

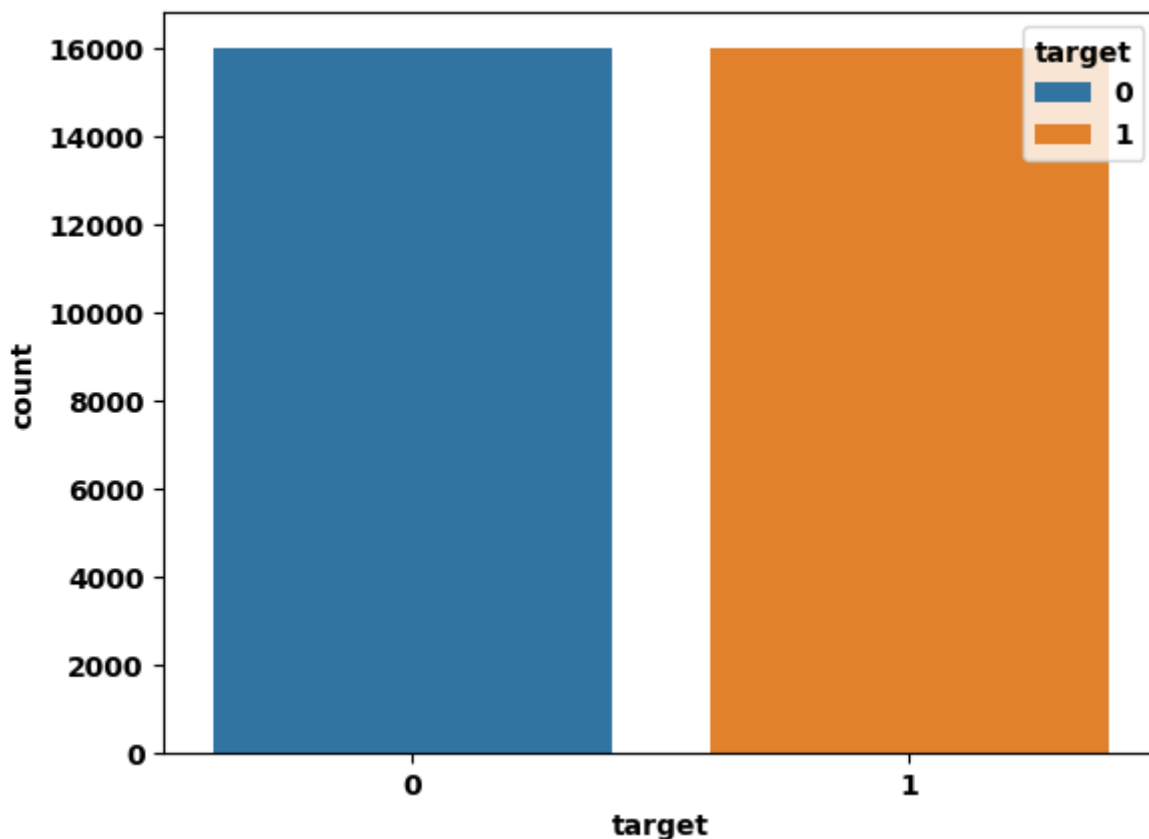
Analyse de sentiments par Deep learning - Modélisation et Déploiement de modèles - une démarche orientée MLOPs

Présentation du dataset

Les modèles ont été entraînés sur le dataset [Sentiment140 dataset with 1.6 million tweets](#) disponible en open source sur *Kaggle*.

	target	ids	date	flag	user	text
1498500	4	2070583051	Sun Jun 07 17:57:16 PDT 2009	NO_QUERY	bgardner	@Corpsman_Com You wouldn't have to pay for the...

Ce dataset contient des informations relatives à des tweets postés entre 2009 et 2020. A chaque tweet est attribué un sentiment (positif ou négatif) défini dans la colonne [target](#). Les deux classes sont équilibrées.



Distribution du sentiment (0 : négatif, 1 : positif)

Ici, seules deux colonnes sont utilisées : la [target](#) et le [text](#).

Modélisation

Plusieurs types de modélisations sont utilisées pour prédire la target à partir du text :

- Modèle Simple linéaire : **Regression Logistique**
- Modèle de réseau de neurones : **LSTM**
- Modèles à base de transformers : **BERT**

Regression Logistique

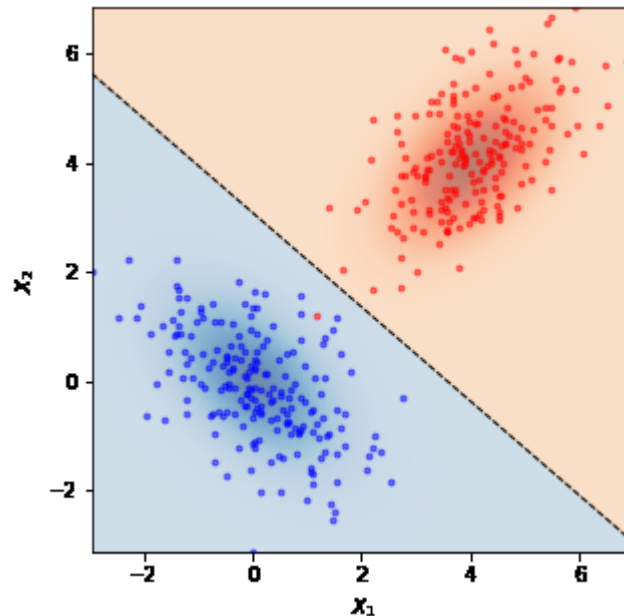


Illustration de régression logistique

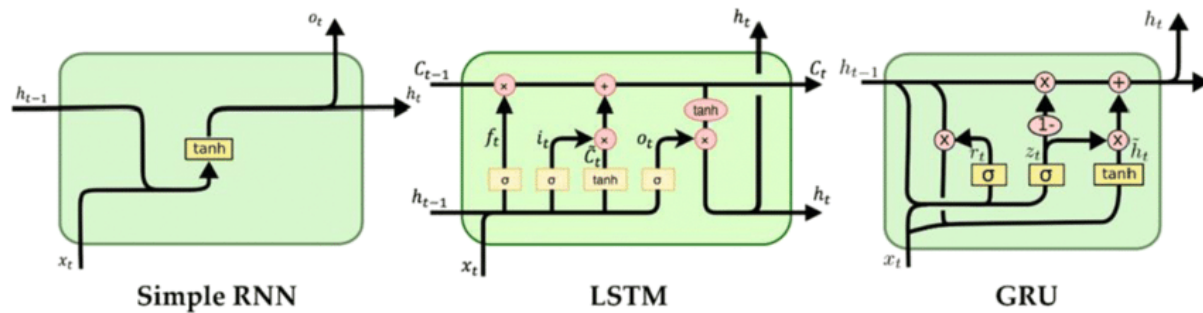
Pour cette approche la partie la plus importante réside dans le feature engineering qui est appliqué pour convertir les chaînes de caractères en variables interprétables par un modèle numérique. Cela passe par plusieurs étapes :

- Nettoyage et normalisation des **text**
- Stemming/Lemmatization
- Vectorisation

Puis une regression logistique est entraînée sur le dataset. Les résultats obtenus avec ce modèle permettent d'atteindre un niveau d'accuracy relativement élevée : **78%** sur le jeu de validation.

Ce modèle a l'avantage d'être très léger et de permettre un déploiement rapide et efficace, cependant il n'est pas capable de comprendre un cas complexe, de détecter le sarcasme ou de comprendre la double négation. Ceci est dû au principe du bag-of-words qui ne prend pas le texte comme une séquence ordonnée de tokens, mais comme un ensemble de mots sans relations entre eux.

Modèle de réseau de neurones (RNN, GRU, LSTM)



Architecture des différents types de cellules récursives pour les RNNs

Les modèles de réseaux de neurones de types récurrents (RNN) permettent de compenser le principal défaut des méthodes de bagging. Ils considèrent les textes comme une séquence ordonnée de jetons.

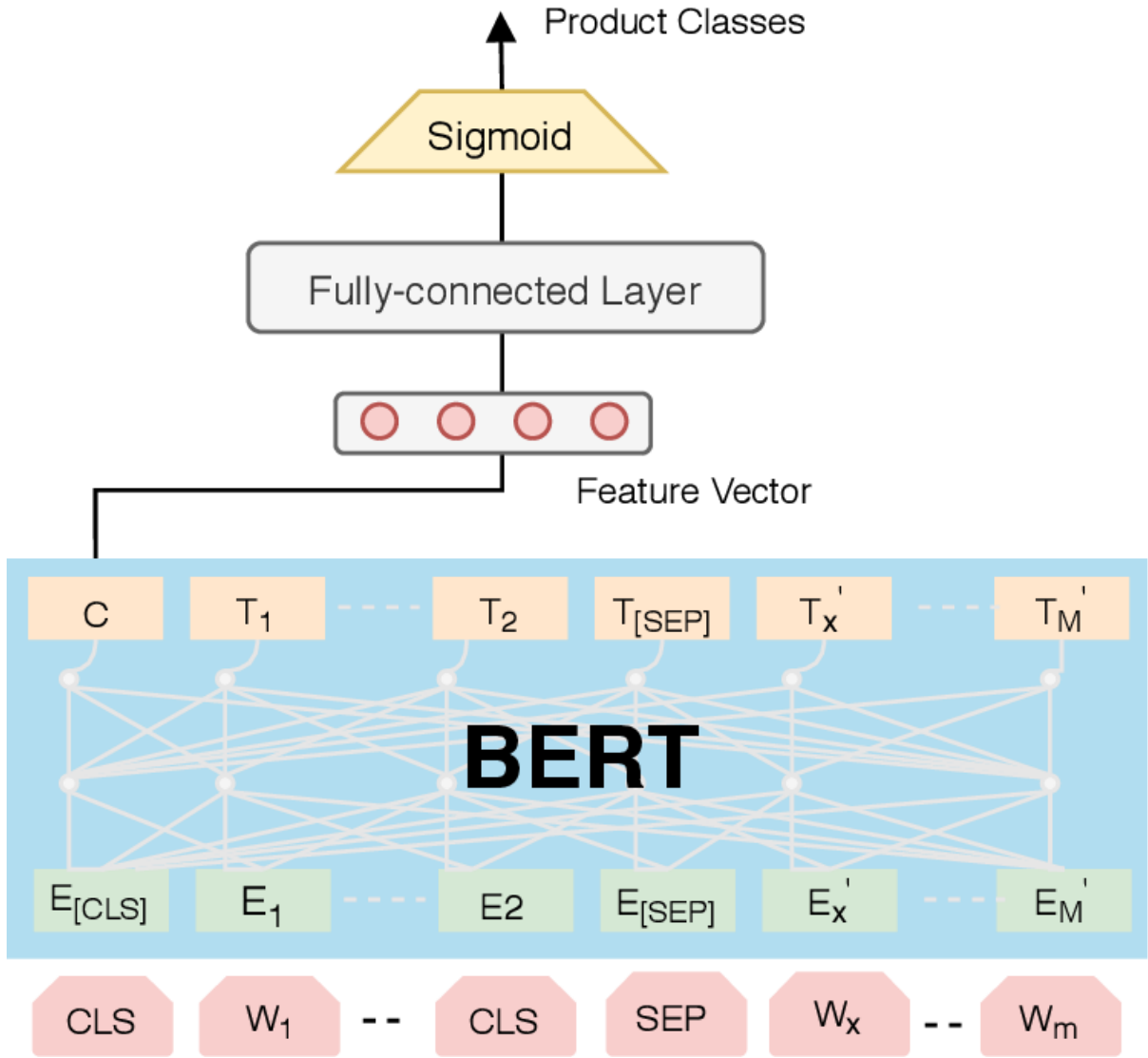
Pour ces types de modèles, la qualité des prédictions va principalement dépendre de plusieurs étapes :

- Nettoyage et normalisation des chaînes de caractères
- Stemming/Lemmatization
- Embedding
- Type de cellule RNN : RNN Simple, GRU, LSTM
- Longueur des séquences (troncage et/ou bourrage) des séquences

Ces modèles ont l'avantage de traiter les différents tokens comme une séquence ordonnée (le traitement du token x_n , prend en compte l'output h_{n-1} issue du traitement du token x_{n-1}).

Ces modèles peuvent cependant être sujets au phénomène de gradient évanescent et ne sont donc pas pertinents pour de très longues séquences même si les cellules **LSTM** ont été créées afin de remédier à ce problème.

Modèle de transformers (BERT)



Principe d'un modèle BERT pour la classification

Le principal avantage des modèles de transformers, basés sur des mécanismes d'attention, sur les modèles à base de LSTM est de fournir pour chaque token un contexte global contrairement aux modèles RNN classiques qui ne prennent en compte dans l'évaluation des tokens que les tokens ayant déjà été traités.

Les modèles de transformers sont téléchargés directement depuis Huggingface et sont associés à un tokenizer pré-entraîné. Ils sont cependant trop gros pour être réentraînés ou fine-tunés localement. Seule la tête de classification ajoutée pour l'application et le dataset à notre disposition est entraînée.

Bilan sur les différents modèles

Les résultats des différents modèles sont compilés dans le tableau suivant :

Modèle	Accuracy	F1-score -	F1-score +	Précision -	Précision +	ROC-AUC	Recall -	Recall +
Régression logistique	0.78	0.77	0.78	0.78	0.77	0.85	0.77	0.79
LSTM	0.79	0.80	0.79	0.78	0.81	0.88	0.82	0.77
BERT	0.82	0.82	0.82	0.825	0.82	0.90	0.82	0.83

Tracking des expériences avec MLFlow et Optuna

Tracking des performances des modèles

Lors de l'entraînement des différents types de modèles MLFlow permet de suivre les performances des modèles lors de l'entraînement pour sélectionner le meilleur modèle.

Tout d'abord il est nécessaire de lancer MLFlow dans l'invite de commande pour lancer le serveur qui servira à enregistrer notre experiment :

```
mlflow server --host localhost --port 8080
```

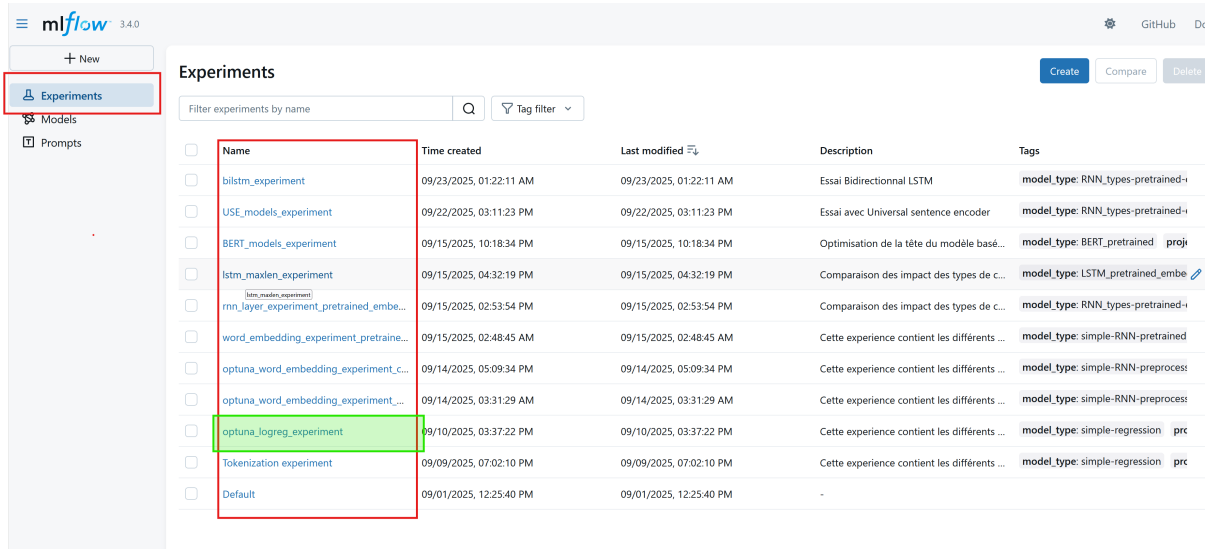
Au début du notebook le client MLFlow est initialisé :

```
from mlflow import MlflowClient
client = MlflowClient(tracking_uri="http://localhost:8080")
mlruns_path = Path("./mlruns").resolve()
mlflow_uri = mlruns_path.as_uri()
mlflow.set_tracking_uri(mlflow_uri)
# Création de notre experiment MLFlow
mlflow.set_experiment("Experiment 1") # <- Nouvelle étude avec MLFlow
```

Puis lors de l'entraînement du modèle choisi:

```
with mlflow.start_run(): # <- Nouvelle run mlflow a
    mlflow.log_input(dataset) # <- Logging du dataset utilisé pour
l'entraînement
    mlflow.log_params(params) # <- Logging des paramètres
    ... # <- La suite du code
    mlflow.log_metrics(output) # <- Logging des outputs
    mlflow.log_artifacts(artifact) # <- Logging d'autre fichiers : images,
textes..
    mlflow.sklearn.log_model(model, "model") # <- Logging du modèle créé en
précisant le framework (sklearn, tensorflow, pytorch ...)
```

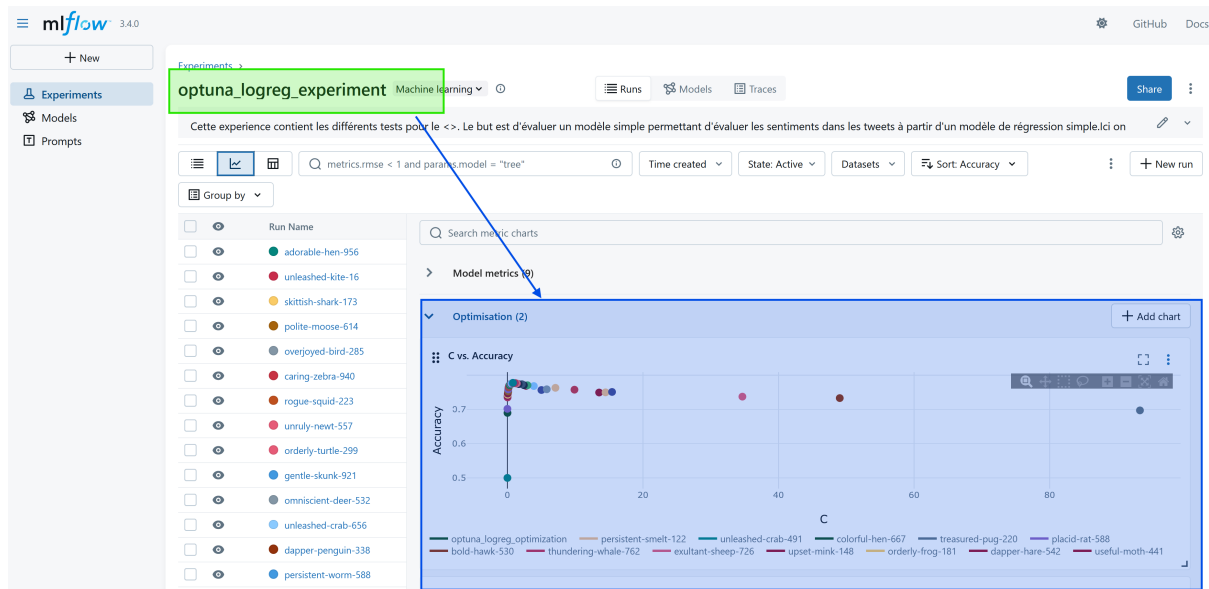
Le suivi de l'avancement des experiment et des performances des modèles se fait en se connectant à l'adresse de tracking dans un navigateur internet :



Name	Time created	Last modified	Description	Tags
bilstm_experiment	09/23/2025, 01:22:11 AM	09/23/2025, 01:22:11 AM	Essai Bidirectionnel LSTM	model_type: RNN_types-pretrained-i
USE_models_experiment	09/22/2025, 03:11:23 PM	09/22/2025, 03:11:23 PM	Essai avec Universal sentence encoder	model_type: RNN_types-pretrained-i
BERT_models_experiment	09/15/2025, 10:18:34 PM	09/15/2025, 10:18:34 PM	Optimisation de la tête du modèle basé...	model_type: BERT_pretrained proj
lstm_maxlen_experiment	09/15/2025, 04:32:19 PM	09/15/2025, 04:32:19 PM	Comparaison des impact des types de c...	model_type: LSTM_pretrained_embe
rnn_layer_experiment_pretrained_embe...	09/15/2025, 02:53:54 PM	09/15/2025, 02:53:54 PM	Comparaison des impact des types de c...	model_type: RNN_types-pretrained-i
word_embedding_experiment_pretraine...	09/15/2025, 02:48:45 AM	09/15/2025, 02:48:45 AM	Cette experience contient les différents ...	model_type: simple-RNN-pretrained
optuna_word_embedding_experiment_c...	09/14/2025, 05:09:34 PM	09/14/2025, 05:09:34 PM	Cette experience contient les différents ...	model_type: simple-RNN-preprocess
optuna_word_embedding_experiment_...	09/14/2025, 03:31:29 AM	09/14/2025, 03:31:29 AM	Cette experience contient les différents ...	model_type: simple-RNN-preprocess
optuna_logreg_experiment	09/10/2025, 03:37:22 PM	09/10/2025, 03:37:22 PM	Cette experience contient les différents ...	model_type: simple-regression prc
Tokenization experiment	09/09/2025, 07:02:10 PM	09/09/2025, 07:02:10 PM	Cette experience contient les différents ...	model_type: simple-regression prc
Default	09/01/2025, 12:25:40 PM	09/01/2025, 12:25:40 PM	-	

Interface de MLFlow pour le tracking des experiments

Les performances des modèles peuvent être comparées graphiquement pour chaque experiment :



Comparaison des performances des modèles en fonction des hyperparamètres choisis

Optimisation des hyperparamètres avec Optuna

La librairie Optuna permet d'optimiser les hyperparamètres des modèles. Les paramètres sont exprimés sous forme de plage de variations (pour les nombres entiers ou flottants) ou sous forme de liste (pour les variables catégorielles ou les chaînes de caractères)

La paramétrisation de l'optimisation est faite de manière suivante :

```
import optuna

# Optuna optimise l'output d'une fonction (run_function)
def run_function(trial)
    param1 = trial.suggest_float("param1", min_float, max_float) # <- Paramètre de
type float
    param2 = trial.suggest_int("param2", min_int, max_int) # <- Paramètre de type
int
```

```
param3 = trial.suggest_categorical("param3", ["cat1", "cat2"]) # <- Paramètre
catégoriel sous forme de liste

# Code de la fonction
...
#
return output # <- Valeur numérique à optimiser
```

Puis l'étude d'optimisation est créée et lancée :

```
# Initialisation
study = optuna.create_study(direction="maximize")
# Lancement
study.optimize(run_function, n_trials=50)
```

Conclusion

Une fois les différents expériences réalisés, le modèle le plus performant est sélectionné et sauvegardé. Il est maintenant possible de le mettre en production.

Mise en production

Le modèle sélectionné est mis en production par déploiement d'une API sur le cloud de MS Azure. Une interface graphique est mise à disposition des utilisateurs finaux afin d'interroger le modèle et d'obtenir les résultats pour chaque tweet proposé.

Création d'une API

Une API est créée via la librairie FastAPI et testée en local. L'API créée a deux points d'entrée :

- Pour la prédiction du modèle (`/predict`)
- Pour le feedback utilisateur (`/feedback`)

Air Paradis Sentiment API 0.1.0 OAS 3.1

/openapi.json

default		^
POST	/predict Predict	▼
POST	/feedback Feedback	▼
Schemas		^
FeedbackIn >	Expand all	object
HTTPValidationError >	Expand all	object
PredOut >	Expand all	object
TweetIn >	Expand all	object
ValidationError >	Expand all	object

Prise de vue de la page de doc de l'API

Création d'une interface utilisateur avec Streamlit

L'interface utilisateur permet de faire appel au modèle et de fournir un feedback sur la prediction fournie :

Air Paradis - Sentiment Analysis

Entrez votre tweet ici:

I love this company the experience was great !

Zone d'entrée du texte

Analyser

Probabilité positive : 0.93

Sortie fournie par le modèle

Sentiment : 😊 Positif

Est-ce la bonne prédiction ?

Renvoi du feedback à l'API

- ☒ Oui
☐ Non

Envoyer mon feedback

Validation de l'envoi du feedback à l'API

✓ Feedback envoyé à l'API

Tweet suivant

Passage au tweet
suivant

Prise de vue de l'interface graphique

Déploiement de l'API sur le cloud Microsoft Azure

Une fois les l'API et l'interface graphique crée, il est nécessaire de mettre à disposition des utilisateurs cette API pour les utilisateurs finaux. Pour ce faire l'API doit être déployée sur le cloud (ici MS Azure).

Une Web App azure est crée sur le portail Azure et liée au repo GitHub de l'application. Afin de lancer le déploiement un workflow est créé en deux phases : build et déploiement.

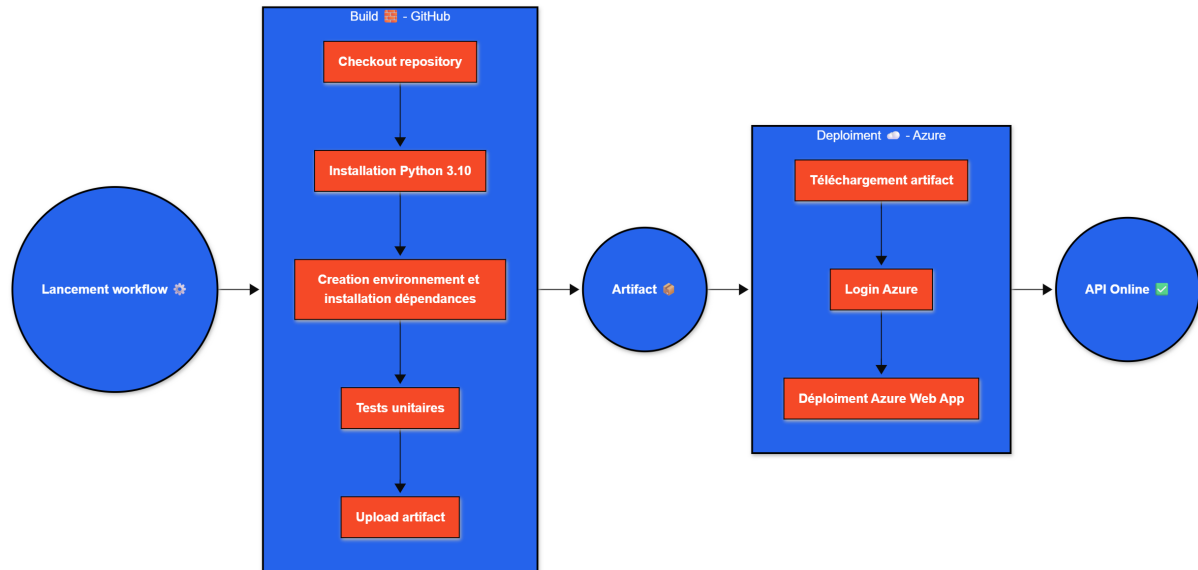


Diagramme du workflow GitHub Actions

Le Build

La phase de **build** permet de construire notre application à partir :

- Du code source et des modèles présents sur le repo GitHub
- De la liste des bibliothèques tierces présentes dans l'environnement python (*requirements.txt*)

Les **tests unitaires** ayant pour objectif de tester indépendamment chaque fonction de l'application sont exécutés durant cette phase.

Tests unitaires

Les tests unitaires permettent de tester chaque fonction du code source :

- Type et dimensions des inputs et outputs de chaque fonction
- Comparaison d'une ou plusieurs réponses

```
python -m unittest discover -s Tests -p "*.py"
```

Le Déploiement

Cette phase consiste à reproduire l'application qui a été build, à partir de l'artefact produit lors de la phase de build :

- Recréation de l'environnement python
- Lancement de l'application au démarrage

Resources locales et sur le Cloud

Les ressources des machines virtuelles fournies par MS Azure sont très différentes de celles disponibles sur une machine de développement :

- Système d'exploitation

- Mémoire
- Stockage

Il faut donc, optimiser les ressources utilisées avant le déploiement :

- Minimisation des packages python à installer
 - Packages graphiques inutilisés
 - Packages lourds inutilisés
- Minimisation des fichiers/dossiers lors du build :
 - Images uploadées sur le repo
 - Notebooks et scripts d'analyse et modélisation
 - Runs MLFlow
- Réduction de la taille du modèle (conversion Tensorflow Lite)

Une fois ces opérations réalisées il est possible d'utiliser l'application créée, mais il est maintenant nécessaire de le maintenir et d'en évaluer les performances, c'est à cet effet que les outils de monitoring proposés par Azure entrent en jeu.

Monitoring de l'application

Le monitoring de l'API est réalié via Azure Application Insights, il permet de déclencher des alertes lors de l'utilisation de l'API. Un exemple d'alerte est créé lorsque trois feedbacks négatifs sont renvoyés par l'utilisateur en moins de 5 minutes.

Lors de la création du monitoring sur Azure, une URI pour le monitoring est crée ainsi qu'une clef d'instrumentation. Ces éléments permettent de lier directement l'API au services Application Insight.

Logging

Tous les évènements à monitorer sont tracés par l'intermédiaire d'un objet `logger` qui communique directement avec le service Application Insight.

```
import logging
from azure.monitor.opentelemetry import configure_azure_monitor

# Configuration du monitoring azure pour l'application
configure_azure_monitor(
    connection_string=
        """InstrumentationKey=AZURE_MONITORING_INSTRUMENTATION_KEY>;
        IngestionEndpoint=AZURE_MONITORING_ENDPOINT_URI>""",
    logger_name="sentiment_api_logger",
)
logger = logging.getLogger("sentiment_api_logger")
logger.setLevel(logging.INFO)
logger.info("Application Insights logging initialized.")
```

Création d'alertes

Les éléments loggés peuvent être consultés sur le portail Azure permettent de définir des règles d'alertes. Cela se fait en trois étapes :

- Requête sur les logs
- Création de condition d'alerte basée sur les résultats de requêtes
- Automatisation d'action au déclenchement de l'alerte (envoi e-mail, SMS etc...)

Edit alert rule ...

Scope

Condition

Actions

Details

Tags

Review + save

Configure when the alert rule should trigger by selecting a signal and defining its logic.

Signal name *

Custom log search

See all signals

Query type

☒ Aggregated logs

Get notified on aggregated data from your logs.

☐ Single event (preview)

Get notified when a specific message appears in your logs.

Define the logic for triggering an alert. Use the chart to view trends in the data. [Learn more](#)

The query to run on this resource's logs. The results returned by this query are used to populate the alert definition below.

Search query *

```
traces
| where message contains "Mauvaise prédiction"
| order by timestamp desc
| take 3
| extend n_2_timestamp = prev(timestamp, 2)
| extend time_diff_minutes = datetime_diff('minute', n_2_timestamp, timestamp )
| where time_diff_minutes <= 5
```

[View result and edit query in Logs](#)

Measurement

Select how to summarize the results. We try to detect summarized data from the query results automatically.

Measure

Table rows

Aggregation type

Count

Aggregation granularity

5 minutes

Génération d'alerte - Condition

Edit alert rule ...

Scope

Condition

Actions

Details

Tags

Review + save

An action group is a set of actions that can be applied to an alert rule. [Learn more](#)

+ Select action groups

+ Create action group

Action group name	Contains actions
Warning on predictions	1 Email

Email subject

Customize the email subject that will be sent when the alert fires. You can add dynamic values from the alert in the email subject. [Learn more](#)

Air-Paradis : Bad predictions alert

Génération d'alerte - Action

Conclusion

Bilan

Un modèle d'analyse de sentiment basé sur des tweets a été créé en comparant plusieurs architectures de modèles :

- Une baseline par regression logistique simple
- Un modèle de réseau de neurone récurrent basé sur les cellules LSTM
- Un modèle moderne de transformers basé sur BERT

Une fois le modèle le plus performant déployé en mettant en place les principes MLOps :

- Analyse de données
- Modélisation (Scikit-learn, TensorFlow)
- Experimentation et tracking des modèles (MLFlow Tracking)
- Optimisation des hyperparamètres (Optuna)
- Contrôle du code source (Git/GitHub)
- Serving API (FastAPI)
- Service de prédiction (Streamlit)
- Déploiement cloud (Azure Web App)
- Monitoring et déclenchement d'alertes (Azure Application Insights)

Pour aller plus loin

Certains principes du MLOps, comme l'automatisation du pipeline d'entraînement, n'ont pas été implémentés dans ce projet, pour des raisons de ressources disponibles . Certaines pratiques peuvent cependant être mises en place dans le cadre de la maintenance du modèle.

- Logging des inputs utilisateurs afin d'étendre la base de données d'entraînement
- Monitoring du taux de mauvaises prédictions :
 - Alerte
 - Réentraînement du modèle