POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Network Slice Reuse in 5G Network: A Machine Learning Approach

Relatori
prof.ssa Chiasserini Carla Fabiana
prof. Casetti Claudio

Studente
Silvio Marcato
matricola: 258037

Anno accademico 2020-2021

Abstract

The rise of 5G networks has been fundamental in these last decades making it possible to handle users demand in a sustainable way. The concept of a flexible and standard architecture brought by the 5G-TRANSFORMER project and later by the 5Growth project has revolutionized the approach in 5G networks. The *network slicing* approach is the key factor of this thesis, it enables a logical and physical separation of network and computation resources, with the possibility of sharing and scaling them. Further improvements can be done in terms of optimization of resources which is a challenging task when handling a wide range of services with different requirements.

Our work is focused on automating the resources optimization using a machine learning model which returns to the system the best configuration parameters for the analyzed context, enabling a proper resource utilization. In order to achieve this goal, we need to define and create a dataset containing information relative to different network contexts which are created through extensive simulations of service instantiation requests. Each simulation considers several set of configuration parameters decided in advance.

Once the dataset is obtained, a machine learning model can be trained to classify correctly most contexts. The model is then given to the system through a new architectural layer defined by 5Growth, the AI/ML platform, responsible for handling model request and creation with respect to the data monitored and the vertical requirements. The final step of our work is to implement the model in the 5Growth architecture with different services as the context requires, simulating instantiation requests using the correct parameters.

Acknowledgements

Before presenting the thesis, I would like to thank my supervisor, prof. Chiasserini Carla Fabiana, and the PhD student who followed me, Avino Giuseppe, for all the support you gave me in these months. Despite the ongoing pandemic, I received a lot of support from both of you and we managed to progress in this work thanks to your knowledge and help.

Thanks to my family and my friends, who supported me during these years in every manner between highs and lows, especially in these last months. If it were not for them, I would not be here right now to achieve this significant goal of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years the network industry went through major development. Wireless service providers have deployed both the Third Generation Partnership Project (3GPP) Long Term Evolution (LTE) and 4G LTE with the objectives of increasing peak data rates both in download and upload and providing a scalable and flexible infrastructure. Nevertheless, the user demand is increasing massively, generating an even greater amount of mobile data traffic, challenging the sustainability of the existent network. According to a study conducted by the International Telecommunication Union (ITU), the global mobile traffic per month is estimated to grow from the actual 57 EB per month to 5 ZB in 2030 [1].

Several projects have been made in the past few years to face this challenge, such as the 5G-TRANSFORMER, an European project focused on the creation of a virtualized layered infrastructure which enables custom service implementation, aggregating and federalizing network and transport functions. It allowed a simpler implementation of various use cases, hiding the complexity to the verticals leveraging different paradigms, such as:

- Network Function Virtualization (NFV), which allows to detach network functions from the hardware, splitting the development time between software and hardware;

- Network Slicing, which enables the isolation of different network functions in order to save resources;

- Multi-access Edge Computing (MEC), which brings the network environment closer to the user reducing latency.

Further innovations have been implemented on this new framework. Thanks to the 5Growth project, prosecution of the 5G-TRANSFORMER one, several architectural enhancements have been done in order to increase performance and security. In this thesis we will focus on the AI/ML platform, a new module introduced in this project which can run several machine learning algorithms using data collected by a monitoring platform, another innovation brought by 5Growth. This workflow can be useful to automate decision-making processes when instantiating vertical slices. Our goal is to create a machine learning model capable of autonomous decision about vertical slices instantiation and reuse, optimizing resources utilization.

1.1 Core elements for 5G networks

5G networks are focused on optimizing the current mobile network in terms of reliability, power consumption, latency, throughput and coverage. Optimized performances allow the deployment of a wide range of vertical services with highly diverse requirements. The crucial technologies are presented in this section.

1.1.1 Network slicing

Network slices are defined by the Next Generation Mobile Network Alliance (NGMN) as a set of network functions that can be logically and/or physically isolated from other slice instances, providing isolation and increased statistical multiplexing. In this way, each network slice represents an independent and virtualized end-to-end network allowing different type of deployment flavors. Moreover, its policies and configuration can be defined in specific descriptors.

Services are deployed as sets of Virtual Network Functions (VNF) which are logically connected as described in the VNF Forwarding Graph (VNFFG). Such decomposition of services in function blocks could enable the sharing of same functions among different slices. In this way the VNF enables different features as migration from one hardware to another, the possibility of scale the computation power to adapt to different situation, high optimization and reliability.

1.1.2 NFV Management and Orchestration

NFV introduces new capabilities to the communications network, requiring a set of functions in order to manage and orchestrate operations, administration, maintenance and provisioning. The decoupling of software and hardware adds new set of relationship and objects which require different management and orchestration functions with respect to the legacy system. The Network Functions Virtualization Management and Orchestration (NFV-MANO) architectural framework has the role to orchestrate the NFV Infrastructure and to manage resources for Network Services and VNFs. It lends itself to different possibilities such as full automation, scaling and distribution of resources, implementations without single points of failures and standard interfaces.

1.1.3 Multi-access Edge Computing

MEC provides cloud-computing capabilities in proximity to mobile subscribers, at the edge of the mobile network. The environment of MEC allows low latency, proximity, high bandwidth and real-time insight into radio network information and location awareness. It enables an efficient and seamless integration of application, through acquirement of contextual information and real-time awareness of the environment using new standardized APIs. The above characteristics are shared with 5G networks based on 3GPP 5G specifications which define enablers for edge computing, allowing interaction between the aforementioned 5G networks with MEC systems in terms of traffic routing and policy control.

In short, deploying service at the edge of the networks brings more efficiency, easing the traffic load of mobile core networks, despite posing some challenges such as resource management, security and privacy.

1.2 5Growth architectural innovations

The goal of the 5Growth project is to enhance usability, flexibility, automation, performance and security of the pre-existent platform. This is done through a series of architecture, algorithm and framework innovations. The combination of the architectural ones relative to vertical service monitoring and AI/ML support is the key factor of our work.

1.2.1 Vertical service monitoring

The implementation of a monitoring platform enables features as scaling, self-healing and can trigger other tasks. It collects, store and process information provided by deployed services. Such monitoring metrics may be useful for other data processing modules, e.g., AIML modules which can build models around this data. Moreover, it adds more agile metric collection, based on dynamic alerts triggered by verticals without any downtime using messaging interaction capability between the submodules of the monitoring platform.

1.2.2 AI/ML support

Machine Learning can be quite useful when it is necessary to take decisions in real time in an highly dynamic environment such as 5G networks. Vertical applications should rely on a robust orchestration and arbitration system that is able to react properly to unexpected events that may occur.

AI/ML algorithms can adapt easily to different patterns or events that may occur within network slices, predicting future situations and problems, guaranteeing the compliance with SLA requirements imposed by verticals. This is possible due to the data made available by the monitoring platform which offers proper train and test datasets available for machine learning algorithms to use.

1.3 Outline of the thesis

The thesis is organized as follows:

Chapter 2 focuses on the 5Growth architecture, the platform on which our model will be deployed. A detailed view of the layers is presented in this chapter as well as the innovations useful for our work which were brought into the platform.

Chapter 3 introduces the concept of network slice reuse automation, the core of our work, and how it can affect the instantiation of VNFs in the system. The definition of the algorithm is presented in this section and refined for our implementation.

Chapter 4 is focused on the *service instance requests simulator* used to generate data suitable for the dataset which will be used to train the model. A detailed view of its workflow and the considered dataset parameters is discussed in this section. The results are reported at the end of the chapter.

Chapter 5 is centered on the Machine Learning paradigm, introducing the different method of classification and selecting the best one based on accuracy results. Results will be shown at the end of this chapter.

Chapter 2

The 5Growth architecture

The 5G networks are envisioned to deploy a wide range of different services, each one with its requirements in terms of latency and resources, spanning from Industry 4.0 to Transportation and Energy. It is necessary to build a new infrastructure which can cope with the aforementioned requirements in order to meet them in a flexible way. To this aim, the 5Growth project [8] provides an infrastructure that supports the current connectivity needs and facilitates the implementation of new innovative digital use cases. The infrastructure is built on the baseline platform, 5G-TRANSFORMER [5], which focuses on designing and building a system fit for purpose which exploits the previously mentioned key pillars for 5G: network slicing and MEC. The former impacts the cost reduction side, introducing sharing among different actors, while the latter enables low-latency services reducing the traffic pushed into the core networks.

The architecture [6] is conceived to support multiple combinations of stakeholders. In order to achieve this purpose, through the new standardized set of Application Programming Interfaces (APIs), the system hides the complexity from the verticals, which are allowed to define their services and the required Service Level Agreements (SLAs).

The system consists in three different components:

- **Vertical Slicer** (5Gr-VS), inherited from 5G-TRANSFORMER Vertical Slicer (5GT-VS), is the entry point for service requests from the verticals, it manages their association of the services to slices as well as network slice management. In the 5Growth model it is expanded to support monitoring, security, and performance assurance for each network slice enabled by machine-learning and analytics. It provides further control and additional support towards multi-domain services.

- **Service Orchestrator** (5Gr-SO), inherited from 5G-TRANSFORMER Service Orchestrator (5GT-SO), is responsible for orchestration of services across different domains, organizing resources to expose to the 5GT-VS. It is enhanced by recent algorithms and frameworks which enable smart orchestration and lifecycle control over slices, optimizing resources for Radio Access Networks (RANs), Transport, Core and cloud/edge computing.

- **Resource Layer** (5Gr-RL), inherited from 5G-TRANSFORMER Mobile Transport and Computing Platform (5G-MTP), provides physical and virtual network resources

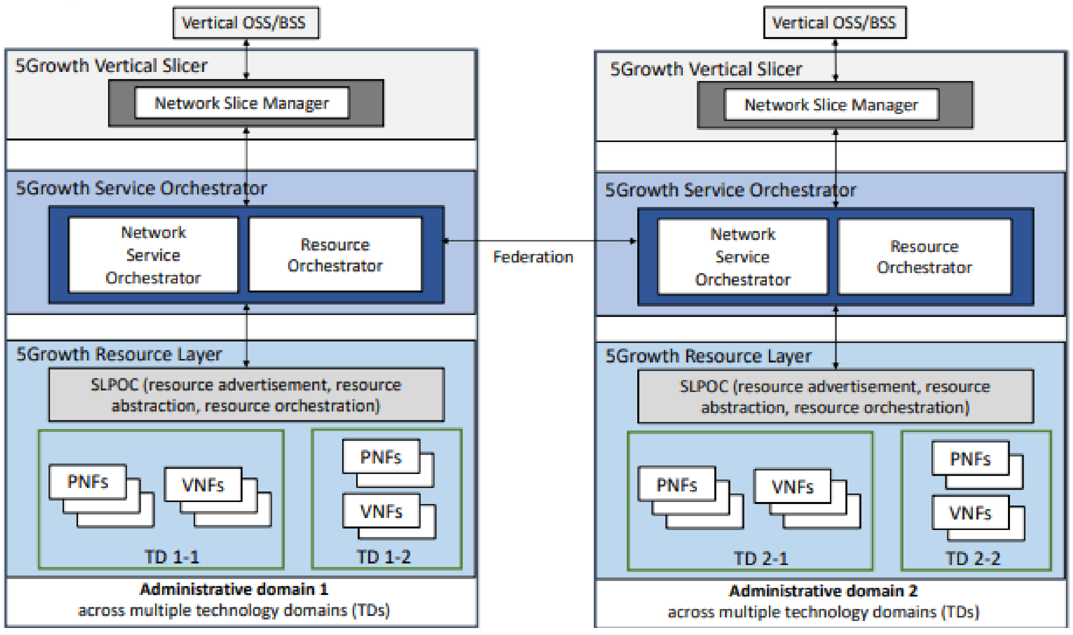

Vertical OSS/BSS
5Growth Vertical Slicer
Network Slice Manager
5Growth Service Orchestrator
Network Service Orchestrator
Resource Orchestrator
Federation
5Growth Resource Layer
SLPOC (resource advertisement, resource abstraction, resource orchestration)
PNFs
VNFs
TD 1-1
TD 1-2
Administrative domain 1
across multiple technology domains (TDs)
Vertical OSS/BSS
5Growth Vertical Slicer
Network Slice Manager
5Growth Service Orchestrator
Network Service Orchestrator
Resource Orchestrator
5Growth Resource Layer
SLPOC (resource advertisement, resource abstraction, resource orchestration)
PNFs
VNFs
TD 2-1
TD 2-2
Administrative domain 2
across multiple technology domains (TDs)

Figure 2.1. 5Growth architecture

for service deployment and network slice execution. It is responsible for the level of abstraction of the 5GT-SO, improved with re-programmable mechanism in order to enhance security, management and control over individual resources.

These three main components are further described below along with a machine learning related component added by the 5Growth project, a crucial factor for the work done in this thesis.

2.0.1 Vertical Slicer

The 5GT-VS is the entry point for verticals into the 5G-TRANSFORMER system. Vertical services are offered through a high-level interface, allowing providers to focus on the requirements and logic side of their services. The layer offers a catalogue of virtual service blueprints (VSB) which are complaint with the SLA requirements and are used by the verticals to generate their service requests. The final specification of a service is expressed through a Vertical Service Descriptor (VSD) which is made of both the correspondent VSB and the user-defined parameters used to fill the VSB. The 5GT-VS identifies what kind of network slice is required for the vertical service deployment and manages their lifetimes.

The identification of a proper network slice is done by the Vertical Service Management Function (VSMF), which is also responsible for multiple service arbitration. It consists in deciding if multiple service could be mapped on the same network slice, following vertical's SLA in terms of isolation and sharing, taking into account the computational resources needed when sharing the slice.

The management and configuration of the Vertical Slicer system allows to configure tenants, their SLAs and the associated policies. Moreover, VS blueprint can be on-boarded in the internal catalogue. These functionalities are implemented through the Tenant management, the SLA & policy management and the VS Blueprint catalog modules and exposed to the system administrator through management REST APIs.

Lifecycle management of one or multiple network slices, arranged in a hierarchy manner, is handled by the Network Slice Management Function (NSMF) and the Network Slice Subnet Management Function (NSSMF). These components are responsible for the creation, modification and termination of network slices, following the vertical management logic.

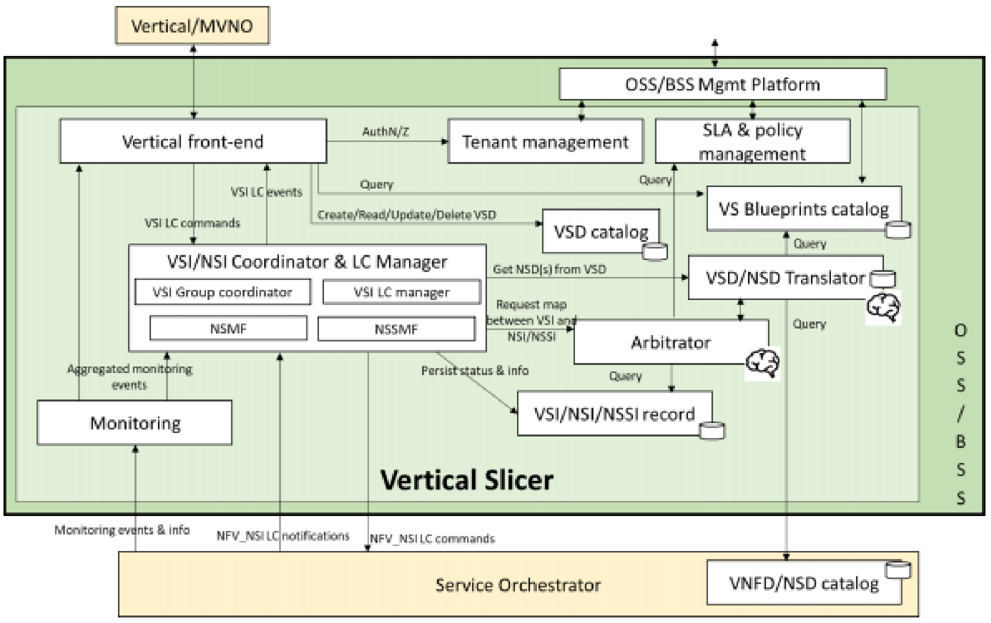


Figure 2.2. 5G-TRANSFORMER Vertical Slicer architecture

The VSI/NSI Coordinator and LC Manager is the core component of the Vertical Slicer. It maps the requests into an existing or a new NFV Network Service (NFV-NS) implementing a network slice which can be shared among different vertical services. It can be made up of network slice subnets that may offer partial required functionality reducing the total load. The NFV-NS will be updated or created through a Network Service Descriptor (NSD), a graph composed by the set of VNF necessary for the proper functioning of the service, along with several instantiation parameters that are sent to the 5GT-SO.

The Vertical Slicer algorithm and decision logic are handled by the VSD/NSD Translator and the Arbitrator. The VSD/NSD Translator maps the vertical services and the network slices using the descriptor of the associated NFV network services. Descriptors enable the identification of several service requirements, such as the number of VNFs required, the amount of requested resources or the bandwidth between virtual links. The Arbitrator, on the other hand, manages contention among concurrent services and it is responsible for the creation, sharing and destruction of network slices with respect to parameters such as sharing requirements and placement.

2.0.2 Service Orchestrator

The 5GT-SO is responsible of providing end-to-end orchestration of the NFV-NS across multiple domains by interacting with the local MTP and peer SOs. It handles the allocation of resources which can be offered by other administrative domains, hiding it to the vertical domain which access to the system through a single 5GT-VS.

This layer has two main components, the network service orchestrator (NFVO-NSO) and the resource orchestrator (NFVO-RO). The NFVO-NSO is responsible of the lifecycle management of the network services, it handles the hierarchy of different NFVO-NSOs if the service is deployed across different domains. The NFVO-RO assign virtual resources which are offered by the local and underlying 5GT-MTP and from peer SOs. The resources information is contained in the Network Functions Virtualization Infrastructure (NFVI) resources repository and the NFV-NS instances repository. Resources will have a higher or lower abstraction level depending on the policies of the MTP and the peer SOs. Ultimately, the 5GT-SO include Virtual Network Function Managers (VNFM) which manage the lifecycle of the VNFs composing the NFV-NS.

In the end, the 5GT-SO main functionalities are the following:

- decides the optimal (de)composition for the whole NFV-NS subject to resources availability offered by the local 5GT-MTP and other domains;
- manages the lifecycle for the NFV-NS and the composing VNFs;
- decides the optimal placement for the VNFs and the deployment of the virtual links between VNFs through optimized mapping;
- requests to other federated 5GT-SOs required network services in order to execute parts of the NFV-NS in other administrative domains;
- performs monitoring tasks and handles fluctuations in the system, triggering self-adapting actions in order to maintain an optimal performance.

The main components of the 5GT-SO are the following:

- **NBI Exposure Layer**: it offers an API towards the 5GT-VS to support requests for service scaling, instantiation, modification and termination;
- **NFV-NS/VNF/VA Catalogue DB/Manager**: repository of all NFV Network Service (NFV-NS), Virtual Network Function (VNF) and Vertical Application (VA) descriptors, used by the 5GT-SO in order to obtain relevant information about needed resources and service lifecycle;
- **Multi-Domain NFV Orchestrator (NFVO)**: it orchestrates virtual resources across different domains, executing NFVO-RO functions as well as coordinating the NFVO-NSO functions;

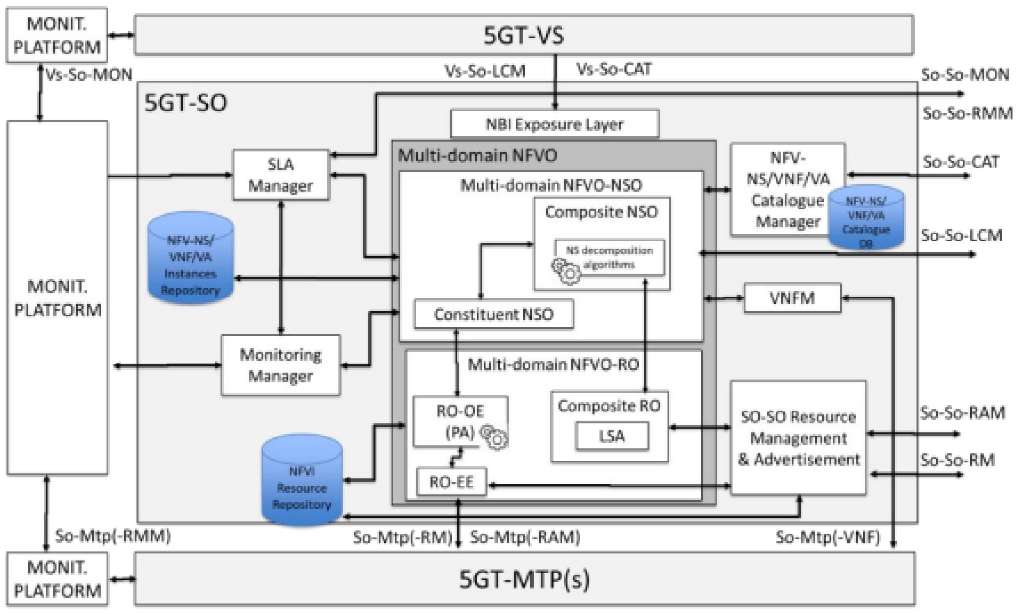


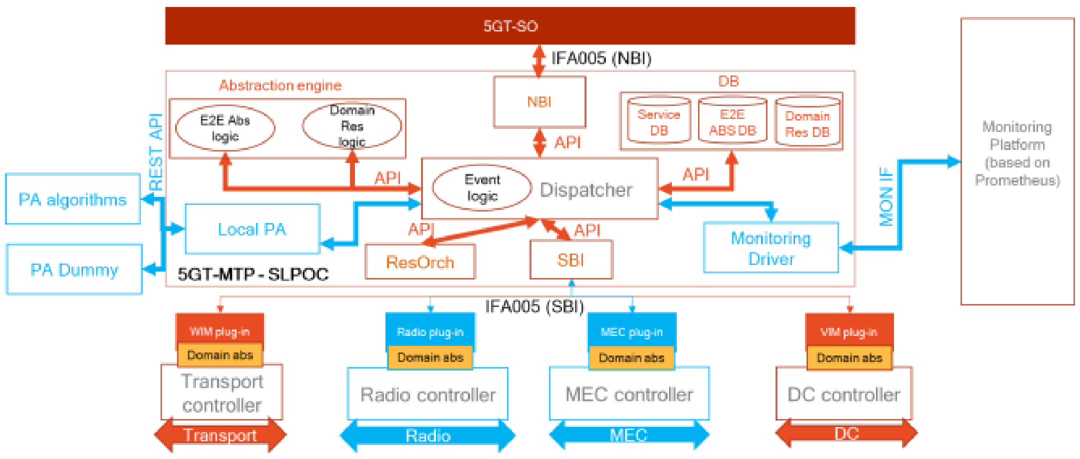
Figure 2.3. 5G-TRANSFORMER Service Orchestrator architecture

- **VNF Manager (VNFM)**: it handles the VNFs lifecycles, provides reconfiguration based on information received from the local NFVO;
- **SO-SO Resource Management & Advertisement**: it is in charge of creating abstract views of the resources with other administrative domains;
- **NFVI Resource Repository**: consolidates the abstract view of the resources received by the underlying 5GT-MTP;
- **NS/VNF/VA Instance Repository**: it contains all the NFV-NS, VNFs and VAs instantiated over time;
- **SO Monitoring Manager**: it is responsible to translating the service requirements into low-level jobs in order to retrieve the necessary information from the virtual infrastructure elements;
- **SLA Manager**: it ensures that the SLA requirements imposed by the verticals are satisified through on-line SLA verification, triggering adapting reactions if a constraint is not respected.

2.0.3 Resource Layer

The 5Growth Resource Layer (5GR-RL) inherits its functions from 5G-TRANSFORMER Mobile Transport and computing Platform (5GT-MTP). It handles the mobile transport network, computing and storage infrastructure. It offers different levels of resource abstraction to the 5GT-SO and it provides solutions to integrate MEC resources from multiple domains in terms of computing and storage resources. Therefore, when receiving a resource allocation request, it generates requests for the relative entities providing part of the virtual resources needed for the VNFs, it interconnects them and configures additional parameters. The requests may be mapped in order to hide the complexity of the underlying mobile network while maintaining flexibility inside the mobile domain.

Figure 2.4. 5G-TRANSFORMER MTP architecture



The MTP is structured as follows:

- **Abstraction Engine**: it generates the abstract view of the available resources to provide to the Service Orchestrator;
- **Database (DB)**: it contains all the information relative to the domain resources. It can be updated through an external SQL server which handles all the queries;
- **Dispatcher**: it manages the communication between the 5GT-MTP components;
- **Local Placement Algorithm (PA)**: it handles the placement of the VNF, selecting the optimal placement and achieving the ideal resource utilization;
- **Monitoring Driver**: it manages communications between the MTP and the Monitoring Platform, creating performance monitoring jobs.

2.1 AI/ML integration

5G networks has opened the doors to a great amount of different use cases with their quite diverse requirements in terms of resources and latency. It is therefore acknowledged how important is the automation of network and service management. This led to the introduction of new features in the 3GPP specifications as the Network Data Analysis

Function (NWDAF) which enables data collection and analysis in network functions, and in the Open Radio Access Network (O-RAN) specifications with regard to radio and network performance improvement through AI/ML.

The 5Growth project [8] is focused on building the 5Gr-AIML platform [7] to achieve such automation, creating a centralized and optimized environment to train, store and serve AI/ML models in the 5Growth infrastructure, which can be integrated in the 5G-TRANSFORMER architecture. Two entities interact together with the platform: the 5Growth Vertical-Oriented Monitoring System (5Gr-VoMS) and a generic entity requiring a trained model (one of the components of the 5G-TRANSFORMER system in our case). The 5Gr-VoMS provides raw data to the 5Gr-AIML platform which collects them as training datasets to be used to train different models. The 5Gr-entity interacts with the platform through the Interface Manager in order to request fresh trained AI/ML models.

The Interface Manager handles the traffic inside the platform. 5Gr entities interact with it making their requests for trained models satisfying the requirements. The request is sent to the computing cluster, based on Apache Hadoop, which returns the trained model requested when it is ready. The Hadoop cluster leverages the Hadoop Distributed File System (HDFS) as dataset and training model storage and YARN for computing resources management. There are several other components as BigDL, used for Deep Neural Networks, and Ray, needed to leverage Reinforcement Learning models, however the crucial component for our objective is Spark, used to train classic supervised and unsupervised model, useful in presence of a label dataset such as the one created in our work.

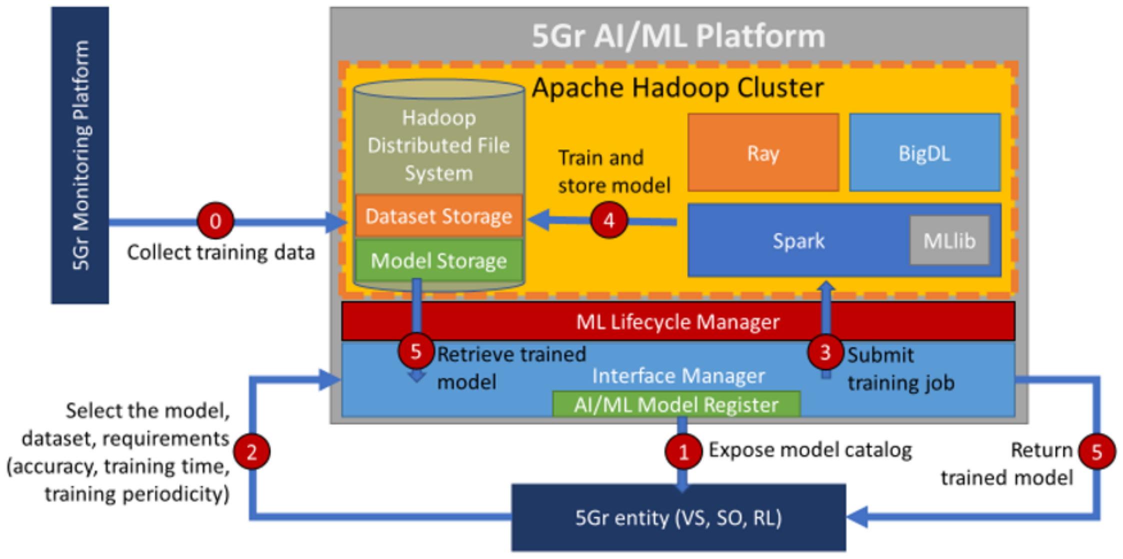


Figure 2.5. 5Gr-AIML Platform Architecture

The workflow is depicted in figure 2.5 and it is as follows:

0. training data are collected from the 5Gr-VoMS and elaborated to build or update a

training dataset saved in the HDFS Dataset Storage;

1. a 5Gr-entity requests to the 5Gr-AIMLP for a model needed to fulfill the optimization requirements (e.g., scaling, deployment, path recalculation) in order to guarantee a correct service lifecycle or to react on some detected irregularity. Therefore, the platform exposes the model catalog containing the suitable ones fit for purpose;

2. the 5Gr-entity select the model and the dataset, specifying also some other requirements such as accuracy and training time. The training periodicity could be optimized by the ML Lifecycle Manager that keeps the model fit using the last data available;

3. if the model has never been trained, its validity has expired or a preliminary training is required, a training job is sent to the computing cluster, otherwise it is directly fetched from the Model Storage;

4. the Hadoop cluster performs a training job using the proper block (Spark, BigDL, Ray) on the appropriate dataset present in the Dataset Storage. Once the training is complete, a training model is created or updated in the Model Storage, consequentially updating the Model Register;

5. the trained model is retrieved from the Model Storage and returned to the requesting 5Gr-entity which will use the model to finalize the initially requested optimization

Chapter 3

ML-driven network slice reuse

The implementation of an AI/ML powered module in the architecture brings several advantages in the 5Growth system. This innovation can simplify the development of algorithms safer and more robust than conventional ones. In fact, AI-based algorithms adapt better to the dynamic behavior of network slices and react in real time to unexpected events.

Our work is focused on the automation of network slice instantiation, done by the Arbitrator inside the Vertical Slicer. This component computes the latency class for each VNF to instantiate, considering its complexity and the number of users handled. In the light of data, it decides whether a new virtual machine must be instantiated for that VNF or if an existent instance of the same VNF could be used to serve the new incoming request.

The goal is to create a ML model which can lighten the Arbitrator code. As a matter of fact, the Arbitrator will only ask to the model which is the best bin configuration instead of calculating it by itself.

The workflow is presented in figure 3.1:

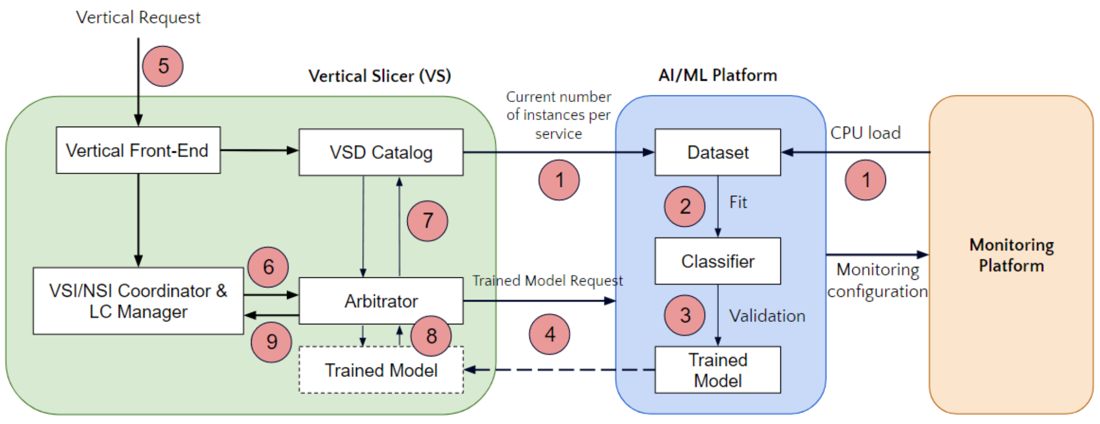


Figure 3.1. ML-driven Network Slice Reuse workflow

1. a dataset is created using information from both the monitoring platform (i.e., the amount of CPU currently used) and the VSD catalog (current number of service instances);
2. the ML model is fit with the training dataset created in the first step;
3. the ML model is validated with the preferred method (i.e., k-fold cross validation) and tested;
4. the Arbitrator requests the trained model to the AI/ML platform to use for VNFs instantiation;
5. a vertical service requests arrives to the Vertical Slicer front-end;
6. after being processed by the VSI/NSI Coordinator, the request is passed to the Arbitrator in order to decide how to instantiate the service and its VNFs;
7. the Arbitrator requests the number of instances from the instance database to give to the trained model for the inference;
8. the trained model predicts the best latency class configuration and returns it to the Arbitrator;
9. the results is sent back to the VSI/NSI coordinator which carries on the service instantiation process, contacting the Service Orchestrator.

In our case the model is trained on data obtained through simulations of service requests without considering the direct contribution of the involved platforms. We leveraged the algorithm 1 used by the Arbitrator to determine the conditions which are to be respected in order to share network slices.

3.1 Slice reuse algorithm

Assuming we have a list of service instantiation requests, the algorithm requires as input:

- the maximum computing capability associated with a single VM ($\bar{\mu}$);
- the per-VNF complexity factor θ_v, indicating the amount of vCPU required to process a traffic unit by the VNF v;
- the network latency between the service area and the service layer l on which the new service is going to run ($d_{l,r}$);
- the request r for the new service

The request r includes:

- the set V_s of the VNFs composing the requested service;
- the service target latency threshold D_s

- the service lifetime L_s
- the maximum number of users that the service instance can handle λ_r
- the service level l^*, which affects the network latency;
- the tenant of the service T

The output of the algorithm is a tuple composed of three elements:

- a service instance;
- a VNF v;
- a VM capability μ_b^* needed for the VNF v

For each VNF of the requested service, their minimum processing delay, (Line 7), target latency (Line 9) and, according to the latter, latency class (Line 11) are computed. In order to look for VMs already instantiated that can be reused for the request r, in Line 14 all the service instances ρ of each running service σ are scanned. Therefore, in Line 16, each VNF $\hat{v}$ composing the current service instance ρ is parsed. If a given $\hat{v}$ is one of the VNFs requested in r, and both their service level, latency class and tenant match, $\hat{v}$ can be shared (Line 17).

If a VNF is shared among services, it is needed to adjust the capability of the VM on which it runs. Since the VMs have a maximum capability $\bar{\mu}$, in Line 19 we check if the VM b^* (on which $\hat{v}$ runs) can be empowered. If so, Line 20 computes the new capability μ_{b^*}. Then, Line 21 adds the tuple "service instance - VNF - capability" (ρ, $\hat{v}$, μ_{b^*}) to the set O (the output of the algorithm). Finally, Line 22 removes $\hat{v}$ from the set V_r of the VNFs that has to be instantiated and, Line , checks if V_r is empty. If so, this means that we found for every VNF composing the new service a VM already running that can be reused. Consequently, the algorithm ends and returns the set O.

Finally, if at least one VNF of r has not been instantiated yet (Line 32), for each $\hat{v}$ still present in V_r: (i) the VM capability μ_{b^*} on which $\hat{v}$ will run is computed, and (ii) the output set O is updated (respectively Line 35 and 36).

Table 3.1. Notation

Value	**Description**
$\bar{\mu}$	Maximum computing capability associated with a single VM
θ_v	VNF complexity factor, equivalent to the amount of vCPU required to process a traffic unit by the VNF v
$d_{l,r}$	Network latency between the service area and the service layer l where the service is instantiated
D_s	Target latency for service s
λ_r	Expected traffic load of the service instance

The complexity factor of each VNF is redefined in a more comprehensible way in order to calculate the processing time of each one and check if the total latency respects the requirements of the service.

Algorithm 1 Sub-slice reuse

```
Require: Service instance request $r = \langle V_s, D_s, \lambda_r, l^* \rangle, \bar{\mu}, \theta_v, d_{l^*,r}$
 1: ▷ Given request $r$, initialize $V_r$ to the set of VNFs composing the corresponding service
 2: $V_r \leftarrow V_s$
 3: ▷ Define $O$ output as empty set
 4: $O \leftarrow \emptyset$
 5: for all $v \in V_r$ do
 6:     ▷ Compute the min processing delay for VNF $v$ of request $r$
 7:     $M_{s,v} = \frac{1}{\bar{\mu} - \theta_v \lambda_r}$
 8:     Compute the target latency for each $v$ composing the service requested
 9:     $D_r^v(l^*) \leftarrow \frac{M_{s,v}}{\sum_{u \in V_s^r} M_{s,u}}(D_s - d_{l^*,r})$
10:     ▷ Determine VNF latency class, given the per-VNF target latency
11:     $j_v^* \leftarrow log_{(1+\epsilon)} D_r^v(l^*)$
12: end for
13: ▷ For each service instance $\rho$ of service $\sigma$ among the running instances $\hat{R}$
14: for all $\rho \in \hat{R}$ do
15:     ▷ For each VNF $\hat{v}$ composing the current service instance $\rho$
16:     for all $\hat{v} \in V_\sigma$ do
17:         if $\hat{v} \in V_r$ and $l^\rho = l^*$ and $j_v^* = j_v^\rho$ then
18:             ▷ Adjust capability of the VM hosting the VNF $\hat{v}$ ($b^*$ denotes the VM and $\hat{r}$ is
        the generic service instance running and using $b^*$)
19:             if $\theta_v[\Lambda(b^*) + \lambda(r)] + \frac{1}{\min_{\hat{r}} D_v^{\hat{r}}(l^*)} \leq \bar{\mu}$ then
20:                 $\mu_{b^*} \leftarrow \theta_v[\Lambda(b^*) + \lambda(r)] + \frac{1}{\min_{\hat{r}} D_v^{\hat{r}}(l^*)}$
21:                 $O \leftarrow O \cup (\rho, \hat{v}, \mu_b^*)$
22:                 Remove $v$ from $V_r$
23:             end if
24:             ▷ Check if all the VNFs composing the new service are instantiated
25:             if $V_r = \emptyset$ then
26:                 break
27:             end if
28:         end if
29:     end for
30: end for
31: ▷ If at least one VNF $v$ has not been instantiated yet
32: if $V_r \neq$ then
33:     for all $v \in V_r$ do
34:         ▷ Create a VM with the correct capability
35:         $\mu_{b^*} \leftarrow \theta_v \lambda(r) + \frac{1}{D_v^r(l^*)}$
36:         $O \leftarrow O \cup (\rho, v, \mu_b^*)$
37:     end for
38: end if
```

For each VNF we define a complexity formula using the evolution of the processing time t_p with respect to the number of active users N and considering one virtual core for each

VNF. We modeled it on a 2nd grade equation:

$$t_p = a * N^2 + b * N + c, \tag{3.1}$$

where a, b, c are coefficient determined either by simulating the considered VNFs with tests or determined *a priori*.

The target delay of each service (D_s) is determined in the following way:

$$D_s = d_{l^*,r} + \sum_{i=0}^{N} t_{pi}. \tag{3.2}$$

The network delay depends on the placement of the service in the network, while the processing time is intended as the processing time of v_i and it is computing according to its formula.

When the D_s is not respected, we scale up the VNF by one virtual core, diminishing the processing time of a factor proportional to the number of cores currently used. When the maximum computation $\bar{\mu}$ is reached, that VNF is no longer considered and the system will scale out instead of scaling up.

Furthermore, the latency class computation for each VNF corresponds to the end-to-end delay of the service to which they belong. In fact, the bin in which they fall is directly dependent on this and it helps to distinguish the possible bin configurations to use in the simulations.

Chapter 4

Dataset creation

The benefits of sharing network slices discussed in the previous chapter are significant in terms of saving resources. In order to achieve the optimal and automated use of resources, a machine learning approach is a viable solution. In fact, the slice reuse issue can be summed up as a *supervised classification problem*. Bearing this in mind, what is missing is a labelled dataset to be used to train a machine learning model.

Extensive service requests simulations are required to get a proper dataset, extracting for each single context the simulation with the best bin configuration which will be taken in account by the machine learning model to determine the best one when assigning it to future data. Each simulation will be redone a number of times equals to the number of different bin configuration in order to identify correctly which one is the best for that specific situation. The parameters are presented in table 4.1 and contains *context-relative parameters*, i.e., the number of instances per service, and *bin configuration-related parameters*, values which depends on the latency classes considered for that specific context, such as the amount of vCPU consumed.

The initial phase of this simulator project, consisted in defining a list of services with different requirements, each one including its set of VNFs, some of which are potentially shareable among different services.

The dataset has to contain useful information regarding the context, a fixed representation when running simulations. As a matter of fact, the same simulation must be redone as much as the number of different bin configurations in order to identify correctly which one is the best for that specific situation. In this way we identify and calculate all the parameters we need. The *bin configuration-related parameters* are required in order to assign the best bin configuration to a specific context.

4.1 Services

The vertical services considered are all related to the automotive use cases, field of application convenient for its variety of different latency and resources requirements. The main trends in this application field are *autonomous driving* and *road safety*, *infotainment* and *traffic efficiency* services [9]. Both safety and non-safety applications are deployed in a demanding and dynamic network topology. This domain, in fact, poses different challenges:

Table 4.1. Dataset Parameters

Dataset parameters	
Value	**Description**
n_r	number of requests over the simulation
n_{rt}	number of requests of each single service type
n_i	mean number of instances in the simulation
n_{it}	mean number of instances per service type (one for each type)
n_{VM}	Total number of instantiated VMs
n_{VNF}	Total number of requested VNF instantiation
μ_{tot}	Total computational power used
B_{set}	Bin configuration for latency classes

- **Highly changing topology**: the vehicles are considered as node in the network. Therefore, the topology is constantly changing due to their high speed and movement variety.
- **High number of connected nodes**, each one with its characteristics
- **Interaction with on-board nodes**, such as automotive sensors and devices brought into the car by the users.
- **Interaction with on-street sensors**, used to provide a better knowledge about the street topology and conditions, raising the vehicle context awareness.
- **Constant reliable connectivity**: must be always available for safety services, especially when not under cellular coverage.
- **Standardization**: it is necessary to leverage a common language to exchange information among different vehicles.

In our work, we are considering three different automotive services:

- **Intersection Collision Avoidance** (ICA), a safety application with a stringent latency requirement due to its purpose, namely that of preventing accidents in real-time.
- **See-Through** (ST), another safety service, but with less tight latency requirement, it provides a front view of the vehicle ahead the one requesting the service.
- **Video Streaming** (VS), the least demanding service in terms of latency among the others, a service which provides broadcast videos.

We considered these three services for our first simulation due to their diverse requirements in order which could lead to a fair amount of context variety. The details on each service and its set of VNFs are discussed below.

4.1.1 Intersection Collision Avoidance service

The ICA [13] is a crucial safety service, it calculates the probability of imminent collision in real time thanks to the communication among vehicles which exchange valuable information about their position and speed contained in Cooperative Awareness Messages (CAM) and take decisions on these data.

The most relevant Key Performance Indicator (KPI) for this service are the following:

- high reliability and availability (99%);
- extremely low latency (<20ms);
- high security and priority;
- enhanced data rate;

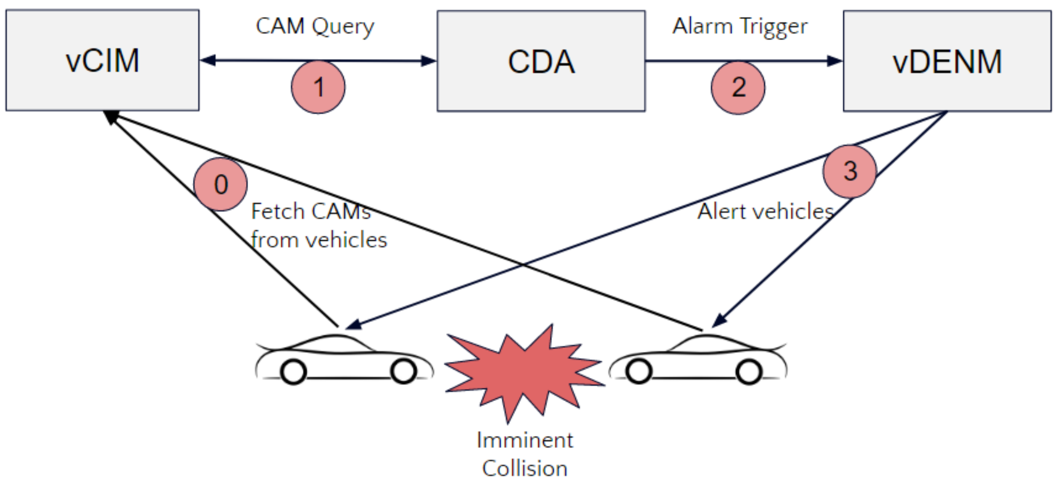


Figure 4.1. ICA service VNFs interaction

The set of VNFs are defined in and they are depicted in figure 4.1. In details, the VNFs are structured as following:

- **Collision Detection Algorithm** , the core of the ICA application, it calculates the speed and the trajectories of the interested vehicles from the respective CAMs.
- **Cooperative Information Manager** (CIM), a third-party entity that acts as a collector of CAMs gathered in the monitored area and it makes them available to the algorithm for its computation.
- **Decentralized Environmental Notification Messages** (DENMs), it is responsible of sending unicast alert messages in case of imminent collision, it is the least demanding VNF in terms of resource utilization.

A user requesting for this service triggers the collision algorithm which queries the CIM in the interested area for the latest CAMs and checks if there are risks of collisions for the requester. If there are, the VNF implementing DENMs is triggered and sends the alert message.

4.1.2 See Through service

The ST service falls in an intermediate category, being a safety service with a slacker latency requirement (<200ms). On the other side we consider it as a more complex service in terms of computing, due to the power needed for encoding and transmitting video frames. We based the set of VNFs on a simplified version of [11], brought in a MEC perception.

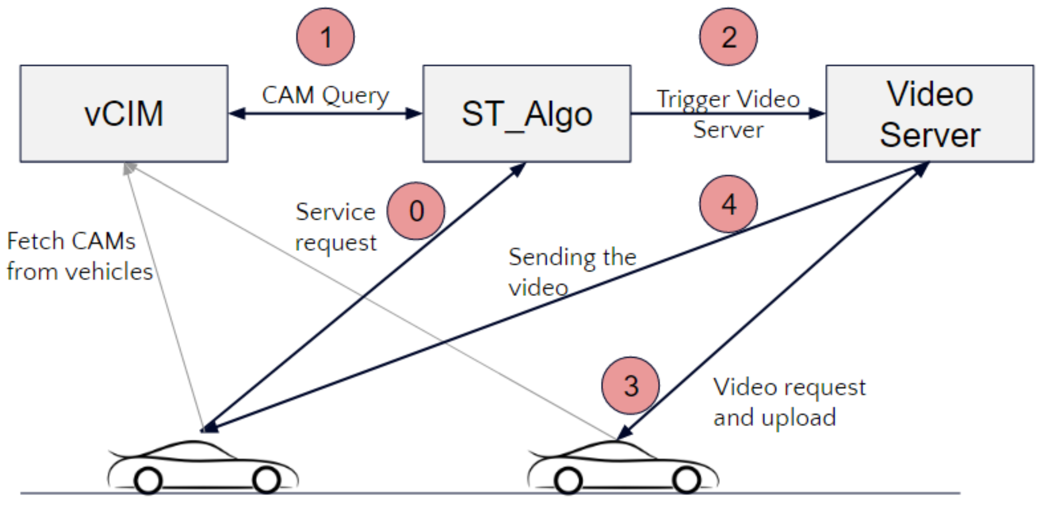


Figure 4.2. See-through service VNFs interaction

The VNFs are the following:

- **See Through Algorithm** (ST_Algo), it determines the relative position between the request sender and the vehicle ahead, as well as camera information to determine the resolution quality of the video.
- **CIM**, the same of the ICA service, queried by the ST_Algo to achieve its purpose.
- **Video Server** (VS), it provides for the encoding and decoding of the video sent from the ahead vehicle.

The vehicle (0) requests the ST_Algo VNF if it can determine if the vehicle in front of it is equipped with a camera analyzing the CAM stored in the CIM (1) previously fetched from the interested vehicles. If the camera is present, the ST_Algo VNF triggers the Video Server VNF (2) which will acquire the captured video from the preceding vehicle (3) and send it to the requester (4).

4.1.3 Video Streaming service

The Video Streaming service is a simple service which can broadcast multiple video streams in broadcast. It is composed with two VNFs:

- **Video Server** (VS), keeps in storage the media files provided to the users.
- **Video Controller** (VC), it can change the quality of the streaming during the transmission to avoid traffic congestion.

The flow of this service is rather simple. The user requests the Video Server for a video and during the transmission the Video Controller asks periodically the Radio Network Information Service (RNIS) information about the Channel Quality Indicator (CQI) which is monitored by the RNIS itself. On the basis of the CQI, the Video Controller chooses the best quality for the video transmitted. In our simulations the complexity of the VC is considered equal to the DENM function of the ICA service.

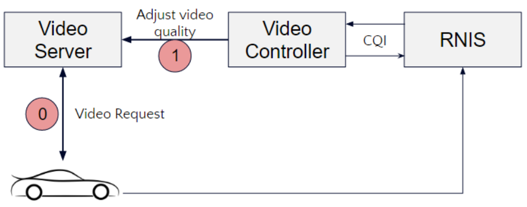


Figure 4.3. Video Streaming service VNFs interaction

4.2 Service structure in the simulator

Each service is presented as a JSON document. This form of standardization simplifies the addition of new different services to the simulator in order to obtain the right context for the specific case. The fields of each service are organized as follows:

- **service ID**, identifying the service in a unique way;
- **maximum number of users**, manageable from the service instance;
- **maximum End-to-End (e2e) delay**, for the service to work properly (in milliseconds);
- **lifetime** of the service instance in the system (in seconds);
- **placement** of the service, either in cloud or in edge;
- **tenant** that provides this configuration(?).
- **set of VNFs**, which are composed, in their turn, by their fields:
  - **VNF ID** to identify the VNF in the system;

- **vCPU**, the number of virtual core used by the VNF;
- **vRAM**, the amount of virtual RAM needed by the VNF;
- **vStorage**, the disk space necessary for the correct deployment of the VNF;
- **placement**, reflecting the service one;
- **maximum number of users**, which value is the same of its service, but it can be modified when sharing the VNF among other instances.

```
{
  "serviceId": "CA",
  "maxNumberUsers": 100,
  "maxE2e": 20,
  "lifetime": 120,
  "placement": "EDGE",
  "tenant": "POLITO",
  "VNFs": [{
    "vnfId": "vCIM",
    "vCPU": 1,
    "vRAM": 8,
    "vStorage": 10,
    "placement": "EDGE",
    "maxNumberUsers": 100
  }, {
    "vnfId": "vDENMg",
    "vCPU": 1,
    "vRAM": 2,
    "vStorage": 10,
    "placement": "EDGE",
    "maxNumberUsers": 100
  }, {
    "vnfId": "EVS",
    "vCPU": 1,
    "vRAM": 2,
    "vStorage": 10,
    "placement": "EDGE",
    "maxNumberUsers": 100
  }]
}
```

Figure 4.4. Sample descriptor of ICA service

4.3 Simulator structure and workflow

The service requests simulator is written in Python 3.8 [10]. It can easily handle different types of service according to the earlier decided structure, generating coherent results with respect to the number and characterization of the services considered in each simulation. The output of each simulation is a dataset containing several lines for each context which number depends on the bin configurations considered. This file is processed later in order to obtain a refined one to give to the model as training set.

There are several parameters considered in the simulation which can be modified freely by the user:

- rates λ_s, they define for each service what is the arrival rate, modeled on a Poisson distribution. The permutations of each rate are considered in order to increase the variety of the results;
- bin configuration B, a list of delimiters of latency classes, they can be added and deleted accordingly to the number and the requirements of the services;
- number of simulations n_s, it defines how many different simulations are considered for each permutation of the rates;
- time of the simulation t_s, chosen by the user, depending on how much time is needed to reach a stable situation;
- number of max core $\hat{\mu}$ for each VNF, kept the same for every VNF;
- network latency d_l, it depends on the service level. Due to the fact that our service are all deployed in the edge network, it is the same for every service.

The workflow of the simulation goes as follows:

1. a list of service instantiation requests is generated according to the chosen rates;
2. for each request, determine if there are some viable instances of the same service not full yet;
3. if there are, the simulator tries to add the user to the selected instance , checking if the end-to-end delay is respected for each service instance using the VNFs of the updated instance;
   (a) if for some services the end-to-end delay is not respected, the selected instance is blocked along with the not shareable VNFs;
4. if there are not, the algorithm defined in the previous chapter is applied, sharing VNFs according to the requirements and instantiating new ones when sharing is not possible;
5. as long as the instantiation is complete, a check on the current running services lifetime is performed;
   (a) if a service has finished its lifetime, it is deleted from the list along with its VNF;
   (b) if some of its VNFs are shared among other services, the number of users handled by those VNFs are updated and eventually it is scaled down by the number of virtual CPU cores needed.

4.4 Parameters chosen and simulations results

We considered a set of parameters based on a mobility study conducted in a area of 11 km^2. We simulated service instantiation requests distributed as a Poisson point process for 1200 seconds in order to obtain a stable context operatively. The chosen latency bin configurations are based on the end-to-end delay of the different services considered:

- 1-bin configuration: all VNFs will be shared;
- 2-bin configuration:
  - [0,20ms], [20ms,∞]: isolates ICA service VNFs;
  - [0,200ms], [200ms,∞]: isolates Video Streaming service VNFs;
- 3-bin configuration [0,20ms], [20ms,200ms], [200ms,∞]: isolates all VNFs.

All the services are considered deployed in the edge network, implying the a low network latency (10ms). The maximum number of available cores for each CPU is set to 8.

Once the parameters are set, the simulation can start and it produces a *csv* file containing a first version of the dataset containing a group of rows for each rate combination. The number of rows considered can vary depending on the number of bin configurations considered. In our case, for each rate combination we obtain 4 rows.

This dataset is then filtered in order to obtain the best distribution of latency classes for those service rates. We can consider different best configuration with respect to the requirements (i.e., low storage consumption or low CPU consumption). We decided to consider the amount of virtual CPU used by the VNFs as the discriminant factor. The best bin configuration will be the one that waste less CPU than the others.

As shown in figure 4.5, the number of instances becomes stable after a certain amount of time depending on the rate of the service considered and its lifetime, respecting the Little's Law whose formula is:

$$L = \lambda W, \tag{4.1}$$

where L is the number of instances in the system, λ is the Poisson rate and W is the lifetime of the instance.

The evolution of the service instances is as we expected, the number of instances reaches a stable number which oscillates since it is highly probable that a new instance is needed immediately after releasing the resources of the leaving one.

We simulated a wide range of different rate combination to increase the variety of our data. As stated before, the rates considered are referred to a mobility study conducted in a central area of Turin, considering different hour slots. The resulting parameters were expanded considering percentage of the rates obtained and completely different ones in order to create a larger number of contexts.

In the end, the obtained dataset contains a row for each context assigned to the best latency class distribution. We obtained more than 20000 different contexts with an irregular distribution of best latency class bins presented in table 4.2. We can affirm that this is due to the nature of the services and their VNFs: the Video Server VNF is quite demanding in terms of computation and it does not worth it sharing it with both the Video Streaming

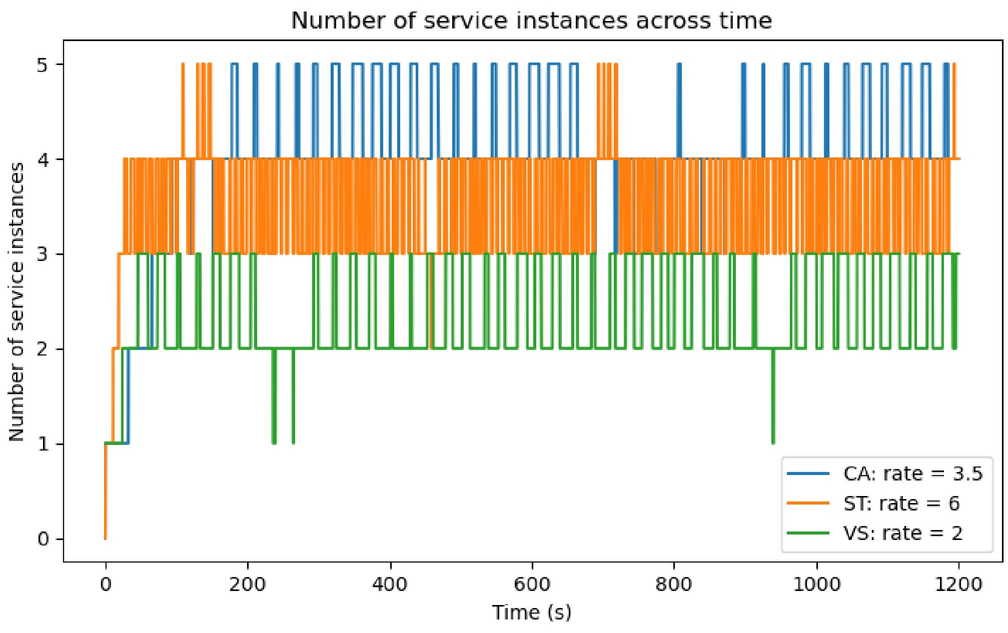


Figure 4.5. Example of number of service instances in a simulation

Table 4.2. Distribution of bin configurations in the dataset

Dataset composition	
Total: 26098	
Class distribution	**Percentage**
[0ms,∞]	8.72%
[0ms,20ms],[20ms,∞]	4.40%
[0,200ms], [200ms,∞]	53.96%
[0,20ms], [20ms,200ms], [200ms,∞]	32.91%

service and the See-through service in most cases. It is worth noting that choosing a different parameter for the best configuration could lead to different results.

In the next chapter we will consider only the average number of instances obtained in the simulations as input of the machine learning algorithms considered, while the bin configuration will act as the row label. However, the dataset is quite unbalanced and can bias different machine learning algorithms, requiring some transformations in order to obtain higher prediction accuracy.

Chapter 5

Machine Learning approach

The dataset we have created within our simulation is the first step in creating a machine learning model which will be able to classify each context to the best bin configuration. As a matter of fact, our issue is a *supervised learning problem*, the machine learning task given to a machine which consists in learning a function that maps an input value to a certain label, based on a set of labelled training data (our dataset). The infered function can be used to map new potential inputs to the correct label.

The steps to resolve a supervised learning problem are as follows:

1. *Determine the type of training samples.* In our case the data considered is all numeric.
2. *Gather a training set.* A set of input objects and the relative labels are gathered according to measurements and experiments.
3. *Determine the representative input features of the learned function.* The input is usually transformed in a feature vector, containing the essential parameters which describes the context. It is necessary to pay attention to the number of features considered in order to avoid the curse of dimensionality.
4. *Determine the structure of the learning functions and the correspondent algorithm.* This is the part of the process which will be described further below.
5. *Adjust the model.* Some algorithms require a certain parameter combination in order to achieve the best accuracy. They can be optimized via cross-validation using different types of parameter search.
6. *Evaluate the accuracy of the function.* After the adjustment and learning, the resulting function is evaluated by measuring its performance on a test set, different from the training one.

The supervised learning brings different tasks depending on the output type of our function. We are considering it a *regression* task when the output is quantitative (i.e., temperature prediction) or a *classification* task when the output is quantitative such as our case.

The first three steps of the standard procedure have been discussed in detail in the previous chapters. The focus in this next one will be on the different machine learning algorithm used to train our model.

5.1 Notation

We will give different meanings to a group of symbols to describe the formulas used by each different algorithm. The input variable will be typically a vector X which components are denoted as X_j. Our quantitative output is given by the letter Y. Observed values are written in lower case (i.e., observed values of X are written as x_i). Matrices are represented with bold capital letters such as $\mathbf{X}$. Vectors will be bold only when they have N components. The prediction of the output Y given an input vector X is denoted by $\hat{Y}$.

5.2 Classifiers

Several machine learning models are available for a supervised classification problem. Considering the nature of our dataset, it is convenient to focus on algorithms which work on numerical data, such as *Support Vector* or *Decision Tree* classifiers, emphasizing on the power of *ensemble classifiers* with *random forests*. All the models used for classification are drawn from *scikit-learn* [14], a Python module dedicated to machine learning methods, built on SciPy, an open-source library for scientific computing and technical computing.

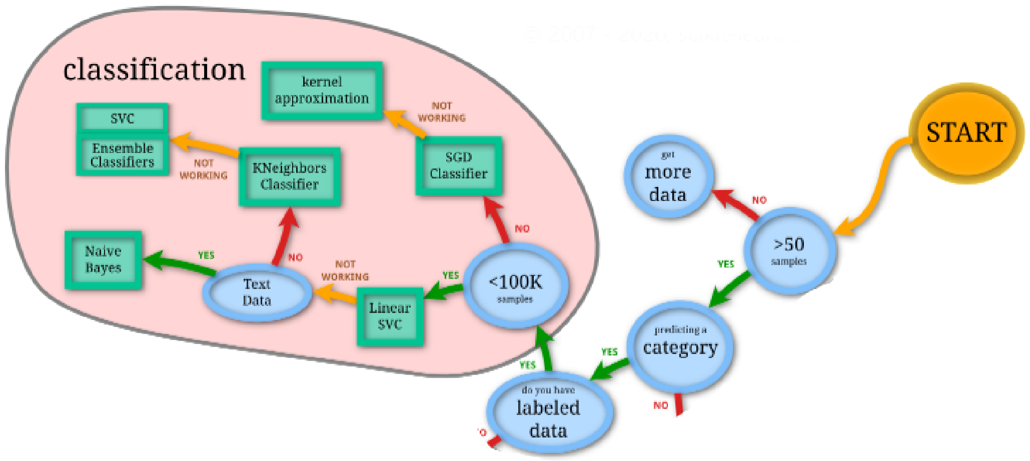


Figure 5.1. Algorithm cheat sheet from scikit-learn

5.2.1 Nearest-Neighbor classifier

Nearest-neighbor methods use the observations in the training set closest in input space to x to form $\hat{Y}$. The k-nearest neighbor fit for $\hat{Y}$ is defined as follows:

$$\hat{Y}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i \tag{5.1}$$

where $N_k(x)$ defines the neighborhood of x defined by the k closest points x_i in the training samples. (Add some other information here)

The classifier based on this equation is *memory-based*, a non-parametric algorithm that compares new test data with training data in order to understand which label is correct depending on what it learned from the dataset. One of its main features is the storage of the entire dataset, a quite costly computation if it is a large one. On the other hand, it does less assumptions than parametric models which make generalization about training data.

Given a query point x_0 we can find the k training points $x_{(r)}, r = 1, ..., k$ closest in distance to x_0 and classify it with the label most present among the k neighbors.

Due to the real-valued nature of our results, we use Euclidean distance in the feature space:

$$d_{(i)} = ||x_{(i)} - x_0|| \tag{5.2}$$

.

Though it is very simple, k-nearest-neighbors is used in a wide range of classification problems, successful where each class has many different prototypes and the decision boundary is very irregular.

5.2.2 Tree classifiers

Tree classifiers are based on partitioning the feature space in a set of rectangles, fitting a simple model in each one. We first describe one of the most popular methods, applied to a regression problem and then applied to our classification.

When growing a tree, we focus our attention to recursive binary partitions, splitting the space into two region and modelling the response by the mean of Y in each region. A variable is chosen to achieve the best split point in order to achieve the best fit. The split is recursive and it ends when a stop condition is met.

One of the tuning parameters of tree classifiers is the *tree size.* It can be controlled with different approaches, such as stop the splitting when the decrease in sum-of-squares due to the split exceeds a threshold. However, this is not optimal because it can prevent the possibility of choosing a worthless split which leads to an optimal one. The solution to this is growing a large tree and then prune it using *cost-complexity pruning.*

In a node m, representing a region R_m with N_m observation, let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k), \tag{5.3}$$

the proportion of class k observations in node m. The majority class in node m is defined as $k(m) = \arg\max_k \hat{p}_{mk}$. The impurity of each node is determined in order to define a termination condition of the growth of the tree. There are different ways to calculate it, such as the *Gini index* or the *cross-entropy*, which are more amenable to numerical optimization due to their differentiability. Moreover, they are more sensitive to changes in the node probabilities.

Random forest

Random Forests is a variant of a technique called *bagging*, or *bootstrap aggregation*, which reduces the variance of an estimated prediction function, working really well with high-variance and low-bias procedures such as trees. Before diving into the random forest technique, we describe what the bagging method is.

The *bootstrap* methods are useful for assessing statistical accuracy, used to estimate further sample prediction errors, reducing the variance by observing a set of observations on a single training set.

We consider the training set as $\mathbf{Z} = (z_1, z_2, ..., z_N)$ where $z_i = (x_i, y_i)$. The basic idea behind this technique is draw random datasets with replacement from the training data, each one the same size as the training set. This procedure is repeated a number of times B producing B *bootstrap datasets.* The model is then refit to each of the bootstrap dataset and we take the average of all predictions in a regression problem, obtaining

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x) \tag{5.4}$$

. In case of classification trees, the class predicted is the one occurring in most of the B predictions, taking a majority vote.

Random forests provides a further modification in the bagging process by decorrelating the trees in order to further reduce the variance when averaging the trees. The procedure of this technique is described as follows:

1. For $b = 1$ to B
   - Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data
   - Grow a random-forest tree T_b to the bootstrapped data, recursively repeating the following steps for each terminal node of the tree until a minimum size for each node is reached:
     - Select m predictors at random from the p predictors
     - Pick the best variable to split among the m
     - Split the node into two daughter nodes
2. Output of the ensemble of trees ${T_b}_1^B$

Regarding classification we can summarize the problem with $\hat{C}_{rf}^{B}(x) = \text{majority vote}\hat{C}_b(x)_1^B$, where $\hat{C}_b(x)$ os the class prediction of the bth random-forest tree.

5.2.3 Support Vector classifier

Support Vector Classifier are based on a technique consisting in constructing an optimal *hyperplane* which separate optimally samples from different classes.

Considering the training data as N pairs $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$ with $x_i \in R^p$ and $y_i \in -1,1$, we define an hyperplane as follows:

$$x : f(x) = x^T\beta + \beta_0 = 0, \tag{5.5}$$

where β is a unit vector: $||\beta|| = 1$. The classification rule imposed by $f(x)$ is

$$G(x) = sign[x^T\beta + \beta_0] \tag{5.6}$$

. Given the fact that the classes are separable, we can find the optimal hyperplane that creates the largest margin M between the training points belonging to different classes. The optimization problem

$$\beta_0, \beta_1, ...\beta_p, \epsilon_0, ...\epsilon_n \text{maximize} M \text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1 \tag{5.7}$$

is a suitable definition of this concept. However, finding the optimal hyperplane is a tough task, due to the presence in the training set of outliers and noises. Therefore, we define a slack variable $\xi = (\xi_1, \xi_2, ..., \xi_N)$ which allows a certain amount of errors in the classification, changing the object of the optimization in a *soft margin* one. The constraint is modified as follows:

$$y_i(x_i^T\beta + \beta_0) \geq M(1 - \xi_i), \tag{5.8}$$

which measures the overlap in relative distance, thus yielding a convex problem. The formula can be rearranged to discuss it with Lagrange multipliers. The equivalent form is

$$\beta, \beta_0 \text{min} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \xi_i \text{subject to} \xi_i \geq 0, y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i \forall i, \tag{5.9}$$

where the cost parameter "C" is the tuning parameter of the classifier.

The Lagrange primal function is

$$L_p = \frac{1}{1}||\beta||^2 + c\sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i[y_i(x_i^T\beta + \beta_0) - (1 - \xi_i)] - \sum_{i=1}^{N} \mu_i\xi_i, \tag{5.10}$$

which is minimized with respect to β, β_0 and ξ_i. Setting the derivatives to zero we obtain

$$\beta = \sum_{i=1}^{N} \alpha_i y_i x_i, \tag{5.11}$$

$$0 = \sum_{i=1}^{N} \alpha_i y_i, \tag{5.12}$$

$$\alpha_i = C - \mu_i, \forall i. \tag{5.13}$$

Substituting these values into the equation 5.10, we obtain the Lagrangian dual function

$$L_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{i'=1}^{N} \alpha_i\alpha_{i'}y_iy_{i'}x_i^Tx_{i'} \tag{5.14}$$

which defines a lower bound on the objective function, bound which is to maximize, subject to $0 \leq \alpha_i \leq C$ and $\sum_{i=1}^{N} \alpha_i y_i = 0$ The solution for β has the form

$$\hat{\beta} = \sum_{i=1}^{N} \hat{\alpha}_i y_i x_i, \tag{5.15}$$

with all $\hat{\alpha}_i \neq 0$ only for those i observations which meet the constraints. Those observations are called *support vectors*, due to the fact that $\hat{\beta}$ representation depends on them alone.

Maximizing the dual function 5.14 is a simpler convex quadratic problem than the primal 5.10 and can be solved with less complex techniques.

5.3 Balancing the dataset

We observed how the final dataset presents an imbalanced proportion of the classes, with (write the proportions). This can lead to a bias in certain machine learning algorithms which could ignore the minority class completely. In order to avoid this we can randomly resample the trained dataset with two main approaches:

- **Random Oversampling**: randomly duplicate samples belonging to the minority class, which could lead to overfitting for some models;
- **Random Undersampling**: randomly delete samples from the majority class which can result in losing information invaluable to some models

Both methods are considered as "naive resampling" since no assumptions are made on the data, thus resulting in a simple and fast resampling method, desired for large datasets. They can be used both in binary classification and multi-class classification and. more importantly, is it to apply only to the training dataset in order to influence the fit of the models.

In our work we use the imbalanced-learn Python library [12] to apply these two resampling procedures. We will choose the best method in terms of prediction accuracy.

5.4 Normalization

Feature manipulation or normalization is a set of transformations applied to the original features. Some transformation can decrease the approximation of estimation errors and increase the speed of the selected algorithm; however it is worth noting that each transformation should be related to the learning algorithm that we are going to apply to the feature vector as well as the prior conditions assumed.

Feature normalization can overcome the problem of having features on different scales which brings suboptimal solution to machine learning algorithm. Moreover, it can also speed up the runtime of some optimization algorithms.

There are several normalization techniques that we can consider for our machine learning algorithm. We indicate $\mathbf{f} = (f_1, ..., f_m) \in \mathbb{R}^m$ the value of the feature f over the m training samples. Also, $\bar{f} = \frac{1}{m}\sum_{i=1}^{m} f_i$ is the empirical mean of the feature over the samples.

- **Centering**: it makes the feature have zero mean, by setting $f_i \leftarrow f_i - \bar{f}$.
- **Unit Range**: it makes the range of each feature be [0,1]. We set $f_i \leftarrow \frac{f_i - f_{\text{min}}}{f_{\text{max}} - f_{\text{min}}}$, where f_{max} and f_{min} are the maximum and minimum value for the feature. The range can also be [1,1] if the transformation is $f_i \leftarrow 2\frac{f_i - f_{\text{min}}}{f_{\text{max}} - f_{\text{min}}} - 1$.
- **Standardization**: it transforms all features in order to obtain zero mean and unit variance. Let $\nu = \frac{1}{m}\sum_{i=1}^{m}(f_i - \bar{f})^2$ be the empirical mean of a feature. Then, we set $f_i \leftarrow \frac{f_i - \bar{f}}{\sqrt{\nu}}$.
- **Clipping**: it clips high or low values of the feature, i.e., $f_i \leftarrow \text{sign}(f_i)$.

- **Sigmoidal Transformation**: it applies a sigmoid function to the feature. For example, $f_i \leftarrow \frac{1}{1+e^{bf_i}}$, where b is a user-defined parameter. It can be seen as a lighter clipping, with small effect on values close to zero and a significant one on values far away from zero.

- **Logarithmic Transformation**: it makes $f_i \leftarrow \log(b + f_i)$, where b is a user-defined parameter. It is useful when dealing with counting features, i.e., number of words.

The *scikit-learn* library offers several normalization methods. We explored different approaches involving standardization leveraging classes such as *StandardScaler*, *MinMaxScaler* and *RobustScaler*.

- *StandardScaler* standardizes the feature by removing the mean and scaling to the unit variance, useful for learning algorithm such as SVM with RBF kernel which works with features centered in zero.

- *MinMaxScaler* scales the features between a minimum and a maximum value, typically between zero and one. It guarantees robustness when working with feature with small standard deviation and preserves zero entries.

- *RobustScaler* removes the median and scales according to the Interquartile Range (IQR) which is the range between the first quartile (25th quantile) and the 3rd quartile (75th quartile)

5.5 Model validation

Each machine learning approach considered relies on a set of hyperparameters which have to be chosen to obtain the best performances. There are several approaches in the hyperparameters search from the manual search via rules-of-thumb to the search on a predefined grid. In our work we leverage the scikit-learn object *GridSearchCV* where CV stands for "cross-validated". During the call to fit the estimator, it selects the parameters from a specified grid, maximizing a specific score (i.e., accuracy).

The *cross-validation* is a model validation technique for assessing the robustness of the trained model in order to avoid the generation of a machine learning model which is very accurate on the training data but has poor performance on unseen data (overfitting). Several cross-validation approaches can be applied, but we consider the basic approach, the k-fold CV. Once the dataset is split into training set and test set, such as in figure 5.2, the procedure goes as follows:

1. split the training set into k smaller sets;

2. train the model using $k - 1$ sets as training data;

3. validate the model on the remaining k set computing the desired score (i.e., accuracy);

4. repeat for every fold.

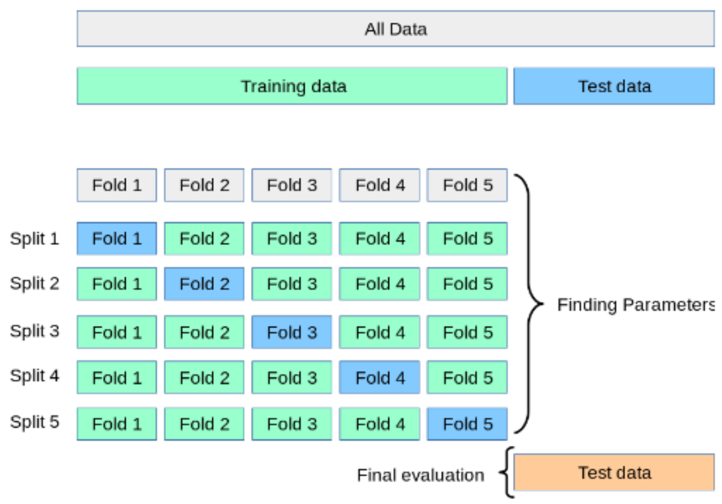


Figure 5.2. Cross-Validation technique

Table 5.1. Hyperparameters grid

SVC parameters	
C	{1,10,100,1000}
Kernel type	{linear, rbf}
gamma	$[1e-3, 1e-4]$
Decision Tree parameters	
split criterion	{gini, entropy}
max depth	[1,10]
minimum sample per split	[2,10]
minimum samples per leaf	[1,5]
Random Forests parameters	
number of estimator	{100,200,300}
max depth	[1,10]
minimum sample per split	[2,10]
minimum samples per leaf	[1,5]
K-Neighbors parameters	
neighbors	[1,10]
weight	{distance, uniform}

The performance measures obtained in each loop are then averaged and given as the final result to the user. It can be computationally expensive but does not waste data.

The list of hyperparameters for each classifier is presented in figure 5.1. Each combination is validated with a k-fold validation with $k = 10$.

Each classifier relies on different parameter which are useful in different ways

5.6 Results

We first analyze the obtained dataset. As we expected, it is convenient to share VNFs for most arrival rates, especially the vCIM, used from both the ICA service and the ST service. We can see in figure 5.3 an example of the average amount of virtual CPU cores consumed when sharing and when not sharing, considering a simulation of 1200 seconds with different arrival rates for each service and taking into account the limitation to the maximum number of available vCPU cores we set in the simulations. The saving of computational resources is clearly visible both in an off-peak scenario and during the rush hour.

In our case we have an high complexity VNF, the Video Server VNF, which brings most of the latency in both the See-Through service and Video Streaming service, two services with largely different latency requirements. It is undeniable that sharing this VNF in presence of several service instances the CPU must scale up faster, thus wasting more resources compared to a non-sharing scenario. Vice versa, the vCIM, shared between the ICA service and the ST service, is less demanding compared the Video Server VNF. Its complexity allows it to be shared between the two services creating an efficient resource saving.

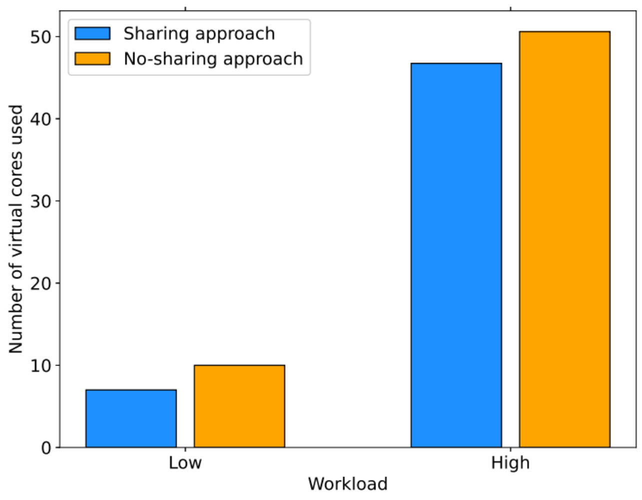


Figure 5.3. Benefit in VNF sharing with low service arrival rates on the left and with medium-high service arrival rates on the right

It is worth noting that all the situations are referred to the combination of the specific services considered. The results should differ when considering services with other requirements and the labels should be chosen accordingly to the nature of the services considered. For example, if we consider services with similar requirements and using the same VNF it is highly probable that the best bin configuration will be the one which enables sharing that specific VNF.

We consider both the oversampling and the undersampling method as well as the different type of normalization described in the previous section when validating and testing our models. In the following tables, the best results are presented with the set of hyper-parameters and the normalization method chosen to assess our results. The test accuracy is obtained on another dataset created for purpose equivalent to the 10% of the training dataset, using different arrival rate combinations for more data variety.

Table 5.2. Classification with oversampling applied

Random Forest			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
number of estimator	200	200	200
max depth	9	9	9
minimum sample per split	2	3	8
minimum samples per leaf	1	1	1
Validation accuracy	95.787%	95.762%	95.721%
Test accuracy	93.080%	93.540%	93.080%
Decision Tree			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
split criterion	entropy	entropy	gini
max depth	5	5	4
minimum sample per split	2	2	2
minimum samples per leaf	1	1	1
Validation accuracy	95.434%	95.432%	95.787%
Test accuracy	95.066%	95.066%	95.066%
Support Vector Classifier			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
C	100	1000	1000
kernel type	rbf	rbf	rbf
gamma	1e-3	1e-3	1e-3
Validation accuracy	96.574%	96.513%	96.513%
Test accuracy	95.487%	95.487%	95.487%
K-Nearest-Neighbors Classifier			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
n° of neighbors	3	3	3
weight	distance	uniform	uniform
Validation accuracy	95.372%	95.513%	95.296%
Test accuracy	89.773%	91.357%	91.456%

The results show that the **Support Vector Classifier** is the best choice in terms of accuracy when predicting new configurations. Moreover, the best class balancing method turned out to be the random oversampling. As a matter of fact, using the undersampling method, a great quantity of examples from the majority class are discarded in our dataset, wasting potential critical information useful to fitting a robust decision boundary. The results obtained using this method can be seen in table 5.3, where we can observe a decrease in the test accuracy of 10%.

Table 5.3. Classification with undersampling applied

Random Forest			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
number of estimator	100	100	100
max depth	5	5	5
minimum sample per split	2	2	2
minimum samples per leaf	1	1	1
Validation accuracy	96.878%	96.840%	96.848%
Test accuracy	87.605%	87.605%	87.605%

Decision Tree			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
split criterion	entropy	entropy	gini
max depth	3	3	3
minimum sample per split	2	2	2
minimum samples per leaf	1	1	1
Validation accuracy	95.434%	95.432%	95.787%
Test accuracy	87.364%	87.364%	87.364%

Support Vector Classifier			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
C	100	1000	1000
kernel type	rbf	rbf	rbf
gamma	1e-3	1e-3	1e-3
Validation accuracy	96.574%	96.513%	96.513%
Test accuracy	87.605%	87.605%	87.605%

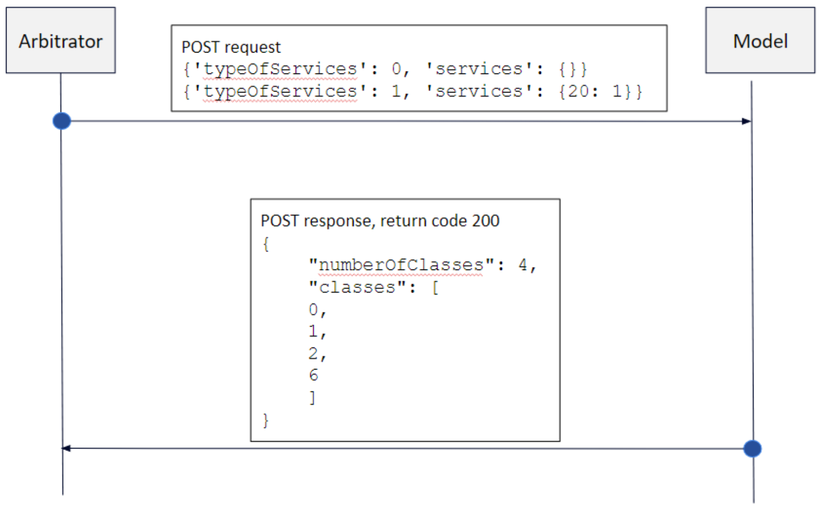
K-Nearest-Neighbors Classifier			
Hyperparameters	StandardScaler	MinMaxScaler	RobustScaler
n° of neighbors	5	6	8
weight	uniform	uniform	uniform
Validation accuracy	95.372%	95.513%	95.296%
Test accuracy	85.740%	86.522%	86.462%

The normalization methods are used in this case to evaluate better the performance of the chosen models. The real model, in fact, is trained on non-normalized data, due to the fact that the Arbitrator works with non-transformed information.

The model is saved and exported with joblib, a python library useful to store objects containing large data in a transparent manner, linking the saved file to the context of the original object. It is quite well-functioning with scikit-learn estimator since they often contain large numpy arrays.

The saved model is passed to the Arbitrator, which uses it to select the bin configuration, gathering information about the context from the VSD Catalog. The Arbitrator sends a JSON request to the ML model, containing the number of instances for every service divided by end-to-end latency.

The JSON exchange represented in figure 5.4 is enabled by a Flask server which listens to a specific port the incoming request. The request is parsed and given in input to the model which predicts the correct bin configuration to return to the Arbitrator, along with the boundaries of the classes in milliseconds.



POST request
```
{'typeOfServices': 0, 'services': {}}
{'typeOfServices': 1, 'services': {20: 1}}
```

POST response, return code 200
```
{
    "numberOfClasses": 4,
    "classes": [
    0,
    1,
    2,
    6
    ]
}
```

Figure 5.4. Example of JSON exchange between Arbitrator and model

In our implementation we considered only the requests for the services considered when creating the dataset. The request contains the list of the services currently handled by the system, organized by latency requirement. The response contains the number of classes predicted along with the boundaries which the Arbitrator will use in its algorithm to decide which VNF is to be shared or not.

Conclusions

The introduction of an AI/ML powered component inside the 5Growth architecture was a crucial step forward in the automation of several operations involving the 5G networks, with a special focus on behavior prediction and efficient response to abnormal events. Thanks to this new platform, it is possible to train a machine learning model on a determined topic and use it in the place of existing traditional algorithms which compute the same tasks, obtaining better performances using data supplied by the other layers presented in the architecture such as 5Gr-VS, 5Gr-SO and 5Gr-RL.

In this thesis we brought a simplification in the VNF arbitration process inside the 5Gr-VS by leveraging the AI/ML platform. We managed to lighten the Arbitrator component, the element in charge of handling decision about VNFs instantiation with respect to resource sharing between same VNFs belonging to different services. We succeed in lowering its computational complexity and leveraging the non-ML decision-making algorithm to build a service request simulator, customizable with different parameters depending on the context the user would like to recreate, i.e., the maximum number of virtual cores available for each VNF.

We simulated requests for three different automotive services, the Intersection Collision Avoidance service, the See-Through service and the Video Streaming service. The choice was dictated by the different and wide-ranged latency requirements (20ms for the ICA service, 200ms for the See-Through service and 1000ms for the Video Streaming service) and the VNFs composition. For example, we had the See-Through who shared the vCIM with the ICA service and the Video Server with the Video Streaming service.

The resulting dataset represented various and heterogeneous contexts, taking into account various combinations of poisson arrival rates and the best latency bin configuration for each one of them. We observed an evident preference for a partial VNF sharing, which is related to the nature of the services considered and the VNFs complexity.

After obtaining the dataset, we focused on finding the most suitable and accurate ML classification algorithm for our case. We considered several models, such as K-Nearest Neighbors and ensemble classifier, Random Forest, for instance. The Support Vector Classifier turned out to be the best among the considered models, with a quite high accuracy which can be improved with more simulations given diverse combinations of service requests rates.

We expect different results with different use cases, i.e., services from the Industry 4.0, such as robotic arm remote control, which has different latency requirements and different VNFs in contrast with the automotive one. These different services can be brought into the system and used to generate another dataset through extensive simulations, considering different bin configurations for the latency classes. The process could be iterated the same as the automotive use case, which continues with the training process of another machine learning model using the resulting dataset as input data. These results can be used to train a new ML model to classify correctly different contexts taking into account the amount of CPU used and the number of services currently handled by the 5Gr-VS.

The imminent future work could be the implementation of the Flask server used in the communication between the Arbitrator and the model inside the 5Gr-VS. Moreover, further improvements in the simulator precision can be obtained deriving more accurate complexity formulas for the VNFs in appropriate testbeds, instead of using the empirical ones we considered in our work. These formulas could also be obtained for other type of services when building a new dataset. Some of the possible candidates for other simulations are the Industry 4.0 services discussed above, which computational complexity could be evaluated for each VNF, assessing the stability of the simulator in a diverse scenario using different latency class bin configurations.

It is worth noting that implementing a recurrent training of the chosen ML model, whether using Industry 4.0 services or automotive ones, could improve the performances in terms of accuracy of the prediction. This could be useful to avoid the generation of another machine learning model and save resources for other tasks, boosting its performances at the same time. It could be efficient if the same classification algorithm performs at its best even with new data, otherwise it is better to search for another ML model that could predict the right bin configuration with a higher accuracy. It may be necessary to implement a multiple training on multiple models in order to find the better solution while expanding the dataset with new combinations of services and rates.

Bibliography

[1] IT Union. “Imt traffic estimates for the years 2020 to 2030.” In: Report ITU (2015), pp. 2370–.

[2] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," in IEEE Communications Surveys & Tutorials, vol. 18, no. 1, pp. 236-262, Firstquarter 2016, doi: 10.1109/COMST.2015.2477041.

[3] NGMN Alliance. “Description of network slicing concept.” In: NGMN 5G P 1.1 (2016).

[4] Yun Chao Hu et al. “Mobile edge computing—A key technology towards 5G.” In: ETSI white paper 11.11 (2015), pp. 1–16.

[5] 5G-Transformer. Visited on 2021-02-10. url: http://5g-transformer.eu/.

[6] 5G-TRANSFORMER. D1.2, 5G-TRANSFORMER initial system design. Tech. rep. May 2018.

[7] J. Baranda et al., "On the Integration of AI/ML-based scaling operations in the 5Growth platform," 2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Leganes - Madrid, Spain, 2020, pp. 105-109, doi: 10.1109/NFV-SDN50289.2020.9289863.

[8] 5Growth project. Visited on 2021-02-10. url: http://5growth.eu

[9] 5G-TRANSFORMER. Report on vertical requirements and use cases. Tech. rep. Dec. 2017

[10] Python. Visited on 2021-03-03. url: https://www.python.org

[11] Gomes, Pedro & Vieira, Fausto Ferreira, Michel. (2012). The See-Through System: From Implementation to Test-Drive. IEEE Vehicular Networking Conference, VNC. 40-47. 10.1109/VNC.2012.6407443.

[12] Guillaume Lemaître and Fernando Nogueira and Christos K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning" in Journal of Machine Learning Research, vol. 18, no. 17, pp. 1-5. url: http://jmlr.org/papers/v18/16-365

[13] G. Avino *et al.*, "A MEC-based Extended Virtual Sensing for Automotive Services," *2019 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*, Turin, Italy, 2019, pp. 1-6, doi: 10.23919/EETA.2019.8804512.

[14] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., "Scikit-learn: Machine Learning in Python" in Journal of Machine

Learning Research, vol. 12, pp. 2825-2830, 2011.