INFORME PRACTICA 2

GESTIÓ D’INFRAESTRUCTURES PER AL PROCESSAMENT DE DADES - CONCEPTES DE XARXA I SERVEIS

Informe realitzat per: Iker Sánchez Catena i Grau Cladera Sensat

[1. Configuració de l’entorn 2](#_Toc114724444)

[1.1 Creació de model-train 3](#_Toc1969715063)

[1.2 Creació de la imatge model-server:default amb servei Flask 5](#_Toc1103020968)

[1.2.1 Creació Imatge 5](#_Toc1823574685)

[1.2.2 Servei Flask 7](#_Toc1690923563)

[2. Desplegar l’aplicació a Kubernetes 9](#_Toc152428605)

[2.1. Carregar les imatges 10](#_Toc1925771427)

[2.2. ConfigMap 10](#_Toc1733694388)

[2.3. Feina Job 11](#_Toc1176316231)

[2.4. Desplegament 13](#_Toc1302354108)

[2.5. Servei 16](#_Toc2140745923)

[2.6. Execució de l’aplicació 16](#_Toc811797987)

[3. Dificultats 17](#_Toc835963956)

[3.1. Problema amb arxiu DockerFile 17](#_Toc862611565)

[3.2. Problema amb disponibilitat dels ports 18](#_Toc1718475873)

[3.3. Problema al carregar les imatges a Minikube 18](#_Toc1768605904)

[3.4. Problema amb connexió MV 18](#_Toc7416667)

[3.5. Problema amb la xarxa 18](#_Toc522020416)

[4. Aprenentatge 18](#_Toc1036275619)

[4.1. Docker 18](#_Toc1327443694)

[4.2. Kubernetes 19](#_Toc1870011411)

[5. Accions per l’excel·lència 19](#_Toc495827433)

[5.1. Repositori Git 19](#_Toc1600625510)

[5.2. Canviar el model proporcionat 19](#_Toc885758249)

La pràctica s’ha dut a terme a l’usuari gixpd-ged-17 i la MV utilitzada té com a ID “Ubuntu - Practica 2”.

# 1. Configuració de l’entorn

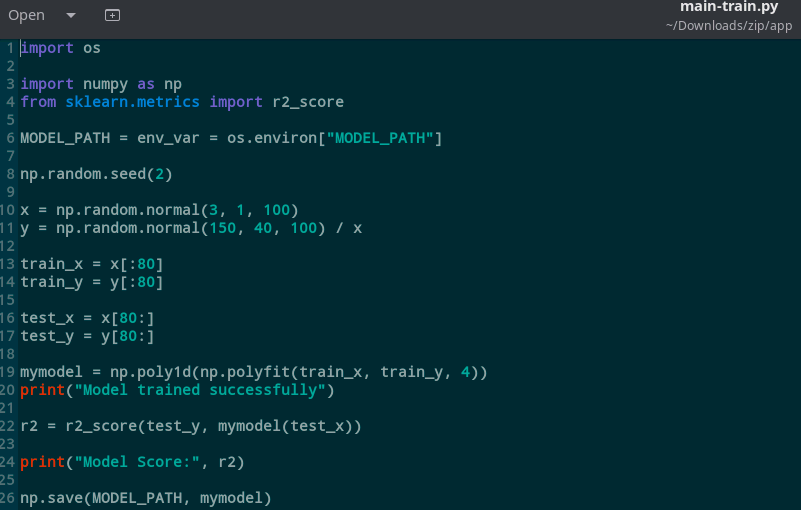
En primer lloc, hem realitzar la tasca 1 seguint les indicacions de les pàgines web indicades a la documentació, per instal·lar *Docker*, *Kubectl* i *Minikube*.

#### 1.1 Creació de model-train

Per crear la primera imatge hem creat una carpeta anomenada “app” i dins d’aquesta un arxiu “Dockerfile”. També hem afegit els arxius “main-train.py” i “requeriments-train.txt” del campus virtual.

Requirements-train.txt conté les llibreries que necessita l’arxiu main-train.py per executar-se correctament.

L’arxiu main-train.py conté el codi per entrenar el model.



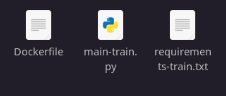
En primera instància, el codi donat crea una variable “MODEL\_PATH” que guarda l’adreça on es troba l’arxiu del model dins del contenidor des d’una variable d’entorn.

A continuació, fixem una llavor perquè les dades siguin pseudoaleatòries i creem dos arrays amb dades aleatòries: l’array *x* amb 100 valors amb distribució normal amb mitjana igual a 3 i desviació estàndard d’1 i, l’array *y* amb 100 valors amb distribució normal amb mitjana igual a 150 i desviació estàndard de 40 i dividit entre els valors de l’array *x*.

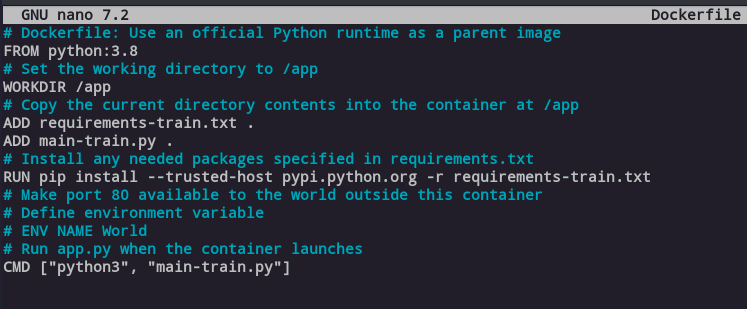
Després, dividim les dades d’entrenament i de test, amb un 20% de dades de test. Entrenem el model utilitzant un polinomi de quart grau i creem el model en base als coeficients obtinguts.

Seguidament, fem l’avaluació del model, calculant R² (valoració de la funció de regressió).

Finalment, guardem el model entrenat a la ruta especificada a la variable “MODEL\_PATH”.

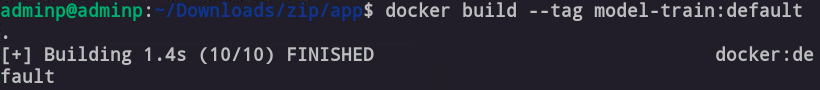


Un cop fet això, hem omplert l’arxiu “Dockerfile”:

Docker file train

A la primera línia de codi especifiquem que utilitzarem python:3.8. Seguidament, especifiquem el directori de treball dins del contenidor, aquest serà “/app”, i hi afegim el contingut del directori actual a dins d’aquest directori, és a dir, els dos arxius esmentats anteriorment. A continuació, instal·lem les dependències necessàries que es troben a l’arxiu “requirements-train.txt”. Amb l’opció “--trusted-host" permetem que “pypi.python.org” sigui una font confiable per evitar errors. Finalment, l’última línia és la comanda que farà que s’executi el script “main-train.py” quan el contenidor s'iniciï.

Un cop completats els passos anteriors, podem crear la imatge Docker.



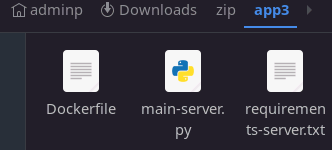
Explicació de la comanda:

* *Docker build*: s’utilitza per crear la imatge Docker.
* *--tag=model-train:default*: Assignem el nom de la imatge com “model-train” i especifiquem la versió “default”.
* *.* : Indica que la construcció de la imatge es farà en base al directori actual.

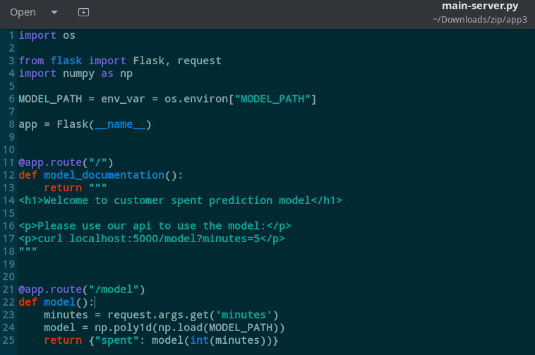
#### 1.2 Creació de la imatge model-server:default amb servei Flask

##### 1.2.1 Creació Imatge

Per la creació de la segona imatge necessitem els mateixos arxius que per la primera.



L’arxiu main-server.py és el següent:

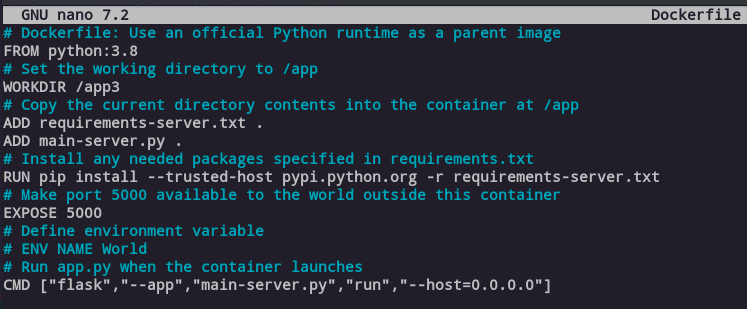


En primer lloc, el codi donat crea una variable “MODEL\_PATH” que guarda l’adreça on es troba l’arxiu del model dins del contenidor des d’una variable d’entorn.

En segon lloc, s’inicia l’aplicació Flask. Aquesta s’utilitzarà per respondre sol·licituds i definir adreces.

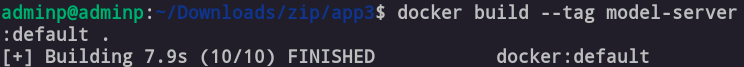
A continuació, la funció *model\_documentation()* que defineix l’adreça “/”. Retorna un missatge HTML explicant com es fa servir l’API. Al accedir al navegador a la adreça “127.0.0.1:5000” ens trobarem amb aquest HTML.

Finalment, definim el que es trobarà l’usuari si fica al navegador l’adreça “127.0.0.1:5000/model” amb la funció *model()*. En aquesta ocasió, es realitzarà una predicció amb un paràmetre que haurà de ficar l’usuari d’aquesta manera: “127.0.0.1:5000/model?minutes=5”.



En aquest “Dockerfile” només tenim dues diferències respecte al “model-train:default”. La primera és *EXPOSE 5000*, aquesta línia farà que el contenidor escolti a través del port 5000. La segona és la comanda que executarà el contenidor quan s'iniciï. Aquesta comanda té els següents paràmetres:

* “flask”: inicia el servidor *Flask*. *Flask* permet construir aplicacions web i APIs a través de Python
* “--app”: indica a *Flask* on es troba l’arxiu que conté l’aplicació *Flask, a l’arxiu “main-server.py”*.
* “main-server.py”: és el nom de l’arxiu on es troba l'aplicació *Flask*.
* “run”: inicia el servidor de desenvolupament. Ens permetrà accedir a l’aplicació web a través del port 5000.
* “--host=0.0.0.0”: paràmetre que indica a *Flask* quina adreça IP ha d’escoltar. En el nostre cas especifiquem que ha d’escoltar totes les adreces IP.



La comanda és similar a la que utilitzem per model-train amb una sola diferència:  
*--tag=model-server:default*: Assignem el nom de la imatge com “model-server” i especifiquem la versió “default”.

Després, comprovem que les imatges s’han creat correctament amb la comanda *docker image ls*, que llista les imatges Docker que es troben en el sistema.

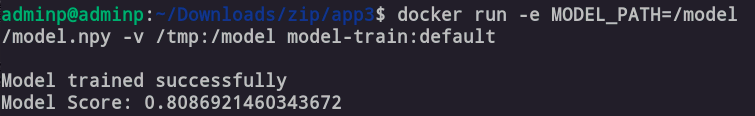
###### 1.2.2 Servei Flask

Podríem dir que Flask és un intermediari entre la API i l’usuari. En aquest cas, Flask accepta sol·licituds HTTP i retorna respostes segons els valors que li inclouen com a paràmetres (les accions que ha de dur a terme s’indiquen a main-server.py). En el nostre cas: /, /model i minutes. Flask realitza les crides corresponents a l’API i les retorna en format web, json o similar.

Per realitzar aquesta tasca cal que fem docker run de **train** i, tenint el model entrenat, realitzem un docker run de server. La comanda és la següent:

*docker run –e MODEL\_PATH=/model/model.npy -v /tmp:/model model-train:default*

* -docker run: comanda per executar un contenidor a partir d’una imatge
* -e: s’utilitza per establir variables d’entorn. En aquest cas, la nostra variable és MODEL\_PATH. Assignem a la variable el valor /model/model.npy, això indica a model-train, la direcció a on ha de guardar el resultat de l’execució.
* -v /tmp:/model: serveix per construir volums. Els volums serveixen per compartir informació entre el contenidor i el host. Fem que el directori del host */tmp* es munti al directori */model* del contenidor.

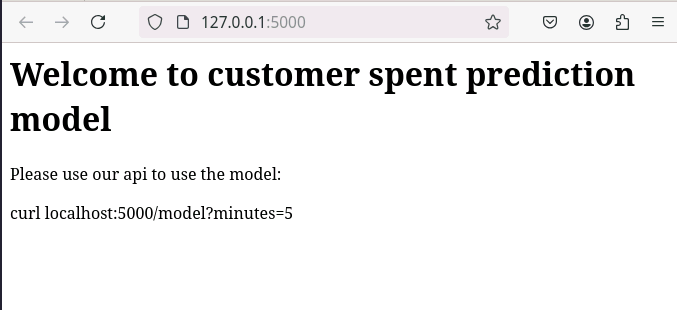


Pel cas de **server** la comanda és similar:

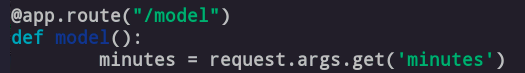
*docker run –e MODEL\_PATH=/model/model.npy -v /tmp:/model -p 5000:5000 model-server:default*

* *-e MODEL\_PATH=/model/model.npy: indiquem a Flask la direcció de l’arxiu a on es troben les dades del model entrenat perquè pugui mostrar informació i dades per la pàgina web.*
* *-v /tmp:/model: funció idèntica a l’altre comanda.*
* -p 5000:5000: amb –p permetem l’accés del host a un port del contenidor. A l’esquerra el port del sistema host i a la dreta el port de la imatge. Per defecte Flask escolta al port 5000 per això podem accedir al contingut del contenidor des de fora d’aquest.

La línia subratllada correspon a la petició per veure la pàgina de la direcció /model.

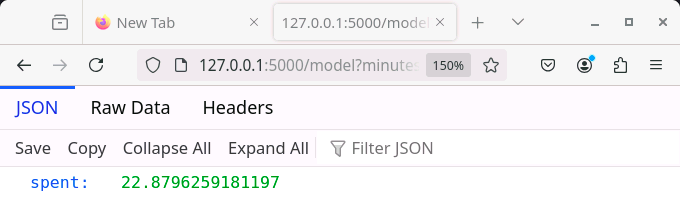
Sortida en format HTTP.

Per accedir a /model, cal que inserim el valor de la variable minutes.



La comanda a inserir al navegador és la següent:  
*127.0.0.1:5000/model?minutes=5*. Tal i com indica a la pàgina de /.

* ?: Indica l’inici dels query parameters de la URL. En aquest cas minutes=5

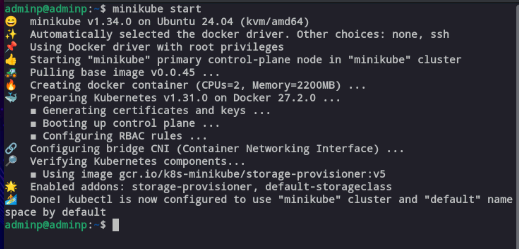
Resultat en format JSON.



Utilitzant la comanda curl

# 2. Desplegar l’aplicació a Kubernetes

En primer lloc, és necessari iniciar el servei minikube amb *minikube start.*



Vam tenir problemes a l’hora d’iniciar kubernetes ja que ens donava un error de connexió. Seguint les recomanacions del professor, vam fer un reboot a la màquina, però l’error continuava. Ho vam ignorar, perquè ens deixava pujar les imatges.tar. Tanmateix, a l’hora de pujar l’arxiu myconfig, ens deia que no era possible a causa de l’error anterior.

Després d’investigar, vam eliminar la imatge de kubernetes a Docker, per forçar-li a crear-la de nou amb una nova configuració, però res canviava. Finalment, vam eliminar tot el contingut de minikube:

*minikube delete*

*minikube start*

Tot arreglat.

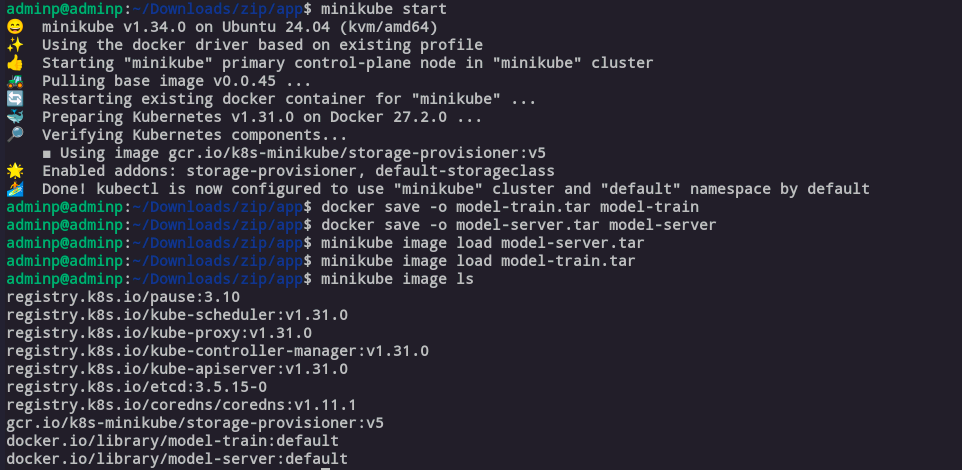
#### 2.1. Carregar les imatges

Seguidament reduïm la mida de les imatges, donat que son grans i ens pot donar error de timeout a l’hora de carregar-les.

Fem servir la següent comanda: docker save –o IMAGE.tar IMAGE



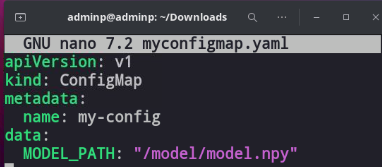
Ara carreguem les imatges amb *minikube image load IMAGE.tar* i amb la comanda *minikube image ls* comprovem que s’han carregat correctament:



#### 2.2. ConfigMap

L’arxiu configmap serveix per guardar els valors de certes variables no confidencials. D’aquesta forma, es millora l’escalabilitat de la nostra aplicació, evitant haver d’entrar en el codi per modificar valors.

En el nostre cas, tal i com es va indicar a la sessió de teoria, el contingut de configmap és el següent:



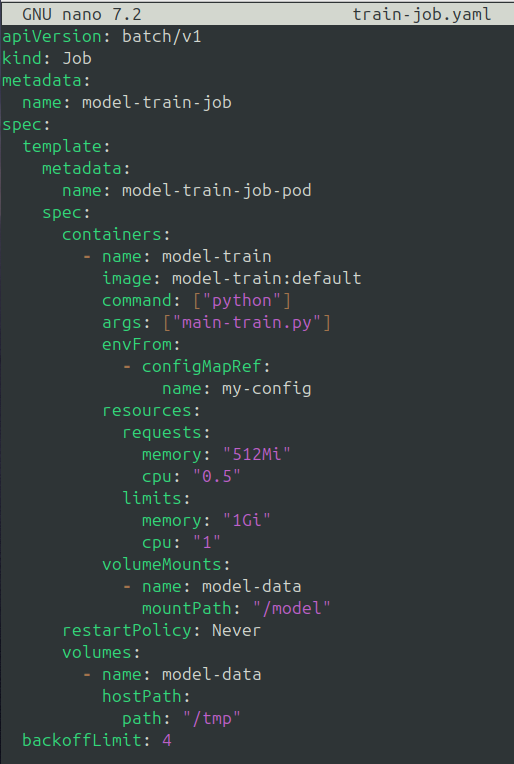
* ApiVersion: es refereix a la versió api de kubernetes
* Kind: especifiquem que es tracta d’un configmap
* Metadata: serveix per donar informació sobre l’arxiu. En aquest cas, indiquem que el seu nom és my-config
* Data: aquí incloem els valors de les variables. En el nostre cas, especifiquem la ruta de MODEL\_PATH: “/model/model.npy”. Els arxius que crearem més tard, job i deployment, utilitzaràn el valor de MODEL\_PATH especificat a myconfigmap per les seves execucions.

Amb la següent comanda kubectl apply -f myconfigmap.yaml apliquem la configuració de configmap al nostre clúster de kubernetes.

#### 2.3. Feina Job

Ha diferència dels Pods que estan dissenyats per executar-se contínuament, els arxius Job estan pensats per realitzar tasques que necessiten executar-se i terminar.

En el nostre cas, el Job entrena el model i el guarda. Un cop hagi acabat l’execució no tornarà a executar-se de nou.



Com a paràmetres d’interès trobem:

* kind: Job. Aquest és vital, perquè especifica que es un arxiu Job. Si no s'especifiqués no funcionaria.
* containers: especifiquem quin container ha d’executar, la seva imatge i la comanda i arguments per executar la app.
* envFrom: indiquem que les variables d’entorn es troben a my-config.
* volumeMounts: especifiquem el directori comú per als contenidors a on cal guardar el resultat de l’execució. D’aquesta forma hi podrem accedir des d’altres contenidors al resultat de l’execució.
* resources: indiquem als resources que pot accedir i els limitem.
  + Requests: indica la quantitat mínima de recursos que necessita el contenidor per funcionar
  + Limits: limita el nombre de recursos als que pot accedir el contenidor.
* restartPolicy: indica sota quines condicions kubernetes ha d’iniciar de nou el contenidor. Pel propòsit de la pràctica, no ens interessa que el procés s’executi múltiples vegades. Tanmateix, un valor que podria ser també adient seria OnFailure, que el tornaria a executar si es detectés qualsevol mena d’error.
* backoffLimit: especifica quants cops s’ha d’executar de nou el Job en cas d’error.

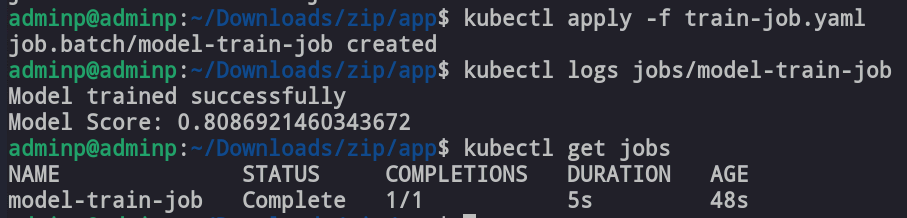
La diferència entre volumes i volumeMounts és la següent:

**volumes**: és a on es defineixen els volums, espais per emmagatzemar dades, que estaran disponibles per a l’arxiu concret en el que estiguem treballant.

**volumeMounts** especifica la ubicació a l’estructura d’arxius del contenidor on es muntarà el volum, permetent que el contenidor llegeixi i escrigui dades en aquella ruta.

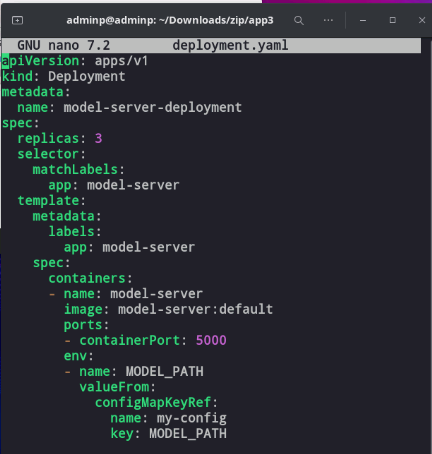
*kubectl apply –f train-job.yaml* crea el Job en el clúster segons la configuració definida dins aquest arxiu Job.

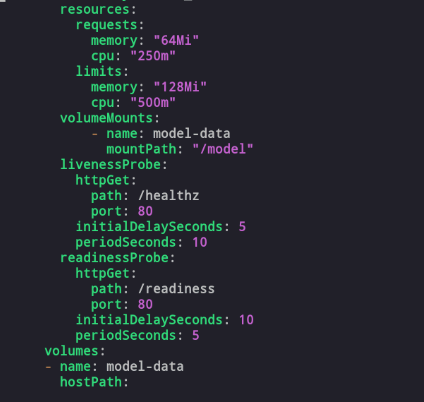
*kubectl logs* executa el job.



#### 2.4. Desplegament

Un arxiu deployment s’utilitza per gestionar y definir la implementació de les aplicacions en un clúster. Assegura que les aplicacions es puguin desplegar, actualitzar i escalar de forma controlada.

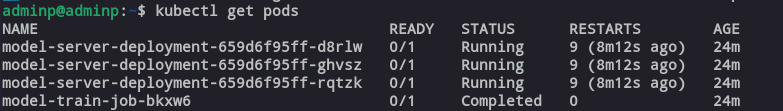






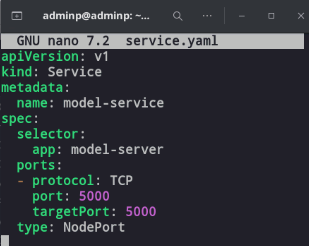
Els paràmetres d’interés són els següents:

* Kind: definim el tipus d’arxiu com a Deployment
* Spec: aquesta secció defineix les especificacions del Deployment.
  + Replicas: 3. Declarem la quantitat de repliques que han d’estar en execució alhora. És un eina per controlar la disponibilitat i per gestionar l’escalabilitat.
  + Selector: Indiquem de quina forma volem seleccionar la nostra aplicació. En aquest cas, amb “matchLabels” indiquem que volem gestionar tots aquells pods que tinguin l’etiqueta app amb el valor model-server
  + Template: definim cóm será la configuració i les característiques dels pods que creem i gestionem.
    - Metadata-labels-app: metadata conté les metadades pels pods. Indiquem que la etiqueta app ha de ser model-server
    - Spec-containers: en aquest apartat especifiquem:
      * -name: nom del contenidor
      * Image: especifiquem la imatge del contenidor que s’utilitzará per crear el pod.
      * Ports – containerPort:5000. El port a on escolta el contenidor.
      * Env: definim les variables d’entorn per al contenidor. És un procés idèntic per a tots els arxius. Indiquem que la variable MODEL\_PATH la trobarà a un confimap específic, en aquest cas my-config.
    - LivenessProbe: proba per saber si el contenidor està operatiu.
      * HhtpGet: utilitzem una solicitud HTTP per verificar si el contenidor està viu.
        + Path: /healthz. Ruta a on fem la solicitud.
        + Port: 80. Port a on fem aquesta solicitud.
      * InitialDelaySeconds: temps d’espera abans de realitzar la primera comprobació. Donem temps a que tot es pugui iniciar correctament.
      * PeriodSeconds: 10. Frecuencia de les comprovacions.
    - ReadinessProbe: configuració per la proba de preparació del contenidor.
      * httpGet: verifica si el contenidor està preparat per rebre trànsit a través de la xarxa.
        + Path: /readiness. Ruta a on fem la verificació
        + Port: 80. Port a on fem la verificació.
      * InitialDelaySeconds i periodSeconds tenen la mateixa funció que per LivenessProbe.
    - Volumes: definim els volums que estaran disponibles pel pod.

Podem comprovar que s’han creat 3 repliques del nostre arxiu model-server-deployment.

#### 2.5. Servei

Un arxiu service exposa una aplicació que s’estigui executant en el clúster amb un punt final cap a l’exterior.



Els paràmetres d'interès són els següents:

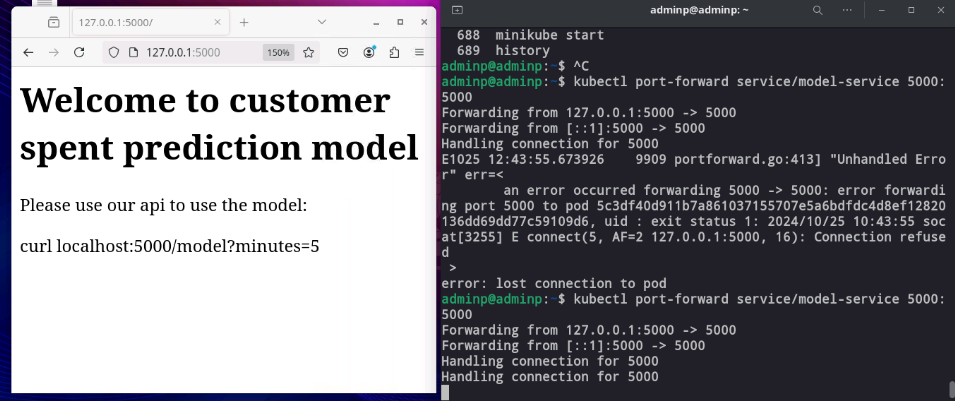
* Kind: definim el tipus d’arxiu com a Service.
* spec: aquesta secció defineix les especificacions del Service.
  + selector: busca els pods amb l’etiqueta “model-server” i dirigeix el tràfic a aquests pods.
  + ports: Utilitzem el protocol TCP. Accés intern pel port 5000 (port) i escolta el port 5000 dins del pod (targetPort). K8s redirigirà el tràfic de “port” a “targetPort”.
  + type: NodePort exposa el servei en un port de cada node del clúster.

#### 2.6. Execució de l’aplicació

Un cop aplicats tots els fitxers de kubernetes especificant la configuració necessària, procedim a provar si funciona correctament. Per dur a terme aquesta comprovació executem la següent comanda:

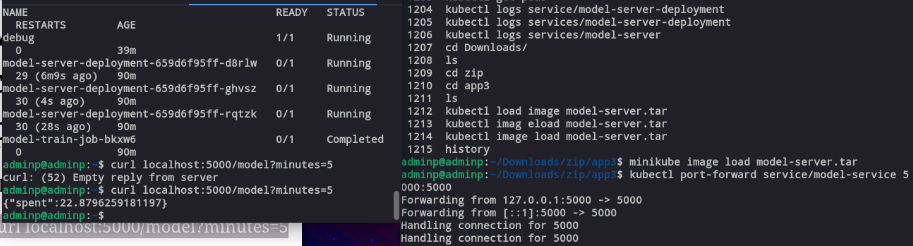
Explicació de la comanda:

* “kubectl”: eina de Kubernetes per gestionar i operar aplicacions dins del clúster.
* “port-forward”: permet reenviar ports d’un recurs de k8s a un port de la màquina local.
* “service/model-service”: especifiquem el recurs a enviar. En aquest cas el nostre servei executat amb anterioritat.
* “5000:5000”: Definim els ports d’enviament, el primer valor és el port de la màquina local i el segon el del servei Kubernetes al que volem accedir.



En executar aquesta comanda, ens indica que la connexió s’ha establert correctament i Kubernetes es queda escoltat al port 5000. Un cop accedim a l’aplicació Flask des del navegador a través del port 5000, kubectl ens notifica amb un missatge dient que està gestionant la connexió per aquest port. Al nostre cas surt dos cops, ja que hem accedit dos cops a l’adreça.

A la part del navegador, podem veure com hem accedit a la mateixa website al que vam accedir a través de Docker, però ara amb Kubernetes.



# 3. Dificultats

En la realització d’aquest exercici ens hem trobat amb dificultats a causa de problemes tècnics i de falta de coneixement.

#### 3.1. Problema amb arxiu DockerFile

A la versió inicial, a l’hora d’indicar els continguts del directori fèiem: ADD /<directori\_arxius> . Aquesta comanda copia tots els arxius del directori en el directori de treball del contenidor.  
Tanmateix, a causa de cert motiu que desconeixem, ens donava errors de que no trobava la carpeta. Per arreglar-ho, vam canviar la sintaxi i ara afegim cada arxiu individualment: ADD <nom de l’arxiu>.

#### 3.2. Problema amb disponibilitat dels ports

En moltes ocasions, quan provàvem a executar quelcom en un port determinat, l’execució donava errors que venien provocats pel fet que el port que li havíem indicat estava en ús per un altre aplicació. Donat la nostra poca experiència, al principi pensàvem que era problema nostre i que havíem fet quelcom malament, però no era el cas. Per resoldre’l teníem dos opcions: fer un reboot de la MV o bé canviar els ports de treball de les nostres execucions.

#### 3.3. Problema al carregar les imatges a Minikube

En el moment de carregar les imatges a minikube, cal iniciar el servei de minikube prèviament. En el nostre cas, ho havíem fet, però per motius externs a nosaltres vam haver de fer reboot. De nou a la màquina, sense fer “minikube start” vam carregar les imatges. En aquesta ocasió, no dona cap error, per tant vam prosseguir amb les següents passes, fins que poc desprès ens vam adonar que el servei de minikube no estava actiu i per tant no havíem carregat les imatges en lloc.

#### 3.4. Problema amb connexió MV

Durant la creació de l’arxiu deployment.yaml, va sorgir alguna mena de problema que va provocar la pèrdua del contingut del nostre arxiu deployment.yaml. Tot i haver guardat l’arxiu prèviament, el contingut sencer es va perdre. Com a conseqüència vam haver de reescriure de nou tot l’arxiu.

#### 3.5. Problema amb la xarxa

Quan intentàvem executar qualsevol comanda de kubectl, ens donava un error de Timeout, a on deia que no havia pogut connectar-se a una IP en concret. Després de consultar-ho amb el professor de pràctiques, vam arribar a la conclusió de que es tractava d’un problema aliè a nosaltres. Això ens va impedir treballar en la pràctica fins el dia següent.

# 4. Aprenentatge

#### 4.1. Docker

En aquesta pràctica hem aprés com crear un contenidor i tots els elements necessaris per al seu bon funcionament, com funciona i quins usos té. Hem aprés molt sobre la forma en la que s’organitzen els directoris de treball per als contenidors: les formes de guardar les dades que produeixen i com podem compartir aquesta informació amb altres contenidors. Com que els contenidors creen la seva pròpia infraestructura, invisible per al SO amfitrió, els permet executar-se en qualsevol entorn amb el mateix resultat sempre.

Hem descobert Flask, per crear una API amb URL i hem vist com integrar-ho en un contenidor per poder crear una aplicació de servei amb Flask.

Finalment, les bones pràctiques de Docker, que són aplicables a molts camps, i que consisteixen en crear imatges lleugeres i segures, optimitzant les instruccions per reduir els temps de construcció i a on es clau utilitzar variables d’entorn per les configuracions millorant la seguretat i el manteniment dels contenidors.

#### 4.2. Kubernetes

Primer entorn de producció amb el que treballem. Hem aprés el seu funcionament i totes les parts necessàries per poder fer que l’entorn sigui escalable i fàcil de mantenir.

Pod, cada pod conté un o més contenidors, comparteixen recursos i s’executen en el mateix entorn, permetent una comunicació molt fàcil entre ells.

Configmap, que permet emmagatzemar valors i configuracions per facilitar el manteniment sense haver de modificar cada imatge dels contenidors.

Job, que s’utilitza per executar tasques úniques, no repetitives.

Deployment, controla la creació i actualització de rèpliques dels pods. Permet gestionar l’escalabilitat del sistema.

Service, actúa com punt d’accés per als pods, equilibra la càrrega de treball. Ofereix un punt d’accés únic a l’aplicació.

5. Accions per l’excel·lència

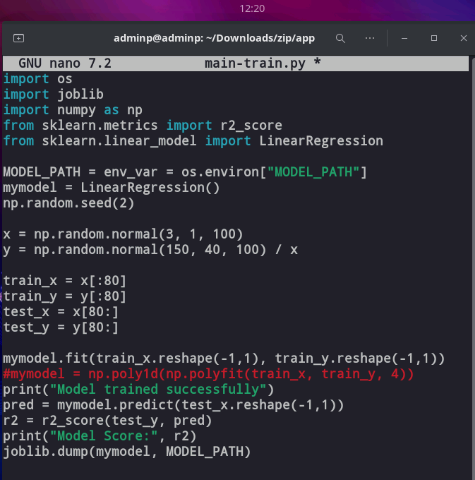
#### 5.1. Repositori Git

Hem creat el següent repositori <https://github.com/Gcladera/GIxPD-Practica2.git> a on es mostra el nostre treball i les versions finals dels nostres arxius.

#### 5.2. Canviar el model proporcionat

En comptes de fer una regressió polinòmica fem una regressió lineal. El resultat que obtenim indica que el nostre model és menys precís que l’original.

Hem modificat l’arxiu main-train.py



Els canvis han estat:

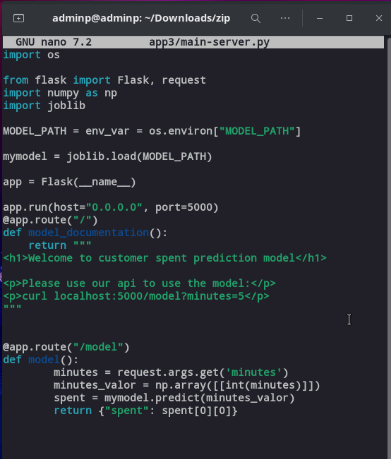
- Importació llibreria joblib

- mymodel = LinearRegression()

- L’entrenament del model amb mymodel.fit. Ha calgut fer reshape de train\_x i train\_y perquè la funció espera matriu.

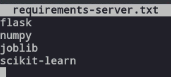
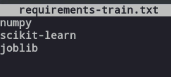
- La forma en com es guarda el resultat del model. Utilitzem joblib.dump()

main-server.py també ha estat modificat:

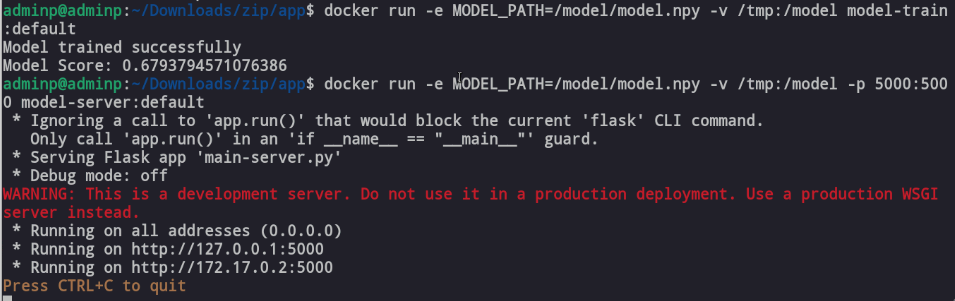


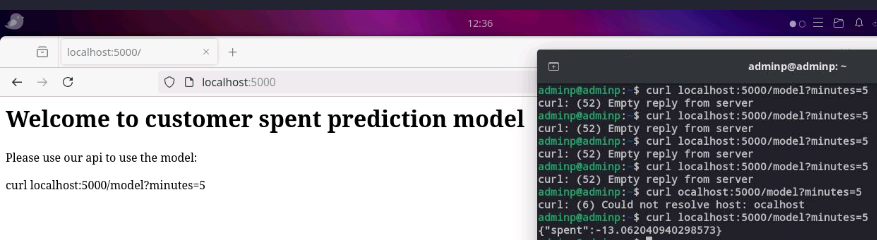
* Importem la llibreria joblib
* Canviem la forma de carregar les dades del model perquè correspongui amb el format amb el que guarda les dades main-train.py. Utilitzem joblib.load().
* A la funció model(), el format del resultat és diferent per tant cal que guardem el resultat d’entrada com un array, el predim i el mostrem per pantalla.

Els arxius requirements.txt tant per main-server.py com per main-train.py han estat modificats afegint la línia “joblib” per poder importar la llibreria dins el contenidor.

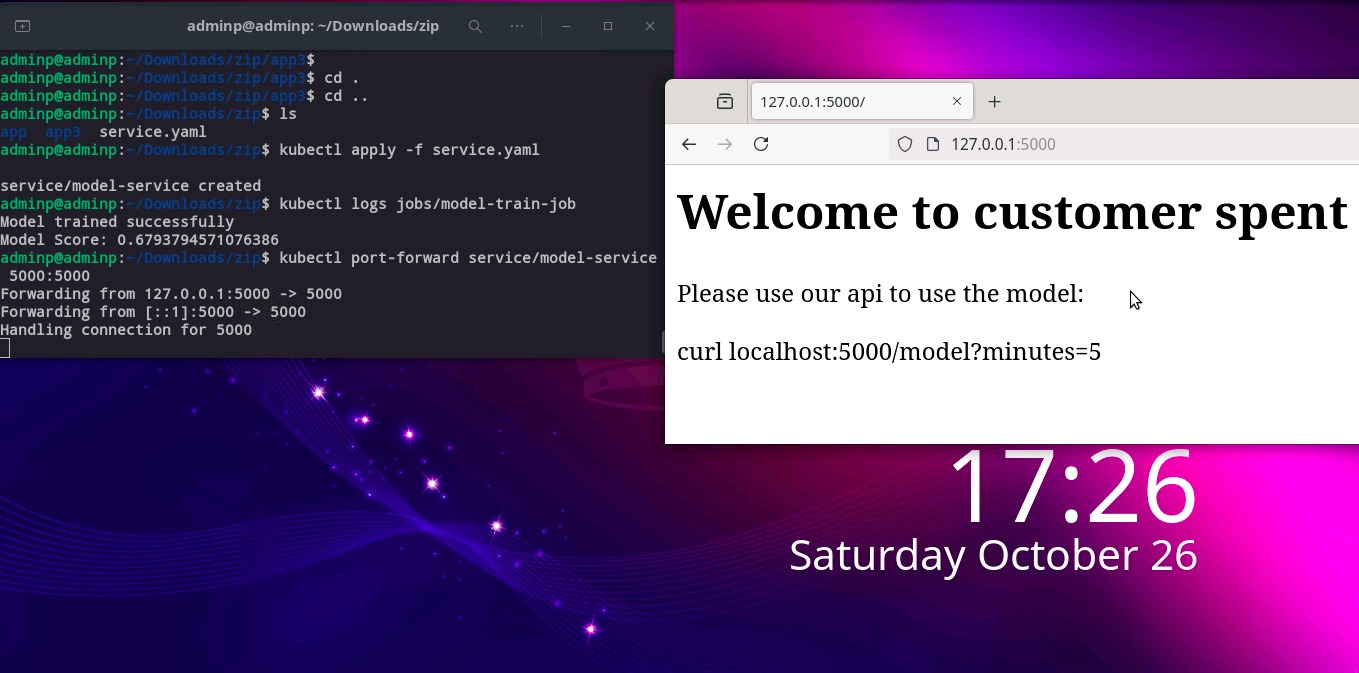
 

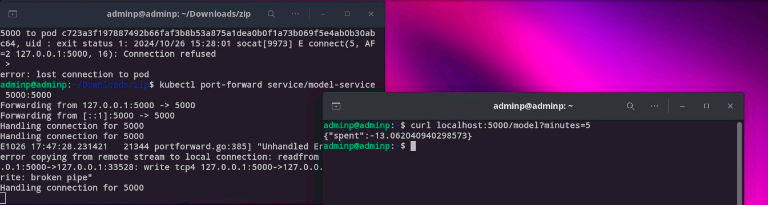
Comprovem que el nostre model funciona:

Obté un resultat de Model Score inferior, però això té sentit, donat que amb regressió lineal no podem aproximar tan bé les dades com amb regressió polinòmica.

main-server.py també funciona correctament.

Desprès de crear les imatges docker, convertir-les a .tar i pujar-les a kubectl, el resultat és el següent:





En un l’error de “broken pipe error”. Això vol dir, que en algun moment hi ha hagut una interrupció entre el client i el servidor. Tanmateix, kubectl port-forward intenta reconnectar automàticament quan detecta aquest tipus d’error.

Donat que en altres execucions no ens funcionava l’execució a causa de problemes amb la disponibilitat del port 5000, creiem que aquest error pot ser provocat pel mateix motiu, és a dir, un altre procés que estigui intentant utilitzar el port 5000.