Méthodologie du test - Le test appliqué aux données

© Pierre-Antoine Guillaume

20 Novembre 2024

I. Plan

- Cours 1 : Place du test dans l'ingénérie moderne & Tests unitaires
- Cours 2 : Culture générale du test, différentes formes de test
- Cours 3 : Les tests sur les pipelines de données et sur les data sciences
- Cours 4 : La stratégie de test

II. Rappels

III. Correction

IV. Sortir du model traditionnel

IV. Sortir du model traditionnel - Les architectures classiques

Une majorité des architectures dans le monde professionnel sont globalement des architectures dites « 3-tiers »

- un client
- un backend
- une bdd

La pyramide des tests est assez alignée avec cette philosophie, où l'interaction est au centre du test.

IV. Sortir du model traditionnel - Les architectures classiques

Si Michel est connecté en tant qu'admin

Quand Michel clique sur le bouton «Archiver le mandat»

Alors le mandat a changé d'état

Alors une notification est envoyée au mandataire

Ce qui sort de ce contexte est couvert par des tests très couteux dits *end-to-end* (cf cours 2).

IV. Sortir du model traditionnel - Travailler avec un pipeline

À la différence des programmes classiques, les programmes associés aux *data* sont souvent des pipelines :

- des process longs
- des process chers à executer
- des process critiques
- avec des jalons

IV. Sortir du model traditionnel - Travailler avec un pipeline

Par exemple, côté engineering, le process est souvent automatisé, mais à haut risque pour les consommateurs.

1. *On prend de la data dans aws et on la met dans Snowflake*
2. *On prend de la data dans un serveur bare metal, et on la met dans Snowflake*
3. *On nettoie la data brut*
4. *On transforme la data en information*
5. *On interprete l'information*
6. *On reroute une partie de l'information dans l'application A*
7. *On reroute une partie de l'information dans l'application B*

IV. Sortir du model traditionnel - Travailler avec un pipeline

Pour le côté data science, la plupart du process est généralement manuel mais également à haut risque

1. *Récupération*
2. *Exploration*
3. *Pré-processing*
4. *Entrainnement*
5. *Évaluation*
6. *Livraison*

IV. Sortir du model traditionnel - Travailler avec un pipeline

Ces modèles s'intègrent mal avec une approche statique d'un programme testé lors d'une phase de pré-production avant d'être déployé

En ce sens, la pyramide des tests n'est pas suffisante car elle engendre des tests trop couteux.
Comment faire pour que tout se passe bien ?

IV. Sortir du model traditionnel

Comment faire pour que tout se passe bien ?

- On ne met pas en prod un système moins performant
- On ne met pas en prod un système moins efficace
- On se focalise sur la valeur produite par notre système
- On se focalise sur le service qu'elle rend à ses consommateurs

IV. Sortir du model traditionnel

Comme pour le reste des *informaticiens* :

- On teste automatiquement
- On trace
- On observe
- On fait des petits pas
- On évalue en continu le bon déroulement du programme

V. Logger

V. Logger - L'intérêt d'avoir de bons logs

- **Traçabilité** : Les logs permettent de suivre l'exécution d'un programme et de comprendre son comportement
- **Diagnostic** : Identifier rapidement les anomalies et les dysfonctionnements
- **Auditabilité** : Garder une trace des actions effectuées pour des raisons de conformité ou d'audit

V. Logger - Les différents niveaux de logs

Il existe habituellement 8 niveaux de logs, dont voici un extrait :

- **DEBUG** : Détails techniques pour le développement
- **INFO** : Informations générales sur l'état du système
- **ERROR** : Problèmes empêchant une opération de se terminer correctement

(Plus de détails ici `https://en.wikipedia.org/wiki/Syslog#Severity_level`)
En python, logger.info("message") permet de créer un log de *sévérité info*. Les frameworks de logs (dont python.logging) permet de paramétrer un seuil pour décider de quels logs doivent être émis ou pas.

V. Logger - Permettre l'inspection

- Les logs doivent permettre de comprendre le déroulement des opérations après coup
- Un bon log inclut des informations comme :
  - La date et l'heure de l'événement
  - Le contexte de l'exécution (ex : ID utilisateur, service impacté)
  - Le détail de l'action effectuée (avec les ID des entités concernées et les données problématiques)

V. Logger - Permettre le debug

- **Rôle clé des logs** :
  - Identifier où une erreur est survenue
  - Comprendre les conditions ayant conduit au problème
- **Conseils** :
  - Ajouter des logs dans les blocs d'erreurs (try/catch)
  - Inclure des informations contextuelles (ex : variables)
  - Ajouter des logs en début et en fin de traitement
  - Dans les cas où le monitoring est important, on peut ajouter des logs de débug dans tous les chemins

V. Logger - Ne pas logger d'informations confidentielles

- **Risque de sécurité** : Les logs peuvent exposer des données sensibles
- **Exemples d'informations à ne pas logger** :
  - Mots de passe
  - Données personnelles identifiables (PII) : noms, adresses, numéros de carte
  - Des tokens de téléphones pour les notifs)
  - Clés API ou secrets
- **Bonnes pratiques** :
  - Masquer ou anonymiser les données sensibles
  - Restreindre l'accès aux fichiers de logs

V. Logger - Permettre une compréhension globale

- Utiliser des **transaction IDs** ou **correlation IDs** :
  - Un identifiant unique attaché à une transaction ou une requête
  - Permet de suivre une requête à travers différents services ou étapes d'un pipeline
- **Exemple d'utilisation** :
  - Une requête utilisateur est traitée par plusieurs microservices
  - Le transaction ID permet de reconstituer tout le parcours de la requête dans les logs

V. Logger - Permettre d'aggréger des informations

- **Agrégation de logs** :
  - Centraliser les logs provenant de plusieurs services
  - Exemple d'outils : ELK Stack (Elasticsearch, Logstash, Kibana), Grafana Loki
- **Avantages** :
  - Identifier des tendances ou des schémas récurrents
  - Simplifier l'analyse grâce à des tableaux de bord

VI. Monitorer

VI. Monitorer - Pourquoi faire du monitoring ?

- **Détection rapide des anomalies** : Identifier des comportements anormaux ou des pannes
- **Prévention proactive** : Repérer des tendances avant qu'elles ne deviennent critiques
- **Mesurer les performances** : Suivre l'état de santé des systèmes et pipelines en temps réel
- **Aider au diagnostic** : Localiser et comprendre les problèmes en production

VI. Monitorer - Bonnes pratiques en monitoring

- **Identifier les KPIs clés** : Choisir les métriques critiques pour votre système
- **Configurer des alertes intelligentes** : Éviter les alertes inutiles ou redondantes
- **Suivre les tendances dans le temps** : Analyser les données historiques pour prévoir les problèmes
- **Automatiser le monitoring** : Intégrer des outils de monitoring dès le développement
- **Corréler logs et métriques** : Faciliter le diagnostic grâce à une vue holistique

VI. Monitorer - Cas pratique : Monitoring d'un pipeline de données

- **Métriques à surveiller** :
  - Taux de traitement des données par étape (latence, débit)
  - Taux d'erreurs de transformation ou de validation
- **Alertes** :
  - Configurer une alerte si le taux d'erreurs dépasse un seuil critique

VI. Monitorer - Un exemple d'outils pour le monitoring

Grafana : Suivi des métriques système et applicatives
`https://grafana.wikimedia.org/`
status wikimedia : `https://www.wikimediastatus.net`

VI. Monitorer

On monitore pour pallier aux défauts qui ont dépassé les tests

VII. Automatiser les vérifications manuelles

VII. Automatiser les vérifications manuelles - Code, Data et Modèles

- **Code** : Vérifications sur le fonctionnement et la robustesse
- **Données** : Validation de la qualité et de l'intégrité des données
- **Modèles** : Tests des performances et de la pertinence des modèles

VII. Automatiser les vérifications manuelles - Code

- **Tests unitaires** : Tester des fonctions spécifiques
- **Tests d'intégration** : Valider les interactions entre plusieurs composants
- **Coverage** : Vérifier que votre code est bien couvert par les tests
- **Linting** : Utiliser des outils automatiques pour vérifier la qualité du code

Rappel : Ce sont les notions abordées pendant les cours précédents

VII. Automatiser les vérifications manuelles - Données

- **Validation des features** :
  - Vérifier chaque champ livré : type, contenu attendu, valeurs aberrantes
  - Raffiner par type de champs (ex : numériques, enums, ...)
- **Validation des pipelines** :
  - Tester chaque étape du pipeline de transformation des données
  - Informer en cas d'échec les acteurs pour qu'ils puissent prendre en main au plus tôt et rétablir leur flux de valeur
- **Lien de données** :
  - Vérifier les relations entre plusieurs sources ou pipelines

VII. Automatiser les vérifications manuelles - Modèles

- **Métriques d'évaluation** :
  - Assurez-vous que chaque époque améliore les métriques choisies
  - Vérifiez que les early stops fonctionnent correctement
- **Qualité des données entrantes** :
  - Détecter automatiquement les anomalies dans les données de formation
  - Valider que les données respectent les critères attendus

VII. Automatiser les vérifications manuelles - baseline

Une fois un modèle édité, assurez vous d'enregistrer un *snapshot* des metrics du modèle
Et à chaque nouveau modèle, assurez vous que le nouveau modèle est *au moins aussi performant que le précédent*.

VII. Automatiser les vérifications manuelles - évaluation automatique d'un modèle

Reservez un petit échantillon de données représentatif en répartition en classes

(*un split*)

VII. Automatiser les vérifications manuelles - Splits

```
from sklearn.model_selection import train_test_split
import pandas as pd

data = pd.read_csv("transactions.csv")
X = data[["amount", "product_id"]] # Features
y = data["label"] # Labels

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print("Repartition des labels dans y_train :")
print(y_train.value_counts(normalize=True))
print("Repartition des labels dans y_test :")
print(y_test.value_counts(normalize=True))
```

VII. Automatiser les vérifications manuelles - évaluation automatique d'un modèle

Et lancez ce modèle sur ces données pour vérifier le resultat de l'apprentissage sur des données que le modèle n'a jamais vu

VII. Automatiser les vérifications manuelles

De manière générale, testez automatiquement pour pouvoir refactoriser en sécurité, et en sortant le facteur humain de l'équation

VIII. Great Expectations

VIII. Great Expectations - Introduction à Great Expectations

- **Great Expectations** est un framework pour valider la qualité des données
- Il permet de définir des règles, appelées **expectations**, qui vérifient automatiquement vos données
- Les tests peuvent être appliqués sur des données brutes, des transformations intermédiaires, ou des données en sortie
- C'est une solution qui s'adapte à plusieurs technologies (pandas, Spark, Sqlite, pgsql, maria, ...)

VIII. Great Expectations - Globalement

```
import pandas as pd
import pytest
import great_expectations as ge

@pytest.fixture
def data():
    return pd.DataFrame({"region": ["North", "East", "Invalid"]})

def test_region_values_in_set(data):
    data_ge = ge.from_pandas(data)
    result = data_ge.expect_property_to_be(...) # exemple factice
    assert result["success"], f"Unexpected:{result['result']['unexpected_list']}"
```

VIII. Great Expectations - Exemple : expect_column_values_to_be_in_set

- **Objectif** : Vérifier que les valeurs d'une colonne appartiennent à un ensemble défini
- **Cas pratique** :
  - Colonne : `region`
  - Contrainte : Les valeurs doivent être dans `["North", "East", "South", "West"]`

VIII. Great Expectations - Exemple : expect_column_values_to_be_in_set

```
def test_region_values_in_set(data):
    data_ge = ge.from_pandas(data)
    result = data_ge.expect_column_values_to_be_in_set(
        column="region",
        value_set=["North", "East", "South", "West"]
    )
    assert result["success"], f"Unexpected: {result['result']['unexpected_list']}"
```

VIII. Great Expectations - Exemple : expect_column_values_to_not_be_null

- **Objectif** : S'assurer qu'une colonne ne contient pas de valeurs nulles
- **Cas pratique** :
  - Colonne : `amount`
  - Contrainte : Toutes les valeurs doivent être non-nulles

VIII. Great Expectations - Exemple : expect_column_values_to_not_be_null

```
def test_column_not_null(data):
    data_ge = ge.from_pandas(data)
    result = data_ge.expect_column_values_to_not_be_null(column="amount")
    assert result["success"], f"Null values found in column: amount"
```

VIII. Great Expectations - Exemple : expect_column_values_to_be_unique

- **Objectif** : S'assurer que les valeurs d'une colonne sont uniques
- **Cas pratique** :
  - Colonne : `transaction_id`
  - Contrainte : Chaque transaction doit avoir un identifiant unique
- **Code avec pytest et Great Expectations** :

VIII. Great Expectations - Exemple : expect_column_values_to_be_unique

```
def test_unique_transaction_id(data):
    data_ge = ge.from_pandas(data)
    result = data_ge.expect_column_values_to_be_unique(column="
        transaction_id")
    assert result["success"], "Duplicate transaction IDs found"
```

IX. Redescendre dans la pyramide

IX. Redescendre dans la pyramide - Redescendre dans la pyramide

- Objectif : Transformer des tests coûteux en tests plus spécifiques et rapides
- Approches : Utiliser des concepts comme les stratégies, variables d'environnement, DSN, et respecter les principes 12 Factor

IX. Redescendre dans la pyramide - Stratégie : Design Pattern

- **Définition** : Le pattern Stratégie permet de définir une famille d'algorithmes interchangeables, et de choisir dynamiquement lequel utiliser
- **Utilité** :
  - Facilite le remplacement d'une dépendance ou d'un comportement
  - Simplifie le test unitaire en isolant les comportements
- **Avantage pour les tests** : Vous pouvez injecter une stratégie simulée ou simplifiée pour tester un composant sans dépendre de l'implémentation réelle

IX. Redescendre dans la pyramide - Exemple : Utiliser le pattern stratégie

```
class BookRepository:
    def save(self, data):
        raise NotImplementedError()

class SqlBookRepository(BookRepository):
    def save(self, data):
        self.database.insert(data)

class InMemoryBookRepository(BookRepository):
    def save(self, data):
        self.dictionnary[data.id] = data
```

IX. Redescendre dans la pyramide - Exemple : Utiliser le pattern stratégie

```
class BookRepository:
    def save(self, data): [...]

class MyLibraryApplication:
    def __init__(self, book_repository: BookRepository):
        self.book_repository = book_repository

    def add_book(self, data):
        return self.book_repository.save(data)

def test_library():
    mock_strategy = InMemoryBookRepository()
    application = MyLibraryApplication(mock_strategy)
    application.add_book({"id": 25, "title": "hello"});
    assert mock_strategy.dictionnary[25] == {"id": 25, "title": "hello"}
```

IX. Redescendre dans la pyramide - DSN : standardiser vos connexions

- **Définition** : Un DSN est une chaîne unique qui contient toutes les informations nécessaires pour se connecter à une source de données
- **Exemples courants** :
  - Base de données : `postgres://user:password@host:port/dbname`
  - Monitoring : URL avec token d'accès
- **Avantages** :
  - Uniformise la configuration des dépendances
  - Simplifie le partage d'informations de connexion

IX. Redescendre dans la pyramide - Variables d'environnement : paramétrer vos tests

- **Utilité** : Configurer vos applications pour différents environnements (local, test, prod)
- **Exemples** :
  - URL de base de données (`DATABASE_URL`)
  - Niveau de log (`LOG_LEVEL`)
  - Mode de debug (`DEBUG`)
- **Avantages** :
  - Facilite les tests dans des environnements variés
  - Découple la configuration du code

IX. Redescendre dans la pyramide - Exemple : Utiliser une variable d'environnement

```python
import os

class Database:
    def __init__(self):
        self.url = os.getenv("DATABASE_URL", "sqlite:///:memory:")

    def connect(self):
        print(f"Connecting to {self.url}")

# En test
os.environ["DATABASE_URL"] = "sqlite:///test.db"
db = Database()
db.connect() # Connecting to sqlite:///test.db
```

IX. Redescendre dans la pyramide - 12 Factor : des applications robustes

- **Définition** : Une méthodologie pour construire des applications modernes, évolutives et fiables
- **Principes liés au test** :
  - Configuration dans des variables d'environnement
  - Logs comme flux de sortie standard (STDOUT/STDERR)
  - Parité entre les environnements de dev, test et prod
- **Avantages** :
  - Réduction des bugs liés à des écarts de configuration
  - Déploiement simplifié et standardisé

`https://12factor.net/fr/`

X. Pour aller plus loin

X. Pour aller plus loin - Des références

- Site : MadeWithMl (`https://madewithml.com/` par Goku Mohandas)
- Livre : Working Effectively with Legacy Code (par Michael Feathers)
- Site : Continuous Delivery for Machine Learning (`https://martinfowler.com/articles/cd4ml.html` hosté sur martinfowler.com)
- Livre : Test-Driven Development by Example (par Kent Beck)