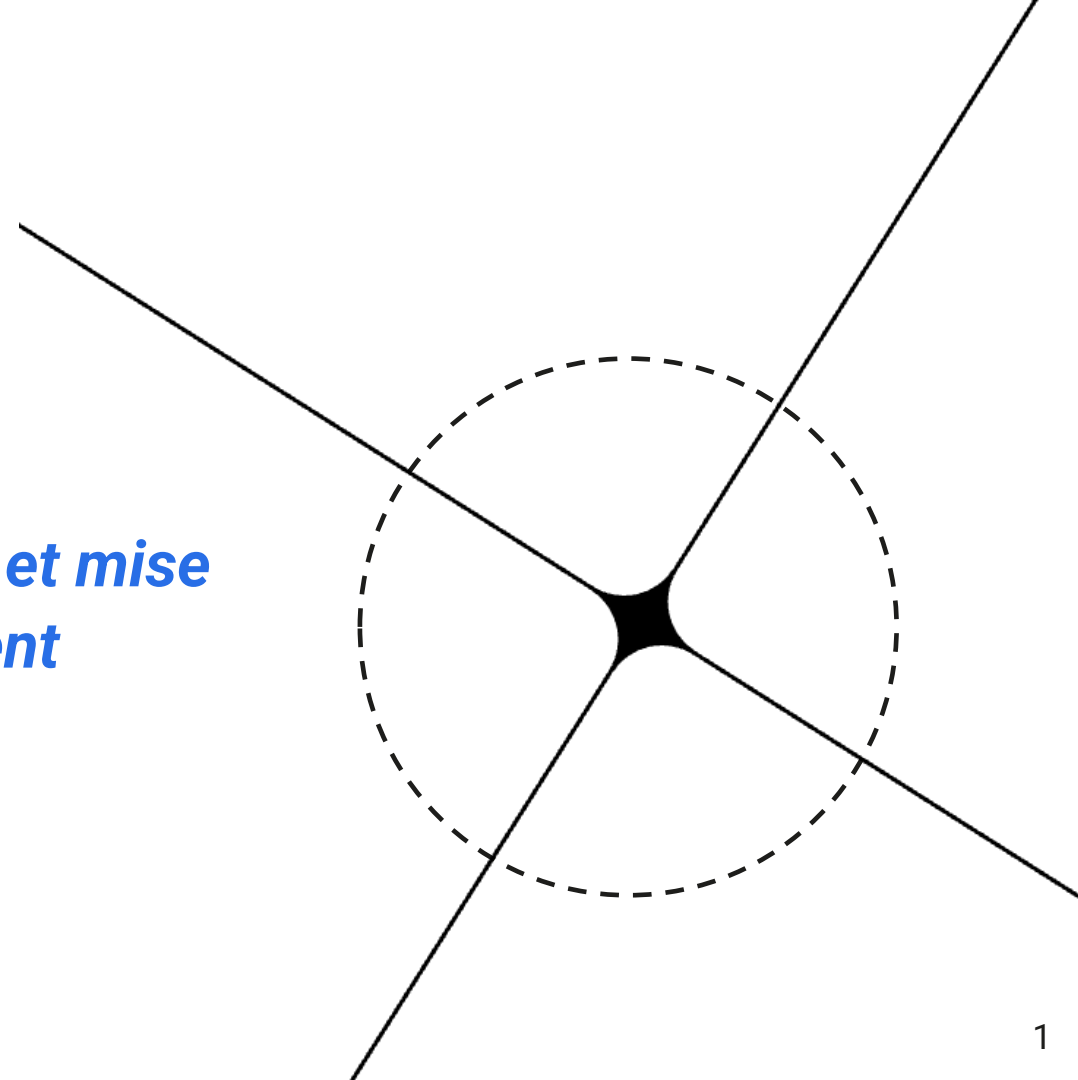
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Fondamentaux du MLOps et mise en place de l'environnement

Julien Romero



Vue d'ensemble du cours

- 6 cours où nous allons apprendre à créer un système pour le machine learning de bout en bout
  - Des données brutes au réentraînement automatique
- Nous allons voir des outils utiles à divers niveaux pour le système
  - Prefect, Feast, MLflow, FastAPI
  - Docker, Docker Compose
  - Prometheus, Grafana
  - Evidently, Great Expectations
  - PostgreSQL
- Évaluation
  - 2 points sur des quizz
  - 6 points sur des comptes rendus de TP
  - 12 points sur un contrôle final sur table

Pourquoi le MLOps ?

- Le machine learning en production est différent du machine learning dans un notebook
- Problèmes classiques:
  - La distribution des données change avec le temps
  - Différence entre les données d'entraînement et d'inférence
  - Manque de versionnage (données, code, modèles)
  - Environnement d'exécution flou et conflit de dépendances
  - Pas de monitoring
- Le MLOps (ou systèmes pour le machine learning) vise les objectifs suivants :
  - Reproductibilité
  - Automatisation
  - Monitoring
  - Scalabilité

Exemple d'un cas d'erreur

- Vous entraînez un modèle sur votre machine, mais les prédictions en production sont mauvaises. Cela peut venir :
  - Une différence de calcul des features entre l'entraînement et la production
  - Préprocessing non synchronisé entre les deux environnements
  - Données manquantes ou arrivant trop tard à l'inférence
  - Différents environnements (OS, bibliothèques)
  - Pas de monitoring: les erreurs passent inaperçues

Déployer un modèle est plus difficile qu'il n'y paraît

Le cycle de vie d'un système pour le machine learning

Un système moderne de machine learning n'est pas un script, mais une pipeline:

1. Ingestion des données
2. Validation des données
3. Calcul des features
4. Entraînement du modèle
5. Suivi des expériences
6. Évaluation
7. Sauvegarde du modèle et code associé
8. Déploiement en production
9. Mise à disposition du modèle
10. Monitoring
11. Détection du drift
12. Ré-entraînement
13. Déploiement continu

Le cycle de vie d'un système pour le machine learning

Un système moderne de machine learning n'est pas un script, mais une pipeline:

1. Ingestion des données
2. Validation des données
3. **Calcul des features**
4. **Entraînement du modèle**
5. **Suivi des expériences**
6. **Évaluation**
7. Sauvegarde du modèle et code associé
8. Déploiement en production
9. Mise à disposition du modèle
10. Monitoring
11. Détection du drift
12. Ré-entraînement
13. Déploiement continu

Étapes vues jusqu'à présent (du moins en partie)

Système ML = logiciel + données

- Quand vous créez et maintenez un système pour le ML, vous devez gérer :
  - La complexité logicielle : APIs, conteneurs, qualité du code
  - La complexité des données : qualité, versionage, recalcul, fuite de données
  - L'infrastructure : orchestration, reproductibilité, scalabilité
- Le MLOps et à l'intersection de ces trois domaines

L'architecture que nous allons construire

- Dans le TP, nous allons construire un système de bout-en-bout en utilisant :
  - PostgreSQL pour les données structurées
  - Prefect pour l'ingestion et les pipelines d'entraînement
  - Great Expectations pour la validation des données
  - Feast pour la gestion des features en ligne/hors ligne
  - MLflow for tracking + registry
  - FastAPI for serving
  - Evidently + Prometheus for drift & metrics
  - Docker Compose to run everything locally
- Maîtriser le système de bout en bout demande de maîtriser beaucoup d'outils !

Les problèmes de non reproductibilité

- La reproductibilité, c'est-à-dire la capacité de pouvoir obtenir de manière déterministe les mêmes résultats facilement, est cruciale pour la mise en production
- Sources de non-reproductibilité :
  - Différentes versions de Python
  - Différentes dépendance à l'OS
  - Pas de versions précises pour les bibliothèques
  - États cachés dans un notebook
  - "Ça marche sur ma machine"
  - Sources de données non-déterministiques
- Nous avons besoin de construire des environnements contrôlés et isolés
  - Solution : les conteneurs

Qu'est-ce qu'un conteneur ? Qu'est-ce que Docker ?

- Un conteneur est :
  - Un environnement d'exécution léger et isolé
  - Contient le code applicatif, des dépendences, les bibliothèques système
  - Partage le noyau de l'OS hôte
  - S'exécute de la même manière partout (Linux, Windows, Mac)
  - Idéal pour reproduire des pipelines de machine learning
- Docker est :
  - Le plus populaire moteur de conteneur
  - Construit des conteneurs à partir de Dockerfiles
  - Gère les images (conteneurs que l'on peut créer) et les conteneurs en cours d'exécution
  - Donne des outils pour :
    - Grouper tous les éléments de l'environnement
    - Exécuter des services isolés
    - Partager des images (DockerHub)
- En bref, un conteneur Docker est un environnement d'exécution de code cohérent et bien emballé

Pourquoi utiliser des conteneurs ?

- Les conteneurs assurent :
  - Le même environnement pour tout le monde
  - Un environnement d'exécution portable et autonome
  - Une isolation par rapport à l'OS
  - Reproductibilité des artéfacts et déploiements
  - Facilité de l'orchestration pour les architectures avec plusieurs services
- Docker joue un rôle fondamental dans ce cours

Conteneur vs machine virtuelle

- Une machine virtuelle :
  - Contient un OS complet
  - Très lourd
  - Lent au démarrage
  - Consomme beaucoup de ressources
- Un conteneur :
  - Utilise le noyau de l'OS hôte
  - Léger
  - Rapide au démarrage
  - Idéal pour les micro-services et les pipelines de machine learning

Image vs conteneur

- Dans Docker, il faut différencier une image d'un conteneur
- Une image :
  - Est un modèle immuable d'un environnement
  - Est construite à partir d'un Dockerfile (une sorte de recette)
  - Contient du code, des dépendances, une configuration, un système de fichier
  - Est comparable à une classe en orienté-objet
- Un conteneur :
  - Est une instance en cours d'exécution d'une image
  - A son propre système de fichier, ses propres processus, et son propre réseau
  - Peut être démarré, arrêté, redémarré
  - Comparable à un objet instancié en programmation orientée objet

Anatomie d'un Dockerfile

- Un DockerFile est un fichier décrivant comment construire un image. Typiquement, il contient :
  - Une image de base que l'on va modifier (ex: un linux, une image spécialisée pour Python, …=
  - Un dossier de travail
  - Des copies de fichiers dans l'image
  - L'installation des dépendances
  - La commande à exécuter au lancement d'un conteneur (ex: démarrer une pipeline ou un site web)
- Cela permet d'avoir un environnement reproductible pour exécuter du code
- Exemple:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

Commandes utiles d'un Dockerfile

- **FROM <image>**: Spécifie l'image de base. Ex: python:3.11-slim, ubuntu:22.04
- **WORKDIR <path>**: Définit le dossier de travail dans le conteneur. Les commandes suivantes s'exécuteront dans ce dossier
- **COPY <src> <dest>**: Copie des fichiers depuis l'hôte dans l'image, typiquement du code, le requirements.txt, des fichiers de config, **...**
- **RUN <command>**: Exécute une commande au moment de la création de l'image. Ex: pip install, apt-get install.
- **ENV <key>=<value>**: Définit des variables d'environnement. Utile pour les configurations par défaut.
- **EXPOSE <port>**: Documente le port utilisé par le conteneur, pour information uniquement.
- **CMD [...]**: Commande à exécuter au démarrage du conteneur. Ex:
  - CMD ["python", "main.py"]
  - CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
- **ENTRYPOINT [...]**: Similaire à CMD, mais n'est pas supprimé avec un docker run.

Les système de couches de Docker

- Une image Docker est construite par une superposition de couches
  - Chaque instruction du Dockerfile crée une nouvelle couche immuable
- Propriétés clefs :
  - **Cache de couches** : Si une couche n'a pas changé, Docker la réutilise
    - Builds rapide, mais ça prend de la place en mémoire
  - **Dépendance de haut en bas** : Changer une ligne/couche invalide le cache de toutes les suivantes
  - **Types de couches** : Couche de base (avec FROM), installation de packages (RUN), Copie de code (COPY), Configuration (ENV, EXPOSE, etc.)
  - **Immutabilité**: Les couches ne peuvent pas être modifiées après la création, on les réécrit
- **Optimisation**: Toujours mettre les commandes qui changent le plus à la fin.
  - Installation de bibliothèques Python vers le début = dans le cache
  - Copie du code à la fin = build plus rapides

Construire une image

- Pour construire une image, Docker va:
  1. Lire le Dockerfile (fichier appelé *Dockerfile* par défaut)
  2. Exécuter chaque instruction dans l'ordre
  3. Créer les couches
  4. Produire une image finale (nommée my-app, ici)
- Options utiles :
  - --no-cache : reconstruit tout
  - -f <Dockerfile> : Si le nom du Dockerfile ne s'appelle pas *Dockerfile*

```
docker build -t my-app .
```

Démarrer un conteneur

- Pour démarrer un conteneur à partir d'une image my-app

```
docker run my-app
```

- Options utiles :
  - -p 8000:8000 : Associe un port à l'intérieur du réseau Docker à un port sur votre machine hôte
    - Format port_hote:port_conteneur
  - --name api : nom du conteneur
  - -e VAR=value : injection d'une variable d'environnement
  - -v host:container : pour monter des volumes (voir plus tard)
- Exemples:
  - docker run -p 8000:8000 my-api
  - docker run --name db -e POSTGRES_PASSWORD=demo postgres

Interagir avec des conteneurs en cours d'exécution

- Commandes utiles :
  - docker ps : liste les conteneurs en cours d'exécution
  - docker ps -a : liste tous les conteneurs (dont ceux arrêtés)
  - docker logs <nom_conteneur> : Lit le logs d'un conteneur
  - docker exec -it <nom_conteneur> bash : ouvre un terminal bash dans un conteneur
  - docker stop <nom_conteneur> : Arrête un conteneur
  - docker rm <nom_conteneur> : Supprime un conteneur
- docker logs et docker exec -it sont souvent utilisées pour le debug

Le réseau pour les conteneurs

- Par défaut chaque conteneur est sur son propre réseau
- Les conteneurs peuvent exposer des ports qui sont accessibles depuis l'hôte (votre machine)
  - Les ports ne sont pas forcément les mêmes
- Exemple
  - docker run -p 5432:5432 postgres
  - Hôte: localhost:5432
  - Conteneur: listens on port 5432
- Nous allons pouvoir aller un peu plus loin (facilement) avec Docker compose (voir plus tard)

Les volumes : Pourquoi en a-t-on besoin ?

- Le système de fichier d'un conteneur est éphémère
  - Quand un conteneur s'arrête, ses fichiers propres sont supprimés
- Les volumes Docker permettent de :
  - Persister les données (ex, bases de données, logs, modèles)
  - Isoler le stockage (contrôlé par Docker)
  - Découpler le stocker du conteneur
- Nous avons deux types de volumes
  - Les volumes nommés : on donne un nom à un volume comme on donne un nom à une image
  - Les bind mounts : on peut lier un dossier sur l'hôte à un dossier dans le conteneur (utile pour le débug et pour partager du code)
- Exemple:
  - docker run -v db_data:/var/lib/postgresql/data postgres
  - Ici, le volume s'appelle db_data

Pourquoi a-t-on besoin de plus que Docker ?

- Dans un système de machine learning, nous avons besoin de nombreux services qui communiquent entre eux
  - API
  - Base de données
  - MLflow tracking
  - Feature store
  - Orchestrator
  - Monitoring
- Lancer tous ces services avec docker run est fastidieux, sujet à erreur, difficile à reproduire, et ne passe pas à l'échelle
- Nous avons besoin d'un outil pour définir les services et les orchestrer
  - **Docker compose**

Introduction à Docker Compose

- Docker compose est un outil pour définir des applications avec plusieurs conteneurs
- Docker compose permet de :
  - Définir plusieurs services dans un seul fichier YAML (docker-compose.yml par défaut)
  - Partager un réseau et la découverte de services
  - Partager des volumes de stockage
  - Facilement démarrer et éteindre notre système
  - Produire des environnement de développement cohérents
- Une seule commande va démarrer tout notre système

Anatomie d'un fichier docker-compose.yml

- L'exemple de droite :
  - Crée automatiquement un réseau
  - Les services sont accessibles par nom (ex: db)
  - L'ordre de déploiement peut-être spécifié
- Chaque conteneur est défini dans la section "services" et a :
  - Un nom (indentation 1)
  - Une image, soit construite avec un Dockerfile (build: ./api), soit une image existante (image: postgres:16)
  - Mapping des post (ports: "host:conteneur")
  - Variable d'environnement (environment: …)
  - Volumes pour stocker des fichiers
    - volumes: PATH_HOST:PATH_CONTENEUR
- On peut aussi définir des volumes nommés dans la section "volumes"
- De même, on peut définir plusieurs réseaux avec "networks" (un seul commun par défaut)

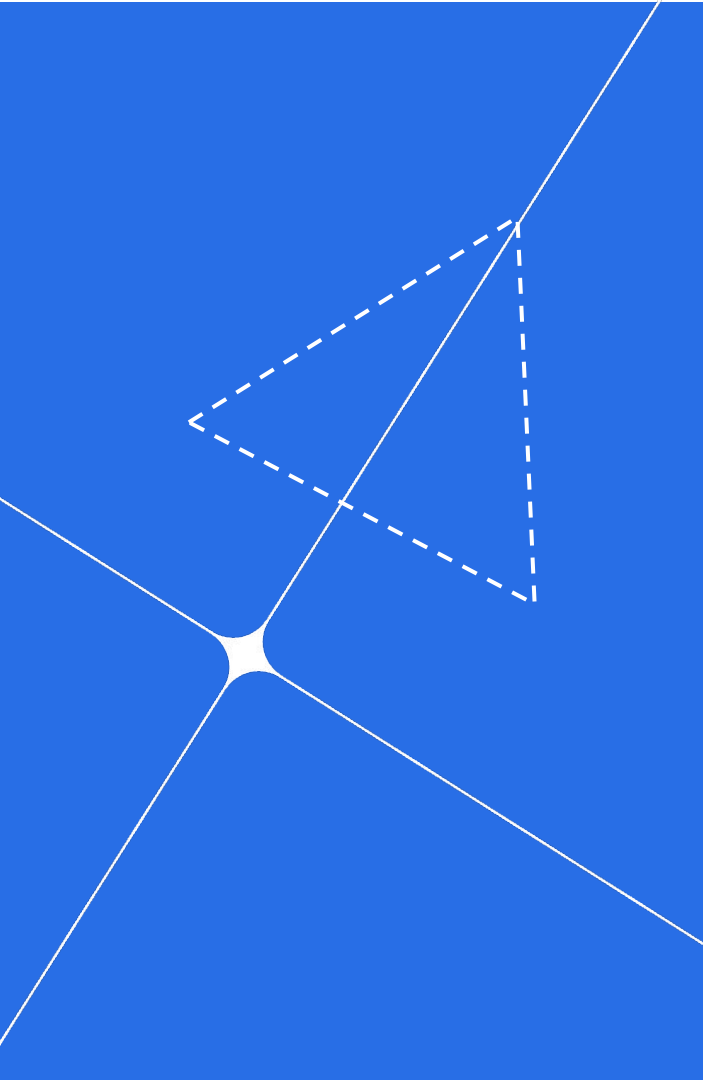
```
services:
  api:
    build: ./api
    ports:
      - "8000:8000"
    depends_on:
      - db

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: demo
      POSTGRES_DB: demo
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Le réseau de Docker compose

- Quand Docker compose démarre, il va :
  - Créer un unique réseau (par défaut)
  - Ajouter au réseau tous les services (comportement par défaut)
  - Rendre accessible chaque service uniquement avec son nom
- Par exemple, l'api peut se connecter à la base de donnée avec
  - db:5432 (pas besoin d'avoir l'adresse IP)

Commandes classiques de Docker Compose

- Démarrer les services : *docker compose up -d* (lit par défaut docker-compose.yml)
- Arrêter les services : *docker compose down*
- Voir les logs : *docker compose logs -f api*
- Lister les services en cours d'exécution : *docker compose ps*
- Redémarrer un service : *docker compose restart api*



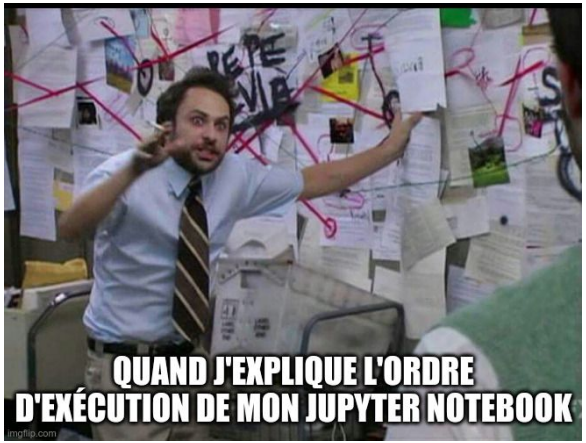
Le projet des TP

Le système complet que nous allons construire

- Composantes principales et rôle :
  - **Postgres** : stockage des données brutes et pré-traitées
  - **Prefect** : exécution des scripts d'ingestion de données et d'entraînement de modèle
  - **Great Expectations** : validation de la qualité des données
  - **Feast** : gestion des features hors ligne et en ligne
  - **MLflow** : suivi des expériences et gestion des versions de modèles
  - **FastAPI** : accès au modèle pour la prédiction en temps réel
  - **Prometheus / Grafana** : monitoring du système et des métriques
  - **Evidently** : Détection du drift dans les données et les prédictions
- Tout sera géré par un seul Docker Compose, qui permettra de simuler un cloud sur votre machine
  - En pratique, plusieurs machines, potentiellement avec Kubernetes
- Beaucoup d'outils à maîtriser en MLOps !

Pourquoi les notebooks ne sont pas suffisants

- Les notebooks sont très utiles pour :
  - L'exploration et la visualisation
  - Le prototypage rapide d'idées
- Cependant, ils sont insuffisants en production
  - Création des états cachés, non reproductible
  - Pas de contrôle sur l'environnement
  - Pas automatisable
  - Pas testable
  - Pas d'intégration avec les features stores, le déploiement
  - Difficile à monitorer ou à logger
- La production demande du code structuré (génie logiciel), des pipelines, et des services, pas des notebooks



Qu’est-ce qu’une API ?

- Une API (Application Programming Interface) REST (souvent sous-entendu) est un système qui :
  - Expose des fonctionnalités à travers des points d’accès HTTP
  - Accepte des requêtes et renvoie (souvent) des données structurées (souvent JSON)
  - Sert à la communication entre les clients et les services
- Exemple d’une API classique en ML :
  - /predict : retourne la prédiction d’un modèle
  - /health : retourne le statut d’un service
  - /metrics : donne des métriques à Prometheus
- En ML, les API permettent de rendre les modèles accessibles en production aux applications, aux dashboards, et aux autres services

Pourquoi FastAPI ?

- Dans les TPs, nous allons utiliser FastAPI pour rapidement définir des API en Python
  - Bonnes performances
  - Syntaxe déclarative for le schéma des requêtes et réponses
  - Validation automatique des entrées
  - Documentation automatique (Swagger UI)
  - Intégration facile avec l'écosystème ML riche de Python
  - Léger, facile à conteneuriser
- FastAPI est largement adopté dans l'industrie

Comment fonctionne FastAPI ?

- Idées principales
  - On défini une application
  - On utilise des décorateurs Python (@... avant une fonction) pour définir les points d'entrée HTTP
    - On a des get et des post
  - On peut renvoyer des dictionnaires Python, compatibles naturellement avec JSON
  - S'exécute avec *uvicorn* dans un conteneur

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok"}
```

FastAPI dans une pipeline pour servir un modèle ML

- Nous allons utiliser FastAPI pour :
  1. Gérer les entrées (user_id, features, un JSON)
  2. Récupérer les features en ligne avec Feast
  3. Charger un modèle en production avec MLflow
  4. Calculer les prédictions
  5. Retourner un JSON avec les résultats
  6. Exposer des métriques pour le monitoring

Aperçu du TP 1

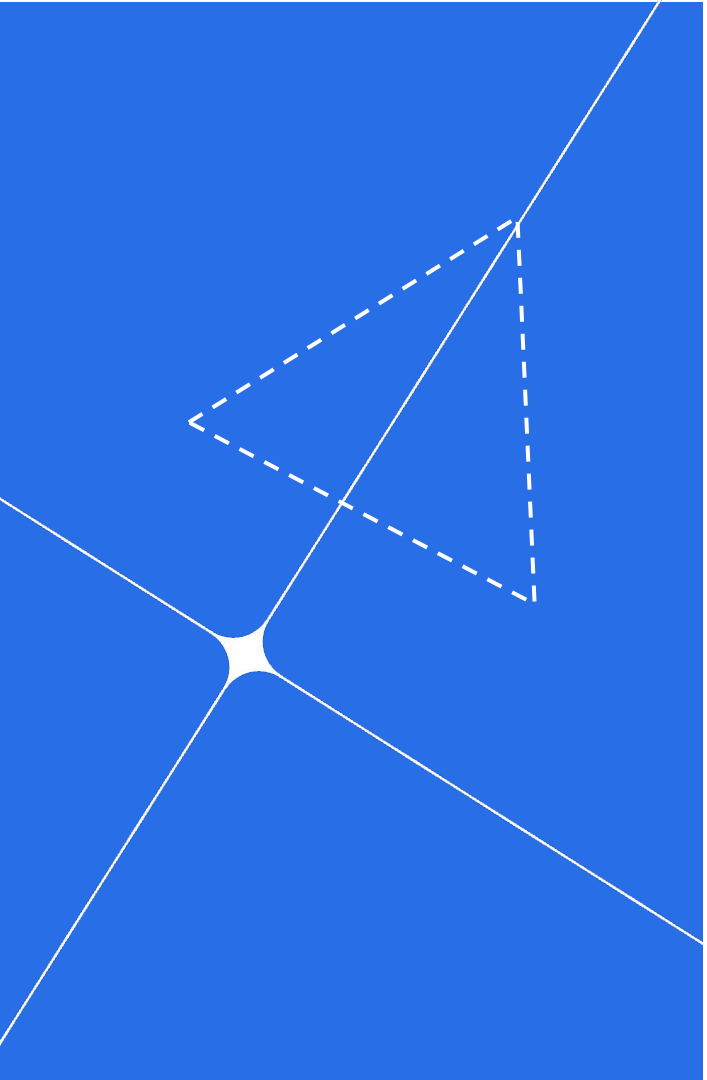
- À la fin du TP, vous aurez :
  - Construit une image Docker avec un Dockerfile
  - Démarré un conteneur et exposé des ports
  - Compris les commandes de base de Docker
  - Démarré plusieurs conteneurs avec Docker Compose
  - Établi une connexion avec une base de données
  - Créé la structure du projet de base

Le premier TP sert de base à tous les autres

À la fin du TP, il faudra m'envoyer un lien vers votre dépôt Git et y mettre le code et rapport du premier TP

En résumé

- Nous avons vu
  - Pour le MLOps existe
  - À quoi un cycle de vie de machine learning ressemble
  - Ce que sont les conteneurs et pourquoi nous les utilisons
  - Comment les Dockerfiles permettent de créer des environnements reproductibles
  - Comment Docker Compose permet de gérer et orchestrer plusieurs services
  - Ce que sont les APIs et pourquoi elles sont utilisées en ML
  - La structure de base du projet



En route vers le TP