Introduction au MLOps

Mohamed Benzerga

Université d'Angers - M2 Data Science
11 décembre 2023

Qui suis-je ?

- **Docteur en Mathématiques de l'Université d'Angers + MSc Smart Data ENSAI**
- **Data Scientist/ML Engineer avec 5 ans d'expérience :**
  - **Consultant/Prestataire chez Expleo pendant plus de 2 ans pour des clients industriels (Airbus, Alstom, SNCF…)**
  - **Senior Data Scientist chez Craft AI : start-up qui développe une plateforme de MLOps**
  - **Aujourd'hui Senior Machine Learning Engineer chez Sodexo, chargé du déploiement et du monitoring d'un produit de prévision de fréquentations de 500 restaurants d'entreprises en France**

1
Le problème

Le problème

- **Voici un data scientist :**

Le data scientist

Le problème

- **Ce data scientist a créé un modèle de ML pour résoudre un problème posé par un client.**

Le data scientist

Son notebook

Son modèle entraîné

Le problème

- **Le client ne connaît rien à la data science ou à Python : comment peut-il utiliser le modèle du data scientist pour obtenir les prévisions dont il a besoin ?**

Le data scientist

Son notebook

Son modèle entraîné

Le client perplexe

Le problème

- **Il manque quelque chose pour mettre à disposition les prévisions du modèle au client…**

Le data scientist

Son notebook

Son modèle entraîné

Le client encore perplexe

Le problème

- **Même si ce problème était réglé, comment le data scientist pourrait surveiller la qualité du service fourni au client (par exemple, de mauvaises prévisions) ?**

Le data scientist

Son notebook

Son modèle entraîné

FAIL

?

Le client toujours perplexe

La solution : le MLOps

- **Le MLOps est la branche de la data science qui cherche à résoudre ces problèmes de mise en production ou déploiement (= mise à disposition des prévisions d'un modèle ML).**

Le data scientist

Son notebook

Son modèle entraîné

MLOps

Le client moins perplexe

Définition du MLOps

MLOps = Machine Learning + Operations

- **Idée d'un ML opérationnel, mis en production, i.e. en fonctionnement réel au service de ses utilisateurs finaux.**

- **Vient d'un principe plus ancien dans le développement logiciel : le DevOps, pour une meilleure collaboration entre développeurs et équipes opérationnelles (= déploiement et maintenance).**

- **Il ne s'agit pas d'une technologie spécifique mais d'un ensemble de bonnes pratiques, de métiers, de techniques, etc. pour faciliter la mise en production de modèles de ML.**

Vue d’ensemble du MLOps

On peut distinguer 3 catégories de problèmes :

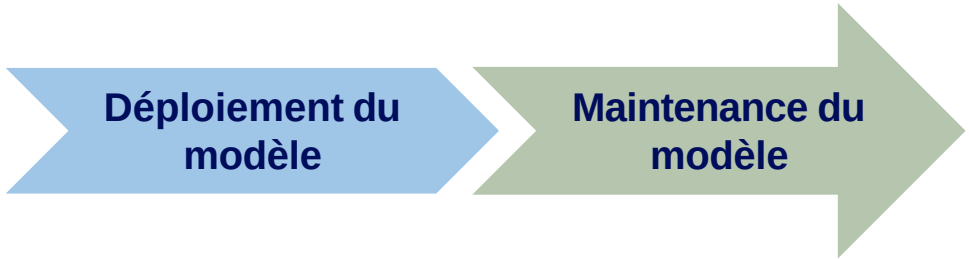
- **le déploiement lui-même : comment rendre les prévisions d’un modèle ML disponibles à ses utilisateurs ?**

Déploiement du modèle

Vue d'ensemble du MLOps

On peut distinguer 3 catégories de problèmes :

- **le déploiement lui-même : comment rendre les prévisions d'un modèle ML disponibles à ses utilisateurs ?**
- **la maintenance après déploiement : comment surveiller la qualité du service rendu et faire face aux dégradations ?**



Vue d'ensemble du MLOps

On peut distinguer 3 catégories de problèmes :

- **le déploiement lui-même : comment rendre les prévisions d'un modèle ML disponibles à ses utilisateurs ?**
- **la maintenance après déploiement : comment surveiller la qualité du service rendu et faire face aux dégradations ?**
- **la préparation en amont du déploiement : comment *bien* développer un modèle ML pour faciliter son déploiement ?**

Avertissement

- **Ce cours contient beaucoup de concepts et techniques exposés en très peu de temps !**
- **Un cours exhaustif sur l'ensemble de ces concepts et techniques nécessiterait beaucoup plus de temps, notamment pour la pratique.**
- **Il s'agit d'une introduction pour :**
  - **vous donner les grands principes**
  - **vous donner des noms de concepts, de technologies… pour que vous puissiez chercher par vous-même si vous souhaitez creuser des points particuliers.**
- **En particulier, vous ne devriez pas chercher à tout maîtriser exhaustivement à l'issue de ce cours.**

2 Déploiement d’un modèle ML

Déploiement : Principes

- **Définition : Déployer un modèle ML entraîné (ou le “mettre en production”), c'est rendre ses prévisions disponibles à ses utilisateurs finaux et/ou à d'autres systèmes informatiques.**

- Cela pose plusieurs questions pratiques, par exemple :
  - **Qui accède aux prévisions ?** Quelques experts métiers, des milliers de personnes “lambdas”...
  - **Quelle méthode d’accès aux prévisions ?** Dans un logiciel déjà existant, dans un dashboard spécifique, par un mail envoyé régulièrement, directement dans une page web sous forme de recommandations…
  - **Quelle fréquence d’accès aux prévisions ?** Une fois par an, par mois, par semaine, par jour, à la demande 7j/7, 24h/24…
- Ces questions vont déterminer les choix techniques de déploiement ou même de ML comme par exemple le fait de savoir s’il faut calculer les prévisions en continu avec un faible temps de latence ou si on peut les calculer par groupe de nouvelles données…

Déploiement : Exemples

Exemple 1 : prévision de ventes en supermarché

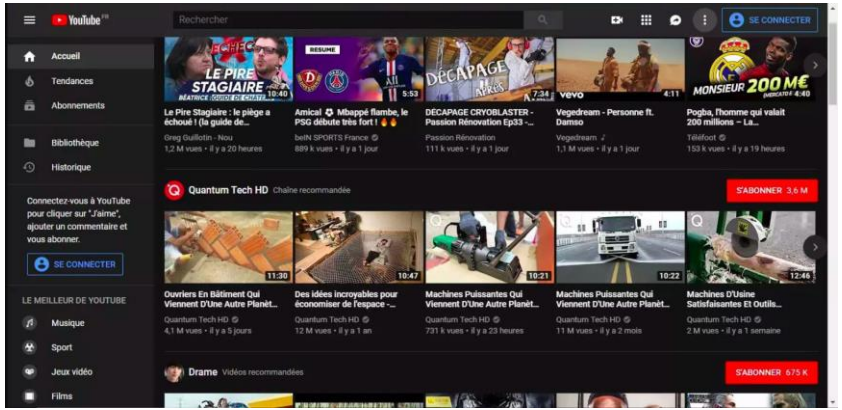
- **Un supermarché veut prévoir les ventes de ses 2000 produits pour préparer son approvisionnement.**

- **Qui accède aux prévisions ?** Le directeur du supermarché, quelques responsables logistique/approvisionnement, quelques chefs de rayons, etc.
- **Quelle fréquence d'accès aux prévisions ?** Une fois par semaine peut suffire si c'est le délai nécessaire pour organiser l'approvisionnement ensuite (dans ce cas, des prévisions journalières seraient inutiles !)
- **Quelle méthode d'accès aux prévisions ?** Sans doute un mail par semaine avec un fichier Excel ou un dashboard spécifique.

Déploiement : Exemples

Exemple 2 : recommandation de vidéos

- **Un site web de contenus vidéos veut proposer des recommandations à ses utilisateurs à leur arrivée sur le site et après visionnage d'une vidéo.**



- **Qui accède aux prévisions ?** N'importe quel utilisateur de la plateforme… donc n'importe qui !
- **Quelle méthode d'accès aux prévisions ?** Via la plateforme directement.
- **Quelle fréquence d'accès aux prévisions ?** Doivent être disponibles sur demande autant que nécessaire.

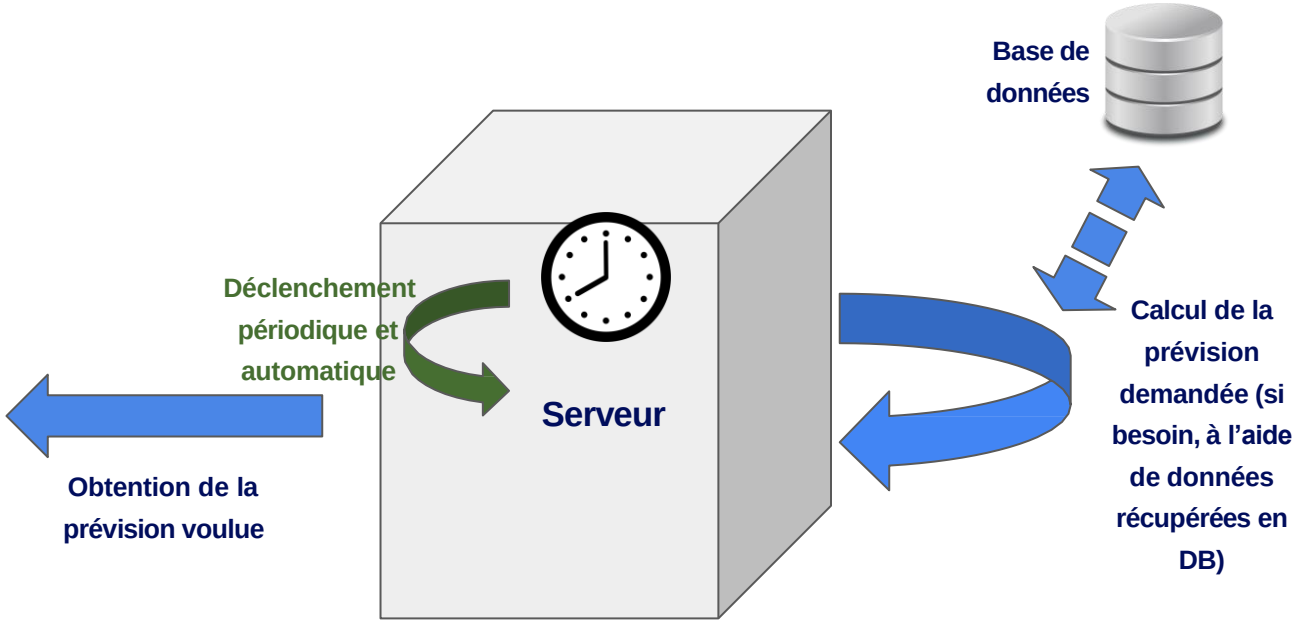
(Sur le cas de YouTube, voir ici).

Déploiement : Aspects techniques

- Plusieurs méthodes possibles selon les situations :
  - **tâches planifiées (“scheduled jobs”) + envoi de mail ou écriture dans une base de données (DB)**
  - **création d’un dashboard (avec Dash en Python ou Shiny en R)**
  - **mise en place d’une API Web**
  - **…**

Déploiement : Aspects techniques

- Plusieurs méthodes possibles selon les situations :
  - **tâches planifiées (“scheduled jobs”) de calcul et d’envoi de prévisions,** par exemple par mail ou écriture en DB & avec un “Cron” (Linux)

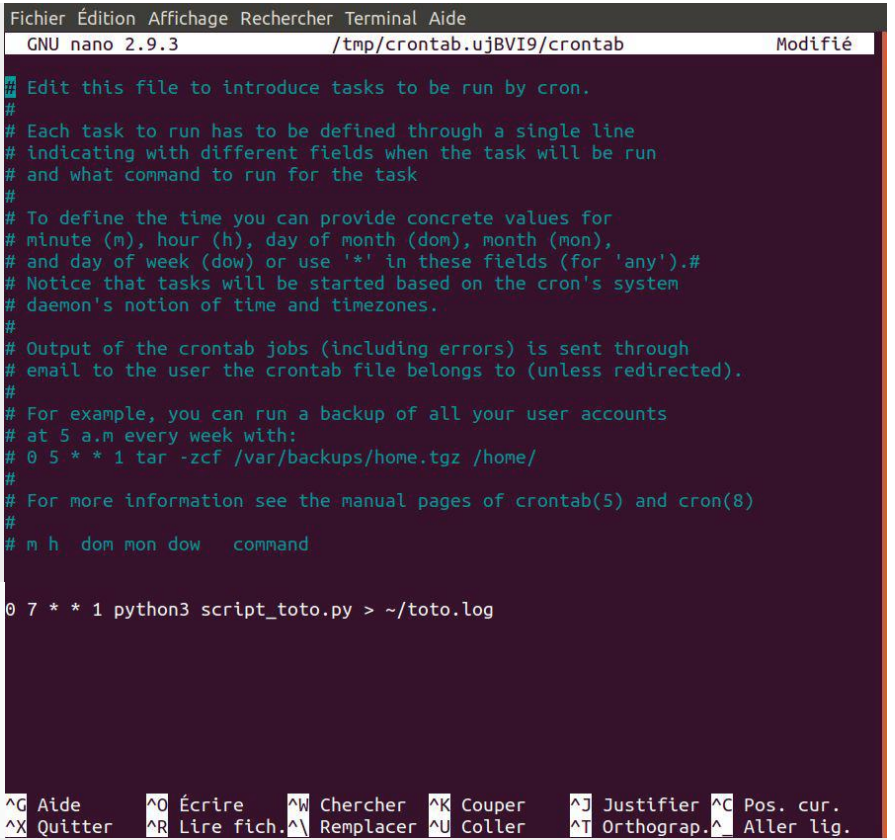


Déploiement : Aspects techniques

Crontab (avec Linux) :

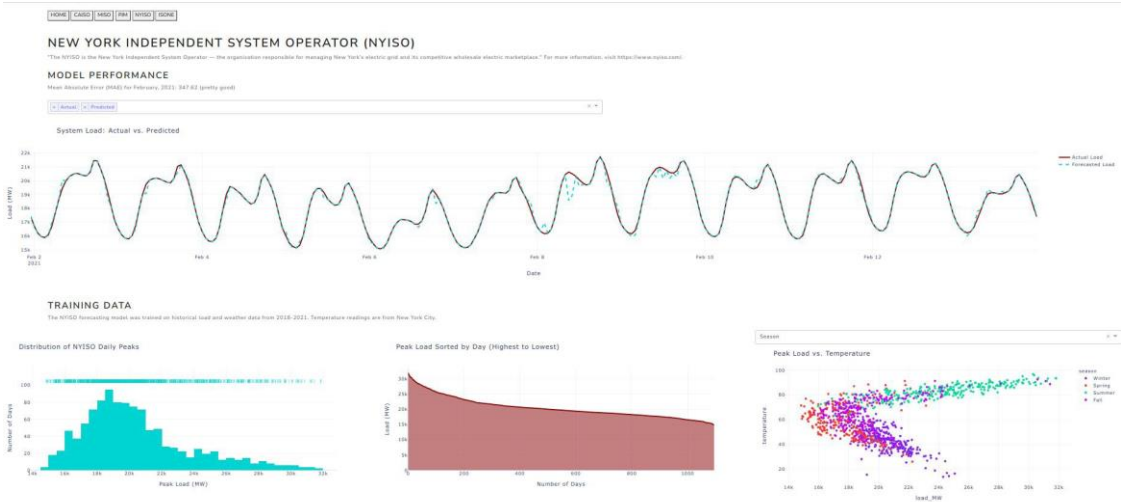
- Sous Linux, un “daemon” nommé **Cron** tourne en continu en arrière-plan pour gérer des tâches planifiées.
- Un utilisateur peut ainsi programmer n’importe quelle tâche (écrite comme une commande du terminal) avec la commande `crontab -e` qui permet d’éditer le fichier crontab de A.
- C’est un fichier texte avec une syntaxe spécifique pour définir les tâches planifiées :

```
# ┌──────────── minute (0 - 59)
# │ ┌────────── heure (0 - 23)
# │ │ ┌──────── jour du mois (1 - 31)
# │ │ │ ┌────── mois (1 - 12)
# │ │ │ │ ┌──── jour de la semaine (6 - 0) (du dimanche au samedi ;
# │ │ │ │ │                                7 est aussi le dimanche sur certains systèmes)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * commande à exécuter
```



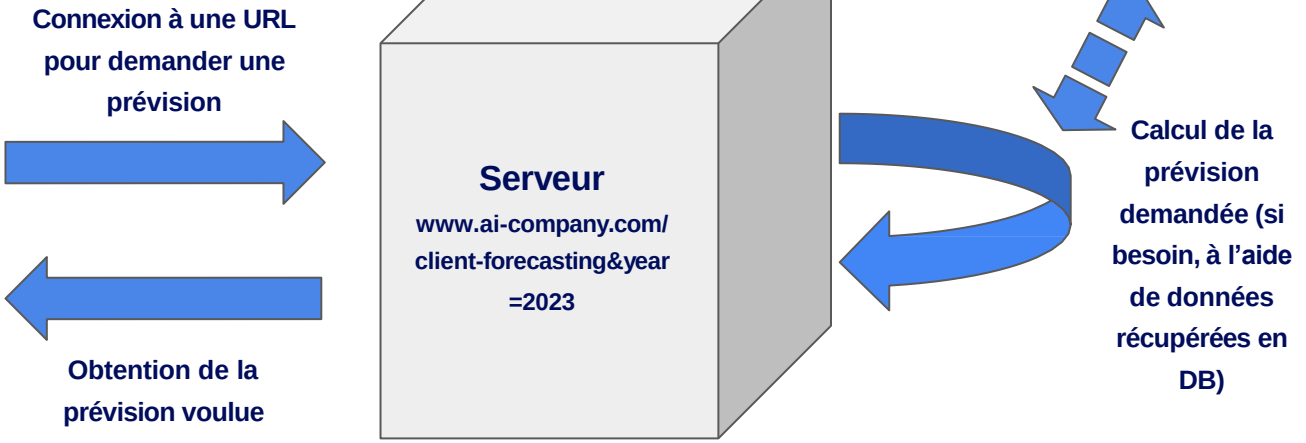
Déploiement : Aspects techniques

- Plusieurs méthodes possibles selon les situations :
  - **tâches planifiées (“scheduled jobs”) + envoi de mail ou écriture en DB**
  - **création d’un dashboard (avec Dash ou Streamlit en Python, Shiny en R ou des outils spécialisés type Power BI, Tableau…) : interface accessible depuis le Web rassemblant graphiques, tableaux statistiques…**



Déploiement : Aspects techniques

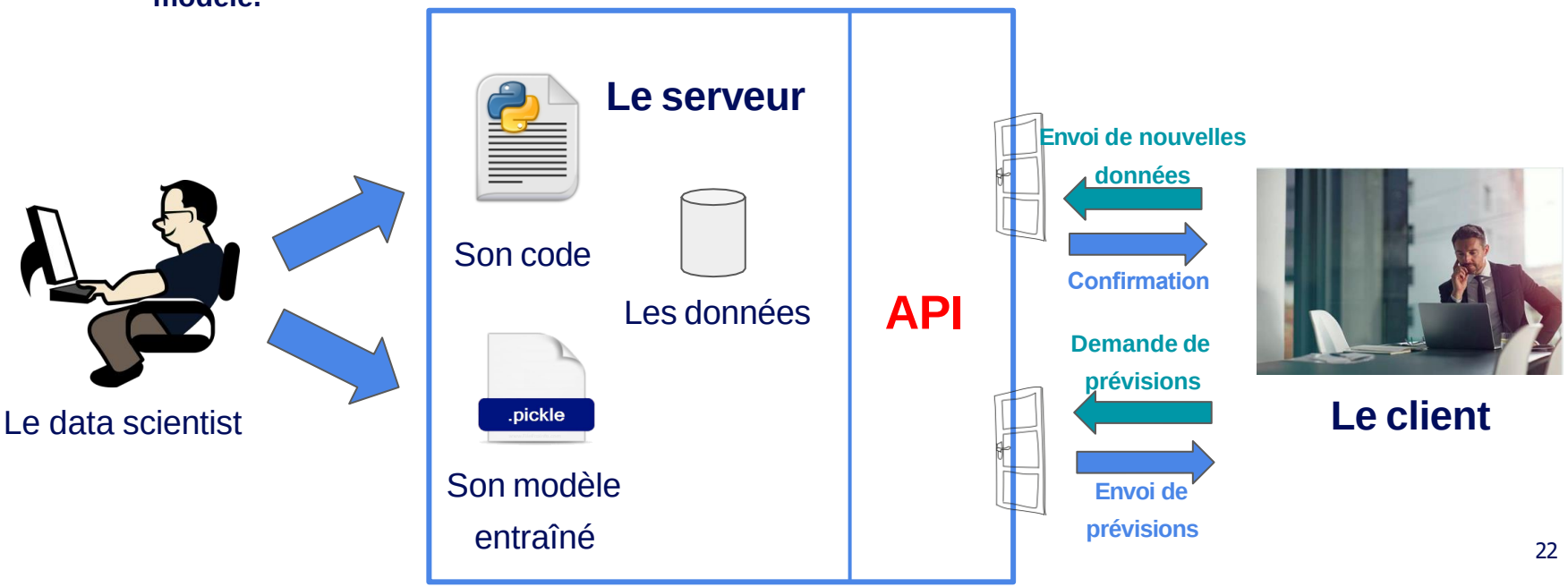
- Plusieurs méthodes possibles selon les situations :
  - **mise en place d'une API Web :** un utilisateur peut requêter une URL qui pointe vers un serveur (ordinateur ou groupe d'ordinateurs, éventuellement dans un cloud) hébergeant le nécessaire pour le calcul des prévisions (modèle, base de données, code de preprocessing, etc.)



Déploiement - API Web

Qu'est-ce qu'une API Web ?

- **Pour mettre à disposition les prévisions du modèle au client, le data scientist peut mettre au point une API Web, i.e. un ensemble de portes d'entrée depuis le Web permettant d'utiliser/exécuter son code, mettre à jour ses données, etc. et d'obtenir les résultats sans accéder au code, aux données ou au modèle.**

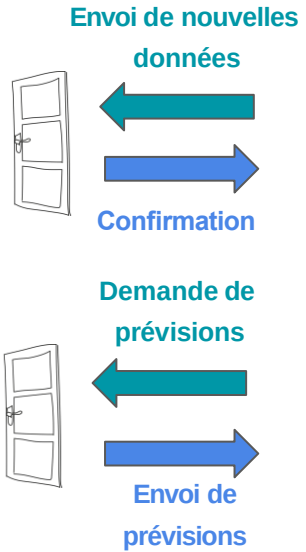


Déploiement - API Web

Qu'est-ce qu'une API Web ?

- Définir ces portes d'entrée, appelées les routes de l'API, nécessite de s'entendre avec le client sur :
  - l'action que chaque route permettra d'effectuer
  - le format attendu pour la requête voire pour les données d'entrée
  - le format attendu pour le résultat renvoyé par le serveur.

- Les requêtes seront envoyées par le client en utilisant le protocole HTTP directement en appelant une URL.



Déploiement - API Web

Le protocole HTTP

- **Le protocole HTTP est l'ensemble des règles permettant la communication de données entre ordinateurs sur le réseau Internet selon un principe client/serveur.**

- **Il définit plusieurs méthodes, i.e. plusieurs types de requêtes : GET, POST, PUT, DELETE… Nous utiliserons uniquement les 2 suivantes :**
  - **GET: requêtes qui réclament des données ou un résultat au serveur (en principe, sans envoyer de données) => par exemple, entrer une URL dans un navigateur envoie une requête GET vers cette URL**
  - **POST: requêtes qui réclament et envoient des données au serveur, causant souvent un changement d'état du serveur.**

Déploiement - API Web

Le protocole HTTP

- Le protocole HTTP définit aussi des codes d'erreurs comme :
  - 200 : tout fonctionne
  - 4xx : erreurs venant de la requête du client :
    - 404 : l'URL n'existe pas
    - 401 : authentification…
  - 5xx : erreurs venant du traitement de la requête par le serveur :
    - 500 : erreur interne du serveur [bug]
    - 503 : service unavailable [maintenance]
    - 504 : gateway time-out [temps de réponse trop long]...
- Une documentation complète des (nombreux) codes d'erreurs possibles est disponible par exemple ici ou là.

Déploiement - API Web

Comment définir une API Web ?

- **Définir une API revient donc à définir ses routes, c'est-à-dire :**
  - **un ensemble d'URL avec une méthode HTTP (GET, POST, etc.) associée**
  - **le format attendu pour une requête sur chaque route (voire pour les données d'entrée dans le cas POST)**
  - **le format attendu pour le résultat renvoyé par le serveur.**

- **Flask est une librairie Python permettant de construire des applications Web**
  ⇒ cf. TP

Déploiement - API Web

Infrastructures : les serveurs

- Dans le cas d'un service Web (API ou dashboard), le déploiement nécessitera des **infrastructures** spécifiques ⇒ des **serveurs Web,** i.e. des **ordinateurs physiques ou virtuels** capables d'être **disponibles 24h/24** et de faire face à de **nombreuses demandes concurrentes et variables.**
- Ces serveurs peuvent prendre plusieurs formes :

Serveurs physiques
on premise

(rares en principe…)

Azure

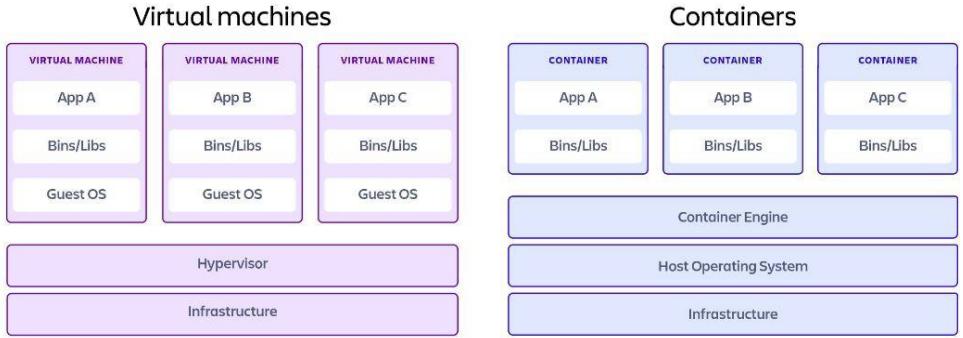
OVHcloud

Google Cloud

Serveurs dans un Cloud
(AWS, Azure, OVH, GCP…)
=> pas besoin d'entretenir beaucoup de machines notamment en cas de pic de demande

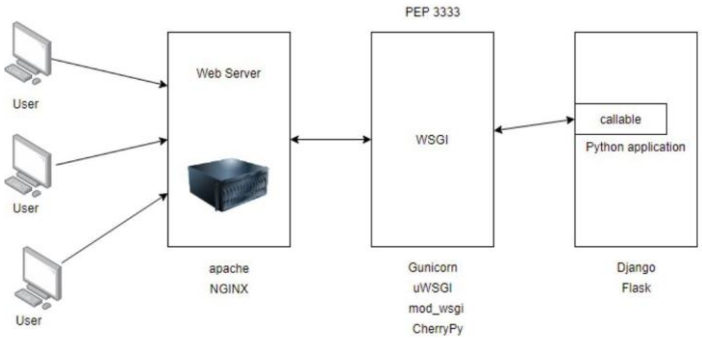
Déploiement : Aspects techniques

- De plus, ces serveurs peuvent être **virtuels,** notamment dans le Cloud, étant sous forme :
  - soit de **machines virtuelles (VM),** qui émulent un 2e ordinateur (OS complet inclus) en réservant une partie des ressources physiques de la machine physique qu'elles utilisent
  - soit de **conteneurs (Docker),** qui sont plus légers en contenant le strict minimum nécessaire pour faire tourner une application spécifique (pas d'OS complet) et en évitant de réserver plus d'espace que nécessaire. Parfois, il sera nécessaire d'avoir plusieurs conteneurs qui tournent en parallèle ou en "série", qu'il faudra alors orchestrer avec Kubernetes par exemple.
- En général, il vous faudra des **serveurs distincts** entre le **développement** (non-accessible à l'utilisateur et sur lequel vous pouvez tester et développer de nouvelles features), la **pré-production** (pour les phases de test) et la **production** (réellement accessible à l'utilisateur final).



Déploiement : Aspects techniques

- Ces serveurs (infra) seront tout à la fois :
  - les **points de réception des requêtes HTTP venues de l'extérieur** : il faudra installer sur ces serveurs un logiciel appelé **reverse proxy,** comme Apache ou NGINX : c'est le point d'entrée des requêtes venues de l'extérieur, qui va pouvoir **contrôler la sécurité** des requêtes, **accélérer leur traitement** en envoyant directement les fichiers statiques (CSS par exemple) au client, **dispatcher les requêtes** vers plusieurs machines (load balancing)...
  - et **les lieux d'exécution de votre code Python**. Or un reverse proxy ne sait pas exécuter du code Python. Un **serveur WSGI** comme Gunicorn est une **librairie Python** qui fait l'intermédiaire entre **le code Python de votre API** (avec Flask qui crée les routes qui utilisent votre code de ML) **et le reverse proxy :** le WSGI reçoit et exécute les requêtes HTTP vers les routes que vous avez codées avec Flask, éventuellement en les répartissant sur plusieurs processus de l'OS du serveur.



Communication between user, web server, WSGI, and Python application.


3
Maintenance d’un modèle ML
%49

Maintenance : Le problème

- Une fois le modèle déployé, il faut **monitorer la qualité du service rendu aux utilisateurs.**
- Deux difficultés de nature différente peuvent apparaître :
  - **les utilisateurs ne reçoivent pas de prévisions** au choix à cause :
    → d'un **bug** dans le code : cas imprévu, données d'entrée problématiques, changement de noms de variables… ou code mal écrit !
    → d'une **indisponibilité** du service : par exemple, trop de connexions simultanées à une API Web par rapport à l'infra qui les reçoit
    → d'un **problème réseau**
    → d'un problème d'**accès à une base de données**…
  - **les utilisateurs reçoivent de mauvaises prévisions** :
    → **données problématiques** : manquantes, de mauvaise qualité, inadaptées…
    → **modèle inadapté** à de nouveaux cas sur lesquels il n'a pas appris (dérive statistique des données) ⇒ c'est le phénomène du **drift** (à suivre).

Maintenance : Le problème

- Il faut donc être capable, pendant que le modèle est déployé :
  - d'être **alerté** sur ces erreurs, si possible en amont
  - de les **diagnostiquer**
  - et de les **corriger**.
- Tout cela nécessite de mettre en place des éléments **d'observabilité :** les responsables de la maintenance d'un modèle ML **doivent pouvoir retrouver tout le contexte** de n'importe quelle prévision calculée y compris plusieurs jours avant :
  - quelles **versions du code, du modèle, des données** ont été utilisées
  - quelles **erreurs** ont eu lieu
  - plus globalement, quels **événements** ont eu lieu et comment a fonctionné le calcul de prévision
- **Objectif :** pouvoir reproduire EXACTEMENT le calcul de la prévision pour pouvoir retrouver l'origine du bug et le débugger.

Maintenance : Quelles solutions ?

1. **Des logs pour le code**
   - Écrivez dans un **fichier log** (qui est un simple fichier texte) les principaux événements et les erreurs qui se produisent pendant l'exécution de votre code.
   - En Python, vous pouvez utiliser la librairie **logging**.

Logging : usage basique

- Vous pouvez écrire dans un fichier log différents niveaux d'alertes : `DEBUG, INFO, WARNING, ERROR, CRITICAL.`

```
import logging
import numpy as np

def main_function(number):
    logging.basicConfig(format='[%(asctime)s] %(message)s',
datefmt='%d/%m/%Y %H:%M:%S', filename='logfile.log')
    logging.info(f'Started with parameter {number}')
    try:
        result = np.exp(number)
        logging.info("Everything is OK")
        return result
    except Exception as e:
        logging.error(f'Error message: {e}')
    logging.info('Finished')

if __name__ == '__main__':
    main_function(1)
    main_function("toto")
```

- À gauche, un script Python.
- Ci-dessous le contenu du fichier 'logfile.log' après exécution de ce script.

```
[01/11/2022 20:18:41] Started with parameter 1
[01/11/2022 20:18:41] Everything is OK
[01/11/2022 20:18:41] Started with parameter toto
[01/11/2022 20:18:41] Error message: ufunc 'exp' not
supported for the input types, and the inputs could not be
safely coerced to any supported types according to the
casting rule ''safe''
[01/11/2022 20:18:41] Finished
```

Maintenance : Quelles solutions ?

1. **Des logs pour le code**
2. **Une data validation : des checks de qualité des données**

Vous ne devez pas partir du principe que votre système recevra des données parfaites !

- Votre code doit donc contenir des fonctions qui **vérifient** entre autres :
  - **qu'il y a des données !** (données manquantes…)
  - **que les données sont du type attendu**
  - **qu'elles sont cohérentes** avec le problème à résoudre, le contexte métier (pas de données aberrantes ou non-conformes à des règles métiers)...
- Cela vous permettra le cas échéant d'alerter le client sur la mauvaise qualité des données qu'il vous envoie.
- En Python, les librairies **marshmallow** et **pandera** peuvent faciliter la création de ces checks de qualité ⇒ **cf. TP**

Maintenance : Quelles solutions ?

1. **Des logs pour le code**
2. **Une data validation : des checks de qualité des données**
3. **Un monitoring de performance des modèles**
   - **Si possible,** il faut calculer et stocker :
     - des **métriques de performance** du modèle (R^2, MAE, MSE, précision, rappel, score F1, etc.)
     - les **valeurs prévues et les valeurs réelles** de ce que vous cherchez à prévoir
     - les données d'entrée avec des **statistiques** sur leur distribution.
   - Vous pouvez même créer un **dashboard** avec les graphes correspondants et des **alertes** quand la performance se dégrade (=> attention au trop-plein d'alertes !).
   - Cela permet de prévenir le **drift** (baisse de performance des modèles).
   - **Mais ce n'est pas toujours possible :**
     - **les "vraies valeurs" ne sont pas toujours faciles à obtenir**
     - **voire même n'existent pas toujours** (quelle est LA bonne recommandation d'une vidéo Youtube, par exemple ?)

Maintenance : Quelles solutions ?

1. **Des logs pour le code**
2. **Une data validation : des checks de qualité des données**
3. **Un monitoring de performance des modèles**
4. **Un ré-entraînement automatique**
   - On peut commencer par prévoir un **ré-entraînement périodique** tous les jours, toutes les semaines… dès que suffisamment de données récentes sont arrivées.
   - Il faudra alors se poser la question de l'évolution de la **fenêtre de temps** sur laquelle porte le ré-entraînement (doit-on abandonner des données trop anciennes et si oui, lesquelles ?).
   - On peut aussi déclencher ce ré-entraînement **en cas de baisse de performance** (par exemple, si le R^2 descend en-dessous d'un seuil à choisir…)
   - Dans certains cas, moins automatiques, il faudra parfois refaire le choix de nouvelles variables, de nouveaux prétraitements ou de nouveaux algorithmes…

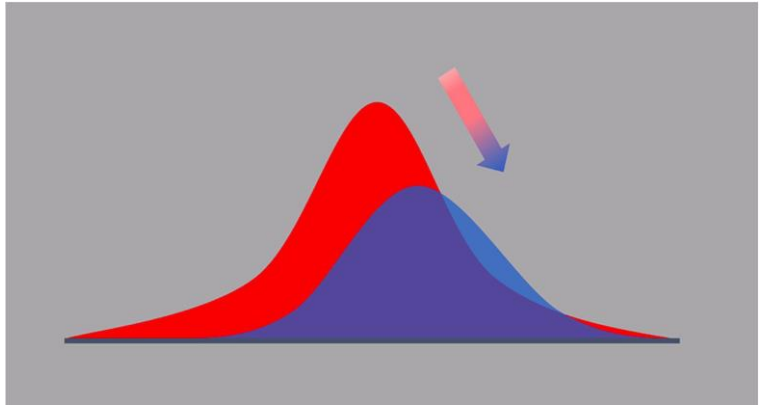
Maintenance : Quelles solutions ?

1. **Des logs pour le code**
2. **Une data validation : des checks de qualité des données**
3. **Un monitoring de performance des modèles**
4. **Un ré-entraînement automatique**
5. **Un versionnage GLOBAL du système**
    - Comme vu précédemment, on veut stocker les versions :
        - du code,
        - des données,
        - des modèles avec les différents choix des (hyper)paramètres,
        - des résultats, etc.
    - **LE standard pour versionner du code est Git.**
    - Un équivalent de Git pour le ML est **DVC (Data Version Control).**
    - Un autre système de versionnage ML populaire est **MLflow.**

Maintenance : Le drift

- **Définition : le "model drift" (dérive des modèles) est la dégradation des performances prédictives d'un modèle ML après son déploiement.**

- Cette dégradation est souvent due à :
  - la dérive des distributions des variables d'entrée X du modèle → **data drift : dérive de P(X)**
  - et/ou la dérive de la relation entre les variables prédictives X et la variable à prévoir Y → **concept drift : dérive de P(Y|X).**

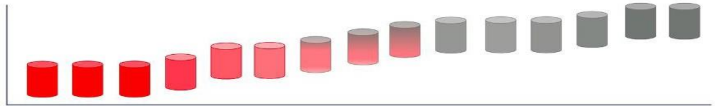


⇒ Les hypothèses sur les données qui ont permis l'entraînement du modèle sont devenues invalides : les données ont “trop changé” par rapport au training set.

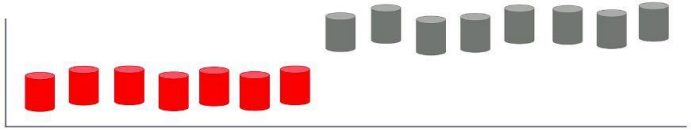
Or, un modèle de ML supervisé ne sait pas extrapoler…

Maintenance : Le drift

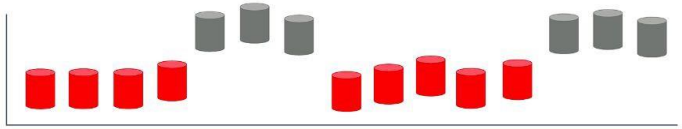
- Le drift peut se manifester différemment dans le temps :
  - **Drift progressif :**



  - **Drift brutal / soudain :**



  - **Drift récurrent :**



Maintenance : Le drift

- **Pour limiter le drift, il faut au moins ré-entraîner le modèle (régulièrement ou après détection du drift) voire créer un nouveau modèle.**

- Dans le cas d'un ré-entraînement, celui-ci devra **prendre en compte les nouvelles données** arrivées depuis le dernier entraînement :
  - soit en plus des données existantes
  - soit avec suppression des données "les plus anciennes"
  - soit en pondérant plus fortement les données plus récentes.

Maintenance : Le drift

- **Exemple 1 :** vieillissement d'une machine en maintenance prédictive.
  - Un modèle de ML a été entraîné pour détecter les pannes d'une machine industrielle.
  - Après une période de bon fonctionnement, le modèle envoie **de plus en plus de fausses alertes.**
  - Une inspection des données d'entrée montre que **la distribution de certaines variables a dérivé hors des valeurs sur lesquelles le modèle a été entraîné, du fait du vieillissement naturel de la machine** → **data drift.**

Maintenance : Le drift

- **Exemple 2 :** détection de spams
  - Un modèle de ML a été entraîné pour détecter des spams dans une boîte mail.
  - Autrefois, il suffisait de détecter des mails promettant une forte somme d'argent d'un cousin éloigné…
  - Maintenant, ça ne suffit plus : certains spams imitent assez précisément de vrais mails provenant de banques, de sociétés de vente en ligne, etc.
  - C'est un **concept drift :** le concept même de spam a changé, les spammeurs se sont adaptés aux méthodes de détection habituelles…

Attention Sir/Madam,

Sequel to United Nations public protection policy against fraudulent activities operating in Europe, US and various African banks. This council was set up to fight against scam and fraudulent activities worldwide, responsible for investigating the legitimacy of unpaid contract, inheritance and lotto winning claims by companies and individuals and directs the paying authorities worldwide to make immediate payment of verified claims to the beneficiaries without further delay.

It was resolved that all unpaid claims will be concluded via e-wire transfer thought First Sunset Bank, which is very reliable and secure bank. Your beneficiary funds the sum of USD 4.8 million has been forwarded and deposited in First Sunset Bank for instant transfer to you once you contact them.

You are advised to contact First Sunset Bank via below email, to guide you further on the wire transfer procedures:

First Sunset Bank.
Email: firstsunsetbank@web.cg
Contact Person: Mrs. Agnes Scott

Please be informed that transfer time is limited sequel to policy, therefore you are advised to attend as soon as you read this email and also reconfirm your full details to them. We have copied all our co-ordinate security agencies for record purposes.

Thank you.

Your Faithfully,
Mrs. Ann Walter.
Director, Special Duties.
United Nations Security Council.

Maintenance : Comment détecter le drift ?

1. Détecter la **baisse de performance prédictive** directement en surveillant les métriques de performance du modèle. Cela nécessite d'avoir les prévisions stockées au fur et à mesure et les "vraies valeurs" à disposition, ce qui n'est pas toujours évident en pratique**.**
2. Examiner les **corrélations entre les variables d'entrée X et celles avec la variable Y à prévoir** puisqu'un modèle apprend souvent sur ces corrélations. Les comparer entre le training set et les données en temps réel.
3. Comparer les **distributions des variables X et Y entre le training set et les données en temps réel** pour "prévoir" le drift faute de le détecter directement. On peut par exemple :
   a. étudier les **distributions** (histogramme, statistiques descriptives, etc.)
   b. vérifier les **valeurs manquantes** et les **valeurs aberrantes**
   c. faire des **tests statistiques** pour dire si la différence entre avant et après l'entraînement est significative (test de Kolmogorov-Smirnov pour comparer des distributions, test de Student pour comparer des moyennes…)
   d. tenter de créer un **autre modèle ML** pour séparer les données en 2 lots consécutifs **"avant/après"** ⇒ si ça fonctionne, il y a drift**.**
4. Utiliser un **algorithme de détection de drift,** comme **ADWIN** (Adaptive Windowing). En Python, ADWIN est disponible avec les librairies River ou scikit-multiflow.

Maintenance : Drift & Algorithme ADWIN

- Au fur de l'évolution d'une série temporelle x_t, l'algorithme fait évoluer une fenêtre de temps W en éliminant des valeurs passées jusqu'à obtenir une fenêtre de moyenne "homogène", i.e. telle que toute subdivision de W en W0 puis W1 donne 2 moyennes $\hat{\mu}_{W_0}$ et $\hat{\mu}_{W_1}$ "proches" selon un seuil ϵ_{cut} choisi à l'avance. (W1 : valeurs les plus récentes de W)

- Le principe global de l'algorithme est donc présenté ci-dessous (son implémentation concrète est plus subtile, pour optimiser le temps de calcul) :

- L'algorithme détecte un drift dès lors qu'il trouve 2 fenêtres W0 et W1 de moyennes "suffisamment différentes". Le point de bascule entre W0 et W1 est un point de drift.

ADWIN0: ADAPTIVE WINDOWING ALGORITHM

```
1  Initialize Window W
2  for each t > 0
3      do W ← W ∪ {x_t} (i.e., add x_t to the head of W)
4         repeat Drop elements from the tail of W
5           until |μ̂_{W0} − μ̂_{W1}| < ε_cut holds
6                for every split of W into W = W0 · W1
7         output μ̂_W
```

Maintenance : Drift & Algorithme ADWIN

- En pratique, ce n'est pas ϵ_{cut} que l'on règle comme paramètre mais un "seuil de confiance" δ entre 0 et 1, dépendant de ϵ_{cut}, de sorte que pour tout temps t :

1. (False positive rate bound). *If* μ_t *remains constant within W, the probability that* ADWIN0 *shrinks the window at this step is at most* δ.

2. (False negative rate bound). *Suppose that for some partition of W in two parts* W_0W_1 *(where* W_1 *contains the most recent items) we have* $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$. *Then with probability* $1 - \delta$ ADWIN0 *shrinks W to* W_1, *or shorter.*

- Autrement dit, l'algorithme ADWIN "réduit à tort" la fenêtre W (i.e. il la réduit alors que la moyenne varie "peu" dans W) avec une probabilité au plus δ
- De plus, l'algorithme ADWIN "réduit à juste titre" la fenêtre W si elle peut être subdivisée en 2 parties de moyennes "assez différentes" avec probabilité 1 - δ

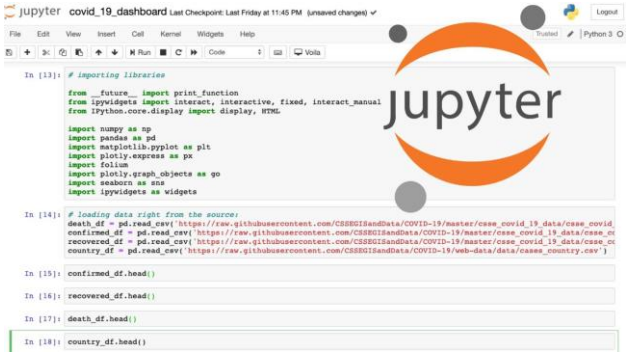
4 Préparation d’un bon déploiement

Notebooks are bad

- Quand un data scientist utilise un notebook pour créer un modèle, c'est souvent en **exécutant les cellules dans un ordre changeant et en changeant des (hyper)paramètres pour tester un nouveau modèle** (sans forcément garder les précédentes versions).
- **Problème :** après quelques itérations, on ne sait pas reproduire les résultats ou retracer les différentes expériences ! C'est le bazar !
- **Pour éviter le "bazar" des notebooks, mieux vaut créer ses modèles ou pipelines dans un script Python organisé en fonctions ou en classes.** On peut ainsi reproduire l'exécution en lançant directement le script plutôt qu'en cliquant dans un ordre mal défini sur des cellules d'un notebook

  ⇒ amélioration de la reproductibilité

  ⇒ automatisation de l'entraînement et de l'évaluation du modèle



Préparation en amont

- Pour faciliter le déploiement et la maintenance d'un modèle de ML, il est en fait nécessaire d'**avoir un système de bonne qualité en amont du déploiement :**
  - **Comment garantir des données de qualité en entrée de votre modèle de ML ?** ⇒ il faudra prévoir un **stockage** unique (cloud ou DB par ex.) **des données brutes** d'une part et **des données transformées** (par feature engineering) d'autre part.
  - **Comment revenir à un état antérieur du système ?** (pour du debug ou pour comparer les performances avant/après un changement du modèle, des données, etc.) ⇒ il faut aussi stocker les modèles, les prévisions envoyées, les métriques associées… et leurs versions, ce qui nécessite un **versioning** et un **tracking d'expériences**, ce qui permet de revenir à une expérience antérieure, comparer des expériences entre elles.
  - **Comment assurer la maintenabilité et la facilité d'évolution du code ?** ⇒ cela passe par l'application de **bonnes pratiques de code**
  - **En cas d'évolution du code, comment vérifier qu'on n'a pas introduit de bug dans le code existant ?** ⇒ grâce à des **tests** et une **CI/CD** permettant de les exécuter automatiquement avant tout déploiement.

Bonnes pratiques de code

- **Principe : le code est fait pour être LU par quelqu'un d'extérieur** !! (soit quelqu'un d'autre soit vous-même dans 6 mois, 1 an…). **Il doit donc être LISIBLE !**

- En particulier, il sera plus facile à comprendre et donc plus facile à débugger, à faire évoluer… ce qui limitera les erreurs et le temps passé sur le code !
- Voici quelques bonnes pratiques :
  - **Nommage des variables : utilisez des noms de variables qui décrivent clairement leur contenu !** Par exemple, préférez `column_name` à `c`, ou `student_data` à `df`.
  - **Utilisez des fonctions, voire des classes, "pas trop longues" pour organiser votre code** au lieu de mettre tout votre code dans un seul script de 500 lignes ! ⇒ Avantage : vous pouvez changer une fonctionnalité spécifique sans toucher au reste du code.
  - **Répartissez ces fonctions ou classes dans des fichiers distincts, nommés adéquatement.** Par exemple, dans un dossier `src`, vous pourriez avoir des fichiers `load_data.py`, `prepare_data.py`, `train_models.py`, `predict.py`…
  - Vous pourriez d'ailleurs organiser votre dossier en plusieurs sous-dossiers `app`, `src`, `notebooks`, `scripts`, `tests`… pour ranger vos fichiers (app pour une API ou un dashboard, src pour le code source principal…)

Bonnes pratiques de code

- Appliquez le principe **DRY (Don’t Repeat Yourself) :** évitez d’avoir un morceau de code similaire à plusieurs endroits différents, mieux vaut en faire une fonction (sinon, en voulant modifier le morceau concerné, vous risquez d’oublier au moins un des cas où vous l’utilisez)
- **“1 function does 1 thing”** : une fonction doit avoir une **seule responsabilité** (sauf éventuellement quelques fonctions “principales” qui déroulent différents appels à des fonctions plus spécifiques pour différentes étapes) pour être davantage réutilisable et plus simple à débugger
- **Documentez vos fonctions :** créez des “docstrings” indiquant à quoi sert la fonction et quelles sont ses entrées et sorties attendues (exemple ci-contre)
- **Appliquez les conventions de style PEP8** par exemple en utilisant le formatting avec la librairie **black**
- Si possible, préférez des **fonctions vectorisées (Numpy, Pandas) à des boucles for : elles sont nettement plus rapides !**
- “Premature optimization is the root of evil” : **il faut d’abord avoir un code fonctionnel avant de chercher à l’optimiser** (temps d’exécution ou mémoire)

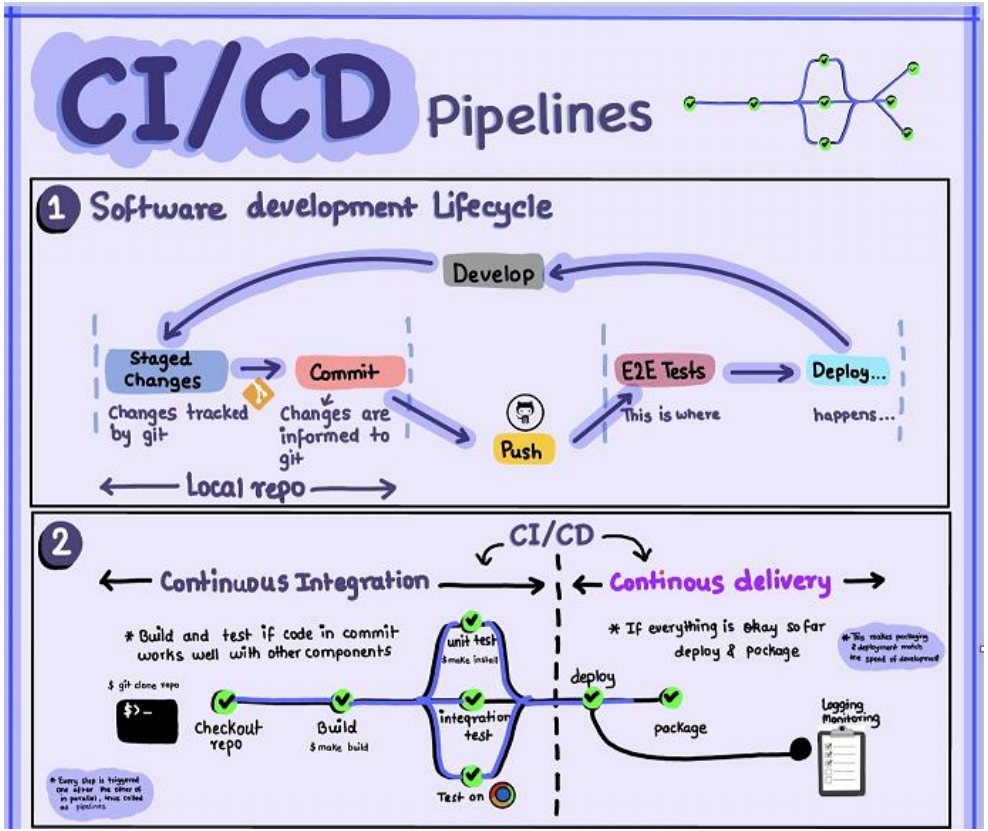
```
def cars(self, distance, destination):
    '''We can't travel distance in vehicles without fuels, so here is the fuels

    Parameters
    ----------
    distance : int
        The amount of distance traveled
    destination : bool
        Should the fuels refilled to cover the distance?

    Raises
    ------
    RuntimeError
        Out of fuel

    Returns
    -------
    cars
        A car mileage
    '''
    pass
```

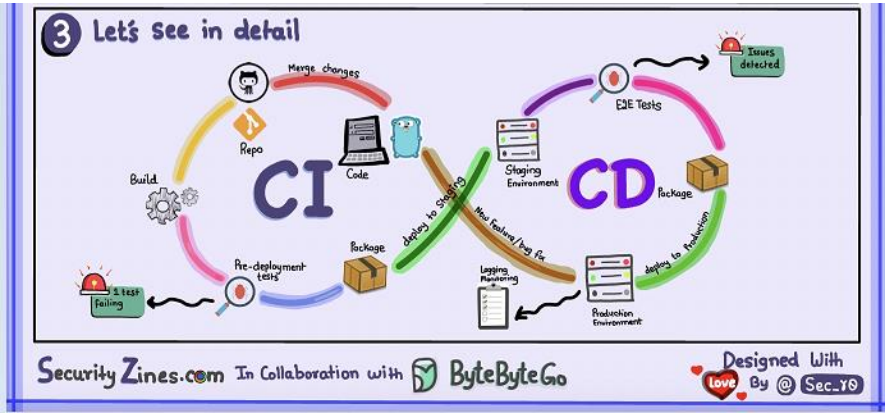
Tester le code

- **Pourquoi faire des tests ?**
  - **Pour automatiser la vérification du bon fonctionnement du code**
  - Pour vérifier que tout fonctionne après un git merge entre plusieurs branches
  - **Pour éviter d'envoyer un code dysfonctionnel en production !**
- **Différents types de tests :**
  - les **tests unitaires :** vérifient le bon comportement d'une fonction/méthode **isolée** (indépendamment du reste du code, de données extérieures…) dans **un seul cas bien défini** (il faut tenter de tester tous les cas possibles d'inputs/outputs, y compris des cas "limites")
  - les **tests d'intégration :** vérifient le bon comportement de plusieurs éléments **en interaction**, par exemple plusieurs fonctions, éventuellement avec des dépendances (ex : vérifier qu'une fonction lit correctement une base de données extérieure)
  - les **tests fonctionnels** : vérifient que les **fonctionnalités** attendues par l'utilisateur donnent le résultat qu'il attend, sans se soucier de leur implémentation.
- En Python, **pytest** et **unittest** sont les principales librairies de création de tests.
- L'idéal est de mettre au point une **CI/CD (Continuous Integration/Continuous Deployment)** permettant de **faire tourner tous les tests automatiquement à chaque merge ou chaque commit** (avec des outils comme GitHub Actions, GitLab, CML…).

Tester le code

Tests & CI/CD


CI/CD Pipelines
1 Software development Lifecycle
Develop
Staged Changes
Changes tracked by git
Commit
Changes are informed to git
Push
E2E Tests
This is where
Deploy...
happens...
Local repo
2
CI/CD
Continuous Integration
Continous delivery
* Build and test if code in commit works well with other components
* If everything is okay so far deploy & package
$ git clone repo
$>_
Checkout repo
Build
$ make build
unit test
$ make install
integration test
Test on
deploy
package
Logging Monitoring


3 Let's see in detail
Merge changes
Repo
Code
CI
Build
Package
Pre-deployment tests
2 test failing
deploy to Staging
Staging Environment
E2E Tests
Issues detected
CD
Package
deploy to Production
Production Environment
Logging Monitoring
New feature/bug fix
Security Zines.com In Collaboration with ByteByteGo
Designed With Love By @ Sec_YO

Tout ça pour un Data scientist, ça fait beaucoup !

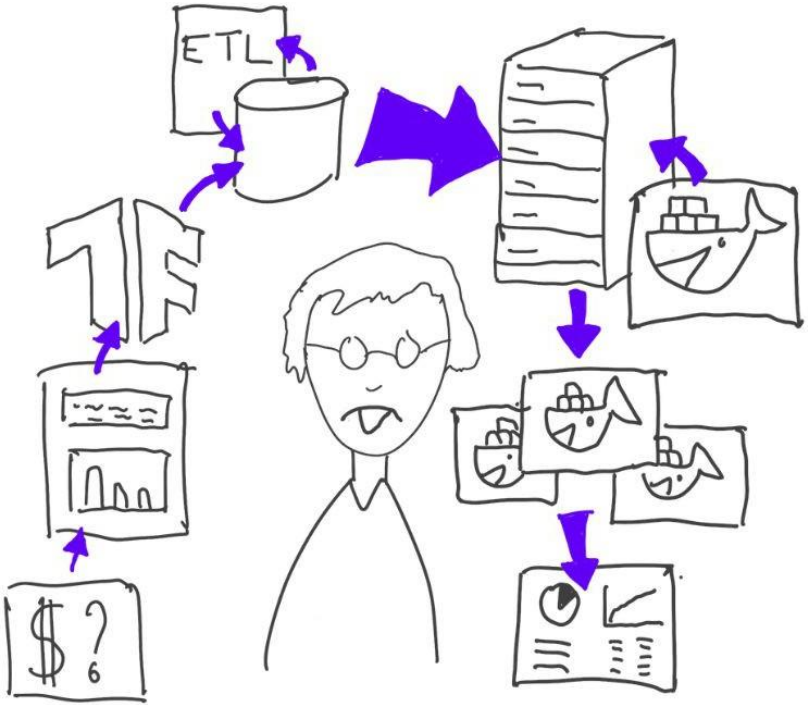


Illustration : Metaflow

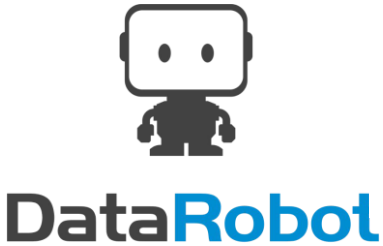
5
Métiers et solutions de MLOps

Métiers du MLOps

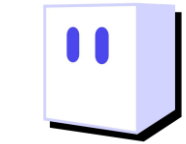
- En fait, tout le MLOps ne repose pas **(en principe)** sur les épaules des seuls Data scientists.
- Ils peuvent compter, selon les cas, sur :
  - des **Data Engineers** chargés des pipelines d'ingestion de données, à partir de technologies Big Data
  - des **ingénieurs DevOps** spécialistes d'infrastructure (Cloud, conteneurs, etc.)
  - des **Machine Learning Engineers,** le nom parfois donné aux data scientists spécialisés dans le déploiement de modèles ML
  - des **développeurs,**
  - **etc.**
- NB : les rôles sont souvent mouvants, leur définition peut varier selon les équipes…

Logiciels de MLOps

- De plus en plus d'entreprises se sont lancées ces dernières années pour créer des logiciels clé en main aidant les Data scientists à réaliser tout ou partie des tâches qui composent le MLOps :



Vertex AI
(Google)



Pour aller plus loin

- *Machine Learning Operations (MLOps): Overview, Definition, and Architecture* : https://arxiv.org/pdf/2205.02302.pdf : revue de littérature, d'outils et recueil d'avis d'experts pour tenter de définir le MLOps.
- Deux cours approfondis sur le MLOps :
  - Made With ML : https://madewithml.com/#mlops
  - ML Systems Design (cours de Stanford) : https://stanford-cs329s.github.io/syllabus.html


FIN