their weights with the server as they please, to allow them to account for particularly sensitive datasets they might prefer not to share. They can also locally personalize the global model to tailor it to the particular needs of their data.

We design this process specifically to be used with SAP's machine learning scanner, known as the Credential Digger[1], a code review aide which allows users to identify sensitive information such as passwords or access tokens in their Github repositories that need to be removed before the code is pushed to the platform. The scanner requires training data that is by nature extremely private and difficult to obtain, as few people would willingly hand over their passwords, especially labelled as such, which is why we turn to federated learning. Working with the Credential Digger's text-based machine learning models, we show the viability of our process on a real-world case using the Credential Digger to scan test data collected from selected public Github repositories.

2 USE CASE

With more than 100 million repositories, GitHub[2] is the world's largest hosting platform for collaborative software development and version control. At least 28 million of these repositories are publicly accessible, and a 2019 survey of 13% of them revealed that, at a conservative estimate, over 100'000 repositories contain sensitive information such as passwords or access tokens, with thousands of leaks more being published each day (Meli et al., 2019).

While there exist scanners such as TruffleHog[3] to identify these leaks so that they can be removed, these tools have been found to suffer from a high rate of false positives (Meli et al., 2019), flooding users with hits incorrectly identified as sensitive information and concealing the true leaks in the metaphorical haystack.

To remedy this problem, SAP developed the Credential Digger, a scanner using machine learning to classify hits into actual leaks and false positives to make it easier for developers to find and remove sensitive information from their repositories before they are published. The tool first scans a repository using regular expressions to identify potential leaks, then feeds each hit into two shallow neural networks for text classification, which analyze the path (for example to exclude documentation) and the code snippet to decide whether it is a true secret or a false positive, as seen in Figure 1. Once the scanner has classified the hits, they are displayed to the user, who can manually fix their code, validating the labels in the process and generating a new labelled training set for the models. Currently, the tool is trained using both real and synthetic data. However, the volume and the diversity of the real data is not sufficient to cover all the programming languages and the coding styles of the developers. Since it is hard for a single research team to manually gather, classify and label the required amount of training data, the federated learning approach with different external contributors appears to be the best solution to reach a decent amount of exploitable training data.

We started by testing a traditional asynchronous federated learning process, which gave us wildly divergent results when applying the final averaged model to local clients due to their heterogeneity. For this reason, we developed a new approach enhancing the existing solutions and algorithms in order to provide more accurate models tailor-made for each client.

3 RELATED WORK

There has been extensive work in recent years in the field of **federated learning** (Yang et al., 2019), (Li et al., 2020), (Kairouz et al., 2019), particularly since the Google keyboard experiment of 2016 (McMahan et al., 2017), (Yang et al., 2018), (Sprague et al., 2019). Federated learning comes in synchronous and asynchronous flavors, with different types of scheduling for the federated nodes. We ourselves use an asyn-

[1] https://github.com/SAP/credential-digger

[2] https://www.github.com

[3] https://github.com/dxa4481/truffleHog

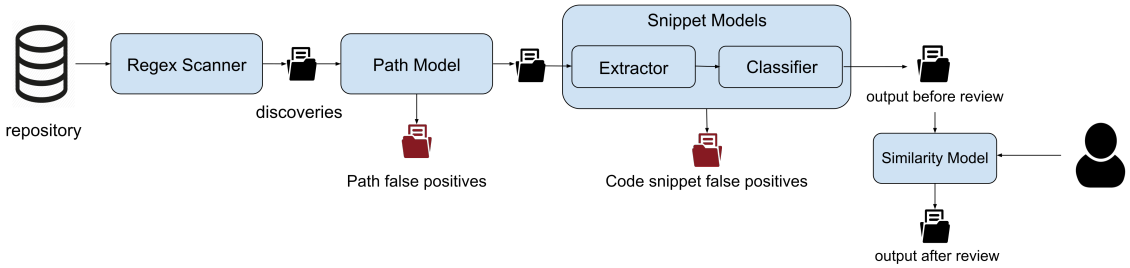

Figure 1: Architecture of the Credential Digger Tool.

chronous process, allowing clients to participate in their own time.

Personalization of machine learning models (Mansour et al., 2020), (Wang et al., 2019) currently focuses mostly on synchronous learning. Techniques such as data interpolation and model interpolation help clients adapt generic global models to their own specific use cases by respectively retraining models on their own data or averaging their results with locally trained models before use. We incorporate these two methods in our learning process to allow edge nodes to tailor their models to their specific data, overcoming the issue of heterogeneity among client datasets.

A further avenue of research goes into different **merging algorithms** to combine client models at the server (McMahan et al., 2017), (Xie et al., 2019), (Sahu et al., 2018), (Wang et al., 2020), (Yurochkin et al., 2019). This ranges from the original *FedAvg* algorithm, which merges the weights layer-wise, to more complex and case-specific formulas such as *FedProx* (Sahu et al., 2018) or *FedMa* (Wang et al., 2020). After testing several of these algorithms, we have found empirically that in our case, the best solution is the *FedAsync* (Xie et al., 2019) algorithm.

Finally, there exist several frameworks for federated learning, most notably Tensorflow Federated [4] and PySyft [5]. We tested both of them, but ultimately found it difficult to integrate our pre-existing machine learning models and to adapt the relatively inflexible set-ups to our specific use case, which is why we make our own from scratch.

4 IMPLEMENTATION

We implement our asynchronous federated learning system in the following way. Clients receive generic models trained on synthetic data from a central server. When they have access to fresh data, these clients can retrain their local model and share its weights with the server. The server can then merge the received model weights with the old model weights and, if it is not deteriorated according to a server benchmark, redistribute the merged model weights to all the other clients to let them profit from the additional training as well. Below is a breakdown of each of the components of the learning process.

4.1 Architecture

We build our federated learning system as a network of Credential Digger scanners at client nodes, connected by a trusted central server. We consider our edge nodes in this scenario to be development teams. Developers run the Credential Digger on their repositories, which shows them the suspected secrets and proposes labels for them. Labels are either ”new discovery” or ”false positive”. If the users wish to do so, they can then manually fix faulty labels as they go through the results to safeguard their repository. This generates a new training dataset of code snippets with manually corrected labels which can be used to retrain the models. We assume that the users have the necessary computing power to handle local retraining and a sufficiently good network access to send and receive model weights to and from the central authority running perpetually on a server safely accessible to the developers. We also assume that none of the edge nodes are malicious.

When the central server receives new model weights from a client, it merges the weights with those of its most recent model, and redistributes the new weights to all its clients, the developers. These developers can update their tool and run a personalization process on the models using their local datasets to tailor the model to their local settings. We assume that repositories within development teams will have similar data. For example, a specific team will probably

[4]https://www.tensorflow.org/federated
[5]https://github.com/OpenMined/PySyft

use the same programming languages, paths, token types and even passwords and dummy values across repositories. The whole process can be seen in Figure 2.

One of the particularities our learning process must account for is that for our models, a good recall is more important than a good precision. Indeed, a false positive, for example a placeholder, being mistaken for real information is less serious than a real secret being dismissed as a false positive by the models, as this makes it more likely for a user to overlook a potentially dangerous leak. We therefore use precision, recall and f1-scores to benchmark our models and design our algorithm to safeguard recall. Before sharing new model weights, the server and edge nodes alike test the models on the original synthetic dataset to ensure that results are not degraded.

4.2 Benchmarking

The benchmarking function determines whether a new model is good enough to be shared with the other participants of the federated learning process. We adapt it to our use case for the Credential Digger, which needs a high recall to prevent the user from overlooking potentially dangerous leaks. Our benchmark is described in Algorithm 1. The new model is tested on all the available data of the model owner. In the server's case, this is the synthetic data used to train the base model. Clients can use this synthetic data as well as the local dataset provided by the user. We compute the recall and the f1 score and verify that neither is deteriorated. This way, we make sure that recall is not abandoned in favor of precision. If the new model is found to be at least as good as the old model, it can be shared.

Algorithm 1: Federated Learning: compare.

Data: old_model, new_model, local_data
recall_ok = recall(old_model , local_data) $\leqslant$ recall(new_model, local_data)
f1_score_ok = f1(old_model, local_data) $\leqslant$ f1(new_model, local_data)
return recall_okay AND f1_score_ok

4.3 Server-side Merging

Updating the central server weights with a client's new weights is done using the server side of the asynchronous federated averaging algorithm *FedAsync* (Xie et al., 2019). We found it empirically to be the most effective merging algorithm for the Credential Digger's shallow neural network models, as well as the simplest. It consists of adding together the weights of models layer-wise, with each model's weights scaled by a factor determined by the staleness of the client model. Indeed, staleness is an important characteristic to consider during asynchronous federated learning. If, for example, client **a** submits a model based on the server model at round **5**, but other clients have already updated the server model multiple times, so that it is now at round **15** (having been incremented at each successive update), merging the new server model with client **a**'s model has the potential to cause divergence because the weights are too different. We therefore mitigate the power of client **a**'s weights using a staleness function.

The central server starts out with the base model, the round $t = 1$ and the scaling factor $\alpha = 0.5$, which was found to be optimal for *FedAsync*. In order to merge a client's weights into the server model, a server must receive two values from said client: the model weights themselves, but also the round τ belonging to the latest server model its new model is based on.

The server then mitigates the effects of any stale client models by adapting α_t using the polynomial staleness function:

$$\alpha_t = (t - \tau + 1)^{-\alpha} \tag{1}$$

Layers of the model are then merged in the following way:

$$\begin{aligned} merged\_layer = \; & (1-\alpha_t) * server\_layer_t \\ & + \alpha_t * client\_layer_\tau \end{aligned} \tag{2}$$

The merged layers form a new model which is locally tested using the synthetic data on which the base model was trained, and benchmarked according to our *compare* algorithm shown in Algorithm 1. If the model passes the test, it becomes the new server model, and the server round is updated by one: $t = t + 1$. The server redistributes the weights of the new model along with the round to all its known clients. The process in its entirety is described in Algorithm 2.

4.4 Client-side Model Personalization

We assume that clients will have datapoints that are more similar to each other than to datapoints of other clients. Indeed, developers all have their own coding habits which depend on many factors, such as the coding language or the occurrences of given keywords. This kind of variation means that the most optimized model for each developer may differ, and that forcing a generic global model might erase such subtleties. It is therefore beneficial for a client to personalize the

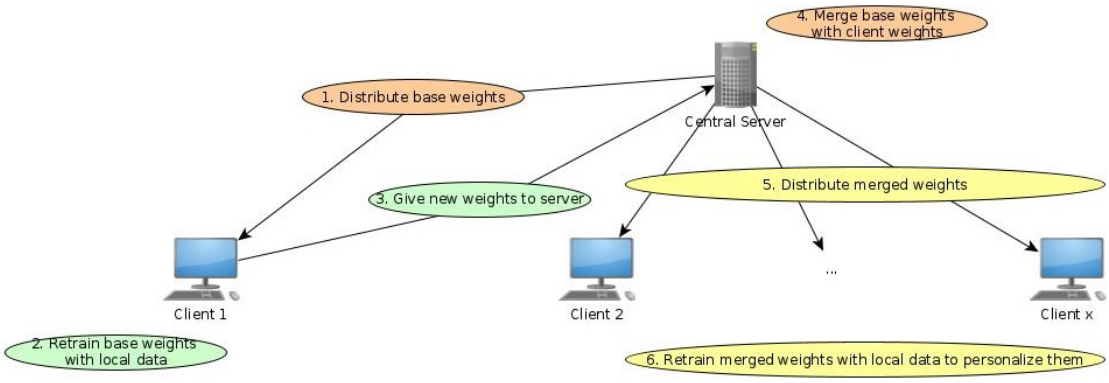

Figure 2: Federated Learning: Overview.

Algorithm 2: FedAsync with staleness.

Data: initial_model_weights
Result: a model created through the fedasync algorithm

server_weights = initial_model_weights
$\alpha = 0.5$
for *round t in 1,2,3,...* **do**
  get (client_weights, τ) from some client
  $\alpha_t = (t - \tau + 1)^{-\alpha}$
  $\beta_t = (1 - \alpha_t)$
  for *l in 1 to nb_layers_in_model* **do**
    merged_weights[l] = α_t * server_weights[l] + β_t * client_weights[l]
  end
  if *compare(server_model, merged_model, synthetic_data) == True* **then**
    t = t + 1
    server_weights = merged_weights
    share (server_weights, t) with clients
  end
end

global model using its available local data. We apply two personalization techniques, model interpolation and data interpolation (Mansour et al., 2020), described respectively in Algorithm 4 and 5. The personalization processes are applied consecutively as described in Algorithm 3.

4.4.1 Model Interpolation

In model interpolation, we reuse *FedAsync* to locally average the client's previously personalized model, if one exists, and the server's newly offered model. To scale the weights, instead of worrying about staleness, we try out different values from a cover of the (0, 1)-range, as explained in Algorithm 4.

Algorithm 3: Federated Learning: update.

Data: local_model, global_model, local_data
Result: an updated local_model

new_model = average_models(global_model, local_model)
if *compare(local_model, new_model) == True* **then**
  local_model = new_model
end
new_model = refine(local_model, local_data)
if *compare(local_model, new_model) == True* **then**
  local_model = new_model
  share local_model with server
end

Algorithm 4: Federated Learning: average_models.

Data: old_model, new_model
Result: an averaged_model at least as good as the old_model

if *old_model exists* **then**
  best_model = old_model
  for λ *in [0.2, 0.4, 0.6, 0.8]* **do**
    avg_model = (1- λ) * old_model + λ * new_model
    if *compare(best_model, avg_model)* **then**
      best_model = avg_model
    end
  end
end
else
  best_model = old_model
end
return best_model

4.4.2 Data Interpolation

The process of data interpolation is fairly straightforward. Whenever new data is provided, the client adds it to their local data and refines the model on the entire dataset, trying out a range of batch sizes β to find the best one. This process is detailed in Algorithm 5.

Algorithm 5: Federated Learning: refine.

```
Data: model, data
Result: the best possible model fit on the data
best_model = model
for β in [16, 32, 48, 64] do
    new_model = model.fit_on(data, β)
    if compare(best_model, new_model) then
        best_model = new_model
    end
end
return best_model
```

5 EVALUATION

We test our method in a real-world scenario: using the Credential Digger to detect leaks in existing public Github accounts. We apply our algorithm to both of the available machine learning models, the model analyzing code snippets as well as the model analyzing file paths, and train them using manually harvested data from Github. Throughout the training, we benchmark our federated learning process against the base models trained using only synthetic data, as well as a traditional learning process using all our training data.

5.1 Snippet Model

The goal of the snippet model is to analyze a pair of words, a keyword and credential value, for example `("password", "snoopy")`, and decide whether the credential is real or not.

5.1.1 Dataset

We select at random five public Github accounts with three repositories each, aiming for repositories whose scan results return a decent number of potential leaked credentials. The scan results are manually labelled, written to file, and deleted after the end of the simulation run to preserve the privacy of the account holders. This yields a diverse dataset of 26'434 labelled code snippets. Furthermore, our base model with which the server starts out is trained on 7478 generic `(keyword, common password)` and `(keyword, placeholder)` combinations, such as `("password", "snoopy")` or `("token", "PUT_YOUR_TOKEN_HERE")`.

5.1.2 Simulation

We run a simulation of five clients, each representing one of the Github accounts. At each round, one of the clients picks one of the repositories of its account and simulates a run of the Credential Digger and subsequent federated learning process on it. The resulting model is tested at each client for that round using the data from the account designated for testing.

5.1.3 Results

As can be seen in the results of Figure 3, our tailored algorithm yields good results. The new real-world data is too noisy and diverse for centralized training to be effective, making the base model trained on generic data imprecise, and the centralized model trained on all our pooled data barely any better. However, the federated learning model, by personalizing for each client and its dataset, manages to overcome this challenge and improve the results.

A question we could ask ourselves is whether sharing model weights through federated averaging is useful in practice, or whether the improvement we see is due solely to local personalization. To resolve this, we look at the breakdown of the federated learning process by client in Figure 5. We find that there are variations in the recalls of clients when other nodes share their weights, which tells us that there are indeed scenarios in which the federated averaging process is impactful, with clients reacting to model updates made by their peers.

5.2 Path Model

We then move on to test our path model, as seen in Figure 1, which analyzes filepaths to determine whether a file is likely to contain false positives, such as test files or documentation.

5.2.1 Dataset

Once again, we select 5 random accounts from Github, this time with 2 repositories each. Note that repositories tend to contain fewer filepaths than snippets, which reduces the size of our test dataset compared to the snippet model evaluation. Path models are also longer, giving us more data per datapoint, meaning we must be cautious of overfitting. During labelling, each time a file contains even a single true positive, it is marked as a true positive. In order

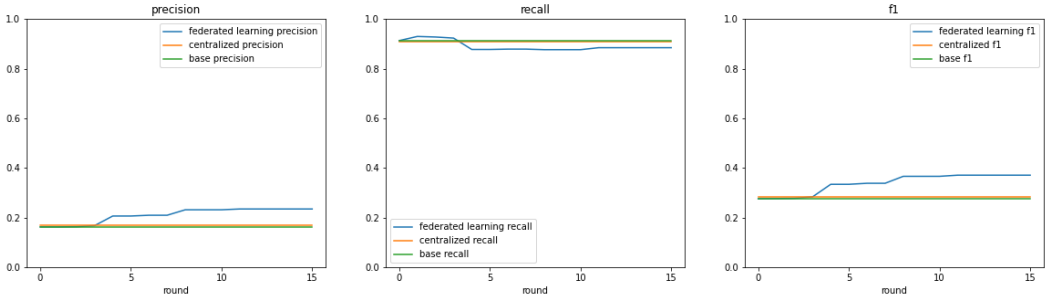

Figure 3: Evolution of the models on a dataset of 26434 snippets over 15 rounds of federated learning, compared to an initial base model of 7478 snippets and a centralized model trained on the whole training dataset.

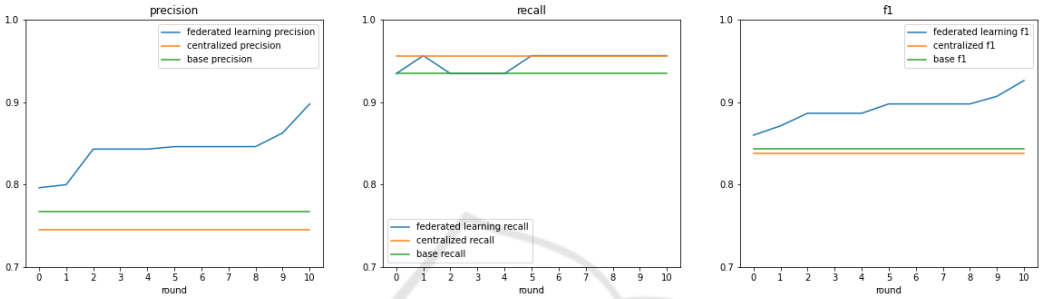

Figure 4: Evolution of the models on a dataset of 314 filepaths over 10 rounds of federated learning, compared to an initial synthetic base model of 1759 filepaths and a centralized model trained on whole training dataset.

to be considered a false positive, a file must contain nothing but dummy discoveries. Our base model is once again trained on a synthetic dataset with typical test and documentation paths. The simulation occurs analogously to the snippet model simulation.

5.2.2 Results

As shown in Figure 4, federated learning outperforms the base model and even the centralized model thanks to personalization, which, on the highly specific local paths of repositories, greatly improves the local models. However, an analysis of the results suggests that in this case, federated learning plays a far smaller role than personalization, as filepaths tend to be extremely specialized to their repositories.

6 CONCLUSION

Leaked credentials on open-source code-sharing platforms represent a pervasive cybersecurity threat for developers, and a time-consuming one to mitigate. Machine learning can help, but requires very sensitive training data and a lot of localized fine-tuning. Current federated learning platforms and applications are mainly focused on a centralized, synchronous approach where the unique features of local models are neglected and accuracy achieved with local users is penalized. The current existing solutions are not efficient with our use case. In this paper we propose a new approach that takes into consideration the specificity of the local clients and reflects it on the accuracy of the local models. We build an asynchronous federated learning and personalization system to give machine learning models access to private training data and improve them for use with the Credential Digger scanning tool. Through this system, users can iteratively improve the detection of hard-coded tokens and secrets in their code while reducing the privacy and security risk to their local training data. We demonstrate the use and effectiveness of our system on data from real-world public Github repositories. As future work, we can enhance our solution with improvements on the fronts of privacy, in order to safeguard locally trained models during transfer and merging (for example with differential privacy, homomorphic encryption of the weights or secure multi-party computation (Kairouz et al., 2019)), vocabulary expansion, and staleness. We can also explore the generalization of our federated learning pattern to other use cases.

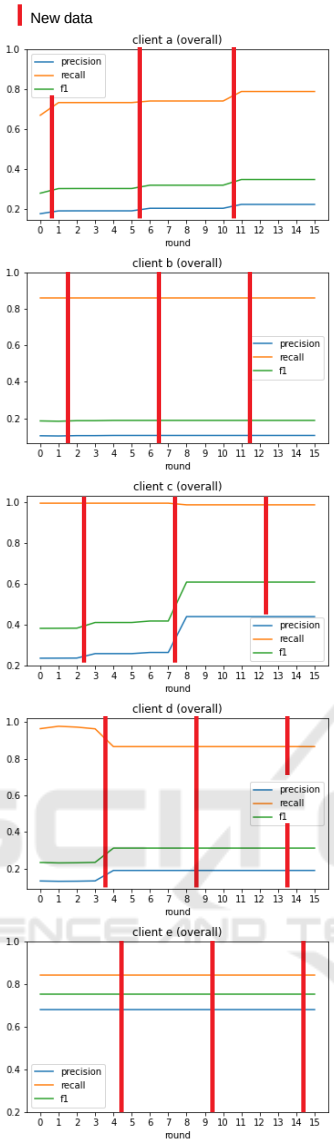

Figure 5: Evolution of the federated model on 26434 code snippets over 15 rounds of federated learning, broken down by client.

ACKNOWLEDGEMENTS

Many thanks to the creators of the Credential Digger, Dr. Marco Rosa, Sofiane Lounici, Carlo Negri and Jarod Cajna at SAP Labs, and to Prof. Boi Faltings at EPFL.

REFERENCES

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Rouayheb, S. E., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak, M., Konečný, J., Korolova, A., Koushanfar, F., Koyejo, S., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F. X., Yu, H., and Zhao, S. (2019). Advances and open problems in federated learning.

Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. (2020). Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60.

Mansour, Y., Mohri, M., Ro, J., and Suresh, A. (2020). Three approaches for personalization with applications to federated learning.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data. In *AISTATS*.

Meli, M., McNiece, M. R., and Reaves, B. (2019). How bad can it git? characterizing secret leakage in public github repositories. In *NDSS*.

Sahu, A. K., Li, T., Sanjabi, M., Zaheer, M., Talwalkar, A., and Smith, V. (2018). On the convergence of federated optimization in heterogeneous networks. *CoRR*, abs/1812.06127.

Sprague, M. R., Jalalirad, A., Scavuzzo, M., Capota, C., Neun, M., Do, L., and Kopp, M. (2019). Asynchronous federated learning for geospatial applications. In Monreale, A., Alzate, C., Kamp, M., Krishnamurthy, Y., Paurat, D., Sayed-Mouchaweh, M., Bifet, A., Gama, J., and Ribeiro, R. P., editors, *ECML PKDD 2018 Workshops*, pages 21–28, Cham. Springer International Publishing.

Wang, H., Yurochkin, M., Sun, Y., Papailiopoulos, D., and Khazaeni, Y. (2020). Federated learning with matched averaging.

Wang, K., Mathews, R., Kiddon, C., Eichner, H., Beaufays, F., and Ramage, D. (2019). Federated evaluation of on-device personalization. *ArXiv*, abs/1910.10252.

Xie, C., Koyejo, S., and Gupta, I. (2019). Asynchronous federated optimization. *CoRR*, abs/1903.03934.

Yang, Q., Liu, Y., Chen, T., and Tong, Y. (2019). Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10(2).

Yang, T., Andrew, G., Eichner, H., Sun, H., Li, W., Kong, N., Ramage, D., and Beaufays, F. (2018). Applied federated learning: Improving google keyboard query suggestions.

Yurochkin, M., Agarwal, M., Ghosh, S., Greenewald, K., Hoang, N., and Khazaeni, Y. (2019). Bayesian nonparametric federated learning of neural networks. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7252–7261, Long Beach, California, USA. PMLR.