Les tests

Introduction- Pourquoi les tests garantissent une plus grande fiabilité du modèle

- Pour tout scientifique des données l'écriture de tests devrait être une priorité absolue pour créer des logiciels **maintenables** et des **modèles fiables** et **faciles** à **surveiller**.
- Il est intéressent d'appliquer différents codes python pour traiter vos données dans votre **notebook**, mais pour rendre votre code **reproductible**, vous devez les mettre dans des **fonctions** et des **classes**.
- Lorsque vous insérez votre code dans le **script**, le code peut être **interrompu** à cause de certaines fonctions. Même si votre code ne casse pas, **comment** savoir si votre fonction fonctionnera comme prévu?

Pourquoi les tests garantissent une plus grande fiabilité du modèle

- l'écriture de tests est la clé de l'écriture de logiciels maintenables. Ceci est particulièrement important pour la science des données qui suit un processus itératif de **recherche** et de **développement** qui passe par les étapes de **nettoyage des données**, **d'analyse des données**, **d'ingénierie des fonctionnalités** et de **développement de modèles**. À chaque fois, cette rotation s'appuie sur le travail effectué lors d'une itération précédente - la modification, par exemple, du code utilisé pour générer une fonctionnalité.
- **Comment** pouvons-nous être sûrs que les changements que nous apportons n'affectent pas involontairement d'autres étapes du pipeline?

Exemple

- Par exemple, nous créons une fonction pour extraire le sentiment d'un texte avec TextBlob , une bibliothèque Python pour le traitement des données textuelles. Nous voulons nous assurer que cela fonctionne comme prévu: la fonction renvoie une valeur **supérieure** à **0** si le test est **positif** et renvoie une valeur **inférieure** à **0** si le texte est **négatif**.

```
from textblob import TextBlob

def extract_sentiment(text: str):
        '''Extract sentiment using textblob.
        Polarity is within range [-1, 1]'''

        text = TextBlob(text)

        return text.sentiment.polarity
```

Suite- Exemple

- Pour savoir si la fonction retournera la bonne valeur à chaque fois, le meilleur moyen est d'appliquer les fonctions à différents exemples pour voir si elle produit les résultats souhaités. C'est à ce moment que les **tests deviennent importants**.

Suite-

En général, vous devez utiliser les tests pour vos projets de science des données, car cela vous permet de:

- Assurez-vous que le code fonctionne comme prévu
- Détecter les cas extrêmes
- N'hésitez pas à échanger votre code existant avec un code amélioré sans avoir peur de casser tout le pipeline
- Vos coéquipiers peuvent comprendre vos fonctions en regardant vos tests

Structurez vos projets

- Lorsque notre code grossit, nous pourrions vouloir placer les fonctions de **science des données** et les fonctions de **test** dans **2 dossiers différents.** Cela nous permettra de trouver plus facilement l'emplacement de chaque fonction.
- Nommez notre fonction de test avec **test_<name>.py** ou **<name>_test.py**.
- **Pytest** recherchera le fichier dont le nom se termine ou commence par **«test»** et exécute les fonctions dont le nom commence par **«test»** dans ce fichier.

❑Il existe différentes manières d'organiser vos fichiers

https://docs.pytest.org/en/latest/

Il existe de nombreux outils Python disponibles pour les tests, mais l'outil le plus simple est **Pytest**.

- Pytest est le framework qui facilite l'écriture de petits tests en Python.
- **Pytest** nous aide à écrire des tests avec un minimum de code.
- Si vous n'étiez pas familiarisé avec les tests, pytest est un excellent outil pour commencer.
- Pour installer pytest, exécutez: **pip install -U pytest**

Projet d'automatisation Tox

- https://tox.readthedocs.io/en/latest/

tox est un outil de ligne de commande de test et de gestion virtualenv générique que vous pouvez utiliser pour:

- vérifier que votre package s'installe correctement avec différentes versions et interpréteurs de Python
- exécuter vos tests dans chacun des environnements, configurer votre outil de test de choix
- agissant en tant qu'interface pour les serveurs d'intégration continue

Exemple- tox

- installez tox: **`pip install tox`**
- Ensuite, placez les informations de base sur votre projet et les environnements de test dans lesquels vous souhaitez que votre projet s'exécute dans un fichier **tox.ini** résidant juste à côté de votre fichier **setup.py**

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py27,py36

[testenv]
# install pytest in the virtualenv where commands will be executed
deps = pytest
commands =
    # NOTE: you can run any command line tool here - not just tests
    pytest
```

Suite-tox

- Vous pouvez également essayer de générer un fichier **tox.ini** automatiquement, en exécutant **tox-quickstart** et en répondant à quelques questions simples.
- Pour tester votre projet avec **Python2.7** et **Python3.6**, tapez simplement: **tox**
- Lorsque vous exécutez **tox** une deuxième fois, vous remarquerez qu'il s'exécute beaucoup plus rapidement car il garde une trace des détails de virtualenv et ne recrée ni réinstalle les dépendances.
- Pour savoir plus: https://tox.readthedocs.io/en/latest/examples.html