

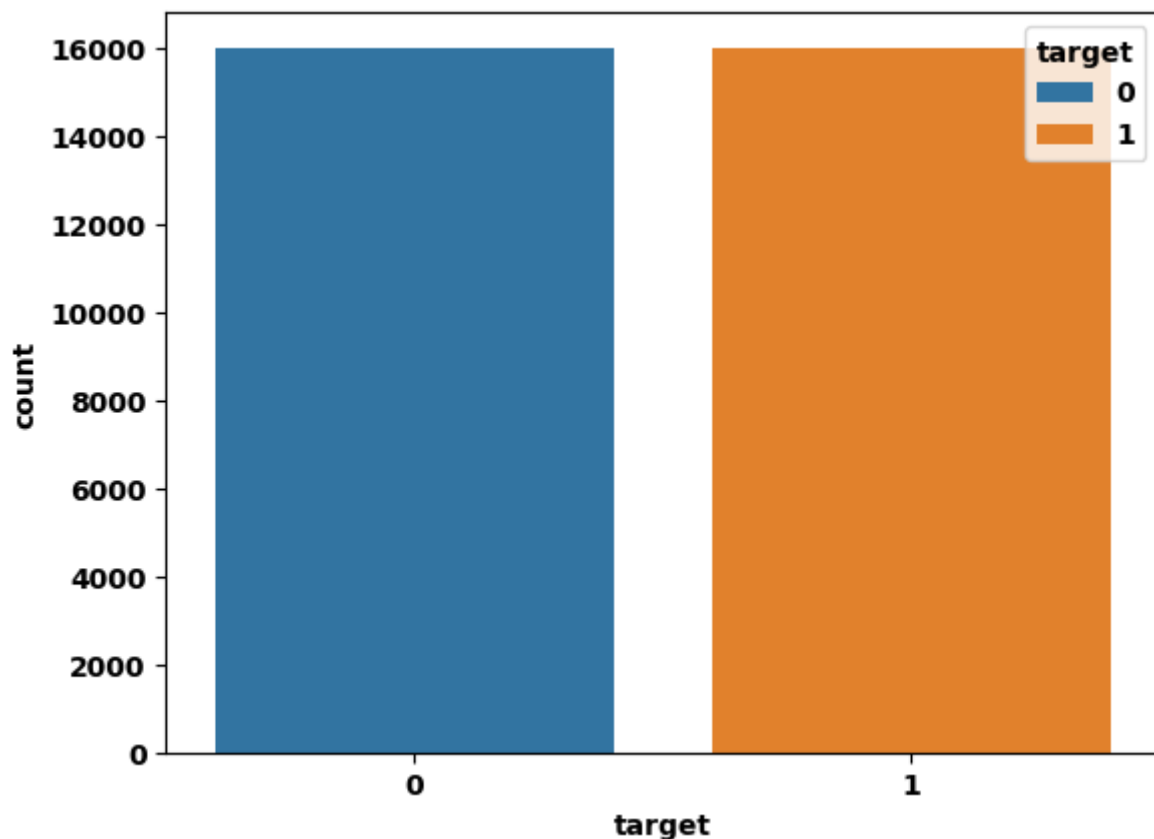
Analyse de sentiments par Deep learning - Modélisation et Déploiement de modèles - une démarche orientée MLOPs

Présentation du dataset

Les modèles ont été entraînés sur le dataset **Sentiment140** disponible sur *Kaggle* contenant **1,6 million de tweets** annotés pour l'analyse de sentiments. Chaque ligne correspond à un **tweet** et contient plusieurs informations :

	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...

Les classes sont **équilibrées**, garantissant un apprentissage stable lors de l'entraînement des modèles.



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

Dans cette étude, seules deux colonnes sont utilisées :

- **text** , qui contient le message à analyser,
- **target**, qui sert de variable cible pour l'apprentissage supervisé.

Ces données constituent la base d'entraînement des différents modèles - linéaires, réseaux de neurones et transformers - pour la prédiction de sentiment.

Modélisation

L'objectif de cette étapes est d'évaluer plusieurs approches pour la modélisation de sentiments à partir de tweets :

- **Regression logistique** : modèle de référence (baseline)
 - **Réseaux de neurones récurrents (RNN, GRU, LSTM)** : prise en compte du contexte séquentiel
 - **Transformers (BERT)** : représentation contextuelle globale
-

Regression logistique

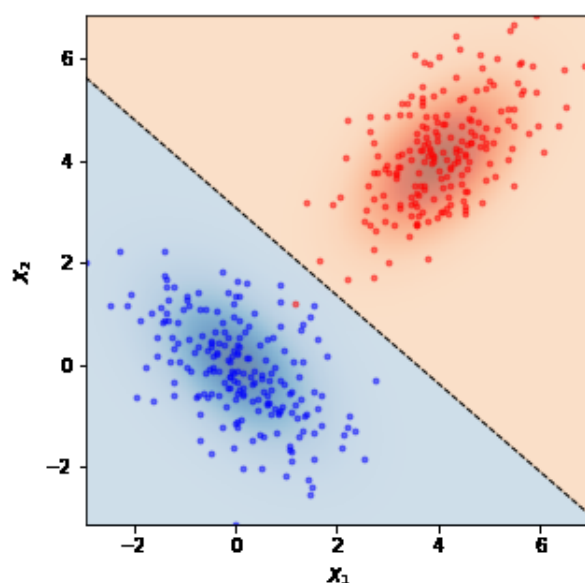


Illustration de régression logistique

La regression logistique sert ici de modèle de référence. Sa performance dépend fortement du **feature engineering** consistant à transformer les chaînes de caractères en représentations numériques exploitables.

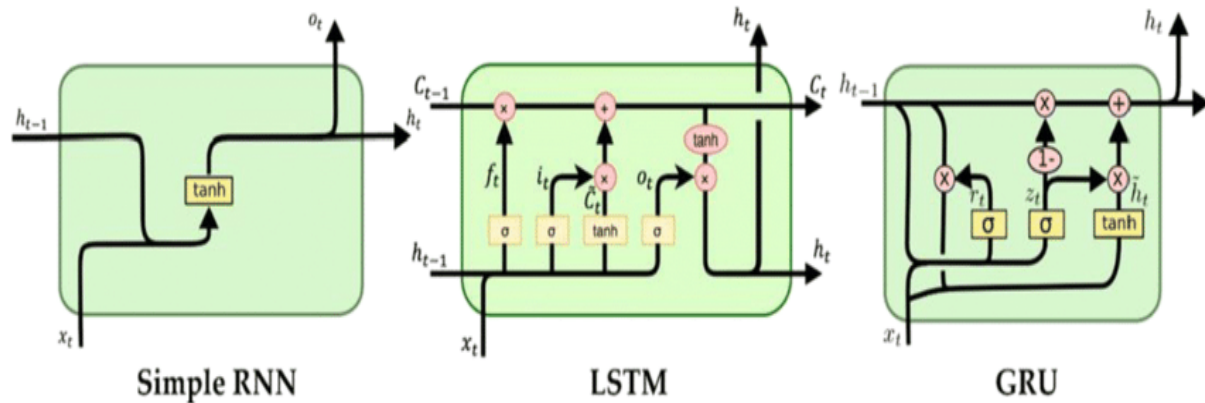
Les principales étapes sont :

- Nettoyage et normalisation du texte
- Stemming / Lemmatization
- Vectorisation (TF-IDF ou CountVectorizer)

Une fois les textes vectorisés, le modèle est entraîné sur le jeu de données équilibré. Les performances obtenues atteignent environ **78% d'accuracy** sur le jeu de validation.

Ce modèle, léger et rapide à déployer, constitue une excellente baseline. Il ne capture cependant pas les relations entre les mots. Le texte est considéré comme un ensemble de tokens ("bag-of-words"). Les structures complexes de phrases ne sont pas correctement identifiées.

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



Architecture des différents types de cellules recursives pour les RNNs

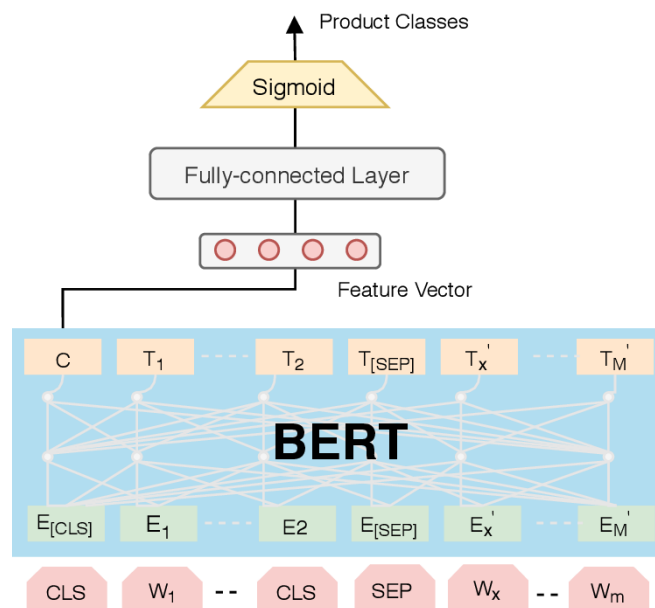
Les modèles de type RNN abordent une limite majeure des approches de type "bag-of-words" : ils considèrent les textes comme des **séquences ordonnées de tokens**. Chaque unité de traitement dépend du mot précédent, ce qui permet de modéliser le contexte.

La qualité de ces modèles dépend de plusieurs paramètres :

- Prétraitement du texte (nettoyage, normalisation, stemming ou lemmatisation)
- Méthode d'**embedding** (Word2Vec, GloVe, FastText)
- Type de cellule (RNN Simple, GRU ou LSTM)
- Longueur des séquences (troncage et/ou bourrage)

Les cellules **LSTM** (Long Short-Term Memory) atténuent le problème du *gradient évanescent* des RNNs classiques, en conservant la mémoire à long terme. Ces architectures offrent de meilleures performances (**79% d'accuracy** sur le jeu de validation) mais leur entraînement est plus long et plus coûteux.

Modèles de transformers (BERT)



Principe d'un modèle BERT pour la classification

Les modèles de type **Transformers**, tels que **BERT**, reposent sur les mécanismes d'**attention** qui permettent d'intégrer un contexte global à chaque token. Contrairement aux RNNs, ils ne traitent pas les séquences mot par mot mais analysent les dépendances entre tous les tokens simultanément.

Le modèle utilisé ici provient de la plateforme *Hugging Face* et est associé à son tokenizer **pré-entraîné**. Seule la **tête de classification** a été fine-tunée sur le dataset, les poids du modèle de base sont conservés pour des raisons de disponibilité de ressources.

Cette approche permet d'obtenir les meilleures performances, au prix d'un coût de calcul et de stockage beaucoup plus élevé.

Bilan sur les différents modèles

Les performances globales des meilleurs modèles pour chaque approche sont compilées 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

Les résultats confirment la progression attendue :

- Le modèle linéaire offre une baseline solide et interprétable
- Le LSTM apporte un léger gain en capturant la dimension séquentielle des textes.
- BERT obtient les meilleurs scores grâce à une compréhension contextuelle approfondie.

Ces performances ont ensuite été exploitées dans une démarche **MLOps** complète visant à suivre, optimiser et déployer les modèles en production.

Tracking des expériences avec MLFlow et Optuna

Dans une approche **MLOps**, le suivi et l'optimisation des expériences sont essentielles pour garantir la reproductibilité, la traçabilité et la sélection du meilleur modèle. Deux outils sont utilisés :

- **MLFlow** pour le **tracking** des entraînements et des performances,
- **Optuna** pour l'**optimisation des hyperparamètres**.

Suivi des performances des modèles avec MLFlow

Lors de l'entraînement des différents types de modèles **MLFlow** permet d'enregistrer et comparer leurs performances à travers plusieurs exécutions (*runs*). Chaque run conserve les paramètres, métriques, artefacts et modèles entraînés permettant ainsi le suivi complet de chaque expérimentation.

Initialisation du serveur MLFlow

Tout d'abord, il est nécessaire de lancer le serveur MLFlow localement pour enregistrer les expériences :

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

Le client MLFlow est également initialisé dans le notebook python :

```
from mlflow import MlflowClient
import mlflow
from pathlib import Path

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")
```

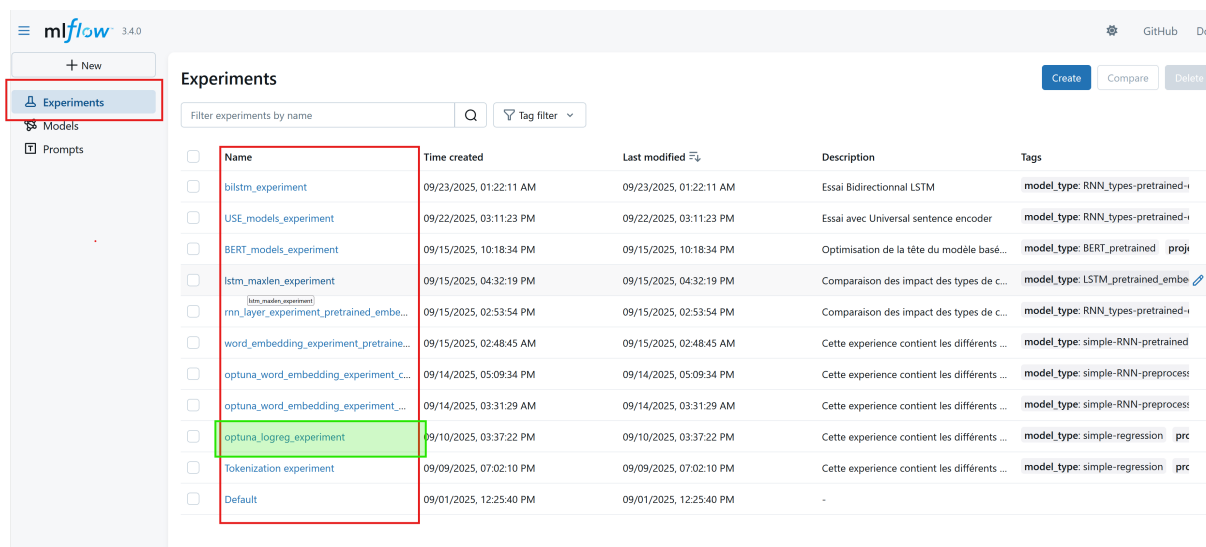
Enregistrement des runs

Lors de l'entrainement d'un modèle chaque run est encapsulée dans un contexte :

```
with mlflow.start_run():
    mlflow.log_input(dataset)           # <- Dataset d'entrainement
    mlflow.log_params(params)           # <- Paramètres
    mlflow.log_metrics(output)          # <- Métriques
    mlflow.log_artifacts(artifact)      # <- Fichiers additionnels
    mlflow.sklearn.log_model(model, "model") # <- Sauvegarde du modèle créé
```

Visualisation et comparaison

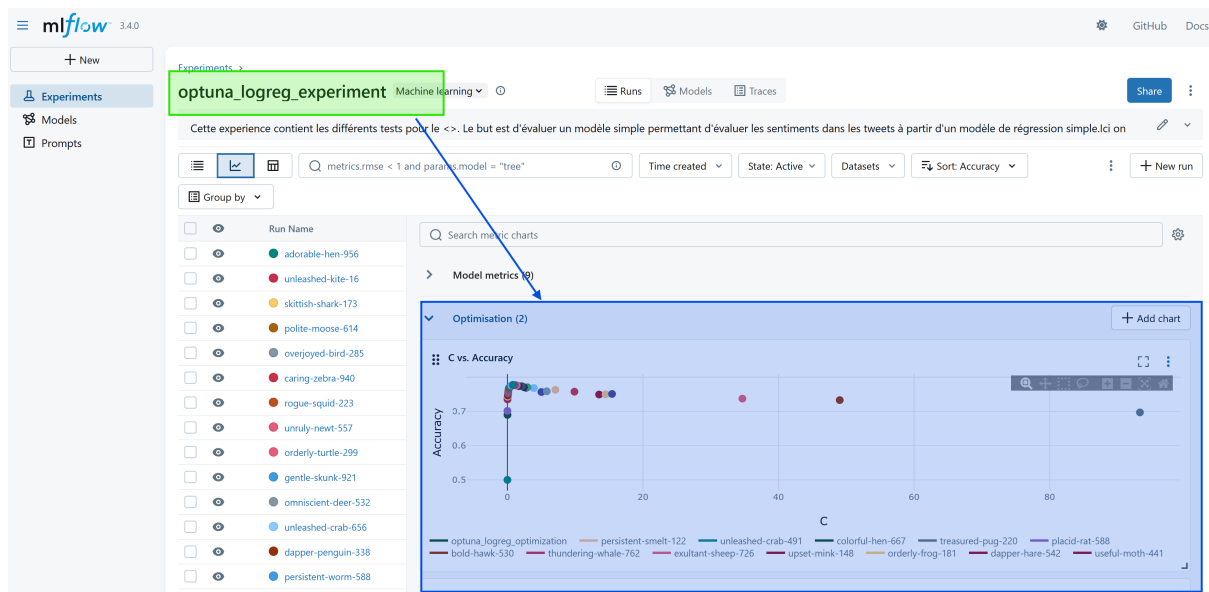
Le suivi des experiences s'effectue via l'interface MLFlow accessible à l'adresse de tracking :



	Name	Time created	Last modified	Description	Tags
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	lstm_maxlen_experiment	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	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
<input type="checkbox"/>	Default	09/01/2025, 12:25:40 PM	09/01/2025, 12:25:40 PM	-	

Interface de MLFlow pour le tracking des expériences

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



Comparaison des performances des modèles

Optimisation des hyperparamètres avec Optuna

Une fois le tracking opérationnel, la librairie **Optuna** permet d'optimiser automatiquement les hyperparamètres des modèles.

Définition de la fonction objectif

L'optimisation est basée sur une fonction objectif à maximiser (ou minimiser):

```
import optuna

def run_function(trial)
    param1 = trial.suggest_float("param1", min_float, max_float)
    param2 = trial.suggest_int("param2", min_int, max_int)
    param3 = trial.suggest_categorical("param3", ["cat1", "cat2"])
    # Entraînement et évaluation du modèle
    ...
    return output # <- Score à optimiser
```

Lancement de l'étude

L'étude Optuna est ensuite créée et exécutée :

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

Optuna cherche ainsi automatiquement la combinaison d'hyperparamètres maximisant (ou minimisant) la fonction objectif.

Conclusion

Une fois les différentes expériences réalisées, le **modèle le plus performant** est sélectionné, sauvegardé et prêt à mettre en production. La prochaine étape consiste à le **déployer** et rendre accessible le service de prédiction via une **API** et une **interface utilisateur**.

Mise en production

Le modèle le plus performant étant sélectionné, il est mis en production pour fournir un service de prédiction accessible aux utilisateurs finaux. Cette étape inclut la création d'une API, d'une interface utilisateur, le déploiement sur le cloud et l'optimisation des ressources.

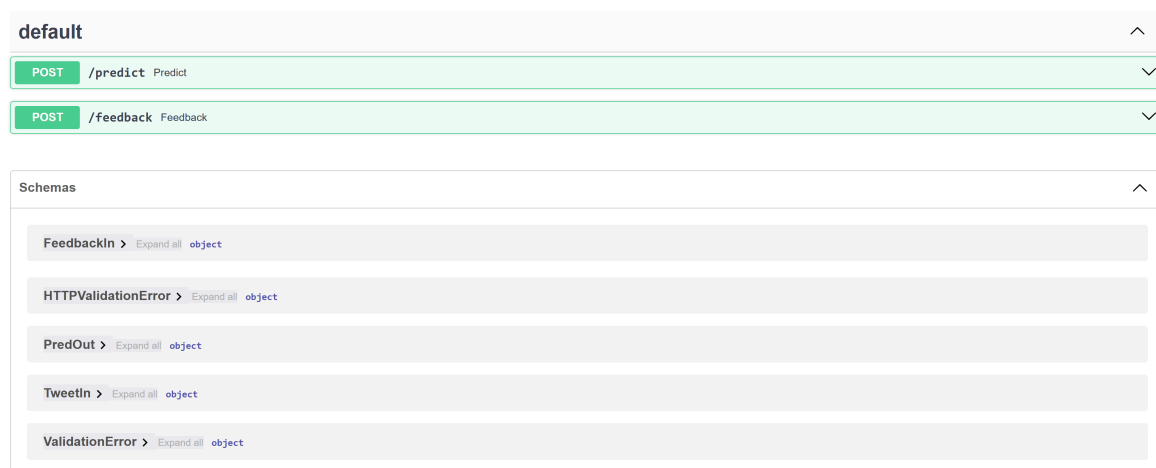
Création d'une API FastAPI

Le modèle est exposé via une **API REST** créée avec **FastAPI**. L'API comporte deux points d'entrée :

- `/predict` : pour la prédiction de sentiment d'un tweet
- `/feedback` : pour recevoir un retour utilisateur

Air Paradis Sentiment API 0.1.0 QAS 3.1

/openapi.json



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

L'API est d'abord testée en local avant déploiement sur le cloud.

Interface utilisateur avec Streamlit

Pour faciliter l'interaction avec le modèle une interface graphique est créée avec **Streamlit**.

Les utilisateurs peuvent :

- Saisir un tweet

- Obtenir la prédiction ddu modèle
- Fournir un feedback sur la qualité de la prédiction

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

Interface utilisateur

Déploiement de l'API sur le cloud Microsoft Azure

L'API est ensuite déployée sur une **Web App Azure**, connectée au dépôt GitHub du projet contenant le code et le modèle. Un **workflow GitHub Actions** automatise le déploiement en deux phases : **build** et **déploiement**.

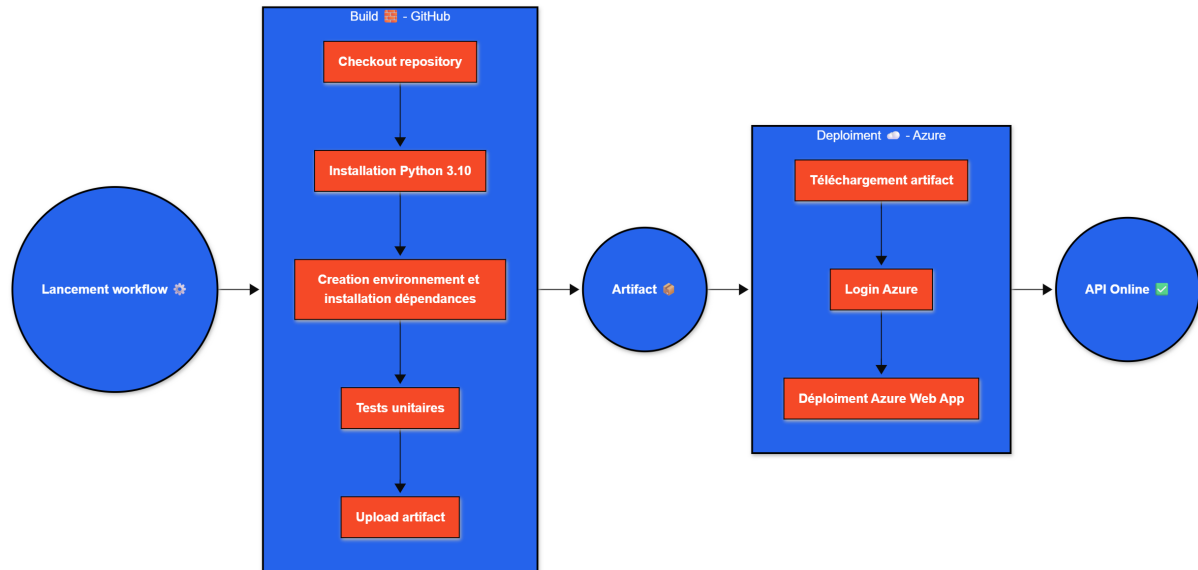


Diagramme du workflow GitHub Actions

Le Build

La phase de **build** construit l'application à partir :

- Du code source et du modèle
- Des dépendances listées dans le fichier *requirements.txt*

Tests unitaires

Les tests unitaires sont effectués durant la phase de build et permettent de tester chaque fonction du code source :

- Type et dimensions des inputs et outputs des fonctions
- Comparaison d'une ou plusieurs réponses

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

Le Déploiement

Cette phase consiste à :

- Reproduire l'environnement python à partir de l'artefacts build
- Lancer automatiquement l'application au démarrage

Optimisation des ressources

Les ressources cloud sont très limitées par rapport aux machines de développement. Il est donc essentiels d'optimiser :

- Les packages Python : retrait des packages lourds ou inutilisés
- Les fichiers et dossiers du build : images, notebooks et runs MLFlow
- La taille du modèle (conversion Tensorflow Lite)

Une fois ces étapes réalisées l'application est prête à être utilisée par les utilisateurs finaux. La section suivante aborde **le monitoring et la gestion des alertes**, indispensable pour assurer la performance continue et la fiabilité de l'API.

Monitoring de l'application

Une fois le modèle déployé, il faut assurer sa performance continue et de détecter rapidement les dysfonctionnements. Le monitoring permet de suivre l'utilisation de l'API, de collecter des événements importants et de déclencher des alertes. Cela est assuré via **Azure Application Insights**.

Logging

Le logging centralisé permet de collecter les événements et de les analyser en temps réel. Chaque retour utilisateur peut être tracé.

Configuration Python

```
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.")
```

Le **logger** envoie automatiquement les événements vers Application Insights, permettant de centraliser le suivi.

Création d'alertes

Les alertes permettent de réagir automatiquement aux événements critiques, comme un taux élevé de feedbacks négatifs en peu de temps.

Cela se fait en trois étapes :

- Requête sur les logs
- Création de condition d'alerte
- Action automatique : envoi d'e-mail ou SMS

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

Le projet a permis de construire un pipeline complet : préparation de données, comparaisons d'approches (regression logistique, LSTL, BERT), tracking (MLFlow), optimisation (Optuna), déploiement (FastAPI, Streamlit, Azure Web App) et monitoring (Application Insights). BERT a fourni les meilleures performances, tandis que la baseline reste pertinente pour un déploiement léger.

Perspectives

Parmi les évolutions utiles : automatiser le pipeline d'entraînement (AirFlow / Azure ML Pipeline), intégrer le feedback utilisateur pour un apprentissage continu, surveiller la dérive et déclencher des réentraînements automatiques, et renforcer les tests automatisés