# Chapitre 01 : les données

## Les API’s :

État et prévision des concentrations de PM10 au niveau national | InerisPlus de détails dans le ReadMe ou l’annexe fournis avec le rendu

**Prev'Air** : c’est une plateforme open source qui fournit des données sur la qualité de l'air, dans notre cas on va scrapper des données brutes plus précisément des cartes géographiques qui représentes la qualité de l’air au niveau de paris. On cible principalement 4 gaz qui sont les plus nocifs et qui affecte d’une façon quasi immédiate l’être vivant.

**DataGouv  :** On recolte les données qui sont disponibles sur Data.gouv.fr. c’est une plateforme qui met à disposition des données publiques en open data. Les données sont disponibles sous forme de datasets, qui sont des ensembles de données structurées. Les datasets sont accompagnés de documentations qui décrivent les données et leur utilisation.  
Elle nous a permis d'accéder aux données de qualité de l'air collectées par le réseau de surveillance de l'air en Île-de-France. L'API est disponible gratuitement

**AirParif :** Elle gère le réseau de surveillance de la qualité de l'air en Île-de-France. L'API permet aux développeurs d'accéder aux données de qualité de l'air collectées par le réseau de surveillance.

A l’inverse de prev’air, il faut demander une clé d’api qui nous permet de récolter ces données. suite à un contact par téléphone et une demande par mail elle nous a été donnée

## Emplacement

Lors de cette partie on a étudié les données et on a pu déduire ce qu’on avait besoin et surtout comment structurer notre récolte

#### Une image contenant texte, capture d’écran, affichage, logiciel Description générée automatiquementUne image contenant texte, capture d’écran, affichage, logiciel Description générée automatiquementPartie Image :

Data : on a stocker le GeoJson qui nous permet de faire du géoréférencement et du découpage Les outputs de nos traitement et près traitement comme les images.tiff

MaxJ : les cartes des Max/j sont divisé sur 4 sous dossier, chaque dossier représente un gaz

##### MoyJ :les cartes des Moy/j sont divisé sur 4 sous dossier, chaque dossier représente un

##### Gaz

##### Prédiction : les outputs de nos traitement d’IA qu’on va voir à la fin du rapport

#### Partie CSV :

##### Gazs : Recuperation depuis l’api en format csv

##### Gazs\_output : Suite à un pré-traitement, nettoyage des csv’s, mais aussi La concaténations des CSV en output

##### Une image contenant texte, capture d’écran, Police, nombre Description générée automatiquementPollens : Les données csv récupérer depuis l’api

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquementUne image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

## Code extraction et traitement: 1. ImageDownloader.py

1.1 Initialisation : Nous avons commencé par initialiser une instance de notre classe `ImageDownloader` avec une URL d'image et un dossier d'image. Nous avons également défini des chemins pour les dossiers `maxj` et `moyj` dans le dossier d'image, et nous avons établi une liste de polluants.

1.2. Création de dossiers : Ensuite, nous avons créé une méthode `create\_polluant\_folders` qui génère des dossiers pour chaque polluant dans les dossiers `maxj` et `moyj`.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement1.3. Téléchargement d'images : Nous avons également mis en place une méthode `download\_images` qui fait ce qui suit :

* + Elle envoie une requête GET à l'URL de l'image et analyse le contenu de la réponse avec BeautifulSoup.
  + Elle extrait tous les liens de fichiers .jpg de la page.
  + Pour chaque lien de fichier, elle ignore ceux contenant le mot 'prevision' pour garder uniquement les résultats non prédit
  + Elle construit l'URL de téléchargement en ajoutant le lien du fichier à une URL de base.
  + Elle détermine le dossier de destination en fonction du nom du fichier (soit `maxj`, soit `moyj`).
  + Elle vérifie si le fichier correspond à l'un des polluants et le stocke dans le bon dossier.
  + Si le fichier existe déjà, elle ne le télécharge pas à nouveau.
  + Sinon, elle télécharge le fichier et l'enregistre dans le dossier de destination.

### 2. ImageProcessor.py :

### 2.1. Initialisation : C'est comme la carte d'identité de notre classe. Lorsqu'on crée une instance de cette classe, on doit fournir ces informations. Les trois premiers arguments sont des chemins de fichiers où se trouvent nos images et où nous voulons stocker les résultats. Les `sub\_dirs` sont les sous-dossiers dans notre répertoire parent qui contiennent les images, et `coordinates` sont les coordonnées pour le recadrage des images.

2.2. process\_subdirectories :On peut dire que c'est le cœur de notre classe. Cette méthode parcourt tous les sous-dossiers et tous les fichiers JPEG dans ces dossiers. Elle recadre chaque image en fonction des coordonnées fournies et enregistre la nouvelle image avec "\_output.jpg" ajouté à son nom.

2.3. load\_images\_with\_output: Cette méthode est comme un assistant qui prépare nos données pour l'entraînement. Elle charge toutes les images traitées, les redimensionne à 256x256, et normalise les valeurs de pixels pour qu'elles soient entre 0 et 1 (en les divisant par 255).

2.4. train\_and\_save : Comme son nom l'indique, cette méthode entraîne notre modèle et sauvegarde les résultats. Elle utilise toutes les images traitées pour entraîner un réseau de neurones convolutif. Après l'entraînement, elle utilise le modèle pour prédire de nouvelles images à partir de l'ensemble de test, sauvegarde une image prédite et utilise `display\_results` pour montrer les résultats.

2.5. display\_results**:** Cette méthode est notre présentateur de résultats. Elle prend les images de test et les images prédites, et les affiche côte à côte pour qu'on puisse voir comment notre modèle a performé.

### 3. imageGeo : La classe `ImageAnalysis` dans le script qu’on a, la classe effectue l'analyse d'une image et sa manipulation géospatiale:

3.1. Initialisation : Cette méthode initialise les chemins vers l'image, le fichier GeoJSON, le chemin de sortie pour l'image traitée et le chemin du fichier TIFF.

3.2. Préparation et affichage de l'image:La méthode commence par ouvrir une image à partir d'un chemin donné. Elle la découpe selon certaines coordonnées et affiche l'image découpée.

3.3. Manipulation du GeoJSON: Ensuite, elle charge un fichier GeoJSON qui contient des formes géographiques. Ces formes sont ensuite affichées sur un graphique pour une meilleure visualisation.

3.4. Superposition d'informations géographiques sur l'image: La méthode superpose ensuite les formes géographiques du fichier GeoJSON sur l'image découpée. Cela donne une vue de l'image avec les entités géographiques superposées.

3.5 Géoréférencement de l'image et création d'un masque: Enfin, la méthode crée une nouvelle image géoréférencée à partir de l'image superposée. Elle définit une bbox (boîte englobante) en coordonnées géographiques, et crée un masque basé sur ces coordonnées. Ce masque est ensuite appliqué à l'image, et le résultat est sauvegardé dans un nouveau fichier TIFF.

En gros, `runImage` est une méthode qui vous permet de charger une image, de la découper, de charger et d'afficher un GeoJSON, de superposer ce GeoJSON sur l'image, et finalement de géoréférencer l'image avec un masque basé sur les coordonnées géographiques.

### 4.Pollens

4.1. Initialisation:Nous avons commencé par initialiser une instance de notre classe `PollensData` avec une URL de pollens et des en-têtes de requête. Nous avons également initialisé une instance de la classe `PollensDataManager` avec l'URL de pollens et un chemin de dossier.

4.2. Récupération des données sur les pollens: Nous avons mis en place une méthode `get\_pollens\_data` qui fait ce qui suit :

* + Elle envoie une requête GET à l'URL des pollens avec les en-têtes appropriés.
  + Elle transforme la réponse en JSON et extrait les données.
  + Elle itère sur chaque observation pour créer les colonnes du DataFrame.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement4.3. Enregistrement des données sur les pollens: Nous avons également mis en place une méthode `save\_pollens\_data\_to\_csv` qui fait ce qui suit :

* + Elle récupère les données sur les pollens en utilisant la méthode `get\_pollens\_data`.
  + Elle crée un nom de fichier avec la date d'aujourd'hui.
  + Elle crée un chemin de fichier en joignant le chemin du dossier et le nom du fichier.
  + Si le fichier existe déjà, elle affiche un message.Sinon, elle enregistre le DataFrame dans un fichier CSV et affiche un message de confirmation.

**5. GazsPolluant**

* 1. Initialisation : Nous avons commencé par initialiser une instance de notre classe `GazsData` avec un dossier de téléchargement et une URL.
  2. Une image contenant texte, capture d’écran, logiciel

     Description générée automatiquementTéléchargement des fichiers CSV: Nous avons mis en place une méthode `download\_csv\_files` qui fait ce qui suit :
  + Elle récupère la liste des fichiers déjà téléchargés dans le dossier de téléchargement.
  + Elle envoie une requête GET à l'URL spécifiée et parse le contenu de la réponse avec BeautifulSoup.
  + Elle parcourt les dossiers de chaque année sur la page web.
  + Si le dossier correspond à l'année cible, elle envoie une requête GET à l'URL du dossier et parse le contenu de la réponse avec BeautifulSoup.
  + Elle parcourt les fichiers CSV dans le dossier de l'année.
  + Pour chaque fichier CSV, elle construit l'URL du fichier et le chemin du fichier.
  + Si le fichier CSV n'a pas encore été téléchargé, elle le télécharge et l'enregistre dans le dossier de téléchargement.

### **Main :** on a pu initialiser des variable et lancer nos traitement d’une façon continue

### Une image contenant texte, capture d’écran Description générée automatiquement

### 

Une image contenant texte, capture d’écran

Description générée automatiquement

## 

#### 4.2.3 Injection dans la base de données

Pour injecter les données récupérer par web scraping/API Rest, on a utilisé ce bloc de code :

A screen shot of a computer program

Description automatically generated

On initialise la connexion avec le seau S3 en utilisant la bibliothèque boto3, et on récupère les données. A la fin on injecte.

On utilise le même process de l’injection des données de pollens dans un autre dossier dans le même S3 Seau.

#### 4.2.4 Création et préparation des messages

Pour la préparation des messages, on a 2 types de messages. Le message pour vérifier si l’utilisateur il est abonné ou pas.

A screenshot of a computer program

Description automatically generated

Et là on a deux possibilités, si l’ID de l’utilisateur existe dans la base de données RDS donc il peut l’utiliser, sinon on l’ajoute.

On utilise send\_message () pour envoyer un message.

#### 4.2.5 Vérification de Verify Token

Avec le Webhook, il y’a une Verify Token à mettre en place pour plus de sécurité, dans notre cas, on utilise ce code pour la mettre.

A screen shot of a computer code

Description automatically generated

#### 4.2.6 Envoie d’image

L’envoie de l’image est fait sur plusieurs étape, on commence par lire la dernière image insérer dans le Seau S3, on passe au traitement et découpage de l’image et en l’enregistre dans un autre dossier dans le seau S3. On a mis en place une fonction dans l’application de BOT qui lis l’image crée du Seau S3, l’enregistre dans le Dyno de Heroku et l’envoi à l’utilisateur, l’image va être supprimer automatiquement après un temps.

A screen shot of a computer program

Description automatically generated

#### 4.2.7 Traitement de message « update me »

Comme on peut vois dans le diagramme de cas d’utilisation pour recevoir les mises à jour, on passe par plusieurs cas. Pour la récupération des données de département demander par l’utilisateur, on utilise une fonction « query data for department » qui cherche le fichier csv de pollen de derniers jours dans le seau S3.

Après on passe à la fonction « query and check gazs data » qui extraire les données des gazs de demain, ces données sont le résultat de deux modèles de Machine Learning qui injecte leurs prédictions dans un tableau RDS.

A screen shot of a computer

Description automatically generated

Dés qu’on a les 4 derniers lignes qui sont les prédictions du jours après, on construit notre message pour l’envoyer à notre utilisateur.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

La fonction main\_predict() s’exécute automatiquement chaque jours a 23h:55 pour injecter la prédictions des gazs dans un tableau de la base de données RDS.

## Pour améliorer notre code, nous pourrions envisager les points suivants :

Gestion des erreurs :Nous pourrions ajouter une gestion des erreurs pour gérer les cas où une étape échoue, par exemple si le téléchargement d'une image échoue ou si le traitement d'une image échoue.

Logs:Nous pourrions ajouter plus de logs pour suivre l'avancement de chaque étape, par exemple en indiquant le nombre total d'images à télécharger et le nombre d'images déjà téléchargées.

fournir l’application.s

Chapitre 05

### 5.1 Introduction

Dans ce chapitre, nous présentons les différentes données que nous avons récoltées et les traitements effectués dessus afin d’ensuite faire des prédictions dessus. Nous expliquerons les différents algorithmes de machine learning et deep learning utilisés et les choix faits.

### 5.2 Images de cartes de France

Dans cette partie, nous nous intéressons aux données récoltées des images de carte de France et au traitement réalisé dessus. L’objectif a été d’ensuite faire des prédictions de polluants en île de France à partir de ces images.

#### 5.2.1 Traitement des images

Nous avons récolté deux images par jour par gaz : une correspondant à la moyenne journalière et une correspondant au pic journalier. Les gazs en question sont le NO2, le O3, le PM10 et le PM2,5.

Une image contenant texte, carte, atlas

Description générée automatiquementLes images récoltées ressemblent à ceci :

Une image contenant texte, carte

Description générée automatiquement

Le traitement des images est fait dans la classe imageProcessor.py comme vu précédemment

Dans Le constructeur de la classe ImageProcessor

#### 5.2.3 Modèle de prédiction CNN

La méthode **train\_and\_save** de la **classeImageProcessor** parcoure tous les dossiers d'images. Pour chaque dossier, les images et leur type de gaz associé sont chargés. L’objectif du modèle est de prédire l’image du jour suivant.

Pour cela on a trié par date consécutive et ca nous a permis de prédire

Les données sont réparties de telle manière que X (les entrées du modèle) contient toutes les images sauf la dernière et que y (les cibles du modèle) contient toutes les images sauf la première.

Les images sont réparties en ensembles d'entraînement et de test, avec 70% des images allant dans l'ensemble de test.

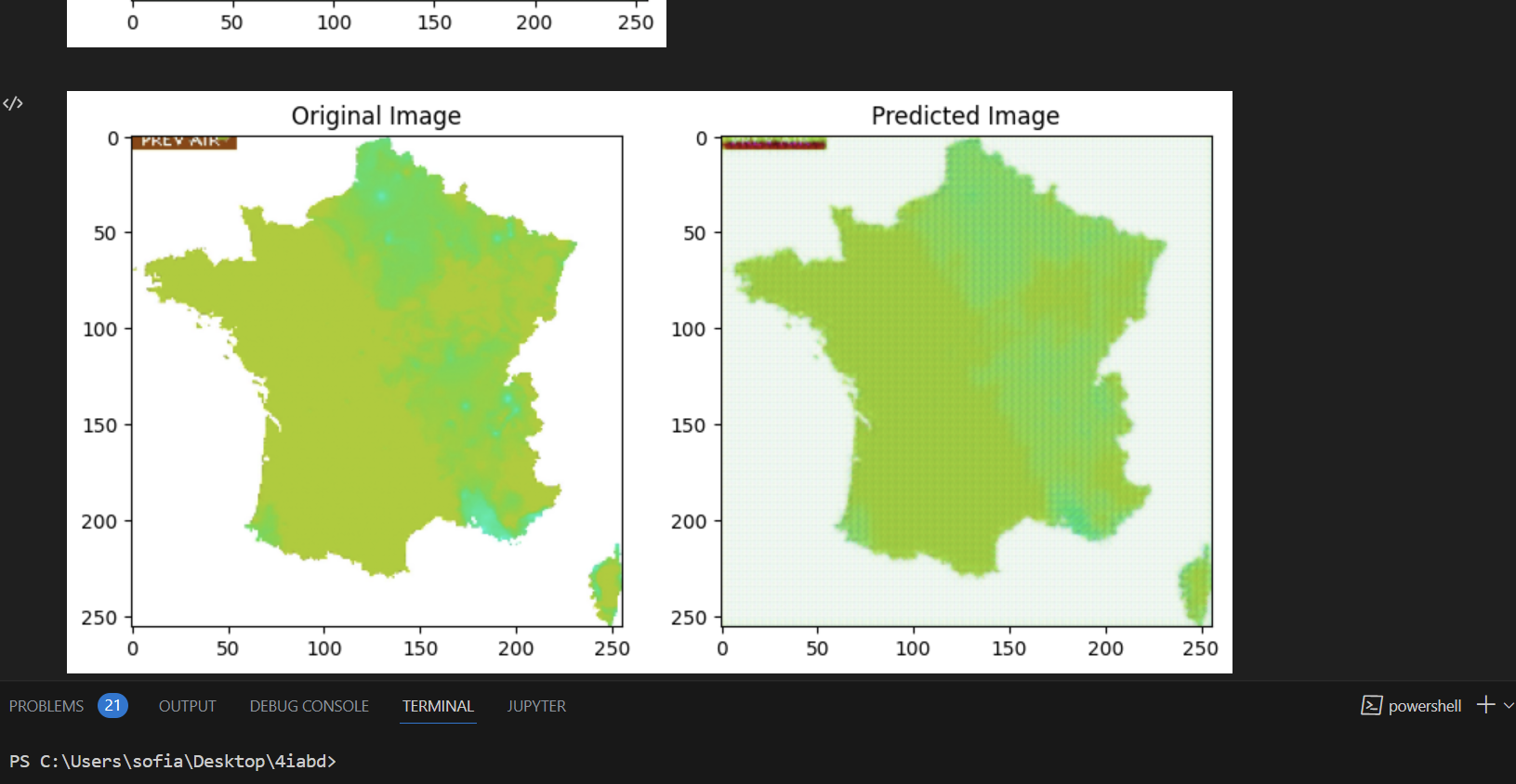
Est défini ensuite un générateur séquentiel qui prend une image d'entrée, la compresse via les deux couches **Conv2D** et la retransforme à la taille originale avec les deux couches de **Conv2DTranspose**. Afin d'accélérer l'apprentissage et réduire les problèmes potentiels lors de l'entraînement on utilise la fonction **BatchNormalization**().

Le générateur est ensuite compilé avec l'optimiseur **Adam** et **la fonction de perte de l'erreur quadratique moyenne**.

Le modèle est ensuite entraîné sur l'ensemble d'entraînement, en utilisant l'ensemble de test comme données de validation, pendant 350 époques.

La première image prédite est affichée. L'image prédite est sauvegardée dans le chemin défini précédemment.

Pour tester notre code, on a implémenté un display pour voir la prédiction sur la date n-1 qu’on avait déjà :



.

#### 5.2.3 Traitement des Images

Comme vu précédemment L’analyse des images est faite dans imageGeo.py ou on va plus s’approfondir

La méthode **runImage** appelle toutes les autres fonctions et méthodes nécessaires pour effectuer l'analyse de l'image.

On récupérer Les limites du GeoJSON sont obtenues avec la propriété **total\_bounds** du **GeoDataFrame**. Le GeoJSON

Une image contenant carte, diagramme, texte, Tracé

Description générée automatiquementL'image de la carte rognée est convertie en tableau numpy et affichée avec le GeoJSON tracé par-dessus la carte en rouge.

Une image contenant texte, carte, capture d’écran, diagramme

Description générée automatiquement

Une Bbox englobante est définie et convertie en un objet Polygon. L'image est alors ouverte de nouveau, convertie en niveaux de gris, et un profil (métadonnées) pour le nouveau fichier TIFF est créé.

Un masque booléen est créé sur la base de la bbox, qui est ensuite appliqué à l'image en niveaux de gris. Cette image masquée est enregistrée comme un fichier TIFF géoréférencé. Le nouveau fichier TIFF est ouvert et les informations de géoréférencement sont affichées.

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Pour tester on lance les coordonnées du pixel (500, 500) qui sont converties en coordonnées géographiques.



Les départements d'Île-de-France sont chargés à partir du fichier GeoJSON et affichés sur l'image à l'aide de GeoPandas et Rasterio.

Un masque est créé en utilisant la géométrie des départements de l'Île-de-France, ce qui permet d'isoler ces régions spécifiques dans l'image. Les départements sont ensuite superposés sur l'image. L’image masquée est créée en utilisant la géométrie des départements d'Île-de-France et enregistrée comme un nouveau fichier TIFF.

L'image finale est affichée avec les limites des départements d'Île-de-France en superposition.

Une image contenant texte, carte, atlas

Description générée automatiquement

### 5.3 Données structurées de gazs :

Dans cette partie, nous nous intéressons aux données structurées de gazs et au traitement réalisé dessus. L’objectif a été d’ensuite faire des prédictions des polluants en France à partir de ces données.

Les données sont structurées dans des fichiers csv. Il y a un fichier csv par jour et les données vont du 01/01/2021 à aujourd’hui.

Chaque fichier contient les colonnes :

Date de début, Date de fin, Organisme, code zas, Zas, code site, nom site, type d'implantation, Polluant, type d'influence, discriminant, Réglementaire, type d'évaluation, procédure de mesure, type de valeur, valeur, valeur brute, unité de mesure, taux de saisie, couverture temporelle, couverture de données, code qualité, validité

Les ZAS sont réparties selon 3 catégories :

* ZAG : 23 agglomérations
* ZAR : 35 zones à risque
* ZRE : 18 zones régionales

#### 5.3.1 Premier traitement des données

Travaillant en groupe en parallèle nous avons abouti à deux prétraitements différents des données et on va commencer par étudier le premier qui a été utilisé pour les prédictions Randomized Forest Regressor (ou Forêt Aléatoire Linéaire). Le but de ce traitement était de par la suite prédire la valeur moyenne de polluant pour le mois suivant.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Nous avons tout d’abord lu tous les fichiers csv récoltés puis commencé par supprimer les colonnes inutiles.

Les colonnes taux de saisie, couverture temporelle, couverture de données ont été supprimés car elles sont entièrement vides. La colonne Date de fin également car elle correspond simplement à Date de début avec une heure en plus.

Les colonnes Organisme, code zas, code site, nom site, discriminant, procédure de mesure, valeur, code qualité et validité ont également été supprimés car leur signification était floue dans la documentation d’où proviennent les données ou alors parce que ces colonnes ne semblaient pas essentielles.

Nous avons ensuite conservé les lignes contenant ‘Oui’ dans la colonne Réglementaire puis supprimé cette colonne également.

Seules les valeurs brutes supérieures ou égales à 0 ont été conservées.

Après toutes ces suppressions il a été nécessaire de regrouper toutes les colonnes restantes dans une dataframe et de calculer la valeur moyenne car pour ne pas avoir plusieurs valeurs brutes au même endroit au même moment.

Nous avons ensuite créé deux colonnes ‘**year’** et ‘**month’** à partir de la ‘Date de début’ puis supprimé la colonne ‘Date de début’ pour tenter d’avoir plus de saisonnalités dans les données en retirant les jours et d’ainsi prédire la valeur moyenne de pollution pour le mois suivant.

Après toutes ces suppressions il a été nécessaire de regrouper toutes les colonnes restantes dans une dataframe et de calculer la valeur moyenne car sinon on avait plusieurs valeurs brutes au même endroit au même moment.

Nous avons obtenu cette courbe de répartition des valeurs moyennes :

Une image contenant capture d’écran, ligne, Tracé, diagramme

Description générée automatiquement

Après tout ce travail sur les données nous nous sommes retrouvés avec ce type de DataFrame :

#### 5.3.3 Prédictions avec Forêt Aléatoire Linéaire

On a utilisé **ColumnTransformer** pour effectuer l’encodage avec **OneHotEncoder** sur les colonnes contenant des chaines de charactère. Chacune de ces colonnes est transformée en plusieurs nouvelles colonnes, une pour chaque valeur **unique** dans la colonne originale.

Une image contenant texte, capture d’écran, noir, Police

Description générée automatiquement

Chaque enregistrement est alors codé avec une sortie binaire ( 0, 1 ) ( True, False) dans ces nouvelles colonnes, en fonction des valeurs qu'ils avaient dans les colonnes originales.

Les colonnes non spécifiées dans categorical\_cols "**year**", "**month**", "**valeur moyenne**" sont laissées telles quelles grâce à l'option **remainder='passthrough'.**

Une image contenant texte, capture d’écran, Police, clavier

Description générée automatiquement Le DF après la transformation ressemble ainsi à ça :

On définit ensuite la variable X qui contient toutes les colonnes de la DataFrame précédente sauf la dernière (la valeur moyenne que l’on veut prédire) et la variable y qui ne contient que la dernière colonne du DataFrame précédent.

Le jeu de données est ensuite divisé :

* + **80%** des données vont dans l'ensemble d'entraînement
  + **20%** vont dans l'ensemble de test.

On a ensuite testé de faire des prédictions sur un **Random Forest simple** avec **100** arbres de décisions pour voir sa performance et on peut voir qu’il a une erreur quadratique moyenne de 7:

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

On a ensuite testé d’améliorer les paramètres du modèle grâce à la fonction **RandomizedSearchCV**. Cette fonction sélectionne des combinaisons d’hyperparamètres aléatoires (ici 100) et fait une validation croisée 3-fold (**cv=3**), ce qui signifie que pour chaque combinaison d'hyperparamètres, le modèle est formé 3 fois sur 2/3 des données et évalué sur 1/3 des données pour trouver la meilleure combinaison d’hyperparamètres.

Les hyperparamètres testés sont les suivants :

param\_grid = {

'n\_estimators': [100, 200, 500],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'max\_features': ['auto', 'sqrt']

}

Nous avons pu trouver : {'n\_estimators': 500, 'min\_samples\_split': 2, 'min\_samples\_leaf': 1, 'max\_features': 'sqrt', 'max\_depth': None} comme meilleure combinaison d’hyperparamètres.

Nous avons ensuite testé les hyperparamètres trouvés par RandomizedSearchCV sur un nouveau RandomForestRegressor et obtenu cette fois-ci une **erreur** quadratique moyenne de : 6.341830874606475.

Nous avons ensuite tenté d’améliorer encore ces paramètres en testant par exemple un nombre n\_estimators encore plus grand mais ces resultats n’ont pas été suffisamment significatifs pour continuer les tests (l’erreur quadratique moyenne était toujours d’environ 6).

Les prédictions avec les données d'origine correspondantes sont ensuite sauvegardées dans un fichier csv.

#### 5.3.4 Deuxième traitement des données

La classe **GazDataProcessor** du fichier **gazs\_data.py** comme vu précédemment lit, transforme et sauvegarder les données sur la qualité de l'air.

La classe est initialisée avec les paramètres **input\_folder** et **output\_folder** qui sont les chemins vers les fichiers csv à traiter et le dossier où les fichiers traités sont enregistrés.

C’est avec la fonction **process\_csv\_files** qu’on effectue le traitement principal des fichiers csv. Elle parcourt tous les fichiers csv du dossier **input\_folder.** Pour chaque fichier elle vérifie que celui-ci n’a pas encore été traité et qu’il n’a pas déjà un fichier de sortie.

* Si le fichier vérifie ces conditions alors on charge le fichier dans un DataFrame et on vérifie que les colonnes ‘**Polluant’** et ‘**Zas’** existent
* sinon on passe au fichier suivant.

La fonction remplace également certains noms de polluants pour qu’ils aient du sens, par exemple ‘**NOX as NO2** ‘ par **NO2**, et supprime les gazs **C6H6**, **SO2** et **C0** et on retire les lignes contenant des valeurs manquantes.

Une image contenant capture d’écran, Police, texte

Description générée automatiquement

La méthode passe ensuite la colonne ‘**Date de fin’** en format **datetime** et groupe les colonnes ‘**Date de fin’**, ‘**Polluant’** et ‘**Zas’** et applique dessus la fonction **custom\_agg.**

* Pour les colonnes de type string (comme **'Organisme'**, **'nom** **site'**, etc.), la fonction **custom\_agg** prend simplement la première valeur du groupe.

**Par exemple :** pour une colonne comme **'Organisme'**, si nous avons **'ATMO GRAND EST'** et **'ATMO SUD'** dans le même groupe, elle prendrait **'ATMO GRAND EST**' car c'est la première valeur du groupe.

* Pour les colonnes de type **numérique** (comme **'valeur'**, **'valeur** **brute'**, etc.), elle calcule la **moyenne** de toutes les valeurs du groupe. Pour les autres types de colonnes, elle assigne une valeur NaN.

La fonction **process\_csv\_files** supprime ensuite les colonnes **taux de saisie, couverture temporelle** et **couverture de données** car elles sont totalement vides.

Finalement, les colonnes conservées pour l’analyse sont :

**Date de fin, Polluant, Zas, type d'implantation, type d'influence, type d'évaluation, procédure de mesure, valeur, code qualité, unité de mesure**

Ce DataFrame finale est enfin sauvegardé dans un nouveau fichier csv dans le dossier de sortie en concaténant tous les csvs dans un seul en sortie.

#### 5.3.5 Prédictions avec Régression Logistique et Régression Linéaire

Finalement, le meilleur modèle que l’on a réussi à obtenir a été avec la Régression Logistique et la **régression linéaire**.

Nous reprenons le fichier que l’on a concaténé à partir de tous ceux récupéré et nous faisons le prétraitement dessus.

Nous créons ensuite deux nouvelles colonnes **'valeur\_shifted'** et **'code\_quality\_shifted'** contenant les données des colonnes ‘**valeur’** et ‘**code\_qualité’** décalées de deux périodes en arrière, pour chaque groupe unique de **'Zas'** et **'Polluant'**.

Les deux dernières lignes de chaque groupe unique de **'Zas'** et **'Polluant'** sont supprimés dans **self.df**, car ces lignes auraient des valeurs **NaN** dans les colonnes **'valeur\_shifted' et 'code\_quality\_shifted'** dues à l'opération de décalage effectuée précédemment.

**Un One-hot encoding** est ensuite fait via la fonction **get\_dummies** pour générer des nouvelles colonnes pour chaque valeur unique des colonnes contenant des chaines de charactères. Chaque ligne de la DataFrame est ensuite mise à jour avec des **True** et **False** pour ces nouvelles colonnes : si la valeur de la colonne d'origine pour cette ligne correspond à la nouvelle colonne générée, alors elle recevra un **True**, sinon un **False**.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

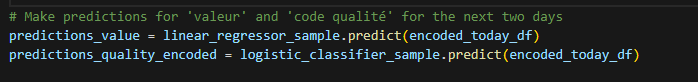
La fonction **train\_model** prend un échantillon aléatoire de **10%** du DataFrame self\_encoded obtenu précédemment.

Les données d’entrées du modèle seront toutes les colonnes de **self\_encoded** sauf **'Date de fin'**, **'valeur\_shifted'** et **'code\_quality\_shifted'**. Les données de sorties seront les valeurs de **'valeur\_shifted'** pour le modèle de régression linéaire et **'code\_quality\_shifted'** pour le modèle de régression logistique.

Pour les 2 algorithmes, 20% des données sont utilisées comme ensemble de test.

Enfin, les modèles sont entrainés en utilisant la fonction fit. Pour la régression logistique, '**code\_quality\_shifted'** est transformé en nombres à l'aide de la fonction **fit\_transform** de **LabelEncoder** pour permettre le traitement par le modèle et est ensuite entrainé sur un nombre d’itérations maximal de 1000.

La fonction **make\_predictions** sélectionne toutes les colonnes de la **DataFrame** d’aujourd’hui, et refais le prétraitement usuel.



Nous vérifions l’alignement des caractéristiques de **'today'** data avec celles de training data. Si des caractéristiques manquent dans les données de **'today'**, elles sont ajoutées avec des valeurs zéro. Si des caractéristiques supplémentaires sont présentes dans les données de **'today'**, elles sont supprimées. Enfin, la fonction réarrange l'ordre des caractéristiques dans les données de **'today'** pour qu'elles correspondent à l'ordre des caractéristiques dans les données d'entraînement.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement Pour finir, nous utilisons les **modèles de régression linéaire** et **de régression logistique** entraînés précédemment pour prédire la '**valeur'** et le **'code qualité'** pour les deux jours suivants.

# Chapitre 2

## Analyse des besoins et Conception

### 2.1 Introduction

Dans ce chapitre, nous présentons les différentes étapes de la phase de conception, de notre bot Messenger. Nous nous basons sur le Workflow dressé par nous pour spécifier les différents cas d’utilisation de notre système, ensuite, nous présentons les diagrammes de cas d’utilisation et séquences.

### 2.2 Analyse des besoins

Dans cette partie, nous nous intéressons aux besoins fonctionnels et non-fonctionnels de notre application.

#### 2.2.1 Besoins fonctionnels

Les différentes fonctionnalités offertes par notre application de bot Messenger sont listées ci-dessous.

1. **Récupération des données de différents pollens dans l’air :** Les données récupérer en temps réel par une API. Ces données présentent les différents pollens dans l’air de l’Ile de France avec le niveau de risque entre 0 et 4.
2. **Affichage des données par département :** Les données sont affichées à l’utilisateur tout dépend de département donné par lui.
3. **Abonnement aux services de TotallyAir :** Les utilisateurs sont capables de s’abonner pour pouvoir interagir avec le Bot TotallyAir.
4. **Réponse automatique :** En cas de recevoir des messages non reconnues, l’application doit être capable de fournir une réponse automatique indiquant comment interagir correctement avec le bot TotallyAir.

#### 2.2.2 Besoins non fonctionnels

Dans cette partie, nous identifions les besoins non fonctionnels de notre bot qui sont nécessaires pour réaliser un Bot Messenger apprécié par les différents utilisateurs et conforme aux normes de Facebook.

1. **L’ergonomie des commandes :** Le Bot doit être utilisée dans le maximum de confort et surtout facile à comprendre.
2. **Fiabilité :** Le fonctionnement de Bot TotallyAir doit être optimal, rapide et ne doit pas contenir des erreurs de fonctionnement.
3. **Sécurité :** Notre base de données doit être bien sécurisé parce qu’on récupère et stocke les Users Id des abonnés.
4. **L’extensibilité :** L’ajout de nouvelles options à notre application doit être facile.

### 2.3 Diagramme de cas d’utilisation

Dans cette section, nous présentons les différents digrammes de cas d’utilisation de note Bot TotallyAir avec la description de chacun.

|  |  |  |
| --- | --- | --- |
| Acteur | Définition | Droits |
| Utilisateur Facebook | Utilise le bot Messenger | * S’abonner aux services de Bot * Demande des nouveautés de pollens dans les différents départements de l’Ile de France * Envoi d’un message aléatoire |

#### 2.3.1 Diagramme de cas d’utilisation globale

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

#### 2.3.2 Description détaillée au diagramme de cas d’utilisation global

Le diagramme de cas d’utilisation décrit les différentes actions prises par l’utilisateur de Bot, il sert à structurer les besoins de l’utilisateur. La Figure x.x présente le diagramme de cas d’utilisation de notre Bot TotallyAir. Nous avons un seul type d’utilisateur qui peut interagir par s’abonner aux services de TotallyAir ou par demander la qualité d’air du jour présent.

#### 2.3.3 Diagramme détaillé du cas d’utilisation S’abonner

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

Dans le tableau x.x, nous présentons une description détaillée du cas d’utilisation « S’abonner au services TotallyAir »

|  |  |
| --- | --- |
| Cas d’utilisation | S’abonner au services TotallyAir |
| Acteur | Utilisateur Facebook |
| Post-Condition | Compte Facebook |
| Scenarios | * L’utilisateur peut s’abonner au Bot pour recevoir les mises à jour de pollens dans l’air. |

#### 2.3.4 Diagramme détaillé de cas d’utilisation de recevoir les mises à jour.

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

Dans le tableau x.x, nous présentons une description détaillée de cas d’utilisation « Recevoir des mises à jour ».

|  |  |
| --- | --- |
| Cas d’utilisation | Recevoir des mises à jour de niveau de pollens dans l’air |
| Acteur | Utilisateur Facebook |
| Pré-condition | S’abonner |
| Post-Condition | Demander les niveaux de pollens dans l’air de jour présent |
| Scenarios | * Envoie de « update me » + numéro de département de l’Ile de France. |

### 2.4 Diagramme de séquence

Dans cette partie, nous enchaînons les diagrammes de séquences pour détailler les différentes fonctionnalités de notre système.

#### 2.4.1 Diagramme de séquence relatif au cas d’utilisation « S’abonner aux services de Bot TotallyAir »

Pour pouvoir consulter les nouveautés concernant les niveaux des pollens dans l’air. L’utilisateur doit simplement envoyer un message « Subscribe »

Une image contenant capture d’écran, ligne, diagramme

Description générée automatiquement

#### 2.4.2 Diagramme de séquence relatif au scénario « recevoir les mises à jour »

Pour recevoir les nouveautés de niveau des Pollens dans l’air, il faut envoyer « update me » + numéro de département.

Une image contenant capture d’écran, texte, ligne, conception

Description générée automatiquement

### 2.6 Conclusion

Dans ce chapitre, nous définissons l’acteur de notre Bot. Nous détaillons aussi les besoins fonctionnels et non fonctionnels de notre système et nous introduisons le diagramme de cas d’utilisation. Nous présentons également les diagrammes de séquence. Le chapitre suivant est consacré à la partie réalisation de notre solution.

# Chapitre 3

## Réalisation, Choix technologique et environnement du travail

### 3.1 Introduction

Dans ce chapitre, nous présentons les différentes configurations effectuées avant de commencer la réalisation de notre application. Nous abordons, ensuite, l’environnement de développement logiciel et les différents choix technologiques effectués dans la réalisation de notre projet.

### 3.2 Architecture d’un bot Messenger

Pour qu’un bot de Facebook soit capable de répondre à un nombre de commande envoyer par un utilisateur, il faut avoir une url Webhook, d’où il le permet de communiquer avec du code python qui crée les messages pour répondre, et communique avec les différents services de Amazon Web Services.

La figure x.x ci-dessous représente une architecture classique d’un bot Messenger.

La mise en place d’un Bot, nécessite comme première étape la création d’une base de données dans Google Cloud, Amazon Web Services etc… avec un code python comme dans notre cas qui gère la relation entre les messages envoyer au bot et la base de données en temps réel. Quand on parle de temps réel, on parle de machine serveur, qui tourne sans arrêt (comme une instance EC2). Et pour faire la liaison entre les deux, on utilise un service de Cloud Computing comme Heroku qui nous permettre de déployer, d’héberger et de gérer notre bot.

* Utilisateur : Bot Messenger, utilisation de l’API Facebook pour recevoir les demandes des utilisateurs de Facebook.
* Cloud Computing : Heroku
* Systéme de gestion de base de données : AWS

Une image contenant capture d’écran, texte, conception

Description générée automatiquement

### 3.3 Cloud Computing utilisé

Nous introduisant dans cette section la définition de Heroku ainsi que son principe de fonctionnement

#### 3.3.1 Définition du Cloud Computing avec Heroku

Le cloud computing avec Heroku fait référence à déployer et à exécuter des applications en temps réel sur la plate-forme cloud de Heroku, en utilisant son infrastructure et ses ressources évolutives.

Le cloud computing d'Heroku implique l'hébergement et la gestion de notre Bot dans un environnement virtuel, nous permettant se concentrer sur l'écriture de code tout en tirant parti des capacités de mise à l'échelle et de déploiement automatiques de Bot.

#### 3.3.2 Principe de fonctionnement de cloud computing

Heroku fonctionne sur une architecture basée sur des conteneurs, isolant et gérant les processus d'application au sein d'unités légères et autonomes appelées dynos.

Avec Heroku, les applications sont déployées et mises à l'échelle horizontalement en ajoutant ou en supprimant des dynos en fonction de la demande, garantissant une allocation et des performances optimales des ressources.

Dans notre cas, on Heroku app est liée avec GitHub, ou chaque mise à jour de code source, on passe par déployer dans Heroku. On utilise les logs en temps réel qui nous aussi à débugger et à voir ce qui se passe lors de la réception des messages des utilisateurs Facebook.

Ci-dessous un schéma d’architecture explique le fonctionnement de Heroku sachant qu’on n’a pas encore utilisé les add-ons.

Une image contenant capture d’écran, texte, Police, conception

Description générée automatiquement

Ci-dessous un Exemple de Logs en temps réel lors de la réception des messages.

Une image contenant texte, Police, nombre, capture d’écran

Description générée automatiquement

Si le Bot ne reçoit pas des messages pour quelques heures, le Worker de Heroku s’arrête, et il se démarre qu’on il y’a un message envoyer, on peut voir ‘Script started’ qui montre le lancement de code Python qui va gérer le message.

### 3.4 Amazon Web Services

Passant vers la base de données mis en place. Nous avons utilisé Amazon Web Services (AWS), une plateforme de cloud computing. AWS nous a fourni une gamme de produits cruciaux pour le développement et le déploiement de notre application.

Pour le stockage de données en temps réel, on utilise Amazon S3, un service de stockage en nuage, pour stocker nos données sur la qualité de l'air. S3 nous a permis de stocker de manière sécurisée et accessible de grandes quantités de données. On à connecter une instance EC2 (avec du code qui tourne chaque 1h de matin) qui injecte les données traiter dans un csv stocker dans le seau S3.

De plus, nous avons utilisé Amazon RDS pour gérer notre base de données où on stocke les utilisateurs Facebook IDs. RDS a simplifié le processus de configuration et de gestion de notre base de données, nous permettant de nous concentrer sur le développement d'applications plutôt que sur la gestion de la base de données.

La figure X.X montre la liaison entre les S3/RDS avec les instances.

### 3.5 Choix Technologiques et Logiciels

Dans cette partie, nous nous intéressons aux langages, aux bibliothèques et aux techniques de programmation utilisées tout au long de la réalisation de notre application en justifiant nos choix.

#### 3.5.1 Choix Technologiques

3.5.1.1 Python

Nous avons choisi Python parce qu’il est un langage de programmation facile à utiliser, à maintenir avec une vaste sélection de bibliothèque disponible qui facilite nos travails.

Bibliothèques Python : Nous avons utilisé plusieurs bibliothèques Python dans notre projet :

* Psycopg2 : cette bibliothèque a été utilisée pour interagir avec notre base de données PostgreSQL (RDS qui a comme engine PostgreSQL.
* Requests : cette bibliothèque a été utilisée pour effectuer des requêtes HTTP afin de récupérer des données à partir d'API.
* Boto3 : Nous l'avons utilisé pour interagir avec les services AWS comme S3 et RDS.
* BeautifulSoup : cette bibliothèque a été utilisée à des fins de grattage Web, pour extraire des données de documents HTML et XML.
* Sklearn et LogisticRegression : Ceux-ci ont été utilisés pour créer un modèle d'apprentissage automatique.

3.5.1.2 WSL (Windows Subsystem for Linux)

Nous avons utilisé WSL avec Ubuntu pour interagir avec nos instances AWS EC2. WSL nous permet d'utiliser un environnement de terminal Ubuntu à part entière dans Windows, ce qui facilite la gestion de nos instances EC2 basées sur Linux.

#### 3.5.2 Choix de logiciels

3.5.2.1 Heroku

Nous avons choisi Heroku comme plate-forme de déploiement parce qu’il est facile à utiliser, et nous permet d’intégrer le code sources et les mises à jour avec GitHub d’une manière transparente pour un déploiement continu et au même temps il nous fournit un URL Webhook qui est nécessaire pour le fonctionnement de Bot.

3.5.2.2 Facebook Messenger Bot

La raison qui justifie le choix de développer un bot sur Messenger et pas sur Twitter ou autre plateforme c’est le cout de l’API où chez Facebook, c’est gratuit. Car il s'agit d'une plate-forme largement utilisée. Cela rend notre application facilement accessible.

3.5.2.3 GitHub

On a utilisé GitHub pour faire des mises à jour dans le code source déployer dans Heroku App.

3.6 Conclusion

Ce chapitre résume les outils et méthodologies utilisés pour livrer notre appli cation. Nous parlons de l’architecture globale de l’application, de l’approche que nous adoptons et de notre environnement de travail, ensuite nous parlons des différent choix technologiques et logicielles que nous utilisons pour