Building Ensemble Models with Web Services on Microservice Architecture

Máté Szabó
Department of Information Technology, Faculty of Informatics, University of Debrecen Kassai út 26, H-4028, Debrecen, Hungary
E-mail: szabo.mate@inf.unideb.hu

Keywords: machine learning, microservice architecture, ensemble, web service

Received: June 2, 2023

The combination of machine learning with web services is not rare, as it is a possible way to make the models reachable to other applications. For example, a mobile or web application with recommendation feature can send requests to query the model's prediction. The advantage of this method is that it does not require to use the same platform or programming language on the model and application side. This paper investigates the building of ensemble models with web services, in a complex microservice architecture-based application. The ensemble models are special, because they rely on other pre-trained models, so they can act as a wrapper model. The advantage of this approach is that it is applicable to multiple models that are written in different programming languages. When we have these wrapper models, all of them can be accessed through web services, which leads to many small services that can be managed together in an application on microservice architecture. In this paper, we combine models from Scikit Learn, Tensorflow, Weka and Deeplearning4j libraries to show how models written in different languages can work together. We propose two similar architecture variants involving machine learning and microservices to combine models from different platforms. The gateway variant uses patterns like API gateway or backends for frontends, the direct variant uses direct access to web services. The integration of these into existing web applications is also presented considering the server or client-side computing load. The analysis shows that both of them can be used, but with different software systems. The direct is preferred when the application partly relies on machine learning services and thus only using few of them, and the gateway is preferred when the application is dependent on these services.

Povzetek: Raziskava predstavlja integracijo ansambelskih modelov v mikroservisno arhitekturo preko spletnih storitev za združevanje različno napisanih modelov.

1 Introduction

When we train machine learning models, it is often done in our favored programming language; however, it might be complicated to integrate it into an application. The easiest scenario is when the model is stored in the application, which means they are possibly written in the same language and the machine learning library is supported. In this case, we can simply just use it for example making predictions, or recognizing images. When it is not supported or usable by our preferred language, the simplest solution is to have a different application just for the machine learning tasks and it will communicate with the original one. This solution requires extra resources for communication as the model or results have to be sent on the network. This first scenario can cover applications built on monolith architecture; the second one can cover the layered architecture. Of course, these are just the obvious examples of integrating machine learning in our web applications. Cloud services can be an option but its results are also queried through web services, so it is similar to the second case.

When we want to integrate machine learning into applications, we have to decide what kind of algorithms we would like to try and how to access it. When we have the answer for accessing the models from applications, our project is already restricted to a given set of libraries. For example, if we chose to make a machine learning module that can be accessed by the application directly, our project should use libraries of the same language. The other question is what kind of models would be trained? Is deep learning needed or K-nearest neighbour is enough, or is data suitable for unsupervised or supervised learning? After decisions like these, the set of available libraries is narrowed. In most of the cases, this is enough to choose, but if we want to use more than just one of them, because one of them supports algorithms that the other does not, we might encounter some trouble. In this case, the applications have to communicate even more.

Microservice architecture structures the application as a collection of services that are independently deployable and loosely coupled. Each microservice consists of data store and application logic, which can only be accessed through its public API. As these software components are

independent, each of them can use proper tools and languages to achieve their goals, so it is possible that one component is written in Python, while others can use Java if it is suitable for their tasks. As a microservice is a small part of the whole application, it can be developed or maintained by a smaller developer team and as it is independent, they can choose their own technology stack. Of course, this architectural style is not applicable to any kind of program, because the huge number of HTTP requests can be costly. The main disadvantage can be the communication between the microservices, because if the application is big enough and contains several services, the number of requests and responses can increase. In a standard web application environment, that means that the program packages the data in JSON format, sends an HTTP request, the other service receives the message, extracts the JSON to its object type, and then writes and sends a response, which is processed by the sender. This messaging can occur even ten thousand times per second. Besides this, we can mention latency, bandwidth, or topology that can affect the performance of microservice based software. The importance of this architectural style can be seen from Google Trends [1]. Many software projects use it, because development time can shorten as the tasks are much smaller and easier to implement. Besides that, the testing of these services is easier, because they are independent and the codebase is much smaller than on traditional architectures. The implementation can be written in any programming language, as it can be started in small, for example we can create 3 web services that communicate with each other and each of them has its own functionality. I chose a hybrid solution with the Spring Boot framework's reference implementation. The needed maven dependencies are spring-boot-starter-web, spring-cloud-starter, spring-cloud-starter-eureka-server. The Eureka server is for the registration of services, so they can discover each other from there. The other dependencies are for creating web services, which can accept or send HTTP requests.

This paper is restricted to web applications, as these are the most popular kinds of applications according to the JetBrains Developer Ecosystem Survey 2021 [2]. There are many ways to structure applications as it is mentioned earlier, but we would like to achieve a machine learning system that connects different libraries from different environments, so the product could be language independent. The idea is that libraries can be wrapped in web services. To support the high number of them, a lot of services should be created, and to manage them as one application we decided to use microservice architecture. It is the best decision as the elements are loosely coupled in the software and data management can be decentralized, so services can send requests to other services through their public API and each of them has the ability to manage its own database. In this way, multiple datasets and their models can be handled, for example we have a web service that wraps the Deeplearning4j library, so we can train models and query the results with HTTP requests. To manage data, train models, and extract the results, we only have to design the appropriate endpoints. When the web services are finished, the application can be assembled with the microservices.

The combination of different machine learning models is not rare as many research state that it can improve performance. There are many ensemble techniques like bagging, boosting, voting, and stacking. The idea of them is that they mostly use only the results of the different models and because of this; we can get these results from different platforms or services. One of the easiest is the voting, which can be majority and weighted. Diettrich's results show that in certain cases, ensemble models can outperform single classifiers [3]

The combination of machine learning and microservice architecture gives us the opportunity to distribute model training in both model and data parallel ways, and besides that, it has the advantage of connecting different platforms in one application. As disadvantages we can mention the dependence on the network and the high number of communications. According to Google Trends [1], the keyword Microservice has been a popular search phrase since 2016; it was at the top in 2020, nowadays the number of related searches decreased to 60 on a scale of 0-100. Although federated learning architecture also has a centralized and decentralized variant like this work, the process of training and the goal are somewhat different. Federated learning often results in a bigger model, because the local clients train models and they update the global models' parameters [4]. In this work, the client can choose between using single classifiers and ensemble models. The direct and gateway approach can align to any kind of client application, which can decide to put the computing load on the server or on its own side. We don't have parameter synchronization or parameter sharing, each model is independent. The common between federated learning and this work is that both of them train local models and both can end up in a single classifier.

2 Related work

There are many papers about connecting microservices and machine learning, but the purpose and the implementation can be different. Pahl and Loipfinger defined machine learning as a service, and their system encapsulated ML in a microservice with a REST interface [5]. DroidAutoML [6] uses the same architecture, but its goal is extended to mobile devices as it can configure and evaluate algorithms to detect Android malware. In this work, the whole ML work is decomposed into small services, like model training, feature database, scanner, apk database, which then communicate with each other. Ribeiro et al. use a similar solution for machine learning deployment [7]. They proposed an architecture to support generic ML pipelines and implemented case studies to test it. This paper uses ensembles to make predictions with different models just like Attota et al. in their work [8]. They use several API gateways with model ensembling, and these gateways communicate with the central server's aggregation service. The cCube architecture [9] has a similar purpose,

to easily develop and deploy ML applications, and for this, it uses an orchestrator service. The user communicates only with this, and it forwards the data to the learners and the scheduler. Our paper presents a centralized solution which based on this work. This technology can be applied in fields like healthcare, where KETOS [10] uses machine learning as a service and the users can use R, Python and DataShield in one application. It is a clinical decision support system with its own development and deployment process. There is a special use case of machine learning in software development, where we provision, and classify microservices with these algorithms like in [11], [12], [13]. For designing the architectures, we used the Guidelines for adopting frontend architectures and patterns in microservices-based systems [14].

Although it seems like microservice is perfectly suitable for most kind of applications; it has several drawbacks and can cause many issues through development. It can increase the system's complexity because of the increasing number of independent services. This means, that the project needs skilled developers. The architecture also has challenges in the areas of monitoring, versioning, and state management. [15]

Distributed machine learning also has its disadvantages like the increase of aggregate processing time and the work with the multiple computing nodes. Because the training is more intense, it needs multiple computers, leading to increased energy consumption. Even with more computing resource, because of the distribution, it is not guaranteed that the training will be faster that much. [16]

3 Architecture of the system

We had many ideas about how to integrate machine learning services in a microservice architecture-based application, but we narrowed it down to 2 variants, the gateway and the direct variant. The first one is a tree-like approach where the smaller services end up in a bigger summing service and this works like as if the application would have a machine learning subsystem. This solution is similar to the MapReduce architecture, but it is specific to web applications. In the direct way, there is no such summing service, so the requestor can process the results and if it wants, can combine them. In either of them, ML microservices should build models on the same datasets, so it is tempting to have one common database behind them, because if not, we must store the same data multiple times within one application. Rules of the architecture forbid this solution, because we would lose the independency of microservices, so all of them will have its own database. There are many patterns for building applications with microservice architecture, like the mentioned API gateway [14] and backends for frontends [14] and beside them we use the gateway aggregation pattern from [17], [18]. Because of this pattern, we will often use the aggregator service and API gateway terms instead of each other. In our reference implementations we built a backend system with Spring Boot, Spring Cloud with the Eureka Server for service discovery [19]. This server manages the services' IP address and port number, so this is from where each client knows the addresses where they can send requests. The Java microservices for training and managing Weka [20] and Deeplearning4j [21] models are written in the same environment. The Python microservices for Scikit [22], TensorFlow [23] and Microsoft Azure models used Flask [24] to be accessible by other web services

3.1 Direct variant

In this variant, there is no API gateway, so we cannot use the backends for frontends pattern. As there is no gateway, the clients can directly access the models and can request predictions for their data. The combination of the trained models is not easy in this variant, because we do not have an independent node that creates ensemble models; instead, the client has to do it. The client can query the results of the model endpoints, and based on these, it can construct its own ensemble model. The advantage of this variant is that the clients can process as little data as needed, because when they need a decision tree, they can query for one, when they need a voting or averaging based ensemble method for two models, they query for the models' results and use the chosen technique. As there is no aggregator service, each of the microservices remain independent and do not rely on other machine learning endpoints' data, so the software remains loosely coupled which is an advantage in microservice architecture. Integrating this solution into a backend system could be challenging as there is no aggregator service, so the caller service must explicitly name a model service to use. We suggest this variant for applications, that use machine learning for a few of its functions. In these systems, developers can use only the needed model from its service, so the network load will be distributed between them. As few of the ML services are used in the application, it should not cause maintenance difficulties. An implementation of this variant can be seen in Figure 1. The blue arrows indicate the path of the data; the green ones are for service registration. In this figure, we can see a web and a mobile application that uses different machine learning models' result using its web service

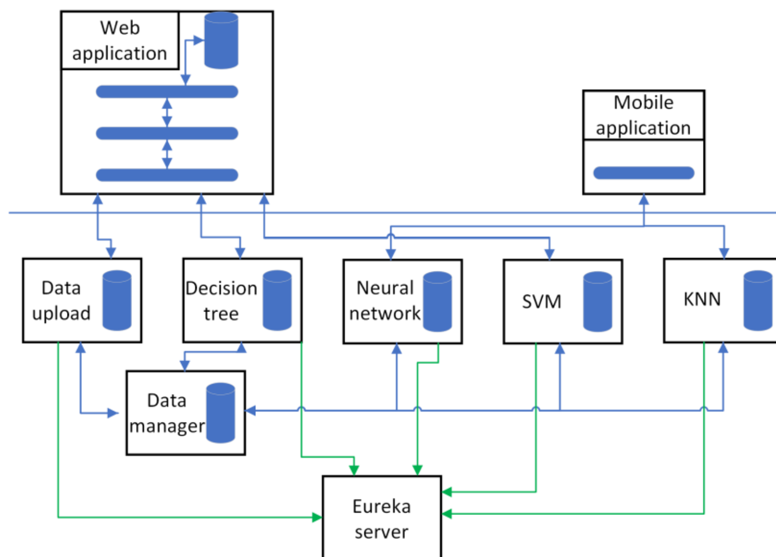

Figure 1: Direct architecture variant with web and mobile clients.

3.2 Gateway variant

The gateway variant can be achieved with patterns like API gateway and backends for frontends, where there are aggregator microservices. If we use the API gateway, there will be one aggregator service that contains client specific APIs for example one for web application and one for mobile. This service communicates with each of the needed machine learning microservice, so it can access models and predictions too. The backends for frontends pattern defines separate API gateways for different kinds of clients, thus we will have a gateway for web clients and another one for mobile clients, each of them accessing the needed machine learning microservices. In the unified API gateway for high availability clusters paper [25], the authors propose a centralized solution for managing clusters. Our work uses a similar idea, but instead of clusters, we manage other services. We consider the aggregator service as a microservice too despite its size and complexity. It is responsible for managing data, training, persisting and accessing models, making ensemble models. The combination of trained models remains on server side in this variant, as we can have a separate microservice for this purpose. In this architecture, the models are only reachable through an aggregator service, which comes with a downside, that it will encounter more network load than the other services. To avoid this problem, we could make the aggregator service scalable with multiple copies of it behind a load balancer. Integrating this solution to a backend system is not a complex task as there is an aggregator service, so each machine learning service can be accessed from it. The downside of this architecture is that both the ensemble service and the API gateway could increase coupling, which depends on the

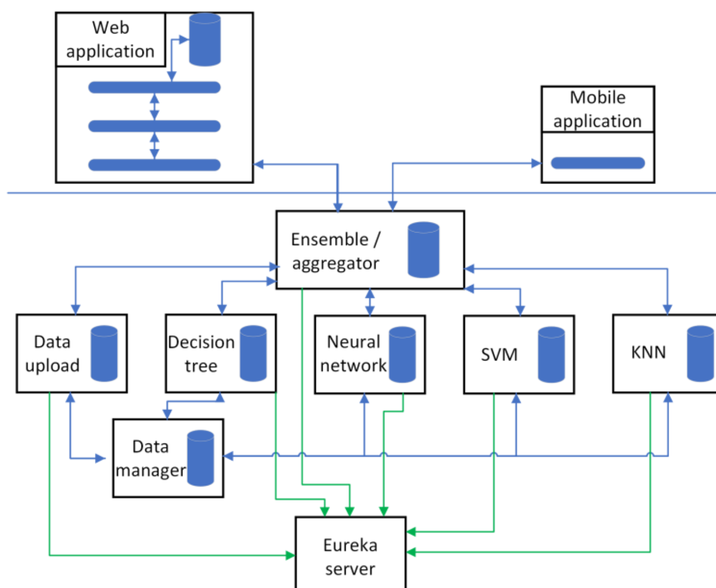

Figure 2: Gateway architecture variant with web and mobile clients.

Table 1: Endpoints of the gateway and direct variants

Service	Endpoint	Data	Description
dataup	POST /upload	list	Persist data in the database.
dtree	POST /train	dataset, parameters	Train a decision tree with the given parameters.
dtree	GET /model	model id	Get the trained model from service.
neural	POST /predict	unlabelled data	Predict the label of an unlabeled data.
dataman	GET /dataset	dataset name	Query for dataset as a list.
ensemble	GET /model	model type, id	Forward request to the appropriate microservice.
ensemble	POST /ensemble	method, list of model type, id	Build the given ensemble model using models from other microservices.
ensemble	POST /upload	list	Forward dataset to the data manager service.

implementation. We suggest this variant for applications that are using many machine learning services as developers have to use only one connection point, thus the code will be more maintainable. This variant can be seen in Figure 2. The blue arrows indicate the path of the data; the green ones are for service registration, but here the mobile and the web application uses the aggregator service to access the needed models' results.

4 Messages between the web services

The whole communication in this microservice application uses the REST API. This means that the web services have to meet the criteria of the REST architectural style. It has a uniform interface as we uniquely identify the resources and send only the necessary data in JSON and they could be manipulated through representations. When we have two microservices that communicate with each other, one can identify as a server and the other as the client. Although we do train models and persist the state of them, the application is stateless and only the resources have states. The responses are cacheable as the client can reuse them later for the same requests. The last constraint of REST is the layered system, and in this application, and especially in microservice architecture, the service does not know if it is an intermediary server or not. Almost each of the services relies on other ones, so at subsystem level, we could see the layering.

It can be seen that every workflow involves the HTTP requests and responses in this system. We send messages to persist data in the database, train a model with parameters, predict with the given data or query a model. The goal of this system is to make ML available on web capable devices, so when we have trained models, there are two ways to use them. The client can request the model itself, which can be used on client side. This way has downsides like the size of these models can be really huge and sending them through network can be challenging, in addition, the client must use the same environment as the server does, so a Java object that contains an ML model cannot be used in Python environment. If we can use the same environment, the advantage of this solution is the reusability as when we successfully transfer a model to the client, it can use it without a network connection, and it will own a local copy of the model. The other way is when the client sends data to the server, which will send back the model's results. This solution is the most commonly used one as the size of messages is usually smaller and they can easily be processed. For example, the client sends the items in a shopping cart and the response contains a set of items with the probabilities of the customer buying it. We do not have to use the same environment on the client and the server side, as the client does not receive model objects; instead, they get data that is not that complex like a set of numbers or a matrix or a list of strings, which can be processed on a wide range of environments. The only downside is that this solution depends on the network as for each prediction it must send an HTTP request and process a response, so it must be implemented with caution for devices that do not always have a stable network connection. On devices like mobiles this problem can be bypassed with a hybrid solution like storing a model on device but update them from web services. We can use TensorFlow and TensorFlow Lite in this case. In Table 1. we can see the endpoints of some microservice in the system.

5 Combining models of different platforms

The main goal of this application is not just to make machine learning usable as a subsystem, but to allow us to combine different models from different platforms with ensembles. Software developers like to use multiple platforms, languages, web services and cloud together, and these architectures below let them to integrate heterogeneous machine learning services with aggregator services. In Figure 3, we can see that there is not much difference between the gateway and direct variant when it comes to model combination. The used color codes are the following: red means direct and yellow means the other one. It can be seen that the only difference between them is the place of the ensemble service or subsystem. When we do not have an aggregator service, the client has to make ensembles, but if we have one, it can be done on server side.

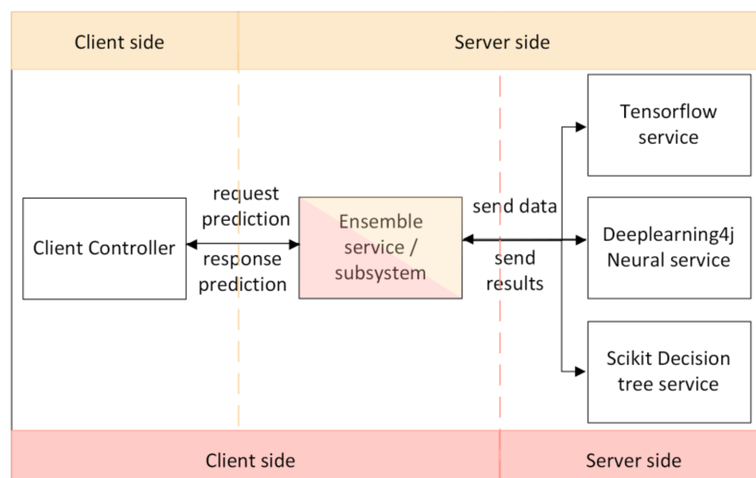

Figure 3: Model ensembling process comparison in the two variants

5.1 Direct variant

As there is no API gateway or aggregator service, this variant lacks the ability to combine models on server side. The client can do this work with an ensemble service or subsystem. In practice it looks like the client needs a prediction that should come from multiple trained models on the server side. The first step is to use the appropriate ensemble technique, so it could choose a bagging, boosting or a simple voting. When we go for voting, the client side sends HTTP requests to the server's model endpoints to get their predictions. For example, it sends data to TensorFlow, decision tree, neural network and a random forest model, that are written in Python, Java, Python with Scikit-learn, and the random forest comes from Azure service. Each of them sends back their results to the client as an HTTP response, which can use the voting technique on them. The direct variant's model combination process can be seen in Figure 3.

5.2 Gateway variant

This variant puts the entire computing load on server side, and thus the combination of models too. As we have an aggregator service, it can use other model microservices' results in an ensemble model, which can be queried from the client side. When we have different trained models behind services, like a Deeplearning4J and Scikit neural network model, they can be part of the microservice architecture-based application, and their results are accessible from the aggregator or ensemble service. The usual use case looks like the client needs a prediction and for that, it would use the mentioned Deeplearning4J and Scikit neural network models. To get their results, it sends an HTTP request to the aggregator service that forwards this request to the model services. The aggregator creates an ensemble model from the responses and then it can respond back to the client with the results. The gateway variant's model combination process can be seen in Figure 3.

6 Integration as a subsystem

The architectures and concepts above focus on applications that mainly train and manage machine learning models, but it is clear that most of the applications use ML as a subsystem or as a service. To meet this need, the variants detailed can act as a subsystem, so they can be integrated into existing software to improve them with the methods of machine learning. In the integration process, we can distinguish three elements: the clients, which are probably user interfaces, the backend and the ML variant.

The integration of the direct variant starts with connecting the frontends with the backend software. There is nothing new here, the web or mobile clients query data from the backend and then display them. The connection of ML services to the backend is unique in each application, but it is common that we need to connect backend classes to model endpoints without aggregator service. This variant is suggested when the main software is smaller or it would use few machines learning services, because the backend system has to maintain connections with multiple endpoints, and the high number of ML usage would negatively affect software maintainability. There are many advantages of the direct variant in the terms of integration. The backend software can minimize network usage by querying only the needed models. Extension can be achieved easily as developers can implement new algorithms and introduce them as new endpoints. Backward compatibility can be kept as a microservice can have multiple endpoints for the same data. Without an API gateway there is no service that processes most of the data and this could result in a lower risk of critical errors and system shutdowns, because if one endpoint is faulty, only some of the services would become unavailable. The main disadvantage is the difficulties of integration, because it requires much programming work on the backend side. An implementation can be seen in Figure 4.

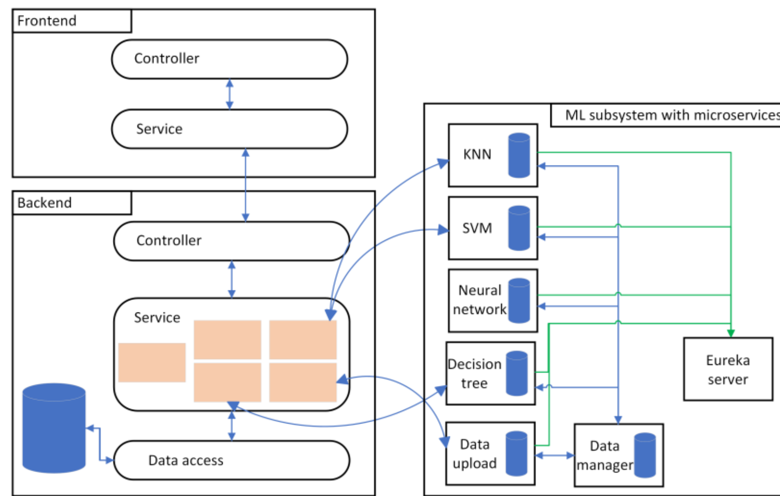

Figure 4: Integrating direct variant into a monolithic application.

The integration of the gateway variant is similar to the other; the only difference is in the connection between the backend and the ML subsystem. As there is an API gateway, multiple backend service is connected to it, so they can query the results of machine learning services through it. This can be seen in Figure 5.

We suggest the use of this variant when the main software would heavily depend on machine learning services as integration and maintenance is much easier than the direct variant, although they share some advantages. For example, both of them are easily extendible and backward compatible. The main risk here is the huge responsibility of the API gateway, because if something goes wrong there, all of the machine learning services would become unavailable.

The developer team can decide which variant is more suitable for their application by the following criteria. How many models would the software use? The more models needed, the more likely the gateway variant is needed. When the application only uses like 2-3 models, it can be faster to use the direct variant as it leaves out the gateway's request/response. How many computing resources the client has? When the clients are simple computers or mobile devices, their computing capacity is limited, so it is suggested to use the gateway variant and put the computing load on the server. Who should store the data? If we do not want to store data on client devices, then the gateway variant is more suitable. How much data do we have? Only smaller datasets or dataset slices can be stored on client devices, so direct variant is suggested when we have this kind of data. How many requests will the clients send? Applications can be heavily dependent on services, so it is possible, that for example a mobile application sends multiple requests per second. The number of request criteria should be interpreted with the number of models. When we need a few models and a lot of requests, direct variant is preferred, but if we have a few requests too, both variant can be suitable.

The protection of data privacy depends on the implementation of these architecture variants, but this can limit the functionality of them. In the direct variant, ensemble model building is often done on the client side, and because of that, some data must be present on client

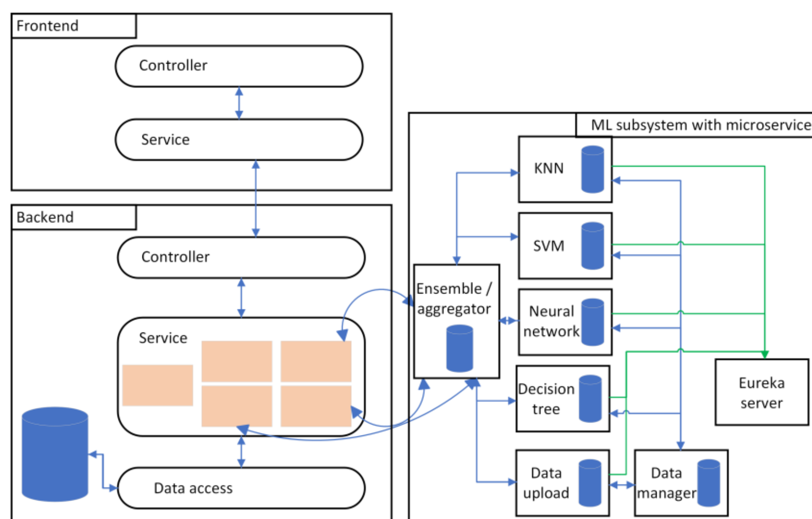

Figure 5: Integrating gateway variant into a monolithic application using aggregator service.

devices. To avoid this, the client-side tasks should be moved to the server side, and then only the models' result can be sent. The gateway variant can care about data privacy by default, as it can only be accessed through the gateway, and the implementation can restrict what kind of data can go out from the server side and who can access it.

6.1 Reference implementation

We made applications for each variant to experience the work with these proposed architectures. Both variants used the same environment, an Intel Core i7-7700HQ processor with 16GB RAM and Windows 10 operating system. The service management software was a Java and Spring-Boot project on a Tomcat application server and the machine learning part uses Python, Scikit-learn and Flask. In both implementations, data upload and data management endpoints were part of the Spring application. Although there are exchange formats we could use for sending machine learning models, like ONNX [26], we chose not to use it, because with it, we would have to use a supported framework and model to make our machine learning solution interoperable. We thought that an architecture should not have restrictions like that.

As the field of use is different for each variant, their performance should be measured differently, because competing them with the same parameters can be misleading. To prove that the direct access can be somewhat faster, we made a performance test on both direct and gateway variants. This shows that how many requests the endpoint can serve within 1 minute, what is the average response time and the error rate. The direct variant could handle 1085 requests from 20 virtual users with an average of 13.71 requests per second. The average response time is 17 ms, the minimum is 9 ms, and the maximum is 317 ms. The 90% of the requests hit the endpoint within 24 ms. The error rate was 0. As the gateway variant means one extra HTTP request/response pair for each request, its results are indeed a bit worse. From 20 virtual users, it could serve 1004 requests with an average of 12.96 requests per second. The average response time is 51 ms, the minimum is 18 ms, and the maximum is 1221 ms. The 90% of the requests hit the endpoint within 121 ms. The error rate was 0. These results are predictable as one extra hop; one extra JSON packaging will always have some cost. Although the results of the gateway variant are worse in this scenario, it does not mean that it is less usable in certain environments.

7 Possible improvements and conclusion

Training machine learning models and making them more accessible to another system with microservice architecture can be very useful in development and this paper presented a unique way of connecting bigger software systems. Making machine learning models portable can be difficult, but hiding them behind scalable web services is easier to implement for many developers, and that is why we avoided the use of ONNX [26] and other formats.

These architectures have many possibilities for future work, like comparing them in real applications, examining the scalability, integrating distributed model training or combining them with different kinds of software architectures. The microservices let us extend these web services to perform model or data parallel distributed training, for example training a neural network, where each perceptron is a microservice.

Acknowledgement

The work is supported by the EFOP-3.6.1-16-2016-00022 project. The project is co-financed by the European Union and the European Social Fund.

References

[1] (2022) Google trends. [Online]. Available: https://trends.google.com/trends/

[2] (2022) The state of developer ecosystem in 2021 infographic. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2021/

[3] Dietterich, T.G. (2000). Ensemble Methods in Machine Learning. In: Multiple Classifier Systems. MCS 2000. Lecture Notes in Computer Science, vol 1857. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45014-9_1

[4] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, Yuan Gao, A survey on federated learning, Knowledge-Based Systems, Volume 216, 2021, 106775, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2021.106775.

[5] M.-O. Pahl and M. Loipfinger, "Machine learning as a reusable microservice," in NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, Taipei, Taiwan, Apr. 2018, pp. 1–7. https://doi.org/10.1109/noms.2018.8406165

[6] Y.-D. Bromberg and L. Gitzinger, "Droidautoml: A microservice architecture to automate the evaluation of android machine learning detection systems," in IFIP International Conference on Distributed Applications and Interoperable Systems, Valletta, Malta, Jun. 2020, pp. 148–165. https://doi.org/10.1007/978-3-030-50323-9_10

[7] J. L. Ribeiro, M. Figueredo, A. Araujo, N. Cacho, and F. Lopes, "A microservice based architecture topology for machine learning deployment," in 2019 IEEE International Smart Cities Conference (ISC2), Casablanca, Morocco, Oct. 2019, pp. 426–431. https://doi.org/10.1109/isc246665.2019.9071708

[8] D. C. Attota, V. Mothukuri, R. M. Parizi, and S. Pouriyeh, "An ensemble multi-view federated learning intrusion detection for iot," IEEE Access, vol. 9, pp. 117 734–117 745, Aug. 2021. https://doi.org/10.1109/access.2021.3107337

[9] P. Salza, E. Hemberg, F. Ferrucci, and U.-M. O'Reilly, "ccube: a cloud microservices architecture for evolutionary machine learning classification," in

Proceedings of the Genetic and Evolutionary Computation Conference Companion, Berlin, Germany, Jul. 2017, pp. 137–138. https://doi.org/10.1145/3067695.3076089

[10] J. Gruendner, T. Schwachhofer, P. Sippl, N. Wolf, M. Erpenbeck et al., "Ketos: Clinical decision support and machine learning as a service–a training and deployment platform based on docker, omop-cdm, and fhir web services," PloS one, vol. 14, no. 10, p. e0223010, Oct. 2019. https://doi.org/10.1371/journal.pone.0223010

[11] M. Chippa, A. Priyadarshini, and R. Mohanty, "Application of machine learning techniques to classify web services," in 2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS), Virudhunagar, India, Apr. 2019, pp. 1–7. https://doi.org/10.1109/incos45849.2019.8951339

[12] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, Dec. 2017, pp. 2433–2441. https://doi.org/10.1109/bigdata.2017.8258201

[13] H. Chang, M. Kodialam, T. Lakshman, and S. Mukherjee, "Microservice fingerprinting and classification using machine learning," in 2019 IEEE 27th International Conference on Network Protocols (ICNP), Chicago, IL, USA, Oct. 2019, pp. 1–11. https://doi.org/10.1109/icnp.2019.8888077

[14] H. Harms, C. Rogowski, and L. L. Iacono, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, Sep. 2017, pp. 902–907. https://doi.org/10.1145/3106237.3117775

[15] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. 2017. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In Proceedings of the XP2017 Scientific Workshops (XP '17). Association for Computing Machinery, New York, NY, USA, Article 23, 1–5. https://doi.org/10.1145/3120459.3120483

[16] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. ACM Comput. Surv. 53, 2, Article 30 (March 2021), 33 pages. https://doi.org/10.1145/3377454

[17] J. A. Valdivia, A. Lora-Gonzalez, X. Lim ´ on, K. Cortes-Verdin, and ´ J. O. Ocharan-Hern ´ andez, "Patterns related to microservice architecture: ´ a multivocal literature review," Programming and Computer Software, vol. 46, no. 8, pp. 594–608, Dec. 2020. https://doi.org/10.1134/s0361768820080253

[18] H. Chawla and H. Kathuria, "Implementing microservices," in Building Microservices Applications on Microsoft Azure. Berkeley, CA, USA: Springer, 2019, pp. 21–41. https://doi.org/10.1007/978-1-4842-4828-7_2

[19] K. S. P. Reddy, Beginning Spring Boot 2: Applications and microservices with the Spring framework. Berkeley, CA, USA: Apress, 2017. https://doi.org/10.1007/978-1-4842-2931-6

[20] F. Eibe, M. A. Hall, and I. H. Witten, "The weka workbench. online appendix for data mining: practical machine learning tools and techniques," in Morgan Kaufmann. Amsterdam, Netherlands: Elsevier, 2016. https://doi.org/10.1016/C2015-0-02071-8

[21] D. Team et al., "Deeplearning4j: Open-source distributed deep learning for the jvm," Apache Software Foundation License, vol. 2, no. 82, 2016.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel et al., "Scikit-learn: Machine learning in python," the Journal of machine Learning research, vol. 12, pp. 2825–2830, Oct. 2011. https://doi.org/10.48550/arXiv.1201.0490

[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean et al., "Tensorflow: A system for large-scale machine learning," in 12th USENIX symposium on operating systems design and implementation (OSDI 16), Savannah, GA, USA, Nov. 2016, pp. 265–283. https://doi.org/10.48550/arXiv.1605.08695

[24] M. Grinberg, Flask web development: developing web applications with python. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018.

[25] Q. Zhang, H. Chu, M. Li, and X. Hu, "A unified api gateway for high availability clusters," in Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC), Shenyang, China, Dec. 2013, pp. 2321–2325. https://doi.org/10.1109/mec.2013.6885428

[26] J. Bai, F. Lu, K. Zhang et al. (2019) Onnx: Open neural network exchange. [Online]. Available: https://github.com/onnx/onnx