

GIXPD Pràctica 2

Entrenament d'un Model amb Docker i Kubernetes

Aránzazu Miguélez i Lucía Revaliente

Contingut

Introducció.....	3
Objectius	3
Configuració i instal·lació de dependències.....	4
Creació de l'aplicació i imatges Docker.....	6
Desplegació de l'aplicació a Kubernetes.....	13
Nou model	18
Dificultats.....	25
Què hem après?	26
Conclusions	27

Introducció

En aquesta pràctica, explicarem el procés de desplegar i gestionar aplicacions en entorns de contenidors utilitzant Docker i Kubernetes. Amb aquests recursos, treballarem en la creació d'un servei capaç d'exposar un model d'aprenentatge automàtic entrenat prèviament. Al llarg de la pràctica, ens centrarem en el desenvolupament de les imatges Docker i en la configuració dels recursos de Kubernetes per assegurar la seva estabilitat i escalabilitat de l'aplicació. El model en qüestió, basat en dades de compra de clients en una botiga, ens permetrà comprendre millor com crear i gestionar una API que pugui ser utilitzada per consultes externes. A través d'aquest exercici, ens familiaritzarem amb les eines fonamentals per a desplegar models en entorns productius, optimitzant la nostra capacitat de crear serveis que s'adaptin a les necessitats reals de dades i rendiment.

Abans de plantejar els objectius, ens agradaria repassar alguns conceptes vitals. En primer lloc, **Docker** és una arquitectura client/servidor. En altres paraules, és una plataforma de contenidors que permet empaquetar i executar aplicacions en entorns aïllats. Cada contenidor conté la seva pròpia aplicació, dependències i configuració però comparteixen el mateix SO. Recordem que un contenidor permet executar aplicacions de manera aïllada en un entorn compartit (sistema físic HW). Per tant, és una unitat que empaqueta una aplicació i totes les dependències en un sol paquet. Tots els contenidors s'executen sobre el mateix sistema operatiu host. Per tant, un contenidor és la virtualització del SO físic i altres recursos (com CPU, memòria, emmagatzematge).

En segon lloc, **Kubernetes** és una plataforma d'orquestració de contenidors. El seu objectiu principal és facilitar el desplegament i gestió d'apps complexes, especialment les que es desglosen en microserveis. Tot i que pot semblar senzill, és complicat perquè les aplicacions poden tenir dependències, necessiten gestionar el tràfic... Kubernetes és l'estàndard (API estàndard) per la creació i gestió d'aplicacions natives en el núvol.

Objectius

L'objectiu d'aquest exercici serà crear un servei per exposar un model entrenat en ciència de dades. Això constarà de 3 tasques:

1. **Configuració i instal·lació de dependències:** Docker, Kubectl i Minikube, per a desenvolupar, contenir i desplegar aplicacions en entorns de Kubernetes.
2. **Crear aplicació i imatges Docker:** una per a l'entrenament del model i una altra per a servir el model mitjançant una API.
3. **Desplegar l'aplicació a Kubernetes:** incloent-hi la configuració de serveis i recursos requerits per garantir l'accés al model de forma segura i escalable.

ID MÀQUINA VIRTUAL: Pràctica 2

USUARI: gixpd-ged-47

Configuració i instal·lació de dependències

En primer lloc, hem instal·lat les dependències (paquets):

1. Docker

A partir de la documentació donada al tutorial de la pràctica hem instal·lat el Docker.

Docker Engine és una tecnologia de contenidorització de codi obert per a la construcció i entronitzant les seves aplicacions. Motor de Docker actua com a client-servidor aplicació.

```
# Add Docker's official GPG key:

sudo apt-get update

sudo apt-get install ca-certificates curl

sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o
/etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:

echo \

  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/debian \

  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \

  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

# Install the latest version

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-
plugin docker-compose-plugin
```

2. Kubectl

A partir de la documentació donada al tutorial de la pràctica hem instal·lat el Kubectl.

Kubernetes és un motor d'orquestració de contenidors de codi obert per automatitzar el desplegament, l'escalat i la gestió d'aplicacions en contenidors.

```
# Download the latest release with the command:

curl -LO https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl

# Install Kubectl

sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

3. Minikube:

A partir de la documentació donada al tutorial de la pràctica hem instal·lat el Minikube.

Minikube és un Kubernetes local, que se centra a facilitar l'aprenentatge i el desenvolupament de Kubernetes. Un cop hem acabat la instal·lació, hem comprovat el funcionament de minikube creant un deployment hello.-world i funcionava correctament.

1 Installation

Click on the buttons that describe your target platform. For other architectures, see [the release page](#) for a complete list of minikube binaries.

Operating system: **Linux** macOS Windows

Architecture: **x86-64** ARM64 ARMv7 ppc64 S390x

Release type: **Stable**

Installer type: Binary download **Debian package** RPM package

To install the latest minikube **stable** release on **x86-64 Linux** using **Debian package**:

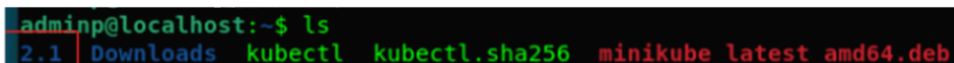
```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb
sudo dpkg -i minikube_latest_amd64.deb
```

Figura 1.

Creació de l'aplicació i imatges Docker

Una vegada hem instal·lat totes les dependències, hem creat dues imatges de Docker.

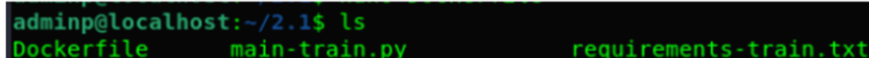
La primera imatge està etiquetada com a **model-train:default**. Aquesta genera un contenidor que entrena un model i el desa en un disc. Per desenvolupar-la, hem creat un directori (`mkdir 2.1`) on hem guardat tots els arxius que volem que contingui el contenidor. Més endavant explicarem el seu ús.



```
adminp@localhost:~$ ls
2.1 Downloads kubect1 kubect1.sha256 minikube_latest_amd64.deb
```

Figura 2.

A continuació, hem copiat els arxius **requirements-train.txt** i **main-train.py** en el directori /2.1.



```
adminp@localhost:~/2.1$ ls
Dockerfile main-train.py requirements-train.txt
```

Figura 3.

El següent que hem fet ha sigut crear un arxiu **Dockerfile** en el mateix directori /2.1. Aquest és un arxiu “de text pla que conté una sèrie d'instruccions necessàries per crear una imatge que, més tard, es convertirà en una sola aplicació utilitzada per un propòsit en concret. En el nostre cas, per entrenar un model. El fitxer **Dockerfile** conté les següents instruccions:

1. **FROM python:** indica que la imatge base pel contenidor serà una imatge oficial de Python. En altres paraules, la imatge es construirà a partir d'una app en Python.
2. **WORKDIR /2.1:** estableix que el directori de treball en el contenidor serà /2.1. Per tant, tot el que passi després de la línia passarà dins d'aquest directori.
3. **COPY ["requirements-train.txt", "main-train.py", "."]:** copia fitxers i directoris des del context de construcció (el teu sistema de fitxers local) a una ubicació dins de la imatge de Docker. Concretament, el fitxer “requirements-train.txt” i “main-train.py” en el destí “.”, que és el directori de treball actual (/2.1) dins de la imatge Docker on s'estan copiant els fitxers.
4. **RUN pip install --trusted-host pypi.python.org -r requirements-train.txt:** executa una comanda durant la construcció de la imatge. Bàsicament, instal·la totes les dependències (paquets) de Python que estan llistades en l'arxiu requirements-train.txt. El flag `--trusted-host pypi.python.org` es una opció per evitar problemes durant la instal·lació de paquets de Python.
5. **# Make port 80 available to the world outside this container:** mostra l'intenció d'utilitzar el port 80 (port estàndard d'HTTP), però no es fa servir perquè la línia està comentada.
6. **# Define environment variable:** suggereix que es podria establir una variable d'entorn. Tot i això, com està comentada no té efecte en la creació del contenidor.
7. **CMD ["python", "main-train.py"]:** defineix la comanda que s'executarà quan arrenqui el contenidor. S'especifica que s'ha d'executar **python main-train.py**, el

qual llença l'script **main-train.py** a dins del contenidor (l'executa). Per tant, considerem que l'arxiu **.py** és l'arxiu principal de l'app (per entrenar el model).

```
adminp@localhost: ~/2.1
GNU nano 7.2 Dockerfile-train
# Dockerfile: Use an official Python runtime as a parent image
FROM python
# Set the working directory to /app
WORKDIR /2.1
# Copy the current directory contents into the container at /app
COPY ["requirements-train.txt", "main-train.py", "."]
# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements-train.txt
# Make port 80 available to the world outside this container
# Define environment variable
# ENV NAME World
# Run app.py when the container launches
CMD ["python", "main-train.py"]
```

Figura 4.

Un cop hem creat el directori 2.1 on està el Dockerfile configurat, executarem la següent comanda : **sudo docker build --tag=app2.1** . Aquesta comanda crearà el docker per crear imatge anomenada app2.1 a partir del Dockerfile. La executarem des del directori 2.1. Per tal d'indicar on es troba el Dockerfile utilitzem el darrer punt de la comanda que indicarà que l'arxiu està en el mateix directori 2.1.

```
adminp@localhost:~/2.1$ sudo docker build --tag=app2.1 .
[+] Building 41.6s (9/9) FINISHED docker:default
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 553B 0.0s
=> [internal] load metadata for docker.io/library/python:latest 1.2s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
[1/4] FROM docker.io/library/python:latest@sha256:4f3780acb8d126492a 22.2s
=> resolve docker.io/library/python:latest@sha256:4f3780acb8d126492a8 0.0s
=> sha256:a47c7f7f31e941e78bf63ca19f0811b675283e2c0 24.05MB / 24.05MB 0.8s
=> sha256:97fc9ec41404bf3b8395f559c4fa381e20fa303f1fb 5.73kB / 5.73kB 0.0s
=> sha256:cdd62bf39133c498a16f7a7b1b6555ba43d02b251 49.56MB / 49.56MB 1.2s
=> sha256:a173f2aee8e962ea19dble418ae84a0c9f71480b5 64.39MB / 64.39MB 3.4s
=> sha256:4f3780acb8d126492a8890a1e1715d36773c0cc1865 9.72kB / 9.72kB 0.0s
=> sha256:09aa001329f10fc6d784a1a48bba50815bc35bee2f3 2.32kB / 2.32kB 0.0s
=> sha256:01272fe8adbacc44afd2b92994b31c40a151f43 211.27MB / 211.27MB 4.8s
=> sha256:e2451c50195ed9cea9f5e115387c359a87446d7c593 6.16MB / 6.16MB 1.4s
=> extracting sha256:cdd62bf39133c498a16f7a7b1b6555ba43d02b2511c508fa 3.6s
=> sha256:c3da17cfdde36ecb3a508c32d3f89f23f29101554 25.76MB / 25.76MB 2.1s
=> sha256:21104ca25c011c6e2e708656224090241094ae0b588badb 250B / 250B 2.3s
=> extracting sha256:a47c7f7f31e941e78bf63ca19f0811b675283e2c00ddea10 1.0s
=> extracting sha256:a173f2aee8e962ea19dble418ae84a0c9f71480b51f768a1 3.9s
=> extracting sha256:01272fe8adbacc44afd2b92994b31c40a151f4324ca39205 9.7s
=> extracting sha256:e2451c50195ed9cea9f5e115387c359a87446d7c5930beb7 0.4s
=> extracting sha256:c3da17cfdde36ecb3a508c32d3f89f23f291015540af2df3 1.1s
=> extracting sha256:21104ca25c011c6e2e708656224090241094ae0b588badb9 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 1.74kB 0.0s
[2/4] WORKDIR /2.1 0.1s
[3/4] ADD . /2.1 0.0s
[4/4] RUN pip install --trusted-host pypi.python.org -r requirements 14.7s
=> exporting to image 3.2s
=> exporting layers 3.1s
=> writing image sha256:436dbd4450069e29836b7a0248aac5dfc77ce99fed2a2 0.0s
=> naming to docker.io/library/app2.1 0.0s
```

Figura 5.

Un cop fet això ens hem adonat de que la imatge havia de tenir com a nom **model-train:default**. Per tant hem executat la comanda **sudo docker tag app2.1 model-train:default** -f Dockerfile- train per tal de canviar el nom.

Seguidament hem esborrat l'altre nom **app2.1** ja que quan l'hem canviat s'ha duplicat amb la comanda **sudo docker rmi app2.1**. En la **Figura 8** podem veure el resultat final de la creació correcta del docker.

```
adminp@localhost:~/2.1$ sudo docker tag app2.1 model-train:default
adminp@localhost:~/2.1$ sudo docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
app2.1	latest	436dbd445006	3 minutes ago	1.37GB
model-train	default	436dbd445006	3 minutes ago	1.37GB
hello-world	latest	d2c94e258dcb	17 months ago	13.3kB

Figura 6.

```
adminp@localhost:~/2.1$ sudo docker rmi app2.1
Untagged: app2.1:latest
```

Figura7.

```
adminp@localhost:~/2.1$ sudo docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
model-train	default	b79a491a78d8	8 minutes ago	1.37GB
hello-world	latest	d2c94e258dcb	17 months ago	13.3kB

Figura 8.

Una vegada hem entrenat el model (training), cal que aquest estigui disponible en un entorn que permeti realitzar prediccions. Això implica empaquetar el model en una aplicació web. Per tant, cal desplegar el model, és a dir, que sigui accessible i usable per altres sistemes i usuaris. Per fer-ho, s'ha de configurar un servei que s'encarregui de rebre les dades d'entrada com peticions HTTP i tornar prediccions basades en el model entrenat.

Per tant, la segona imatge Docker que hem creat està etiquetada com **model-server:default**. Aquesta imatge és un servidor web creat amb el framework Flask (framework lleuger de Python). El seu propòsit principal és exposar el model entrenat perquè es pugui interactuar mitjançant una API. El servidor tindrà dues rutes (URL) que realitzaran diferents funcions:

- **Ruta "/" (arrel):** carregarà una pàgina HTML que explica com funciona el servei. En altres paraules, la web descriurà el propòsit de l'API i/o donarà informació general sobre el model. Tot i això, la pàgina web serà molt bàsica i no tindrà funcions complexes.
- **Ruta "/model":** carregarà el model entrenat (primera imatge), per tant, és la ruta més important. En cridar aquesta ruta, es rebran els paràmetres d'entrada necessaris per executar el model i tornarà la sortida en format JSON.

Per desenvolupar la imatge del server, hem copiat els arxius **requirements-server.txt** i **main-server.py** en el directori /2.1. A continuació, hem generat el Dockerfile que configura un servidor Flask en un contenidor Docker. El contingut d'aquest és el següent:

1. **FROM python:** indica que la imatge base pel contenidor serà una imatge oficial de Python. En altres paraules, la imatge es construirà a partir d'una app en Python.
2. **WORKDIR /2.1:** estableix que el directori de treball en el contenidor serà /2.1. Per tant, tot el que passi després de la línia passarà dins d'aquest directori.
3. **COPY ["requirements-server.txt", "main-server.py", "."]:** copia fitxers i directoris des del context de construcció (el teu sistema de fitxers local) a una ubicació dins de la imatge de Docker. Concretament, el fitxer "requirements-server.txt" i "main-server.py" en el destí ".", que és el directori de treball actual (/2.1) dins de la imatge Docker on s'estan copiant els fitxers.
4. **RUN pip install --trusted-host pypi.python.org -r requirements-server.txt:** executa una comanda durant la construcció de la imatge. Bàsicament, instal·la totes les dependències (paquets) de Python que estan llistades en l'arxiu requirements-server.txt. El flag --trusted-host pypi.python.org es una opció per evitar problemes durant la instal·lació de paquets de Python.
5. **# Make port 80 available to the world outside this container:** mostra l'intenció d'utilitzar el port 80 (port estàndard d'HTTP), però no es fa servir perquè la línia està comentada.
6. **# Define environment variable:** suggereix que es podria establir una variable d'entorn. Tot i això, com està comentada no té efecte en la creació del contenidor.
7. **CMD ["flask", "--app", "main-server.py", "run", "--host=0.0.0.0"]:** defineix la comanda que s'executarà quan arrenqui el contenidor. S'especifica que s'executa el Flask (framework web). "--app" i "main-server.py" especifiquen que Flask ha de carregar el fitxer **main-server.py** com a punt d'entrada. "run" indica a Flask ha d'executar l'aplicació. Finalment, "--host=0.0.0.0" determina que l'aplicació estarà disponible per a totes les adreces IP (no només localment dins del contenidor). Això és necessari perquè es pugui accedir a l'aplicació des de fora del contenidor. En resum, en aquesta línia s'executa el servidor Flask quan s'inicia el contenidor, escoltant peticions de totes les IPs.

```
adminp@localhost: ~/2.1
GNU nano 7.2 Dockerfile-server
# Dockerfile: Use an official Python runtime as a parent image
FROM python
# Set the working directory to /app
WORKDIR /2.1
# Copy the current directory contents into the container at /app
COPY ["requirements-server.txt", "main-server.py", "."]
# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements-server.txt
# Make port 80 available to the world outside this container
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["flask", "--app", "main-server.py", "run", "--host=0.0.0.0"]
```

Figura 9.

Seguidament, hem executat la comanda **sudo docker build --tag=model-server:default -f Dockerfile-server ..**. Aquesta comanda crearà la imatge anomenada model-server:default a partir del Dockerfile de la Figura 9. Cal destacar que s'ha d'executar des del directori /2.1. Si analitzem bé la comanda, veiem que el punt final “.” indica que el Dockerfile es troba en el directori actual. Ara és important especificar el paràmetre -f per indicar al sistema quin dels dos Dockerfiles ha de seguir (ja que hi ha dos en el mateix directori).

```

adminp@localhost:~/2.2$ sudo docker build --tag=model-server:default .
+ Building 10.8s (9/9) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile              0.0s
=> => transferring dockerfile: 596B                             0.0s
=> [internal] load metadata for docker.io/library/python:latest 0.4s
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                    0.0s
=> [1/4] FROM docker.io/library/python:latest@sha256:a31cbb4db18c6f09e33 0.0s
=> [internal] load build context                                0.0s
=> => transferring context: 672B                                  0.0s
=> CACHED [2/4] WORKDIR /2.2                                    0.0s
=> [3/4] ADD . /2.2                                             0.0s
=> [4/4] RUN pip install --trusted-host pypi.python.org -r requirements- 9.5s
=> exporting to image                                           0.9s
=> => exporting layers                                           0.8s
=> => writing image sha256:0e74fc7b445576296abbfcfbefcfb779e0d20b52c12e1 0.0s
=> => naming to docker.io/library/model-server:default          0.0s

```

Figura 10.

Si revisem les imatges Docker amb la comanda **sudo docker images**, podem comprovar que ambdues imatges s'han creat amb èxit.

```

adminp@localhost:~/2.2$ sudo docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
model-server        default      0e74fc7b4455     About a minute ago 1.11GB
model-train          default      b79a491a78d8     12 days ago      1.37GB
hello-world         latest      d2c94e258dcb     17 months ago    13.3kB

```


Figura 11.

Per tal de verificar si hem creat els contenidors correctament, els hem executat. En primer lloc, hem entrenat el model dins del contenidor, emmagatzemant el resultat en la ruta /home/adminp/model. Per fer-ho, hem executat la comanda **sudo docker run -e MODEL_PATH=/mnt/model/model.npy -v /home/adminp/model:/mnt/model model-train:default**.

- **sudo docker run**: executa el contenidor.
- **-e MODEL_PATH=/mnt/model/model.npy**: estableix variable d'entorn anomenada 'MODEL_PATH' dins del contenidor. Aquesta variable indica la ubicació on es guardarà el model en el sistema d'arxius del contenidor.
- **-v /home/adminp/model:/mnt/model**: munta un volum del sistema d'arxius local ('/home/adminp/model') dins del contenidor, permetent que el model entrenat es mantingui fora del contenidor i es pugui accedir a ell fins i tot després que el contenidor es tanqui.

- **model-train:default:** nom de la imatge Docker que s'utilitza per executar aquest contenidor, i que conté el codi necessari per entrenar el model.

Com podem observar en la Figura 12, el model s'ha entrenat correctament, mostrant una mètrica de rendiment (score) de 0.8086.



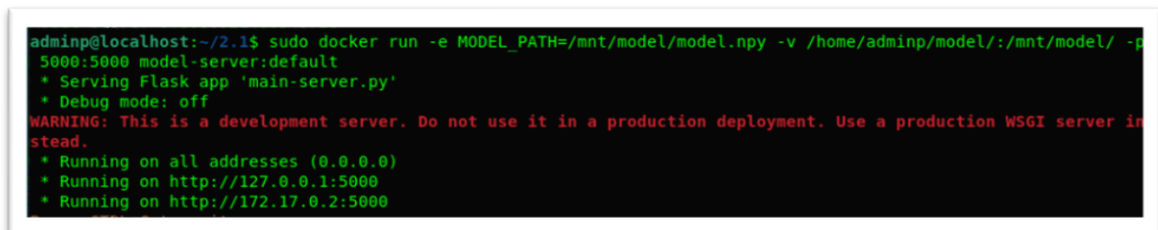
```
adminp@localhost:~/2.1$ sudo docker run -e MODEL_PATH=/mnt/model/model.npy -v /home/adminp/model:/mnt/model/ model-train:default
Model trained successfully
Model Score: 0.8086921460343728
```

Figura 12.

En segon lloc, hem inicialitzat un servidor Flask que desplega el model entrenat per poder fer prediccions a través de sol·licituds HTTP. Per fer-ho, hem executat la comanda **sudo docker run -e MODEL_PATH=/mnt/model/model.npy -v /home/adminp/model:/mnt/model -p 5000:5000 model-server:default**.

- **sudo docker run:** inicia el contenidor.
- **-e MODEL_PATH=/mnt/model/model.npy:** estableix una variable d'entorn anomenada 'MODEL_PATH' dins del contenidor. Aquesta variable indica la ubicació on es guardarà el model en el sistema d'arxius del contenidor.
- **-v /home/adminp/model:/mnt/model:** munta un volum del sistema d'arxius local ('/home/adminp/model') dins del contenidor, permetent que el model entrenat es mantingui fora del contenidor i es pugui accedir a ell fins i tot després que el contenidor es tanqui.
- **-p 5000:5000:** mapeja el port 5000 del contenidor al port 5000 de la màquina local. En altres paraules, qualsevol sol·licitud feta a localhost:5000 es redirigirà al contenidor, on està executant-se l'app Flask.
- **model-server:default:** imatge de Docker que conté l'aplicació Flask que servirà el model per fer prediccions. Aquesta aplicació està executant-se dins del contenidor.

Tal i com volíem, s'inicia un servidor Flask que corre a localhost (127.0.0.1) i en l'adreça interna del contenidor 172.17.0.2 al port 5000.



```
adminp@localhost:~/2.1$ sudo docker run -e MODEL_PATH=/mnt/model/model.npy -v /home/adminp/model:/mnt/model/ -p 5000:5000 model-server:default
* Serving Flask app 'main-server.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
```

Figura 13.

Per comprovar si l'execució és existosa, hem d'obrir el navegador mentre tenim el contenidor del server "corrents" i cerquem l'IP indicada (127.0.0.1) amb el port 5000. Observem que tal i com hem explicat, la sol·licitud feta es redirigeix al contenidor de l'app del Flask.

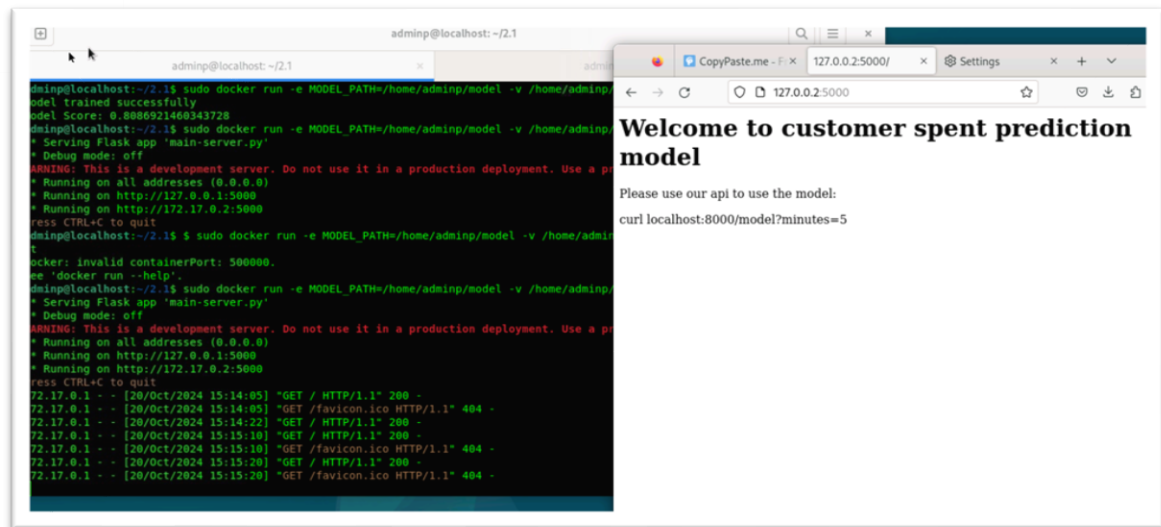


Figura 14.

La redirecció al contenidor no funciona si no executem el servidor Flask dins del contenidor perquè el servidor és qui controla les sol·licituds HTTP.

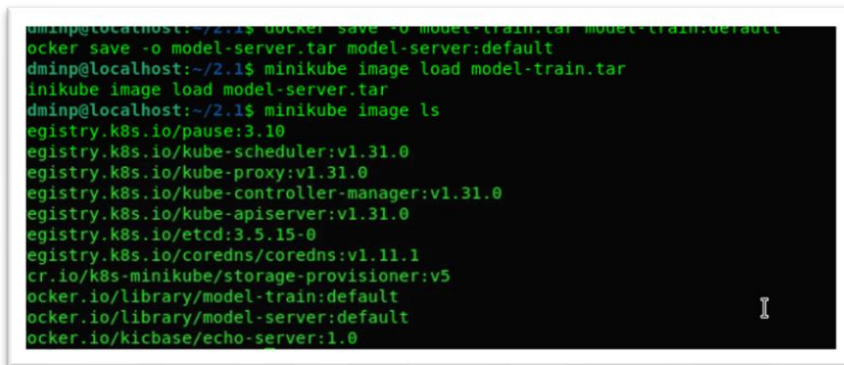
Desplegació de l'aplicació a Kubernetes

Un cop hem emmagatzemat les imatges d'entrenament i del servidor en els contenidors, hem desplegat l'aplicació del nostre entorn de “Producció” a Kubernetes. En altres paraules, hem implementat l'app que entrena un model i desa els resultats en un volum persistent (emmagatzematge permanent, no volàtil). Desplegar una aplicació a Kubernetes implica diversos **recursos** que juguen rols específics en la gestió i operació d'aplicacions en contenidors. En aquest apartat, exposem l'ordre que hem seguit per realitzar la implementació, juntament amb la importància dels recursos utilitzats.

a. Carregar les imatges:

Per crear els següents arxius, el primer que hem fet ha sigut guardar les imatges del Docker en format .tar per així poder copiar les imatges entre diferents entorns. La comanda que hem executat és **sudo docker save -o train.tar model-train:default** i **sudo docker save -o server.tar model-server:default**. En aquest cas, el paràmetre -o indica en quin és l'arxiu de sortida.

En segon lloc, hem carregat les imatges.tar al minikube. Per fer-ho, és vital inicialitzar minikube amb **minikube start**. Per pujar les imatges.tar, hem fer servir la comanda **sudo minikube image load train.tar** i **sudo minikube image load server.tar**.



```

minip@localhost: ~/2.1$ docker save -o model-train.tar model-train:default
minip@localhost: ~/2.1$ docker save -o model-server.tar model-server:default
minip@localhost: ~/2.1$ minikube image load model-train.tar
minikube image load model-server.tar
minip@localhost: ~/2.1$ minikube image ls
registry.k8s.io/pause:3.10
registry.k8s.io/kube-scheduler:v1.31.0
registry.k8s.io/kube-proxy:v1.31.0
registry.k8s.io/kube-controller-manager:v1.31.0
registry.k8s.io/kube-apiserver:v1.31.0
registry.k8s.io/etcd:3.5.15-0
registry.k8s.io/coredns/coredns:v1.11.1
cr.io/k8s-minikube/storage-provisioner:v5
docker.io/library/model-train:default
docker.io/library/model-server:default
docker.io/kicbase/echo-server:1.0
  
```

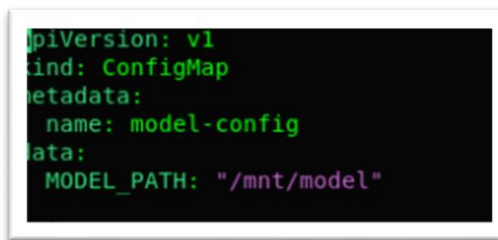
Figura 15.

El primer recurs que hem creat ha sigut **ConfigMap**. Aquest és un objecte en Kubernetes que permet emmagatzemar les dades de la configuració en parells clau-valor. S'empra per “injectar” configuracions en els contenidors sense haver de modificar o reconstruir la imatge. Per tant, ConfigMap aporta flexibilitat i eficiència en l'app.

Doncs, el primer que hem fet ha sigut crear l'arxiu **configmap.yaml** (ConfigMap). Aquest emmagatzemarà la ruta del model entrenat, la qual serà accedida pels recursos que explicarem més endavant (Job i Deployment).. El contingut de **configmap.yaml** és la següent:

1. **apiVersion: v1**: especifica la versió de l'API de Kubernetes que s'està utilitzant per crear el recurs. En aquest cas, s'està utilitzant v1, que és la versió estable i més comú per als recursos de configuració com ConfigMap.
2. **kind: ConfigMap**: especifica que el tipus de recurs és un ConfigMap. Com hem dit, els ConfigMaps s'utilitzen per emmagatzemar dades de configuració que poden ser utilitzades per contenidors en un pod.

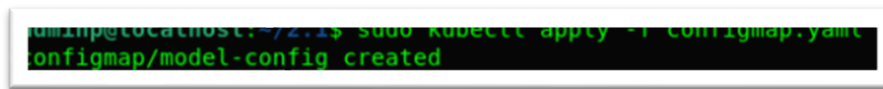
3. **metadata:** aquesta secció conté metadades sobre el ConfigMap:
 - a. **name: model-config:** indica el nom del ConfigMap. Aquest nom s'utilitzarà per referenciar el ConfigMap en altres recursos com els Deployments o els Pods.
4. **data:** aquesta secció conté les dades que s'emmagatzemen en el ConfigMap:
 - a. **MODEL_PATH: "/mnt/model":** es defineix un parell clau-valor. En aquest cas, la clau és MODEL_PATH i el seu valor és la ruta /mnt/model. Aquesta variable (MODEL_PATH) s'està utilitzant per especificar la ubicació d'un model (probablement un model de màquina o un fitxer de configuració d'aplicació) dins del sistema de fitxers del contenidor. El valor /mnt/model és una ruta que pot ser utilitzada per l'aplicació per accedir al model durant la seva execució.



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: model-config
data:
  MODEL_PATH: "/mnt/model"
```

Figura 16.

Un cop l'hem definit el carreguem i creen amb la següent comanda.



```
admin@localhost: ~/2.1$ sudo kubectl apply -f configmap.yaml
configmap/model-config created
```

Figura 17.

En segon lloc, hem hagut de crear un **Job**. Un Job en Kubernetes s'encarrega de gestionar l'execució d'una tasca que ha de ser realitzada una única vegada o un nombre limitat de vegades, com l'entrenament d'un model de machine learning. Un Job assegura que un nombre específic de pods (grup de contenidors que es despleguen junts en un mateix host i comparteixen recursos) s'executi fins que la tasca es completi. A continuació podem veure com hem definit el job i els paràmetres corresponent. És important tenir en compte el mountPath que ha de ser ell mateix que hem indicat prèviament al volum al executar els dockers.

1. **apiVersion:** Utilitza l'API batch/v1, que és específica per a recursos de tipus Job.
2. **kind:** Defineix que aquest recurs és un Job.
3. **metadata:** Definim el nom (model-train-job) per identificar el Job.
4. **spec:** Conté la configuració principal.
5. **template:** Definim els contenidors i configuracions necessàries per al Job.
 - a. **containers:** Especifica el contenidor que s'executarà (model-train-container), que utilitza la imatge model-train:default.
 - b. **envFrom:** Fa referència a model-config per carregar la variable MODEL_PATH.

- c. **volumeMounts:** Munta el volum en /mnt/model, que és la ubicació on es desarà el model.
6. **restartPolicy:** Estableix que, en cas de fallida, el Job es tornarà a iniciar.
 7. **volumes:** Defineix un volum (model-volume) que apunta a /home/adminp/model per persistir el model entrenat fora del contenidor. Aquest ha de ser el que hem indicat al crear el docker.

```
adminp@localhost:~/2.1$ cat job.yaml
# job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: model-train-job
spec:
  template:
    spec:
      containers:
      - name: model-train
        image: model-train:default
        env:
        - name: MODEL_PATH
          valueFrom:
            configMapKeyRef:
              name: model-config
              key: MODEL_PATH
        volumeMounts:
        - name: model-storage
          mountPath: /mnt/model
      resources:
        requests:
          memory: "512Mi"
          cpu: "500m"
        limits:
          memory: "1Gi"
          cpu: "1"
      restartPolicy: OnFailure
    volumes:
    - name: model-storage
      hostPath:
        path: /home/adminp/model
```

Figura 18.

Seguidament, carreguem i creen el job.

```
adminp@localhost:~/2.1$ sudo kubectl apply -f job.yaml
job.batch/model-train-job created
```

Figura 19.

Això entrenarà el model i el desarà en el volum. Verifiqui l'estat del job amb:

```
adminp@localhost:~/2.1$ sudo kubectl get jobs
NAME                STATUS    COMPLETIONS  DURATION  AGE
model-train-job     Complete  1/1           23s       74s
```

Figura 20.

A continuació definirem el deployment.yaml.

1. **apiVersion:** Utilitza apps/v1, l'API per a recursos com els Deployments.
2. **kind:** Defineix aquest recurs com un Deployment.

3. **metadata:** Defineix el nom per identificar el Deployment.
4. **spec:** Conté la configuració principal.
 - a. **replicas:** Defineix el nombre de rèpliques del servidor a 3.
 - b. **selector:** Indica quins Pods han de ser gestionats pel Deployment, en aquest cas, aquells amb l'etiqueta app: model-server.
 - c. **template:** Defineix el Pod que executa el servidor:
 - **containers:** Especifica el contenidor (model-server-container) amb la imatge model-server:default.
 - **envFrom:** Carrega les variables d'entorn des del ConfigMap model-config.
 - **volumeMounts:** Munta el volum en /mnt/model per accedir al model.
 - d. **volumes:** Defineim un volum que apunta a /home/adminp/model.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: model-server
  template:
    metadata:
      labels:
        app: model-server
    spec:
      containers:
        - name: model-server
          image: model-server:default
          env:
            - name: MODEL_PATH
              valueFrom:
                configMapKeyRef:
                  name: model-config
                  key: MODEL_PATH
          volumeMounts:
            - name: model-storage
              mountPath: /mnt/model
      resources:
        requests:
          memory: "512Mi"
          cpu: "500m"
        limits:
          memory: "1Gi"
          cpu: "1"
      readinessProbe:
        httpGet:
          path: /
          port: 5000
        initialDelaySeconds: 5
        periodSeconds: 5
      livenessProbe:
        httpGet:
          path: /
          port: 5000
        initialDelaySeconds: 15
        periodSeconds: 20
      volumes:
        - name: model-storage
          hostPath:
            path: /home/adminp/model
```

Figura 21.

Per últim, creen el service. El definim d'aquesta manera:

1. **apiVersion:** Utilitza v1, l'API per a recursos com els Services.
2. **kind:** Defineix aquest recurs com un Service.
3. **metadata:** Defineix el nom (model-server-service) per identificar el servei.
4. **spec:** Conté la configuració principal.
5. **selector:** Indica que el servei exposa els Pods amb l'etiqueta app: model-server.

- a. **ports:** Defineix els ports utilitzats:
- **port:** El port 80 és el punt d'entrada extern del servei.
 - **targetPort:** Indica el port intern (5000) del contenidor.
- b. **type:** Especifica NodePort, que exposa el servei a un port de cada node del clúster, fent accessible el servei externament.

```
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: model-server
spec:
  type: NodePort
  ports:
    - port: 5000
      targetPort: 5000
  selector:
    app: model-server
```

Figura 22.

Un cop desplegada la nostra aplicació a Kubernetes comprovem que funcioni amb la següent comanda.

```
adminp@localhost:~/2.1$ kubectl port-forward service/model-server 5000:5000
Forwarding from 127.0.0.1:5000 -> 5000
Forwarding from [::1]:5000 -> 5000
Handling connection for 5000
Handling connection for 5000
```

Figura 23.

Per tal de comprobar-ho, obrim un navegador i posem 127.0.0.1:5000 com se'ns ha indicat. Podem veure com la pàgina càrrega correctament i per tant, tota la configuració és correcta.

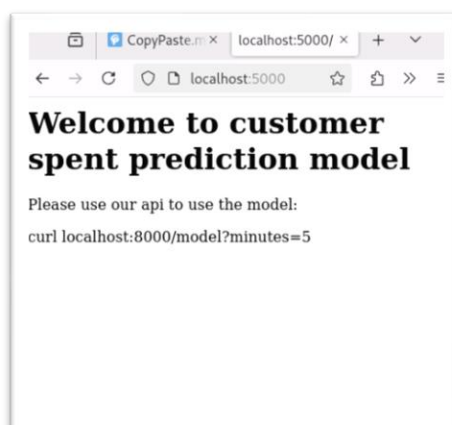


Figura 24.

Nou model

Una de les bones pràctiques en Machine Learning, és canviar el model d'entrenament per comparar les prediccions i decidir quin és el model més òptim. Per tant, en aquest apartat explicarem com hem executat l'aplicació sent el nou model `LinearRegression`.

En primer lloc, hem modificat els arxius `main-train.py` i `main-server.py`, sent ara `main-train2.py` i `main-server2.py`. Si analitzem els canvis:

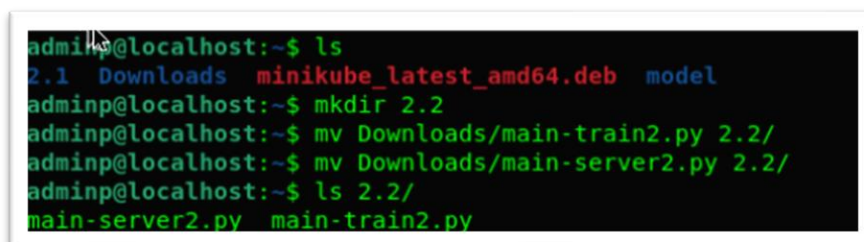
1. Arxiu d'entrenament:

- El model utilitzava un model polinòmic `np.polyfit` i `np.poly1d`. Per canviar a un model de regressió lineal, hem fet servir `LinearRegression` de scikit-learn i guardem el model complet utilitzant `joblib` (en lloc de `np.save`, que només funciona per a dades simples, com ara coeficients d'un polinomi).
- Per a `LinearRegression`, cal que les dades d'entrada (`train_x` i `test_x`) estiguin en un format de matriu 2D. Per aquest motiu, hem aplicat `reshape(-1, 1)` per transformar els arranjaments en matrius d'una columna.
- Canviem `np.save` per `joblib.dump` per guardar el model complet (`joblib` és més adequat per guardar i carregar models complexos de scikit-learn).

2. Arxiu de server:

- En lloc de `np.load`, utilitzem `joblib.load(MODEL_PATH)`, ja que el model de `LinearRegression` va ser guardat amb `joblib` a l'entrenament.
- Al codi original, minuts es passava directament a `model()` com un valor sencer. Amb `LinearRegression`, hem necessitat convertir minuts en un array d'una sola columna (matriu 2D), per la qual cosa fem servir `np.array(minutes).reshape(-1, 1)`.
- En lloc de `model(int(minutes))`, que només aplica per a models polinòmics, hem fet servir `model.predict(...)` per fer la predicció amb el model de `LinearRegression`.
- Convertim `prediction[0]` a un float perquè el valor sigui serialitzable a JSON i es torni en un format adequat per a la resposta API.

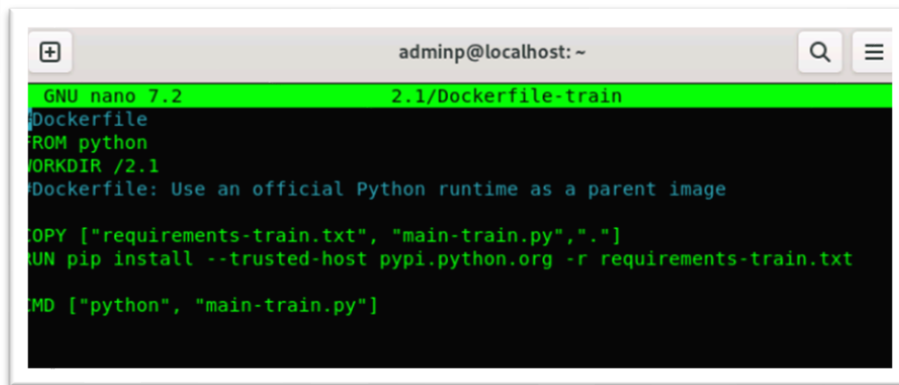
En segon lloc, hem enviat els documents a la MV a través de `copypaste.me`. I d'aquí a un nou directori, on emmagatzemarem les imatges i contenidors del nou model que crearem a continuació.



```
adminp@localhost:~$ ls
2.1 Downloads minikube_latest_amd64.deb model
adminp@localhost:~$ mkdir 2.2
adminp@localhost:~$ mv Downloads/main-train2.py 2.2/
adminp@localhost:~$ mv Downloads/main-server2.py 2.2/
adminp@localhost:~$ ls 2.2/
main-server2.py main-train2.py
```

Figura 25.

El següent pas, ha sigut modificar els Dockerfiles per especificar que les imatges del contenidor s'han de crear a partir del nou model (`main-train2.py` i `main-server2.py`).

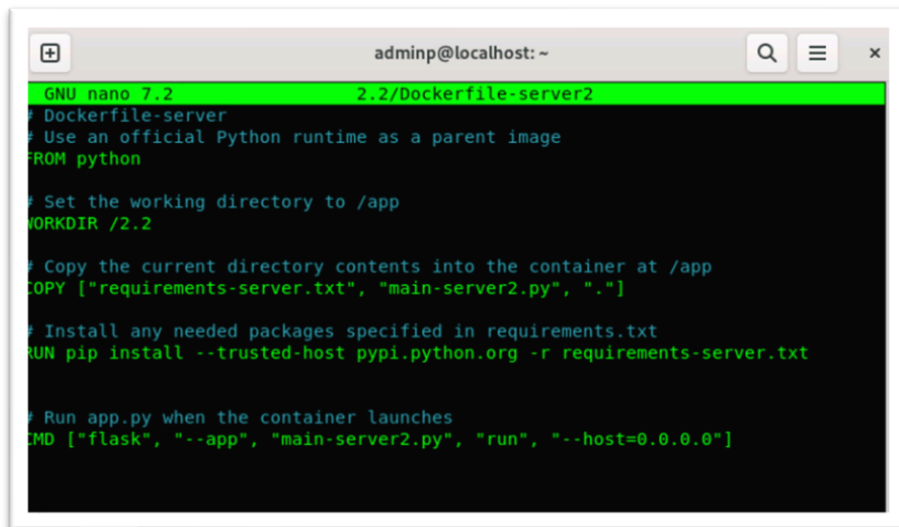


```
adminp@localhost: ~
GNU nano 7.2 2.1/Dockerfile-train
# Dockerfile
FROM python
WORKDIR /2.1
# Dockerfile: Use an official Python runtime as a parent image

COPY ["requirements-train.txt", "main-train.py", "."]
RUN pip install --trusted-host pypi.python.org -r requirements-train.txt

CMD ["python", "main-train.py"]
```

Figura 26.



```
adminp@localhost: ~
GNU nano 7.2 2.2/Dockerfile-server2
# Dockerfile-server
# Use an official Python runtime as a parent image
FROM python

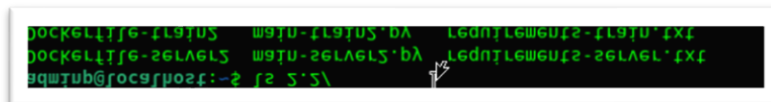
# Set the working directory to /app
WORKDIR /2.2

# Copy the current directory contents into the container at /app
COPY ["requirements-server.txt", "main-server2.py", "."]

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements-server.txt

# Run app.py when the container launches
CMD ["flask", "--app", "main-server2.py", "run", "--host=0.0.0.0"]
```

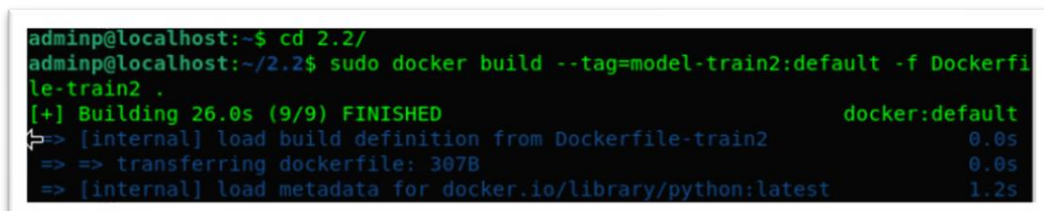
Figura 27.



```
dockerfile-train-19105  w91u-19105'bx  L6dnJL6w6uf2-1910'ixf
dockerfile-train-261665 w91u-261665'bx  L6dnJL6w6uf2-26166'ixf
mqw1ub@roc9fuo2f:-$ ls 5'5\
```

Figura 28.

El següent pas, ha sigut crear les imatges a partir dels Dockerfiles: Dockertfile-train, Dockerfile-service.



```
adminp@localhost:~$ cd 2.2/
adminp@localhost:~/2.2$ sudo docker build --tag=model-train2:default -f Dockerfile-train2 .
[+] Building 26.0s (9/9) FINISHED docker:default
=> [internal] load build definition from Dockerfile-train2 0.0s
=> => transferring dockerfile: 307B 0.0s
=> [internal] load metadata for docker.io/library/python:latest 1.2s
```

Figura 29.

```
adminp@localhost:~/2.2$ sudo docker build --tag=model-server2:default -f Dockerfile-server2 .
[+] Building 9.8s (9/9) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile-server2    0.0s
=> => transferring dockerfile: 548B                             0.0s
=> [internal] load metadata for docker.io/library/python:latest 0.4s
=> [internal] load .dockerignore                               0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build context                                0.0s
=> => transferring context: 1.04kB                               0.0s
=> [1/4] FROM docker.io/library/python:latest@sha256:a31cbb4db18c6f09e33 0.0s
=> CACHED [2/4] WORKDIR /2.2                                   0.0s
=> [3/4] COPY [requirements-server.txt, main-server2.py, .]    0.0s
=> [4/4] RUN pip install --trusted-host pypi.python.org -r requirements- 8.4s
=> exporting to image                                           0.9s
=> => exporting layers                                           0.9s
=> => writing image sha256:0d1d142c59149449b5b95db6cddb0f27ebb661110d422 0.0s
=> => naming to docker.io/library/model-server2:default         0.0s
adminp@localhost:~/2.2$
```

Figura 30.

```
adminp@localhost:~/2.2$ sudo docker images
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE
model-server2       default    0d1d142c5914  34 seconds ago 1.11GB
model-train2        default    b56dc4d93bbb  About a minute ago 1.37GB
model-server        default    8b52933731ba  15 hours ago   1.11GB
model-train         default    94ef72299b6e  15 hours ago   1.37GB
gcr.io/k8s-minikube/kicbase v0.0.45    aeed0e1d4642  8 weeks ago    1.28GB
```

Figura 31.

A partir d'aquestes imatges, hem creat els contenidors que contenen el nou model. Com podem veure en la Figura 32, veiem que el nostre model de train obté un valor diferent de Score, pel que s'està entrenant d'una manera diferent. Podem concloure doncs, que la modificació s'ha realitzat amb èxit.

```
adminp@localhost:~/2.2$ sudo docker run -e MODEL_PATH=/mnt/model2/model.npy -v /home/adminp/model2:/mnt/model2/ model-train2:default
Model trained successfully
Model Score: 0.6793794571076386
```

Figura 32.

Tot i això, en crear el contenidor del server, ens ha saltat un error ja que no havíem modificat les dependències del .py.

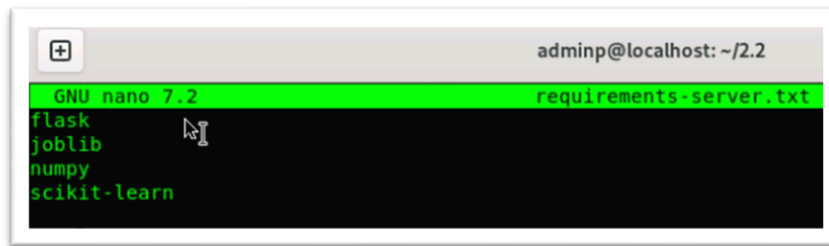
```
adminp@localhost:~/2.2$ sudo docker run -e MODEL_PATH=/mnt/model2/model.npy -v /home/adminp/model2:/mnt/model2/ -p 5000:5000 model-server2:default
Usage: flask run [OPTIONS]
Try 'flask run --help' for help.

Error: While importing 'main-server2', an ImportError was raised:

Traceback (most recent call last):
  File "/usr/local/lib/python3.13/site-packages/flask/cli.py", line 245, in locate_app
    __import__(module_name)
    ~~~~~^~~~~
  File "/2.2/main-server2.py", line 4, in <module>
    import joblib # Para cargar el modelo guardado con joblib
    ~~~~~^~~~~
ModuleNotFoundError: No module named 'joblib'
```

Figura 33.

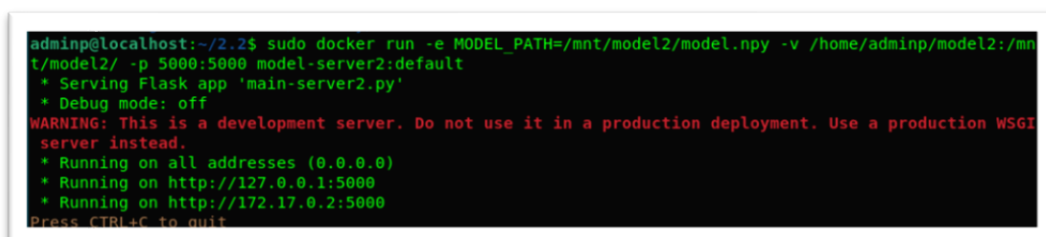
És per aquest motiu que hem afegit als requeriments del server joblib i scikit-learn.



```
adminp@localhost: ~/2.2
GNU nano 7.2 requirements-server.txt
flask
joblib
numpy
scikit-learn
```

Figura 34.

Com que els contenidors i les imatges són inmutables, hem hagut d'esborrar-los i crear-los de nou. Ara, el Dockerfile-sever podrà importar les llibreries esmentades i no saltarà cap error.



```
adminp@localhost:~/2.2$ sudo docker run -e MODEL_PATH=/mnt/model2/model.npy -v /home/adminp/model2:/mnt/model2/ -p 5000:5000 model-server2:default
* Serving Flask app 'main-server2.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

Figura 35.

Si obrim el localhost:5000 en el cercador, veiem que el model s'executa amb èxit. Tot i això, surt el mateix missatge que abans perquè no s'ha modificat des del fitxer .py.

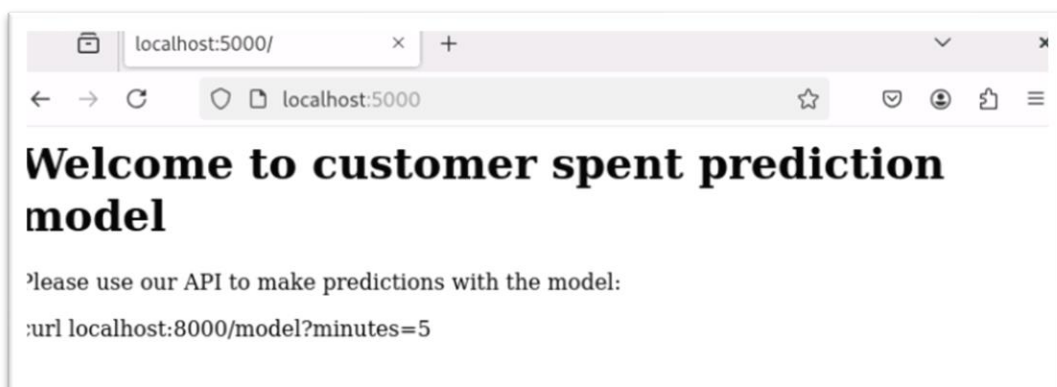


Figura 36.

Per desplegar l'aplicació, hem realitzat els mateixos passos que abans: hem penjat les imatges al minikube, creat el configmap, job, deployment i service...

```
adminp@localhost:~$ docker save -o server2.tar model-server2:default
adminp@localhost:~$ minikube image load server2.tar
adminp@localhost:~$ minikube image ls
registry.k8s.io/pause:3.10
registry.k8s.io/kube-scheduler:v1.31.0
registry.k8s.io/kube-proxy:v1.31.0
registry.k8s.io/kube-controller-manager:v1.31.0
registry.k8s.io/kube-apiserver:v1.31.0
registry.k8s.io/etcd:3.5.15-0
registry.k8s.io/coredns/coredns:v1.11.1
gcr.io/k8s-minikube/storage-provisioner:v5
docker.io/library/model-train:default
docker.io/library/model-train2:default
docker.io/library/model-server:default
docker.io/library/model-server2:default
docker.io/kubase/echn-server:1.0
```

Figura 37.

```
adminp@localhost:~$ cp 2.2/configmap.yaml 2.2/configmap2.yaml
adminp@localhost:~$ cp 2.1/deployment.yaml 2.2/deployment2.yaml
adminp@localhost:~$ cp 2.1/service.yaml 2.2/service2.yaml
adminp@localhost:~$ ls 2.1/
dockerfile-server  configmap.yaml  job.yaml  main-server.py  model-server.tar  requirements-server.txt  service.yaml
dockerfile-train  deployment.yaml  kubectrl  main-train.py  model-train.tar  requirements-train.txt
adminp@localhost:~$ cp 2.1/job.yaml 2.2/job2.yaml
```

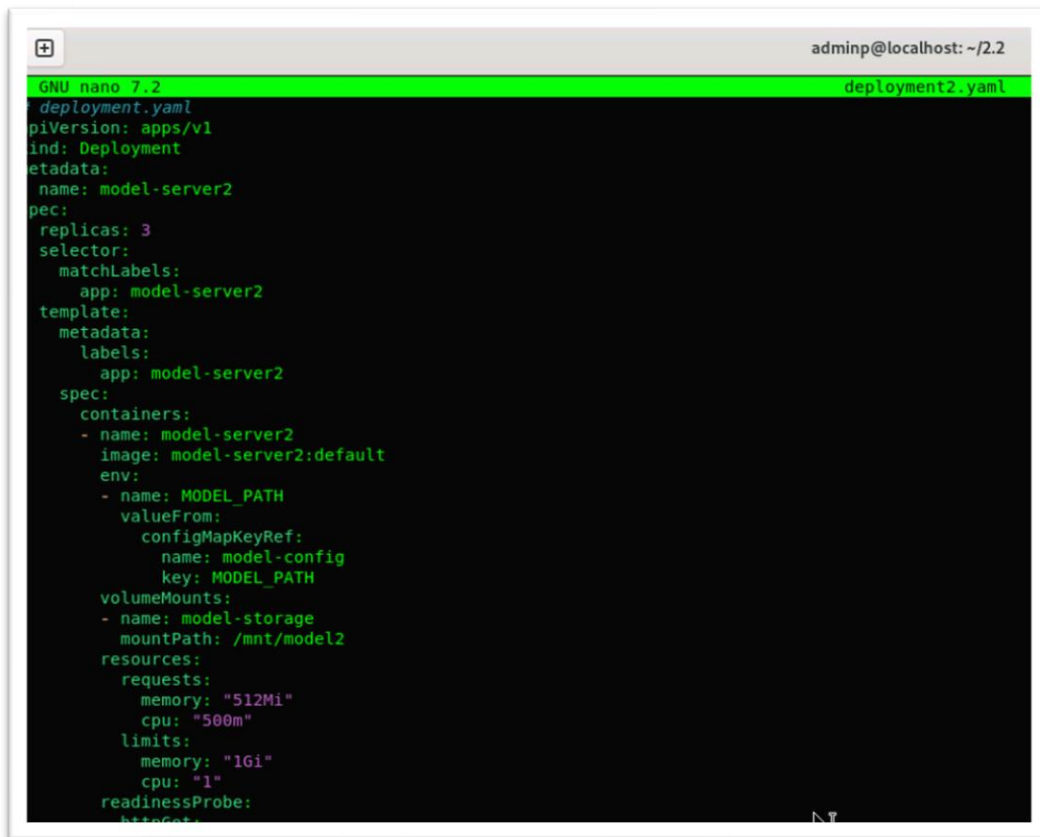
Figura 38.

```
adminp@localhost: ~/2.2
GNU nano 7.2 configmap2.yaml
# configmap2.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: model-config
data:
  MODEL_PATH: "/mnt/model2/model.npy"
```

Figura 39.

```
adminp@localhost: ~/2.2
GNU nano 7.2 job2.yaml
job2.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: model-train2-job
spec:
  template:
    spec:
      containers:
        - name: model-train2
          image: model-train2:default
          env:
            - name: MODEL_PATH
              valueFrom:
                configMapKeyRef:
                  name: model-config2
                  key: MODEL_PATH
          volumeMounts:
            - name: model-storage2
              mountPath: /mnt/model2
      resources:
        requests:
          memory: "512Mi"
          cpu: "500m"
        limits:
          memory: "1Gi"
          cpu: "1"
      restartPolicy: OnFailure
      volumes:
        - name: model-storage2
          hostPath:
            path: /home/adminp/model2
```

Figura 40.




```

adminp@localhost: ~/2.2
GNU nano 7.2 deployment2.yaml
deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-server2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: model-server2
  template:
    metadata:
      labels:
        app: model-server2
    spec:
      containers:
      - name: model-server2
        image: model-server2:default
        env:
        - name: MODEL_PATH
          valueFrom:
            configMapKeyRef:
              name: model-config
              key: MODEL_PATH
        volumeMounts:
        - name: model-storage
          mountPath: /mnt/model2
      resources:
        requests:
          memory: "512Mi"
          cpu: "500m"
        limits:
          memory: "1Gi"
          cpu: "1"
      readinessProbe:
        httpGet:

```

Figura 41.



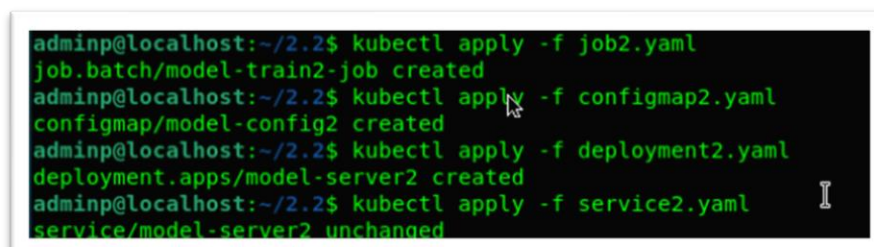
```

adminp@localhost: ~/2.2
GNU nano 7.2 service2.yaml
service.yaml
apiVersion: v1
kind: Service
metadata:
  name: model-server2
spec:
  type: NodePort
  ports:
    - port: 5000
      targetPort: 5000
  selector:
    app: model-server2

```

Figura 42.

Finalment, hem creat els pods i executat la comanda que executa el servidor amb l'app.



```

adminp@localhost:~/2.2$ kubectl apply -f job2.yaml
job.batch/model-train2-job created
adminp@localhost:~/2.2$ kubectl apply -f configmap2.yaml
configmap/model-config2 created
adminp@localhost:~/2.2$ kubectl apply -f deployment2.yaml
deployment.apps/model-server2 created
adminp@localhost:~/2.2$ kubectl apply -f service2.yaml
service/model-server2 unchanged

```

Figura 43.

```
adminp@localhost:~/.2.2$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-minikube-7d4899fd6-4p424	1/1	Running	4 (49m ago)	17h
model-server-65f79b946b-m8gdm	1/1	Running	4 (48m ago)	17h
model-server-65f79b946b-p7qt6	1/1	Running	4 (48m ago)	17h
model-server-65f79b946b-t7bm9	1/1	Running	4 (48m ago)	17h
model-server2-597cc97c5b-6sntr	1/1	Running	0	54s
model-server2-597cc97c5b-76c77	1/1	Running	0	54s
model-server2-597cc97c5b-mwvvn	1/1	Running	0	54s
model-train-job-2r6bf	0/1	Completed	0	17h
model-train2-job-tr44h	0/1	Completed	0	67s

Figura 44.

Concloem doncs, que l'aplicació amb el nou model funciona, ja que si obrim un navegador amb localhost:8080, observem la pàgina mostrada anteriorment.

Com l'objectiu d'aquest apartat és desplegar l'aplicació amb un model diferent al plantjat, no analitzarem quin dels dos models és millor.

The screenshot shows a terminal window on the left and a web browser on the right. The terminal window displays the following commands and output:

```
minip@localhost:~/.2.2$ kubectl port-forward service/model-server2 5000:5000
Error: timed out waiting for the condition
minip@localhost:~/.2.2$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
hello-minikube-7d4899fd6-4p424      1/1      Running   4 (47m ago) 17h
del-server-65f79b946b-m8gdm         1/1      Running   4 (45m ago) 17h
del-server-65f79b946b-p7qt6         1/1      Running   4 (45m ago) 17h
del-server-65f79b946b-t7bm9         1/1      Running   4 (45m ago) 17h
del-server2-597cc97c5b-6sntr        1/1      Running   0           54s
del-server2-597cc97c5b-76c77        1/1      Running   0           54s
del-server2-597cc97c5b-mwvvn        1/1      Running   0           54s
del-train-job-2r6bf                  0/1      Completed 0           17h
del-train2-job-tr44h                 0/1      Completed 0           67s
minip@localhost:~/.2.2$ kubectl apply -f job2.yaml
batch/model-train2-job created
minip@localhost:~/.2.2$ kubectl apply -f configmap2.yaml
configmap/model-config2 created
minip@localhost:~/.2.2$ kubectl apply -f deployment2.yaml
deployment.apps/model-server2 created
minip@localhost:~/.2.2$ kubectl apply -f service2.yaml
service/model-server2 unchanged
minip@localhost:~/.2.2$ nano job2.yaml
minip@localhost:~/.2.2$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
hello-minikube-7d4899fd6-4p424      1/1      Running   4 (49m ago) 17h
del-server-65f79b946b-m8gdm         1/1      Running   4 (48m ago) 17h
del-server-65f79b946b-p7qt6         1/1      Running   4 (48m ago) 17h
del-server-65f79b946b-t7bm9         1/1      Running   4 (48m ago) 17h
del-server2-597cc97c5b-6sntr        1/1      Running   0           54s
del-server2-597cc97c5b-76c77        1/1      Running   0           54s
del-server2-597cc97c5b-mwvvn        1/1      Running   0           54s
del-train-job-2r6bf                  0/1      Completed 0           17h
del-train2-job-tr44h                 0/1      Completed 0           67s
minip@localhost:~/.2.2$ kubectl port-forward service/model-server2 5000:5000
Error: unable to listen on port 5000: Listeners failed to create with the following errors: (unable to listen: Error listen tcp4 127.0.0.1:5000: bind: address already in use unable to listen: Error listen tcp6 [::]:5000: bind: address already in use)
rdr: unable to listen on any of the requested ports: [{5000 5000}]
minip@localhost:~/.2.2$ kubectl port-forward service/model-server2 8080:5000
Forwarding from 127.0.0.1:8080 -> 5000
Forwarding from [::]:8080 -> 5000
Waiting connection for 8080
Waiting connection for 8080
```

The web browser on the right shows the following content:

Welcome to customer spent prediction model

Please use our API to make predictions with the model:

```
curl localhost:8000/model?minutes=5
```

Figura 45.

Dificultats

En aquest apartat, exposarem les dificultats més importants que hem trobat durant la realització de la pràctica.

En primer lloc, vam cometre una confusió en l'execució dels arxius Dockerfile de **train** i **server**. Com que els vam anomenar de la mateixa manera, no sabíem com crear la imatge del servidor seguint el Dockerfile corresponent. Encara que vam anomenar un dels fitxers **Dockerfile-train**, vam aprendre que, per activar el contenidor, era necessari especificar el nom del fitxer amb el paràmetre **-f**. Així, la comanda correcta és: **sudo docker build --tag=model-server:default -f Dockerfile-server ..** Aquesta dificultat ens va servir per adonar-nos que es pot crear més d'una imatge en el mateix directori.

En segon lloc, a l'hora de fer el docker run havíem d'especificar diversos paràmetres. Un d'ells era **-v** (volum) que un paràmetre necessari per a muntar directoris locals dins del contenidor. Aquest pas era essencial perquè el contenidor tingués accés als fitxers necessaris per executar el codi, com els models d'entrenament o els conjunts de dades. Inicialment, no érem conscients de la importància d'especificar el volum, cosa que ens va portar a errors en l'execució, ja que el contenidor no trobava els fitxers requerits.

A més, vam trobar dificultats amb la gestió de les variables d'entorn. Alguns paràmetres, com el port o les rutes d'accés, necessitaven ser definits a l'entorn per a garantir una configuració correcta del servei. No obstant això, en alguns intents inicials, ens vam oblidar de declarar aquestes variables, fet que va provocar problemes en la connexió i l'execució del servidor.

Què hem après?

All llarg d'aquesta pràctica, hem adquirit coneixements valuosos sobre les diferents infraestructures i recursos clau per al desplegament d'aplicacions en contenidors. A continuació, detallem els aprenentatges principals segons els diferents apartats.

En la **configuració i instal·lació de dependències**, ens hem adonat de l'importància de cercar la dependència correcta segons el nostre SO. En cas contrari, hi haurà conflictes i no es podran executar correctament les aplicacions.

En la **creació de l'aplicació i imatges Docker**, hem après a crear imatges Docker segons les funcions desitjades: entrenar models o fer-los mitjançant APIs. A més, hem utilitzat Flask per construir l'API que serveix el model entrenat, coneixent així aquesta eina lleugera i flexible per al desenvolupament de serveis web. Flask ens ha permès definir rutes específiques, com la de `/model`, per interactuar amb el model i retornar resultats en format JSON. Doncs, ens ha donat una visió pràctica de com estructurar una API simple i eficient que pugui respondre a sol·licituds de clients externs. Per últim, hem entès la importància de seguir les bones pràctiques en la construcció d'imatges per reduir-ne la mida i augmentar la seguretat. Algunes de les bones pràctiques que hem dut a terme són:

1. Al crear els Dockerfiles, hem col·locat les instruccions que canvien amb menys freqüència al principi de l'arxiu per aprofitar la caché de les capes i reduir el temps de construcció. Per exemple, *from python o workdir ./2.1*.
2. A més, hem utilitzat COPY en comptes d'ADD per copiar els arxius ja que la primera és més predecible que la primera i ens permet un major control.
3. Respecte la documentació, hem afegit comentaris en els Dockerfiles per explicar les decisions del disseny i els passos importants.
4. Al crear el contenidor, hem fet servir el volum per emmagatzemar dades persistents i evitar la pèrdua de dades a l'eliminar contenidors.
5. Per tal de limitar els permisos, hem executat els contenidors mab l'usuari no root.

En la **desplegació de l'aplicació a Kubernetes**, ens hem familiaritzat amb els recursos principals Jobs, Deployments i Services. Cada recurs ens permet configurar aspectes específics per gestionar les aplicacions amb eficàcia. En el cas dels *Jobs*, ens han ajudat a automatitzar tasques d'entrenament i emmagatzemar el model. D'aquesta manera, ens hem assegurat que l'aplicació s'executa només una vegada i que compleix els requisits de memòria i CPU establerts. En el cas dels *Deployments*, hem après a desplegar i gestionar l'API del model en múltiples còpies. Aquestes rèpliques han garantit que, si una instància deixa de funcionar, les altres continuaran servint el model, assegurant que el servei es manté disponible. També ens ha permès actualitzar la nostra aplicació de forma controlada, gestionant quantes instàncies es renoven alhora. Finalment, *Services* ens ha proporcionat una forma segura d'exposar el servei, gestionant la comunicació de les rèpliques del Deployment.

Cal destacar que la utilització de *ConfigMaps* ens ha ensenyat com gestionar variables d'entorn de manera eficient i centralitzada, facilitant l'accés a les ubicacions dels models en els diferents recursos del clúster.

Conclusions

Al finalitzar aquesta pràctica, hem aconseguit desenvolupar i desplegar un servei funcional que exposa un model d'aprenentatge automàtic mitjançant una API, utilitzant Docker i Kubernetes, seguint unes bones pràctiques.

Durant el procés, hem après a crear imatges Docker i a gestionar-les adequadament en un entorn de contenidors. A més, hem aplicat els coneixements sobre *Flask* per construir una API que permet interactuar amb el model de manera senzilla i efectiva. Per últim, hem configurat correctament els recursos de Kubernetes (*Jobs*, *Deployments* i *Services*), garantint així la disponibilitat i escalabilitat del nostre servei.

La implementació de gestió de variables d'entorn (*ConfigMap*) ens ha permès assegurar que l'aplicació funcionés correctament. Doncs, la pràctica no només ens ha aportat coneixements sobre el desplegament d'aplicacions, sinó que també ens ha ajudat a comprendre la importància de l'estructura i desenvolupament del software.

Finalment, hem creat i desplegat una nova aplicació on s'hi prediu el valor de les dades amb un model diferent: *LinearRegression*. No només hem modificat els arxius *.py*, sino que hem adaptat els Dockers i els recursos de Kubernetes.

En conclusió, hem assolit els objectius de la pràctica. No només això, hem adquirit coneixements fonamentals en futurs projectes sobre dades i desenvolupament d'aplicacions.