Módulo 3: ML in Production Environments

Packaging ML Models

Why packaging matter

Los modelos de ML deben ser empaquetados y desplegados en un ambiente de producción; estos deben ser compatibles con diferentes sistemas y fáciles de desplegar. Algunas formas de empaquetar son:

- Serialización: simple, ligero e independiente del lenguaje.
- Empaquetado del ambiente: captura todo el ambiente del software
- Contenedorización: un ambiente aislado, portable y reproducible

How to package ML models

Serialización es el proceso de convertir un modelo de ML a un formato en el que pueda ser almacenado y recuperado.

El empaquetado del ambiente involucra crear un ambiente consistente y reproducible en el cual el modelo pueda ser ejecutado.

La contenedorización involucra empaquetar el modelo, sus dependencias y el ambiente en el que este se ejecuta en un único contenedor para que sea fácil de ejecutar.

Serializing scikit-learn models

Serializing an sklearn model using **pickle**:

```
import pickle

model = ...  # Train the scikit-learn model

# Serialize the model to a file
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)

# Load the serialized model from the file
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)
```

Serializing an sklearn model in **HDF5** format:

```
import h5py
import numpy as np
from sklearn.externals import joblib

model = ...  # Train the scikit-learn model

# Serialize the model to an HDF5 file
with h5py.File('model.h5', 'w') as f:
    f.create_dataset('model_weights',
    data=joblib.dump(model))

# Load the serialized model from the HDF5 file
with h5py.File('model.h5', 'r') as f:
    model = joblib.load(f['model_weights'][:])
```

Serializing PyTorch and Tensorflow models

Serializing a PyTorch model:

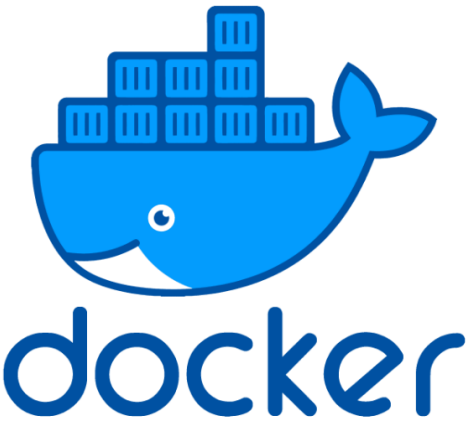
```
import torch

# Train a PyTorch model and store it in a variable
trained_model = ...

# Serialize the trained model to a file
serialized_model_path = 'model.pt'
torch.save(trained_model.state_dict(), serialized_m

# Load the serialized model from a file
loaded_model = ... # Initialize the model
loaded_model.load_state_dict(
    torch.load(serialized_model_path))
```

Serializing a Tensorflow model:

```
import tensorflow as tf

# Train a Tensorflow model
trained_model = ...

# Save the trained model to a directory
saved_model_directory = 'model/'
tf.saved_model.save
    (trained_model, saved_model_directory)

# Load the saved model from the directory
loaded_model = tf.saved_model.load(
    saved_model_directory)
```

ML environment packaging with Docker

- Ensure the model's environment lets it run
- conda or virtualenv create consistent and reproducible environments
- Docker containers are self-contained units that are easily deployable



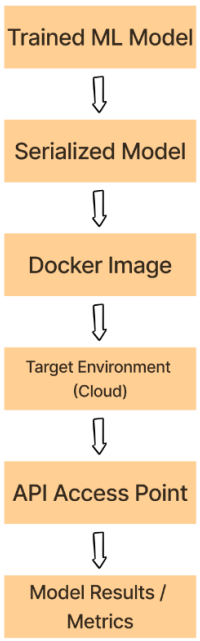
Example Dockerfile

```
# Use an existing image as the base image
FROM python:3.8-slim

# Set the working directory
WORKDIR /app

# Copy the requirements file to the image
COPY requirements.txt .

# Install the required dependencies
RUN pip install -r requirements.txt

# Copy the ML model and its dependencies to the image
COPY model/ .

# Set the entrypoint to run the model
ENTRYPOINT ["python", "run_model.py"]
```

```
<---- Use Python 3.8 base image

<---- Set the working directory

<---- Copy the requirmentes.txt file

<---- Install the model's dependent packages

<---- Copy the model into the continer

<---- Tell the container how to start up
```

Experiment -> Docker workflow

1. **Serialize** the trained ML model using a format such as pickle, HDF5, or PyTorch.
2. **Containerize** the serialized ML model, its dependencies, and the environment
3. **Deploy** the Docker image to a target environment such as a cloud platform
4. **Run** the Docker container from the deployed Docker image and run the ML model.



Scalability

La escalabilidad es importante para asegurar que el modelo de ML es capaz de manejar conjuntos de datos más grandes y complejos a medida que se incrementa su uso

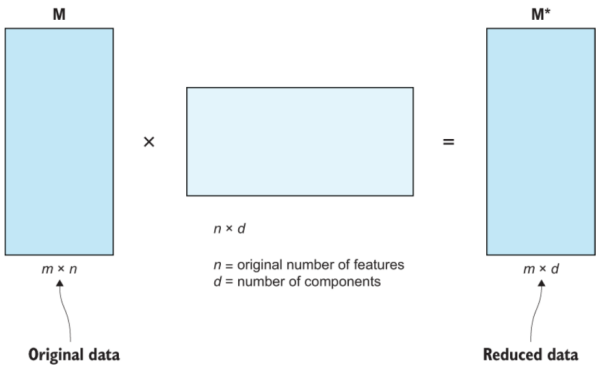
Compute constraints in scalability

CPU, memoria, capacidad de almacenamiento, entre otros, puede impactar la escalabilidad del modelo de ML. Si el modelo necesita más recursos de los disponibles entonces es posible que se vuelva lento, haciendo difícil usarlo en aplicaciones de la vida real. Medir el uso de CPU, memoria y del disco duro durante el entrenamiento y las inferencias ayuda a identificar las restricciones de compute, facilitando la escalabilidad.

Model complexity and scalability

La complejidad del modelo afecta su escalabilidad, pues a mayor complejidad, mayor uso de recursos computacionales.

- Model complexity affects scalability
- Balancing complexity and scalability is challenging
- Strategies for balancing:
  - Feature selection techniques (e.g. Chi-squared, PCA)
  - Model compression techniques (e.g. pruning)



Principal Component Analysis (PCA)

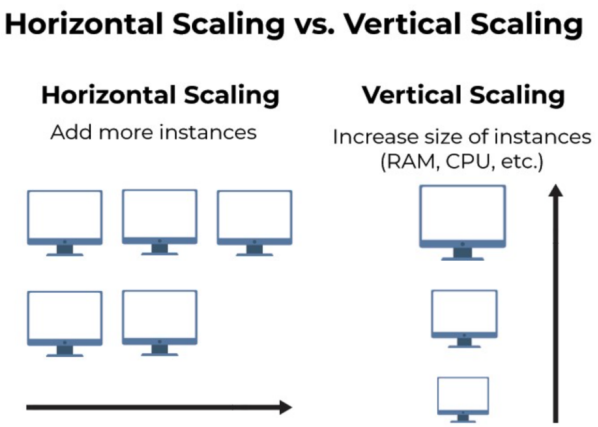
Velocity of deployments and scalability

La velocidad del despliegue mide qué tan rápido se integran los modelos actualizados y es un factor clave para la escalabilidad.

- Strategies
  - continuous integration/deployment - covered in the next lesson
  - online learning - learning with new data in real time

Optimal scaling strategies

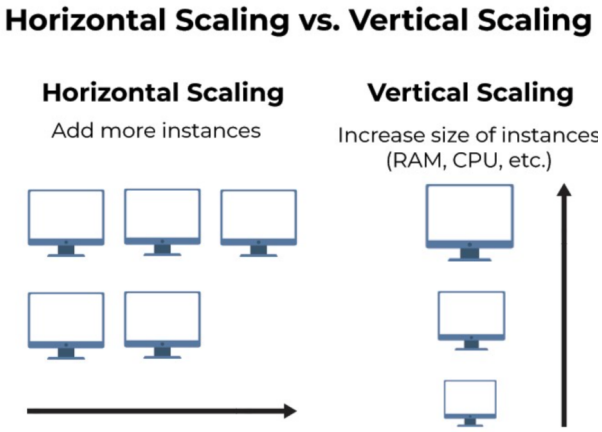
- Trade-offs include cost of serving, re-training, and velocity of deployments
- Scaling strategies: horizontal scaling, vertical scaling, auto-scaling
  - Horizontal scaling: adding more machines
  - Vertical scaling: increasing machine size
  - Auto-scaling: adjusts number of machines based on workload



Horizontal vs. VerticalScaling

- Horizontal scaling uses load balancing, partitioning
- Horizontal scaling trade-offs: complexity, cost ($)
- Vertical scaling increases machine size
- Vertical scaling trade-off: cost ($)

Ideally - auto-scale horizontally and vertically.



Automation

Automatización es un componente clave de MLOps que asegura tanto la confiabilidad como la eficiencia de los pipelines de ML, reduciendo los riesgos de errores humanos.

Los cuatro principios de MLOps son:

- Continuous Integration CI
- Continuous Delivery CD
- Continuous Training CT
- Continuous Monitoring CM

Continuous Integration (CI) Integrating code changes into a shared repository regularly

Continuous Delivery (CD) Automatically building, testing, and deploying code changes

Continuous Training (CT) Continuously training & updating the model with new data

Continuous Monitoring (CM) Monitoring model performance and accuracy on an ongoing basis

Continuous Integration

- Asegura que el código siempre esté en un estado funcional
- Reduce el riesgo de errores humanos
- Encuentra errores de forma temprana

Continuous Delivery

- Asegura que los modelos se despliegan de forma rápida y consistente
- Reduce el tiempo necesario para llevar nuevos modelos a producción
- Reduce el riesgo de errores humanos

CI/CD tools: Git, AWS CodePipeline, Jenkins, Travis CI

Jenkins

Continuous Training

- Asegura que el modelo es preciso y está actualizado
- Reduce el riesgo de un deterioro del modelo
- Reduce el tiempo de reentrenamiento

Continuous Monitoring

- Reduce el riesgo de un deterioro del modelo
- Mejora el desempeño global
- Proporciona acceso a métricas consistentes y confiables del modelo

Example of ML automation at scale

1. CI: The code for the model is committed to Git.
2. CD: The committed code is built and tested using a CI/CD tool like Jenkins. If they pass, we deploy.
   - The model is serialized.
   - Dependencies are set using Docker.
   - The Docker image is deployed.
3. CM: Model performance is continuously monitored.
   - Monitoring informs decisions about the model.
   - CM tools include Prometheus & Grafana
4. CT: The model is trained on new data.
5. New code is written / models are updated ... back to step 1