

Intégration et déploiement en continue (CI/CD) d'une API Flask et Docker sur AWS Elastic Beanstalk à l'aide de Travis-ci



Table des matières

INTRODUCTION	3
1 ARCHITECTURE DU STACK AWS	4
1.1 DESCRIPTION DU FLUX UTILISATEUR	4
1.1.1 <i>Route 53</i>	4
1.1.2 <i>Elastic Beanstalk</i>	5
1.1.3 <i>Docker</i>	7
1.2 DESCRIPTION DE L'INTEGRATION ET DEPLOIEMENT EN CONTINUE DU MODELE AVEC TRAVIS-CI.....	7
2 ARCHITECTURE DU CODE PYTHON ET FLASK.....	9
2.1 ARBORESCENCE DU CODE.....	9
2.2 1 ^{ER} SOUS PROJET : APPRENTISSAGE ET SAUVEGARDE DU MODELE DE PREDICTION	10
2.2.1 <i>Formatage des données Excel</i>	10
2.2.2 <i>Séparation de la donnée en train, test avant processing</i>	11
2.2.3 <i>Création d'un Pipeline de processing de la donnée</i>	11
2.2.3.1 Création d'une liste de Transformers.....	12
2.2.3.2 Instanciation d'un objet Pipeline	13
2.2.3.2.1 Méthode fit_transform de classe Pipeline.....	14
2.2.3.2.2 Sauvegarde du Pipeline en format pickle	14
2.2.4 <i>Création du modèle de Machine Learning pour la prédiction</i>	15
2.2.4.1 Pipe de création, tuning d'hyperparamètre et sauvegarde du modèle.....	15
2.3 2 ^{EME} SOUS PROJET : API FLASK	15
2.3.1 <i>Description du script application.py</i>	15
2.3.2 <i>Decorator et route dans controller.py</i>	16
2.3.3 <i>Fonction predict</i>	16
2.3.4 <i>Résultats du call API</i>	18
CONCLUSION.....	18

Introduction

Dans le cadre du test technique proposé par Fairmoney, où l'objectif était de produire un script d'entraînement et de déploiement d'un modèle de Machine Learning sous forme d'API.

J'ai ainsi décidé de développer une architecture modulaire (programmation objet) en Python permettant d'éviter toute redondance de code, maintenable et réutilisable pour la prédiction de nouvelles données ou pour tout autre projet. Une architecture Flask a été mise en place pour pouvoir faire de la prédiction sur de nouvelles données à l'aide d'un modèle de pipeline de transformation de données et un algorithme (LightGbm) entraîné sur le jeu de données fourni. Pour faciliter la mise en production et éviter toutes erreurs de dépendances et d'incompatibilité de système d'exploitation, le projet est « containerisé » grâce à la technologie Docker.

Une fois les modèles de Machine Learning entraînés et sauvegardés, et une fois le script Flask développé et déployé sur Github sur la branche Master, la plateforme d'intégration et de déploiement en continue se charge de construire l'image du docker, de faire des tests unitaires puis de déployer le container Docker sur le service AWS Elastic Beanstalk qui permet de facilement déployer et monter en charge, c'est-à-dire d'augmenter le nombre de serveurs en cas de fort trafic. Le résultat se trouve sur le site déployé : <https://ngjohn.site>

L'objectif ici, est d'illustrer au maximum à travers ce projet mes compétences en termes de programmation, de Machine Learning et de devops mais également de montrer ma motivation à rejoindre FairMoney.

Ces compétences sont requises puisque selon moi, le poste de Machine Learning Engineer est au croisement du métier de Data Scientist et du Software Engineer. Le Data Scientist se charge de développer des algorithmes de Machine Learning, et le Software Engineer se charge quant à lui de construire une architecture en réutilisant le modèle développé par le Data Scientist. Le Machine Learning Engineer est ainsi constamment challengé par le fait d'arbitrer entre performance de la prédiction et la déployabilité du modèle en production.

Ainsi dans une première partie, il sera question d'une description du choix de l'architecture globale du projet (Stack AWS et Travis-CI). Puis dans un second temps, nous rentrerons en détail sur l'architecture du projet Python Flask.

1 Architecture du Stack AWS

Backend

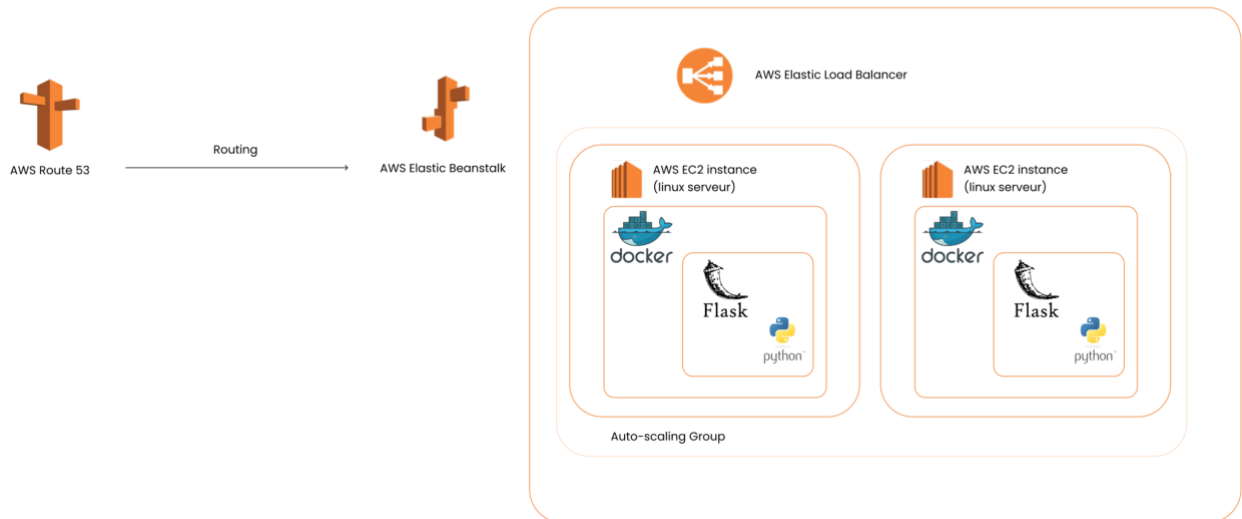


Figure 1 Flux globale de l'architecture du projet sur AWS

Il sera question dans cette partie d'une description du flux de l'API Flask

1.1 Description du flux utilisateur

1.1.1 Route 53

Lorsqu'un utilisateur arrive sur l'endpoint <https://ngjohn.site>, le service **Route 53** d'AWS est activé et se charge d'acheminer l'utilisateur vers des applications Internet en traduisant le nom de domaine en adresse IP. L'adresse IP associée dans notre cas est celui du service Elastic Beanstalk correspondant à la ressource hébergeant l'API Flask



Figure 2 Flux AWS Route 53

1.1.2 Elastic Beanstalk

Il a été choisi d'utiliser le service **AWS Elastic Beanstalk**, car il permet de déployer et mettre à l'échelle très facilement des applications ou des services Web développés en Node.js, Python, et Docker sur des serveurs familiers, tels qu'Apache ou Nginx.

Il suffit de charger le code en format .zip sur la plateforme ou via des lignes de commandes ou même par intégration continue grâce à des plateformes comme Travis-ci (solution utilisée ici) ou AWS CodePipeline.

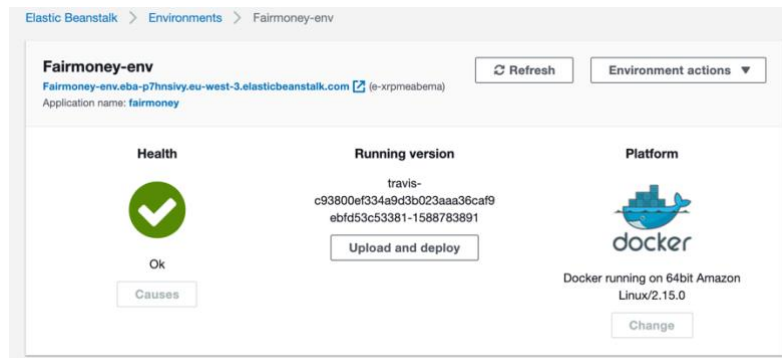


Figure 3 Capture d'écran du service Elastic Beanstalk en mode console

« Elastic Beanstalk effectue automatiquement les étapes du déploiement que sont le dimensionnement des capacités, l'équilibrage de la charge, le dimensionnement automatique et la surveillance de l'état de l'application. Ce faisant, vous conservez la maîtrise totale des ressources AWS alimentant votre application et pouvez accéder aux ressources sous-jacentes à tout moment. »

(Source : <https://aws.amazon.com/fr/elasticbeanstalk/>)

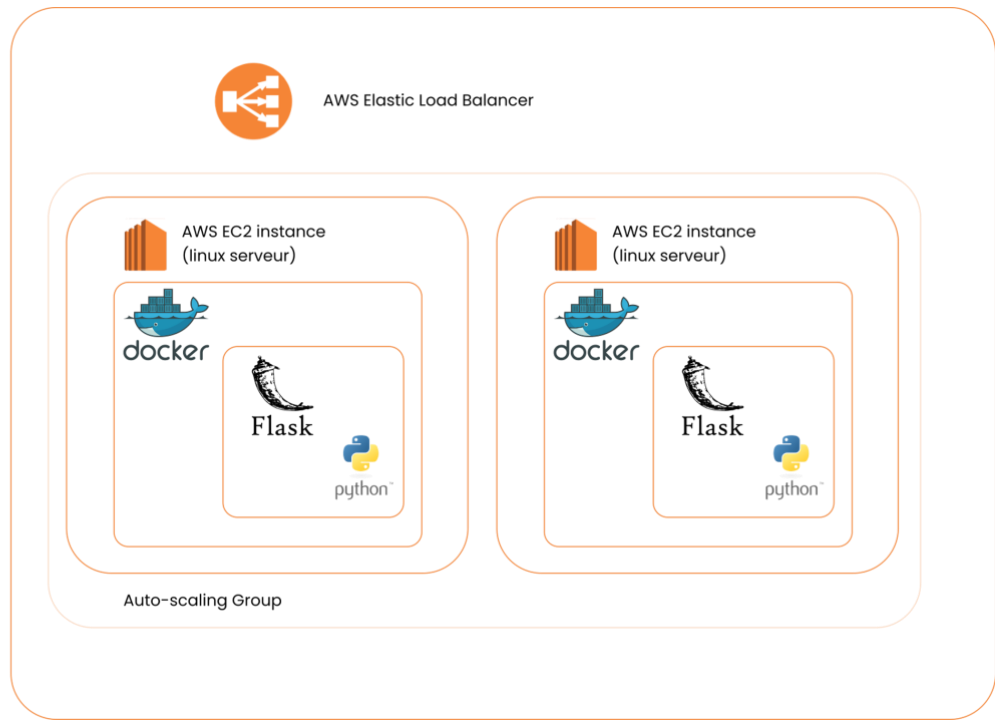


Figure 4 Détail de l'architecture Elastic Beanstalk

Le service Elastic Beanstalk permet ainsi de mettre à l'échelle les serveurs EC2 contenant le site, c'est-à-dire dupliquer les serveurs pour pouvoir répondre au trafic croissant. Ceci est possible grâce à l'équilibreur de charge Elastic Load Balancer (ELB) qui est contenue dans le service Elastic Beanstalk (EB à ne pas confondre avec ELB). (Voir Figure 4)

Le programme Python Flask a été containerisé via Docker ce qui nous permet de ne pas se soucier des dépendances, et d'éviter tout problème d'incompatibilité de système d'exploitation (OS).

1.1.3 Docker

```
FROM ubuntu:latest
MAINTAINER "Johnathan NGUYEN"

RUN apt-get update \
    && apt-get install -y python3-pip python3-dev \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 install --upgrade pip

WORKDIR '/app'

COPY requirements.txt ./
RUN pip3 install -r requirements.txt
```

Figure 5 Dockerfile du projet flask

Pour containeriser du code, il est nécessaire d'avoir un fichier « Dockerfile » à la racine de notre projet pour instancier un système d'exploitation contenant seulement les fichiers de notre projet, les dépendances à installer et la commande permettant d'exécuter le code. En d'autres termes, ceci est appelé une image et permet de créer des containers avec ses spécificités.

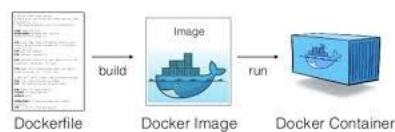


Figure 6 Schéma simplifié de Docker

Après avoir vu les explications sur l'architecture globale du stack AWS, nous allons voir plus en détail le code Python permettant de créer le modèle de prédiction. Il convient également de voir en détail le code Flask permettant la prédiction.

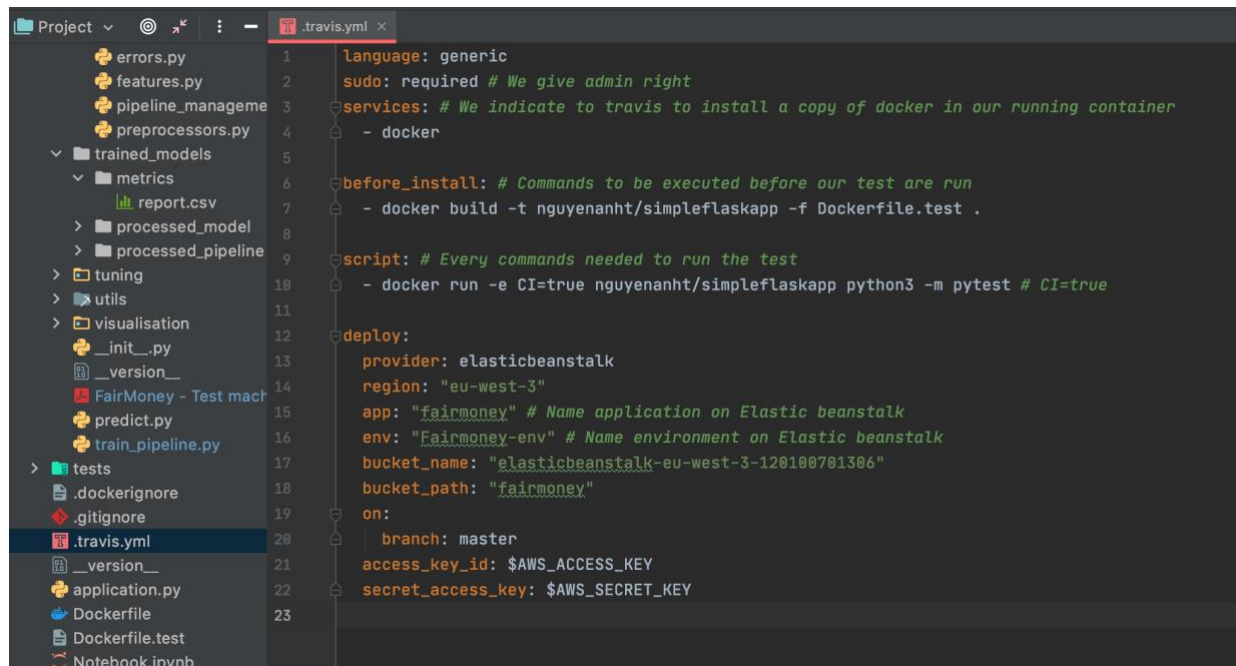
1.2 Description de l'intégration et déploiement en continue du modèle avec Travis-ci



Figure 7 Flux intégration et déploiement en continue avec Travis-ci

Il a été choisi, d'utiliser un stack Github, Travis-ci pour l'intégration et déploiement en continue. En effet à chaque « push » de code sur la branche Master du repository github, une batterie de test est effectuée grâce à Travis-ci et si ces tests sont positifs, travis se charge de déployer sur AWS Elastic Beanstalk.

Toutes ces étapes sont renseignés dans un fichier `.travis.yml` à la racine du projet



```
1 language: generic
2 sudo: required # We give admin right
3 services: # We indicate to travis to install a copy of docker in our running container
4   - docker
5
6 before_install: # Commands to be executed before our test are run
7   - docker build -t nguyenanht/simpleflaskapp -f Dockerfile.test .
8
9 script: # Every commands needed to run the test
10   - docker run -e CI=true nguyenanht/simpleflaskapp python3 -m pytest # CI=true
11
12 deploy:
13   provider: elasticbeanstalk
14   region: "eu-west-3"
15   app: "fairmoney" # Name application on Elastic beanstalk
16   env: "Fairmoney-env" # Name environment on Elastic beanstalk
17   bucket_name: "elasticbeanstalk-eu-west-3-120100701306"
18   bucket_path: "fairmoney"
19   on:
20     branch: master
21   access_key_id: $AWS_ACCESS_KEY
22   secret_access_key: $AWS_SECRET_KEY
23
```

Figure 8 Fichier `travis.yml` permettant de configurer l'intégration et déploiement en continue

Ainsi, nous avons vu dans cette première partie le détail du fonctionnement sur le stack AWS et le détail de l'architecture d'intégration et de déploiement en continue.

2 Architecture du code Python et Flask

2.1 Arborescence du code

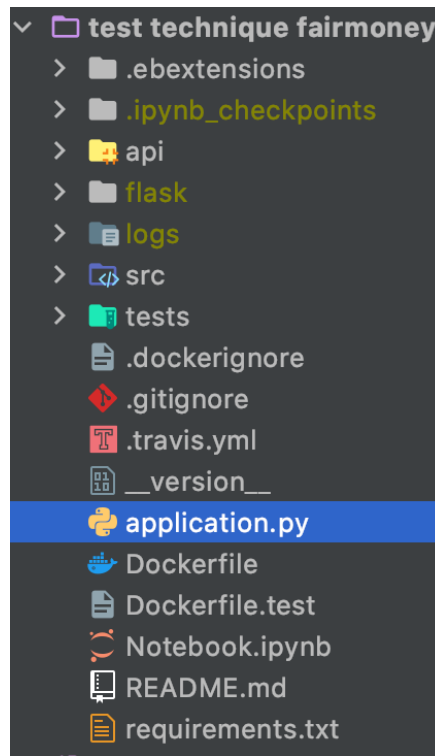


Figure 9 Structure générale du code Python Flask

Le projet est découpé en 2 sous projets ou modules importants :

- Le module api qui contient « l'intelligence » du code Flask
- Le module src qui contient tout le code permettant de générer un modèle de transformation de données et un modèle de prédiction.

Ainsi, nous verrons en détail le code permettant de générer les modèles évoqués ci-dessus.

2.2 1^{er} sous projet : Apprentissage et sauvegarde du modèle de prédiction

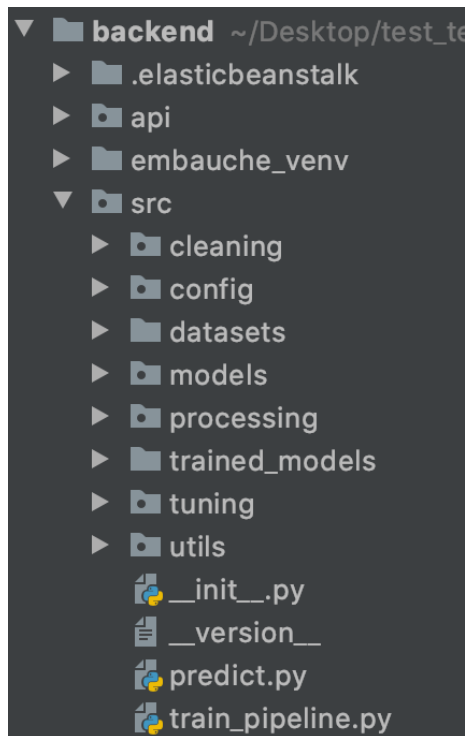


Figure 10 Arborescence module src

Le code train_pipeline.py est le script principal du module src. C'est lui qui orchestre l'apprentissage et la sauvegarde des modèles.

2.2.1 Formatage des données Excel

```
train_pipeline.py x
23 def run_training() -> None:
24     """Train the model."""
25
26     # Read Training set
27     # =====
28     data_mngmnt = Data()
29     data_mngmnt.format_excel_to_csv(filename=config.EXCEL_FILE)
30     data_mngmnt.from_csv(config.DATA_FILE, sep=',')
31     data = data_mngmnt.df
```

Figure 11 Function run_training

2.2.2 Séparation de la donnée en train, test avant processing



```
train_pipeline.py x
33 # Divide dataset
34 # =====
35 X_train, X_test, y_train, y_test = train_test_split(
36     data[config.FEATURES],
37     data[config.TARGET],
38     test_size=0.2,
39     random_state=123,
40     stratify=data[config.TARGET]
41 )
```

Figure 12 Séparation de la donnée en 2 jeux de données : train (70%) et test (30%)

Il convient de séparer le dataset en 2 parties avant tout pre-processing. En effet, sinon une partie de l'information contenue dans le jeu de test serait induite dans les données du train. Ce concept de fuite de données est appelé « data leakage ».

2.2.3 Création d'un Pipeline de processing de la donnée

Pour la phase de processing de la donnée avant l'apprentissage du modèle de Machine Learning, il convient de construire un pipeline de transformation. Un pipeline est un regroupement d'une série d'instruction, ici de transformation de la donnée. Créer un pipeline a l'avantage de condenser les transformations et permet de réutiliser ces transformations plus facilement sur de nouvelles données.

```

train_pipeline.py x
43 # DATA Pipeline
44 # =====
45 # Features
46 transformers_features = [
47     preprocessors.ObjectTypeToCategory(),
48 ]
49
50 preprocessing_pipeline = Pipeline(transformers=transformers_features)
51
52 # TARGET, We remap the target between 0 and 1
53 transformers_target = [
54     preprocessors.RemapTarget(target=config.TARGET, mapping=config.MAPPING_TARGET)
55 ]
56 preprocessing_target_pipeline = Pipeline(transformers=transformers_target)
57
58 # Fit transform X_train
59 X_train = preprocessing_pipeline.fit_transform(X_train[config.FEATURES])
60 X_test = preprocessing_pipeline.transform(X_test[config.FEATURES])
61 # Fit transform y_train
62 y_train = preprocessing_target_pipeline.fit_transform(y_train[config.TARGET])
63 y_test = preprocessing_target_pipeline.transform(y_test[config.TARGET])
64

```

Figure 13 Création de pipeline personnalisé

2.2.3.1 Création d'une liste de Transformers

A la ligne 46 de la figure 11, nous pouvons voir une liste de transformers contenant des objets de type Transformer.

```

train_pipeline.py x preprocessors.py x
69
70 class ObjectTypeToCategory(Transformer):
71
72     def __init__(self, variables=None) -> None:
73         if not isinstance(variables, list):
74             self.variables = [variables]
75         else:
76             self.variables = variables
77
78     def fit_transform(self, df: pd.DataFrame, y: pd.Series = None) -> pd.DataFrame:
79
80         return self.transform(df)
81
82     def transform(self, df: pd.DataFrame) -> pd.DataFrame:
83
84         df = df.copy()
85         obj_list_to_one_hot_encode = list(df.select_dtypes(include=['object']).columns)
86
87         for col in obj_list_to_one_hot_encode:
88             df[col] = df[col].astype('category')
89
90         return df
91

```

Figure 14 Exemple de classe héritant de Transformer

Chacun des objets contenus dans la liste, hérite de la classe Transformer issue du module `src/utls/transformers.py`.

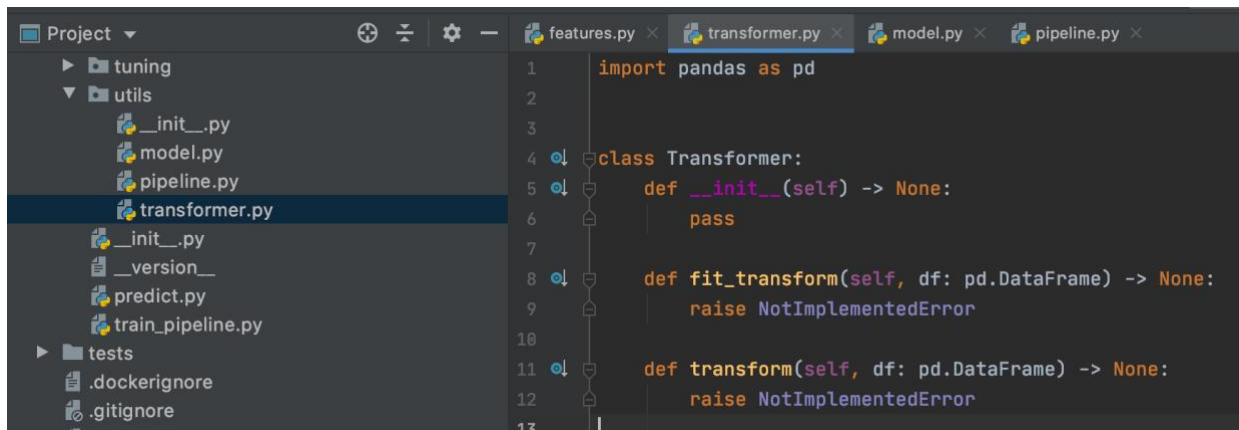


Figure 15 Classe parente Transformer

Ceci nous permet de forcer les classes à avoir une méthode `fit_transform` et `transform` ce qui sera utile pour l'apprentissage du modèle Pipeline de transformation de la donnée.

En effet, pour la lisibilité du code et ne pas oublier de définir des méthodes dans les classes, nous faisons appel au concept d'héritage. Les classes parentes sont toutes contenues dans le module `src/utils`.

2.2.3.2 Instanciation d'un objet Pipeline

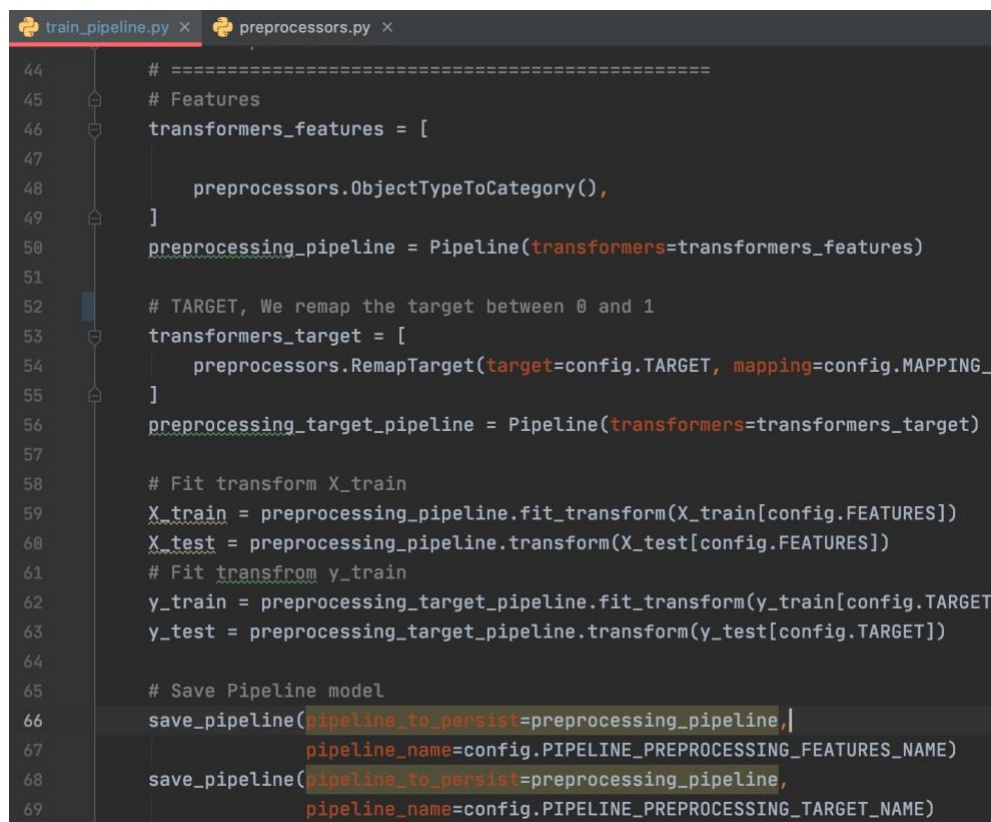


Figure 16 Création du pipeline de Transformers

Nous instancions à la ligne 50 sur la figure ci-dessus, un objet Pipeline avec comme paramètre la liste de Transformers.

2.2.3.2.1 Méthode fit_transform de classe Pipeline

A la ligne 59 sur la figure 14, nous pouvons voir l'avantage d'utiliser le concept d'héritage puisque pour chaque Transformer, nous appliquons la méthode fit_transform associée.

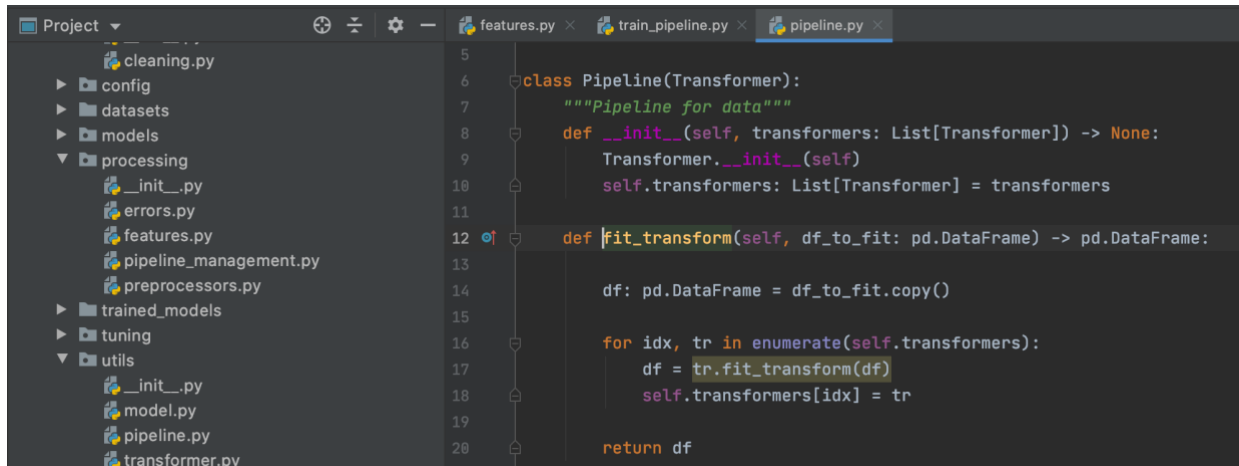


Figure 17 Méthode fit_transform de la classe Pipeline faisant appel aux méthodes fit_transform respectives aux Transformers

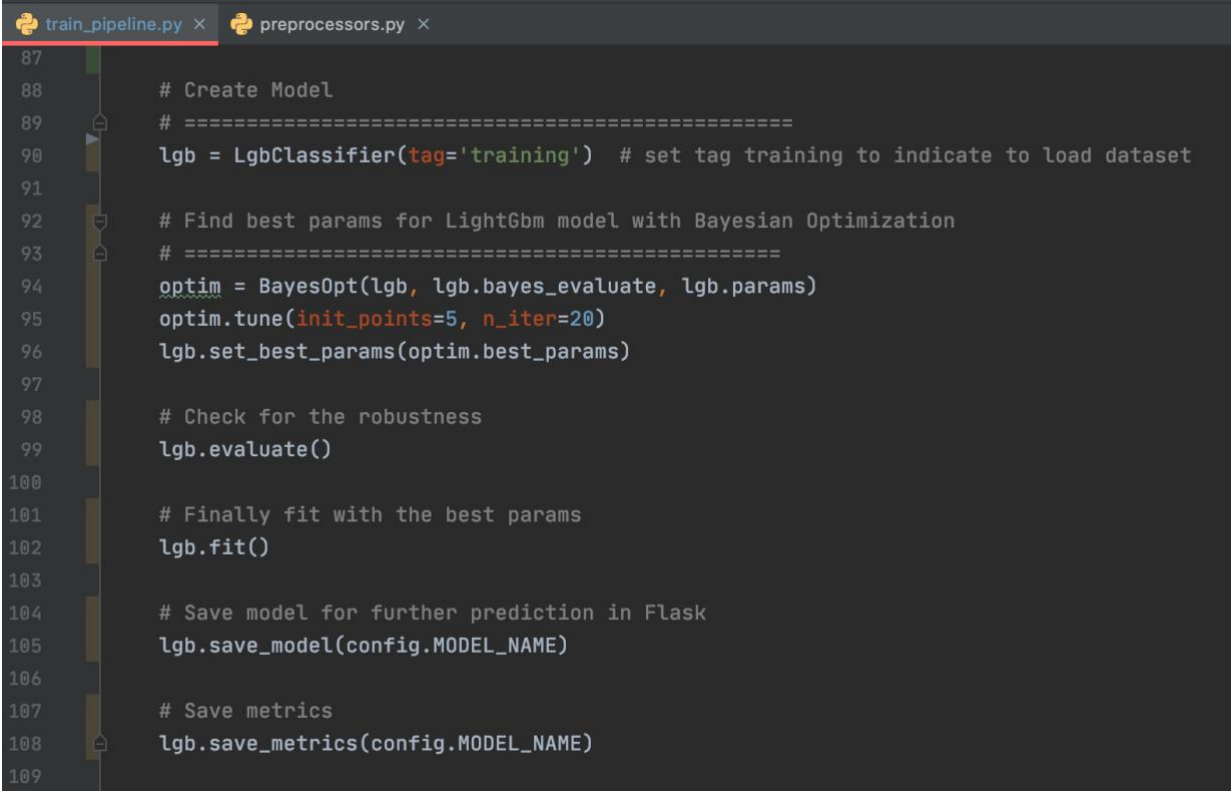
Après avoir effectué le fit_transform, le pipeline peut réutiliser les transformers en mémoire pour appliquer à nouveau une transformation soit sur le jeu de test ou sur des données reçues en production (Ligne 60 figure 14).

2.2.3.2.2 Sauvegarde du Pipeline en format pickle

Puis le pipeline est sauvegardé grâce à la librairie Joblib (ligne 66 figure 14).

2.2.4 Création du modèle de Machine Learning pour la prédiction

2.2.4.1 Pipe de création, tuning d'hyperparamètre et sauvegarde du modèle



```
87
88 # Create Model
89 # =====
90 lgb = LgbClassifier(tag='training') # set tag training to indicate to load dataset
91
92 # Find best params for LightGbm model with Bayesian Optimization
93 # =====
94 optim = BayesOpt(lgb, lgb.bayes_evaluate, lgb.params)
95 optim.tune(init_points=5, n_iter=20)
96 lgb.set_best_params(optim.best_params)
97
98 # Check for the robustness
99 lgb.evaluate()
100
101 # Finally fit with the best params
102 lgb.fit()
103
104 # Save model for further prediction in Flask
105 lgb.save_model(config.MODEL_NAME)
106
107 # Save metrics
108 lgb.save_metrics(config.MODEL_NAME)
109
```

Figure 18 Pipe de création, tuning et sauvegarde de modèle

Nous instancions deux objets :

- Un objet LgbClassifier
- Un objet BayesOpt qui permet de trouver les meilleurs paramètres du classifieur Lightgbm

Puis nous sauvegardons le modèle entraîné pour le réutiliser lors de la prédiction à l'aide de l'API Flask. Nous sauvegardons également les métriques d'accuracy et de courbe de roc auc.

2.3 2ème sous projet : API Flask

2.3.1 Description du script application.py

Le script application.py se charge de créer l'application Flask. Cette application permet de recevoir des requêtes et d'envoyer en retour un résultat.

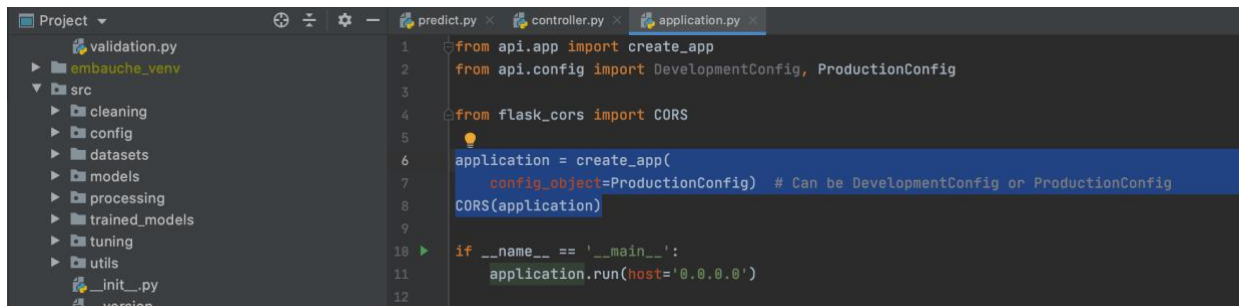


Figure 19 application.py à la racine du projet

2.3.2 Decorator et route dans controller.py

L'intelligence du code se trouve dans le script controller.py où les différentes routes sont spécifiées.

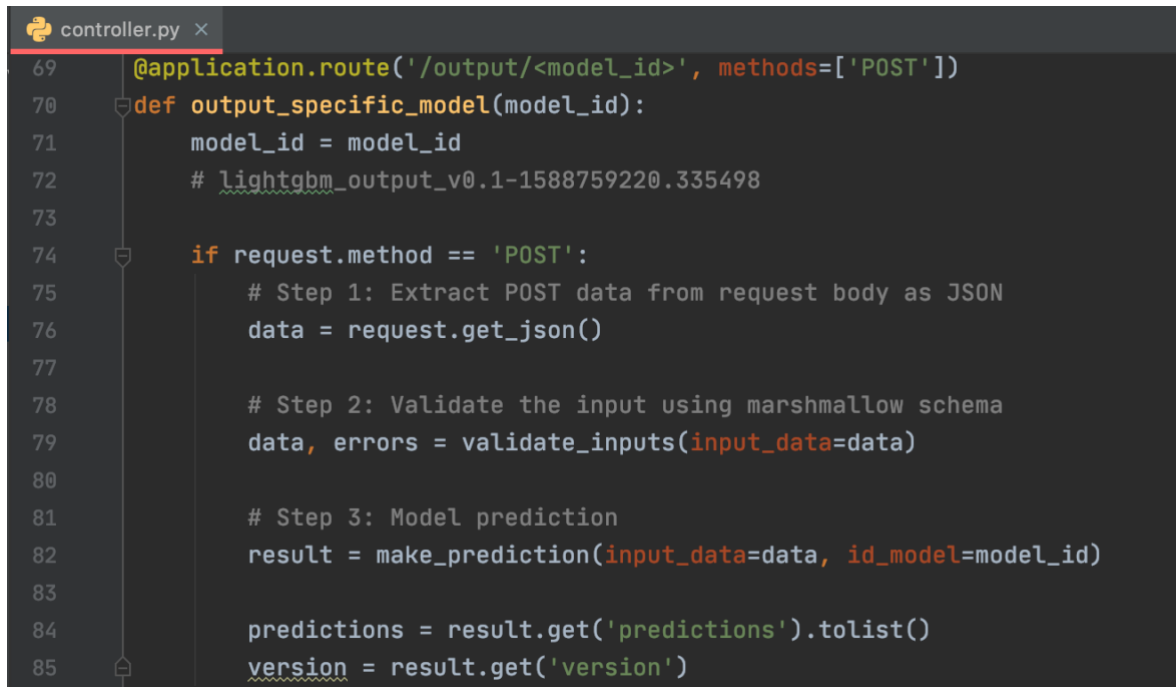


Figure 20 decorator et route endpoint

La ligne 69 dans la figure 18 est un décorateur. Un décorateur englobe une fonction. En d'autres termes, lorsque l'utilisateur tente d'accéder à https://nomdedomaine/output/model_id on lui renvoie ici le résultat de la fonction output_specific_model.

2.3.3 Fonction predict

A la ligne 82 de la figure 18, nous appelons la fonction make_prediction qui se trouve dans le script predict.py.


```

1 import pandas as pd
2
3 from src.models.xgboost_classifier import XgbClassifier
4 from src.processing.pipeline_management import load_pipeline
5 from src.config import config
6 from src import __version__ as _version
7
8 import logging
9 import typing as t
10
11 _logger = logging.getLogger(__name__)
12
13 # Load Pipeline for preprocessing as a global variables
14 pipeline_file_name = f"{config.PIPELINE_SAVE_PREPROCESSING_FILE}_{_version}.pkl"
15 _pipe = load_pipeline(file_name=pipeline_file_name)
16 _model = XgbClassifier()
17 _model.load_model(filename=config.MODEL_NAME)
18

```

Figure 21 import des modèles de pipeline et de prédiction

Ce script predict.py charge en mémoire, au lancement de l'application, les modèles de Pipeline de transformations et de prédiction Lightgbm basé sur le timestamp le plus récent.

Si nous spécifions un model_id comme sur la figure 18, ligne 82, le modèle correspondant est utilisé et non celui par défaut.

```

20 def make_prediction(*, input_data: t.Union[pd.DataFrame, dict]) -> dict:
21     """Make a prediction using a saved model pipeline.
22
23     Args:
24         input_data: Array of model prediction inputs.
25
26     Returns:
27         Predictions for each input row, as well as the model version.
28     """
29     if isinstance(input_data, dict):
30         data = pd.DataFrame(input_data, index=[0])
31     else:
32         data = pd.DataFrame(input_data)
33
34     transformed_data = _pipe.transform(data[config.FEATURES])
35     output = _model.predict(transformed_data)
36     results = {"predictions": output, "version": _version}
37
38     _logger.info(
39         f"Making predictions with model version: {_version} "
40         f"Inputs: {data} "
41         f"Predictions: {results}"
42     )
43

```

La fonction `make_prediction` appelée se charge de réutiliser ces modèles pour transformer la donnée reçue et faire une prédiction. Ainsi le Flask renvoie un résultat binaire à l'utilisateur qui a fait une demande de prédiction via une requête POST.

2.3.4 Résultats du call API

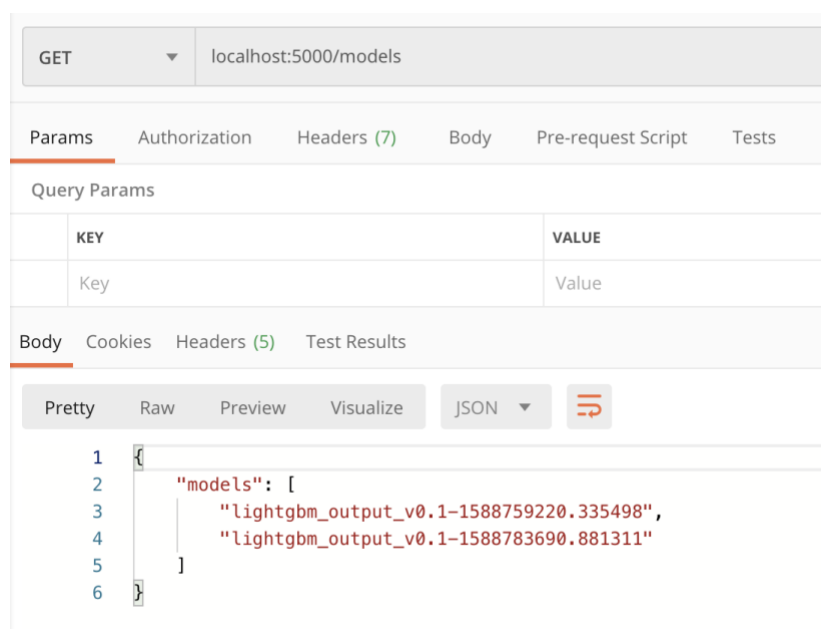


Figure 22 Résultats possibles sur le site <https://ngjohn.site>

Conclusion

Travailler sur ce projet a été l'occasion d'illustrer toutes les compétences apprises durant ma dernière expérience en tant que CTO de la startup Umamy où j'ai appris à utiliser les services AWS pour le déploiement des algorithmes de Machine Learning avec Docker et Travis-CI.

J'ai veillé à bien respecter au maximum les bonnes pratiques de la PEP8, ajouter des docstrings dans les fonctions, programmer de manière modulaire et en objet pour éviter la répétition de code (concept du DRY, don't repeat yourself) dans une limite de 2H30.

Cependant par manque de temps, le code et le flux de déploiement sont perfectibles.

En voici des pistes d'améliorations :

Pour avoir un code propre et professionnel, il aurait fallu faire des tests unitaires sur toutes les fonctions pour la maintenabilité. En effet lorsque les projets grossissent, les

fonctions peuvent être amenées à être modifiées par d'autres développeurs et faire des tests unitaires nous assurent que les changements n'altèrent pas les comportements souhaités.

Ils sont également les socles permettant de l'intégration et déploiement en continue (CI/CD) puisqu'avant de déployer en production, les plateformes comme Travis-ci exécutent les tests unitaires. Ici, un test sans lien avec le code a été mis en place pour montrer le flux le plus complet possible.

Lors de la phase de training de modèle, des seuils pour mesurer la performance et la présence d'overfitting devraient être fixés dans les tests de robustesses des algorithmes de Machine Learning avant de pouvoir être sauvegardé pour la production.