Automatisation et intégration

Déploiement sur le Cloud - ESLSCA

Maxime Jumelle

www.blent.ai

Plan

1. Versioning sous `git`

2. L'approche CI/CD

    Continuous Integration (CI)

    Tests unitaires

    Tests d'intégration

3. Continuous Delivery (CD)

Versioning sous `git`

Le versioning

Lorsque plusieurs développeurs travaillent sur un même projet, il est important que chacun ait une version **à jour** du programme. Une approche très ancienne consistait à s'envoyer les fichiers soit par transfert FTP, soit du des réseaux intranet ...

Heureusement, ce temps est révolu, et `git` (dont l'ancêtre est souvent considéré comme étant `svn`) est aujourd'hui un **outil indispensable pour toute collaboration en équipe**.

Création du dépôt `git`

La création d'un dépôt **local** doit être effectuée afin la commande `init`.

```
git init
git add .
```

Après `git add`, on peut ajouter n'importe quel fichier ou dossier. Le point indique le répertoire courant et, par défaut et sans fichier *.gitignore*, traque tous les fichiers et dossiers de manière récursive.

Création du dépôt `git`

Pour mettre à jour le dépôt **local**, un `commit` est nécessaire.

```
git commit -am "First push"
git push origin master
```

La commande **push** permet ensuite de pousser le dépôt local vers le dépôt **distant** sur la branche `master`.

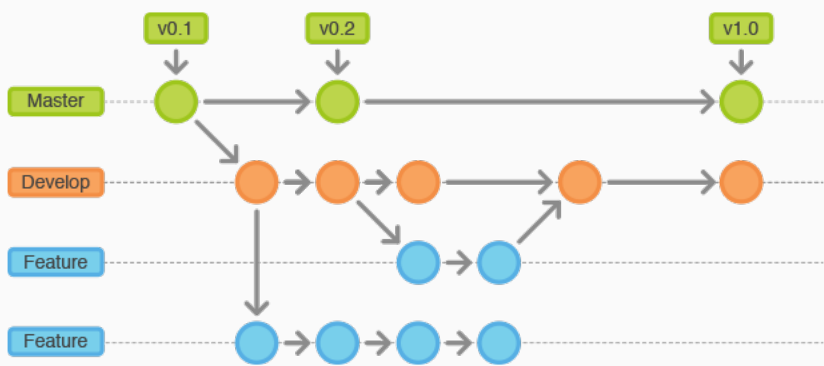
Une histoire de branches

Un des points forts de `git` est l'utilisation des **branches**. Une branche est utilisée pour développer des fonctionnalités isolées, comme par exemple une élaboration d'une nouvelle version.

- Par défaut, la branche *master* est créée et représente souvent la branche « de référence ».
- D'autres branches peuvent alors être créées (*dev* pour version en développement) en fonction de la taille du projet et du nombre de fonctionnalités disponibles.

```
git checkout -b dev
git commit -am "Updated dev version"
git push origin dev
```

Une histoire de branches

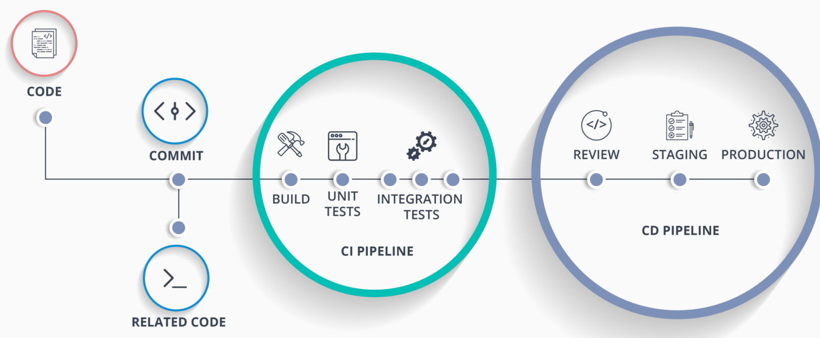


Quoi utiliser ?

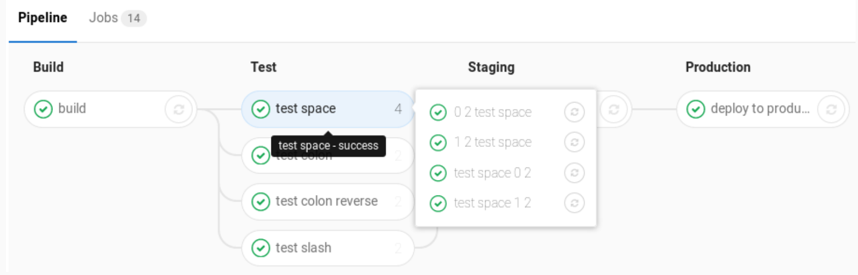
vs.

L’approche CI/CD

La route CI/CD



Pipelines et jobs



Intégration Continue

L'Intégration Continue (ou *Continuous Integration*) est une pratique qui consiste à tester et implémenter de nouvelles fonctionnalités dans un code source, le plus souvent en collaboration avec d'autres développeurs.

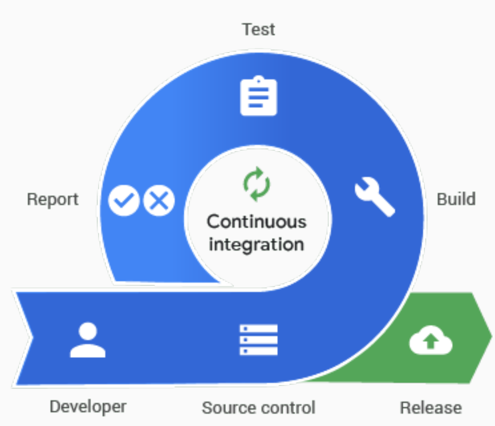
Dans de nombreuses entreprises de différentes tailles (de start-ups jusqu'aux multinationales), **l'approche CI/CD est incontournable**. En effet, de nombreux projets industrialisés s'exécutent de manière entièrement automatisé, et l'approche CI/CD permet de mettre en harmonie toutes les règles qui interviennent lors de l'automatisation.

Pourquoi la CI ?

La CI présente de nombreux avantages dans les environnements de développement (et, dans une autre mesure, dans les environnements de production) :

- cela permet d'éviter l'introduction de conflits entre différentes sources et l'apparition de bugs ;
- réparer les erreurs commises par un autre développeur ;
- s'assurer du bon fonctionnement du programme sur plusieurs architectures cibles (systèmes d'exploitation, navigateur web, ...).

Le cycle CI



Configuration de l'environnement CI

Tout d'abord, il est **fortement conseillé** de créer un **environnement virtuel** dans chaque projet : cela permet de ne garder que les dépendances nécessaires au projet, quitte à tout réinstaller en cas de problèmes.

```
virtualenv venv
source venv/bin/activate
```

Pour cet exemple, nous aurons besoin de scikit-learn pour l'algorithme de Machine Learning, ainsi que de :

- `flake8` pour le *linting*, qui examine en particulier les défauts de syntaxe et le respect à la norme de codage PEP8 ;
- `pytest` pour l'exécution des tests unitaires ;
- `pytest-cov` pour la couverture de code.

Configuration de l'environnement CI

L'installation des dépendances est réalisée par `pip`, donc la commande *freeze* permet de renseigner tous les packages (avec les versions correspondantes) du `pip` local utilisé (en particulier, celui de l'environnement virtuel).

```
pip install sklearn flake8 pytest pytest-cov
pip freeze > requirements.txt
```

Linting

L'outil `flake8` s'assure du bon respect des normes de codage PEP8, et donc de la reproducabilité et maintenabilité du code.

```
flake8 --exclude=venv* --statistics
```

```
./test_classifier.py:3:1: E302 expected 2 blank lines, found
./test_classifier.py:6:7: E114 indentation is not a
multiple of four (comment)
./test_classifier.py:7:7: E111 indentation is not a
multiple of four
./test_classifier.py:8:7: E111 indentation is not a
multiple of four
./test_classifier.py:9:7: E111 indentation is not a
multiple of four
./test_classifier.py:11:5: E301 expected 1 blank line, found
./test_classifier.py:12:7: E111 indentation is not a
multiple of four
4       E111 indentation is not a multiple of four
1       E114 indentation is not a multiple of four
(comment)
1       E301 expected 1 blank line, found 0
1       E302 expected 2 blank lines, found 1
```

L'intérêt de `pytest` est qu'il permet d'associer directement les tests unitaires pour les fichiers dont on souhaite exécuter les tests et vérifier la couverture du code (si l'organisation et le nommage des fichiers est assez intuitif).

```
pytest -v --cov=classifier
```

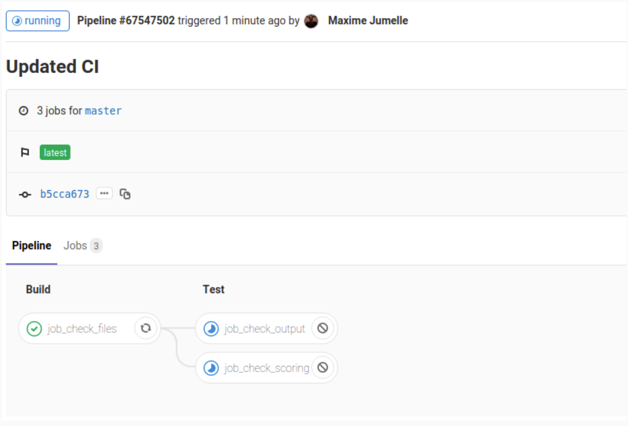
La couverture du code vise à savoir à quel point les tests atteignent assez de fonctionnalités pour éviter des bugs issus de code que l'on aurait pu oublier de prendre en compte.

```
cachedir: .pytest_cache
rootdir: /home/maxime/.../
plugins: cov-2.7.1
collected 2 items

test_classifier.py::TestClassifier::test_prediction
PASSED [ 50%]
test_classifier.py::TestClassifier::test_score
PASSED [100%]

---coverage: platform linux, python 3.5.2-final-0---
Name                Stmts    Miss   Cover
-----------------------------------------
classifier.py          15       0    100%
```

TP

Dans ce TP, nous allons ré-utiliser le code existant, mais cette fois-ci, nous allons rajouter de nouveaux tests.

- Tout d'abord, on vérifiera que certains fichiers sont présents dans le dépôt `git`.
- Ensuite, on testera les fonctionnalités suivantes :
  - La bonne consistance des prédictions du modèle ;
  - Un score minimal à atteindre sur un ensemble de test supérieur à un estimateur naïf ;
  - Un score minimal à atteindre pour éviter une régression (fixé à 80% de la métrique 0/1).


running
Pipeline #67547502 triggered 1 minute ago by Maxime Jumelle
Updated CI
3 jobs for master
latest
b5cca673
Pipeline
Jobs 3
Build
Test
job_check_files
job_check_output
job_check_scoring

Continuous Delivery (CD)

Heroku comme SaaS

Créer un compte sur Heroku et installer le client `heroku`.

MacOS

```
brew install heroku/brew/heroku
```

Ubuntu 16+

```
sudo snap install heroku --classic
```

Windows : installer l'exécutable (32 ou 64 bits).

Authentifier la connexion.

```
heroku login
```

```
heroku: Press any key to open up the browser to login
or q to exit
      Warning: If browser does not open, visit
      https://cli-auth.heroku.com/auth/browser/***
heroku: Waiting for login...
Logging in... done
Logged in as me@example.com
```

Variables d’environnements CD

Lors du CD, il est souvent nécessaire de déployer le projet sur un serveur distant : une authentification (SSH, LDAP, SAML, ...) est requise. **Il ne faut jamais placer d’informations sensibles dans le dépôt !** Il est préférable de configurer des **variables d’environement** CD.

Job de déploiement

Le job de déploiement a pour objectif de pousser le projet vers le serveur distant. Notons que l'exécution du projet **est à la charge du serveur distant**, l'opération CI/CD de GitLab s'arrêtant uniquement à la mise à jour du code source du serveur distant.

```
job_deploy:
  stage: deploy
  script:
  - dpl --provider=heroku
    --app=eslsca-bigdata-test
    --api-key=$HEROKU_KEY
  only:
  - master
```

TP

La dernière partie du TP concerne la mise en place du CD sur un serveur distant géré par Heroku. En particulier, les tâches à réaliser sont

- Configurer le Runner GitLab et les variables d'environnement.
- Définir le *job* de déploiement.
- Mettre à jour le dépôt pour effectuer une première exécution de la *pipeline*.
- Ajouter une route */predict* avec la méthode **POST** dont l'objectif est de prédire la probabilité l'issue d'une observation à partir d'un fichier JSON.

TP