Machine learning et analyse de séries temporelles pour la Smart City

En appliquant des algorithmes SARIMAX/VAR

Projet TX

Etudiant : Jinshan GUO

GI04 FDD(IA)

Professeur : Gilles MOREL

Automne 2020

Université de Technologie de Compiègne

Résumé

Ce projet a débuté par la recherche de données de séries temporelles qui sont générées par des capteurs, soient installés dans la ville, soient dans les bâtiments, répartis dans des zones qui sont proches l'un de l'autre. Cela nous permet d'explorer la corrélation entre capteurs et d'essayer de trouver la possibilité de pouvoir quand même prédire les valeurs d'un capteur, par exemple, qui est tombé en panne brusquement, en utilisant des données mesurées par d'autres capteurs.

Basé sur les données de la pollution atmosphérique et des données de météo qui satisfont notre besoin, le cadre d'usage proposé est de pouvoir prédire le niveau d'alerte des polluants afin de pouvoir donner des conseils aux citoyens dans quelle période pour le prochain jour ils pourront sortir sans soucis de nuire la santé à cause de pollution dans l'air.

En appliquant des modules statistiques SARIMAX, VAR sur les séries temporelles, on a non seulement bien atteint notre objectif en trouvant le bon modèle pour notre cas, mais aussi bien compris dans quelle situation, quel modèle sera le meilleur choix en comparant les avantages et les inconvénients pour chaque algorithme.

Pour faciliter la gestion et la compréhension des fonctions personnalisées, on a regroupé des fonctions en morceaux en ajoutant des commentaires et des explications. Le développement s'est basé sur Google Colab qui permet de travailler sous Jupyter notebook en ligne (langage Python).

Enfin, Je remercie sincèrement Monsieur Gilles MOREL de me diriger, de me donner des conseils, de m'encourager, de prendre le temps pour m'expliquer de nouvelles choses et aussi de me proposer des ressources à apprendre tout au long du semestre. Ce projet m'a permis d'avoir un pas initial sur Machine Learning, de bien connaître la difficulté et la passion d'analyse des données et surtout d'avoir développé mes compétences de codage ainsi que mes capacités à résoudre de nouveaux problèmes.

Mots clés : Machine Learning, Séries Temporelles, Smart City, SARIMAX, VAR

Table des matières

1 Introduction

En fonction de la base des données des séries temporelles de la pollution atmosphérique, et de la météo captées par les capteurs déployés dans la ville de Copenhague au Danemark, mises à jour toutes les 30 minutes du 01/10/2020 au 31/10/2020, notre objectif est de visualiser les données mesurées et les prédictions dans un tableau d'abord en indiquant le niveau de qualité de l'air pour chaque heure, et aussi de prédire le niveau d'alerte des polluants à l'aide des meilleurs modèles qui s'adaptent à notre situation, obtenus en mettant en place des algorithmes SARIMAX, VAR.

En suivant la procédure classique pour un projet de Machine Learning, on le divise en trois phases.

Première phase : la recherche de données, la proposition de cas d'usage, la préparation de données pouvant alimenter notre modèle.

Deuxième phase :la répartition de données en un ensemble d'entraînement et un ensemble d'essai, la sélection des modèles, l'entraînement ainsi que l'évaluation des modèles.

Troisième phase : la prédiction et l'optimisation du paramètre pour obtenir meilleur modèle.

On va maintenant entrer dans les détails de chaque phase pour vous expliquer comment on a réussit à mettre en place des algorithmes statistiques pour prédire la serie temporelle.

2 Données et capteurs

2.1 Description de données

2.1.1 Données de pollution

Les données de pollution atmosphérique contiennent 7 types différents de polluant avec son unité indiquée : NO2(µg/m³), NOx (µg/m³), CO (mg/m³), O3 (µg/m³), PM10 (µg/m³), PM25 (µg/m³), SO2 (µg/m³).

Afin d'évaluer la gravité de pollution de l'air, on doit calculer l'indice de qualité de l'air (Air Quality Indice, AQI) qui est une norme de qualité de l'air, divisée en six catégories indiquant des niveaux croissants de préoccupation pour la santé et chaque catégorie a sa propre couleur représentative[2].

IQA	Niveau de pollution de l'air	Impact sur la santé
0 - 50	Bon	La qualité de l'air est jugée satisfaisante, et la pollution de l'air pose peu ou pas de risque.
51 -100	Modéré	La qualité de l'air est acceptable. Cependant, pour certains polluants, il peut y avoir un risque sur la santé pour un très petit nombre de personnes inhabituellement sensibles à la pollution atmosphérique.
101-150	Mauvais pour les groupes sensibles	La qualité de l'air est acceptable; Cependant, pour certains polluants, il peut y avoir un problème de santé modérée pour un très petit nombre de personnes qui sont particulièrement sensibles à la pollution de l'air.
151-200	Mauvais	Tout le monde peut commencer à ressentir des effets sur la santé; les membres des groupes sensibles peuvent ressentir des effets de santé plus graves.
201-300	Très mauvais	Avertissements de santé de conditions d'urgence. Toute la population est plus susceptible d'être affecté.
300+	Dangereux	Alerte de santé: tout le monde peut ressentir des effets de santé plus graves.

FIGURE 1 – Indice de Qualité de l'Air (AQI)

Comme l'unité de l'AQI est ppb (parties par milliard en volume) au lieu de µg/m³ (microgrammes de polluant gazeux par mètre cubique) pour les polluants (NO2, NOx, O3, SO2), on doit d'abord convertir l'unité. (NO2 : 1 ppb = 1.88 µg/m3, NOx : 1 ppb = 1.25 µg/m3, O3 : 1 ppb = 2.00 µg/m3, SO2 : 1 ppb = 2.62 µg/m3).

Pour calculer l'AQI, il faut utiliser l'équation suivante[3] :

$$I = \frac{I_{high} - I_{low}}{C_{high} - C_{low}}(C - C_{low}) + I_{low}$$

— I est l'indice (de qualité de l'air)
— C est la concentration de polluants
— C_{low} est le point de rupture de la concentration qui est $\leq C$
— C_{high} est le point de rupture de la concentration qui est $\geq C$
— I_{low} est le point de rupture de l'indice correspondant à C_{low}
— I_{high} est le point de rupture de l'indice correspondant à C_{high}

Basé sur cette formule, on peut consulter dans le tableau suivant des valeurs pour chaque point de rupture et effectuer le calcul :

O_3 (ppb)	O_3 (ppb)	$PM_{2.5}$ (µg/m^3)	PM_{10} (µg/m^3)	CO (ppm)	SO_2 (ppb)	NO_2 (ppb)	AQI	AQI
C_{low} - C_{high} (avg)	C_{low} - C_{high} (avg)	C_{low}- C_{high} (avg)	C_{low} - C_{high} (avg)	C_{low} - C_{high} (avg)	C_{low} - C_{high} (avg)	C_{low} - C_{high} (avg)	I_{low} - I_{high}	Category
0-54 (8-hr)	-	0.0-12.0 (24-hr)	0-54 (24-hr)	0.0-4.4 (8-hr)	0-35 (1-hr)	0-53 (1-hr)	0-50	Good
55-70 (8-hr)	-	12.1-35.4 (24-hr)	55-154 (24-hr)	4.5-9.4 (8-hr)	36-75 (1-hr)	54-100 (1-hr)	51-100	Moderate
71-85 (8-hr)	125-164 (1-hr)	35.5-55.4 (24-hr)	155-254 (24-hr)	9.5-12.4 (8-hr)	76-185 (1-hr)	101-360 (1-hr)	101-150	Unhealthy for Sensitive Groups
86-105 (8-hr)	165-204 (1-hr)	55.5-150.4 (24-hr)	255-354 (24-hr)	12.5-15.4 (8-hr)	186-304 (1-hr)	361-649 (1-hr)	151-200	Unhealthy
106–200 (8-hr)	205–404 (1-hr)	150.5–250.4 (24-hr)	355–424 (24-hr)	15.5–30.4 (8-hr)	305–604 (24-hr)	650–1249 (1-hr)	201–300	Very Unhealthy
-	405-504 (1-hr)	250.5-350.4 (24-hr)	425-504 (24-hr)	30.5-40.4 (8-hr)	605-804 (24-hr)	1250-1649 (1-hr)	301-400	Hazardous
-	505-604 (1-hr)	350.5-500.4 (24-hr)	505-604 (24-hr)	40.5-50.4 (8-hr)	805-1004 (24-hr)	1650-2049 (1-hr)	401-500	

FIGURE 2 – Le tableau des points de rupture pour calculer AQI

```
# Convert pollutant concentration unit (just one time)
Pol_BA = conversion_to_ppb(Pol_BA)
```

2.1.2 Données de météo

Les données de météo contiennent 5 indicateurs avec son unité indiquée : Direction du vent, Rapidité du vent(m/s), Température(°C), Humidité relative(%), Radiation solaire(W/m2).

2.2 Information des capteurs

Il y a quatres capteurs, trois capteurs pour mesurer la concentration de pollution, un capteur pour mesurer le chagement des indicateurs de météo, on va lister ci-dessous les capteurs et leur information :

— H.C. Boulevard Andersens/1103 (Capteur Pollution 1)

Position géographique : Station de rue ($12^o34'20"E, 55^o40'30"N$)

Installation : Mesuré au niveau de la rue (dans les rues à fort trafic)

Description : La station orientée trafic (station N^o.1103) est située du côté nord d'une rue à 6 voies. Des deux côtés de la rue principale, il y a un parking le long des rues latérales et des trottoirs parallèles. Du côté nord, il y a des propriétés résidentielles de 5 à 6 étages qui abritent l'administration municipale, tandis que du côté opposé se trouve un parc d'attractions avec des arbres et des bâtiments bas. La densité du trafic est d'environ 70 000 véhicules à moteur par jour.

— Jagtvej/1257 (Capteur Pollution 2)

Position géographique : Station de rue ($12^o33'12"E, 55^o41'54"N$).

Installation : Mesuré au niveau de la rue (dans les rues à fort trafic).

Description : La station orientée trafic (station N^o.1257) est située dans un rayon de 20m de large entre la piste cyclable et la chaussée. L'endroit est entouré de bâtiments résidentiels de cinq étages des deux côtés de la rue. La zone se compose principalement de maisons avec de petits magasins et les bâtiments sont des propriétés résidentielles de 3 à 5 étages. Le site est parcouru quotidiennement par environ 22 000 véhicules automobiles.

— Institut H. C. Ørsted/1259 (Capteur Pollution 3).

Position géographique : Station de toit ($12^o33'41"E, 55^o42'1"N$).

Installation : Mesuré au niveau du contexte urbain (sur les toits ou dans les exploitations agricoles des zones urbaines).

Description : La station de toit (station $N^o.1259$) est située sur un toit plat d'un immeuble de sept étages qui abrite des instituts et des laboratoires appartenant à la Faculté des sciences de l'Université de Copenhague. L'endroit est ouvert et arboré entre les départements universitaires. La distance jusqu'à la gare routière ($N^o.1257$) est d'environ 300m Il y a plusieurs rues avec une intensité de trafic de plus de 5 000 véhicules par jour dans les kilomètres les plus proches.

— H.C. Ørstedvej (Capteur météo)

Position géographique : Station de toit ($12^o54'85"E, 55^o68'05"N$).

Installation : Mesuré de l'Information météorologique au niveau du contexte urbain.

Les capteurs sont proches l'un de l'autre au niveau de l'emplacement géographique :

— Du H.C. Ørstedvej vers Jagtvej : 7 mins en voiture, 3.9 km
— Du Jagtvej vers Institut H. C. Ørsted : 2 mins en voiture, 600 m
— Du Institut H. C. Ørsted vers H.C. Boulevard Andersens : 10 mins en voiture, 4.5 km
— Du H.C. Boulevard Andersens vers H.C. Ørstedvej : 4 mins en voiture, 1.9 km

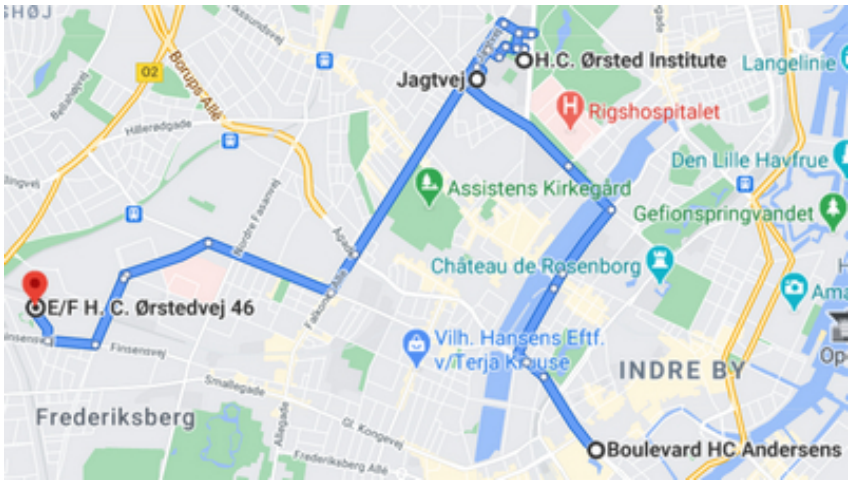

FIGURE 3 – Position géographique des capteurs

Cela a répondu à notre besoin et nous donne la possibilité d'explorer la corrélation parmi les capteurs et de prédire les valeurs d'un capteur à l'aide de l'enregistrement d'autres capteurs. Jusqu'à présent, on a déjà eu des bonnes ressources de données, et on peut commencer à manipuler les données.

2.3 Visualisation de données

La première étape est de télécharger les données et les stocker dans un fichier en format csv à partir du site[3]en appelant le DATA API. On constate que les données brutes sont vraiment désordonnées, donc il faut qu'on les traite soit par du script, soit manuellement. Notre objectif est de rendre les données soient assez propres même si il existe des valeurs manquantes pour qu'ils puissent être importées correctement.

La deuxième étape est d'importer les données, d'afficher tout leurs caractéristiques, les indices pour vérifier si les données soient bien des séries temporelles, et de aussi visualiser les données pour avoir une vue initiale de leurs valeurs maximum et minimum, leur tendances, leur variations etc. On a aussi visualisé les valeurs manquantes grâce à la librairie python 'missingno'.

```
# Importing pollution data
Pol_BA = pd.read_csv("Pollution_Boulevard Andersens.csv", sep=',', names=
    ['DateTime','NO2', 'NOx', 'O3', 'CO', 'SO2', 'PM10', 'PM2.5'], header=0,
    parse_dates=['DateTime'], index_col='DateTime')

# Inspecting pollution data
inspect_data(Pol_BA, visualize_missing_value=True)

# Display pollution data
plot_data(Pol_BA)
```

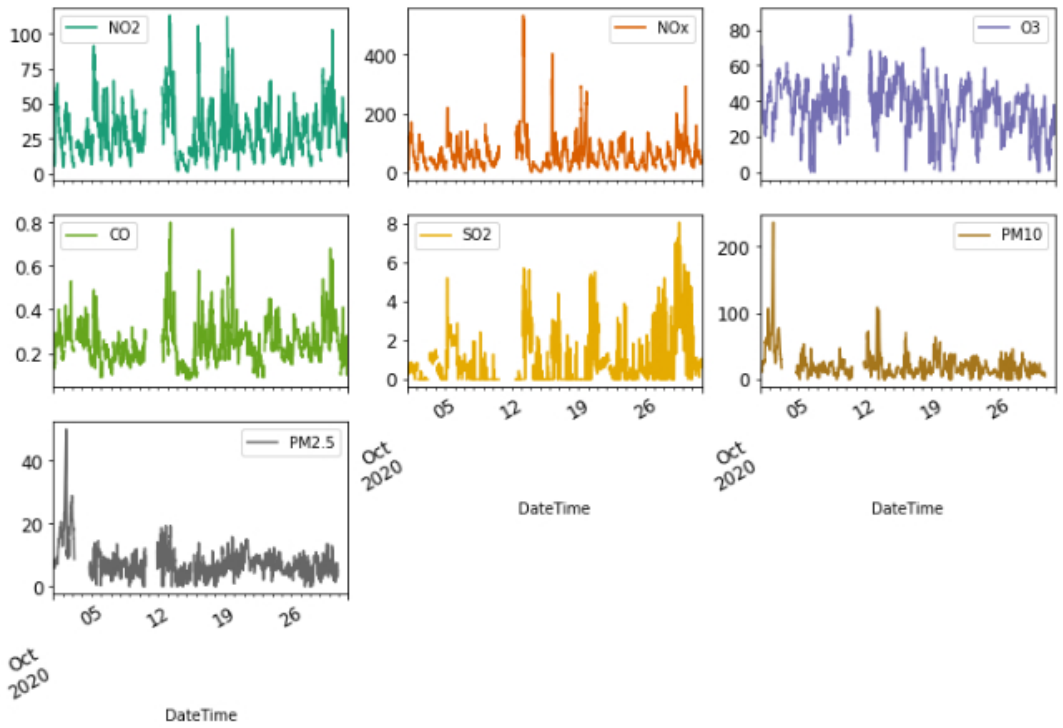

FIGURE 4 – Visualisation de données - Capteur pollution 1

On constate qu'il existe des données manquantes pour chaque série temporelle, on les visualise donc en affichant une matrice avec des lignes blanches qui représentent les valeurs manquantes. On peut aussi afficher en figure 'bar' pour voir le nombre de valeurs manquantes.

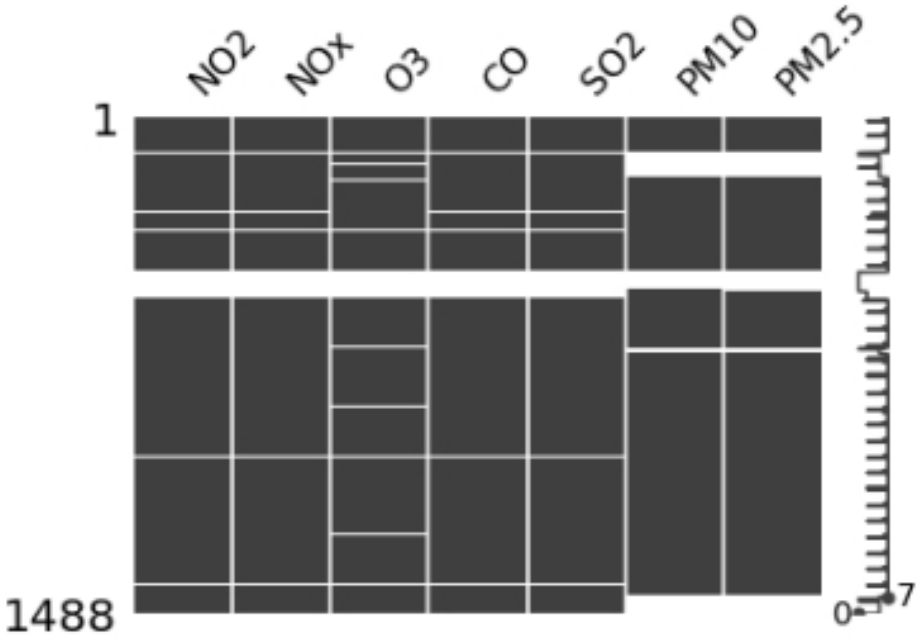

FIGURE 5 – Visualisation de valeurs manquantes - Capteur pollution 1

3 Prétraitement des données

3.1 Remplissage de valeurs manquantes

On peut remplir les valeurs manquantes de plusieurs façons différentes : par une valeur fixée ou statistiques (moyenne, variance, médian), par la valeur précédente dans la série, par la valeur suivante dans la série, ou par la méthode d'interpolation linéaire, polynomiale etc.

3.2 Normalisation et Reconversion

Comme les données sont à des échelles différentes, il est préférable de normaliser les données pour que les modèles de Machine Learning soient plus performants. Après l'entraînement de données, on est toujours obliger de reconvertir le résultat de prédiction pour qu'il puisse garder les mêmes échelles que les données originales.

Pour notre cas, on a d'abord utilisé les données non normalisées pour entraîner les modèles, le résultat qu'on a obtenu est relativement bon mais mauvaise que celui après la mise en œuvre de la normalisation.

```
# Normalize the data
Pol_BA_df_copy, Pol_BA_df_avgs, Pol_BA_df_devs = nomalize_data(Pol_BA_df_copy)

# Reconvert the data
prediction_osa_1 = Reconvert_prediction(df_to_convert=prediction_osa_1,
    df_trans=Pol_BA_df_copy, avgs=Pol_BA_df_avgs, devs=Pol_BA_df_devs)
```

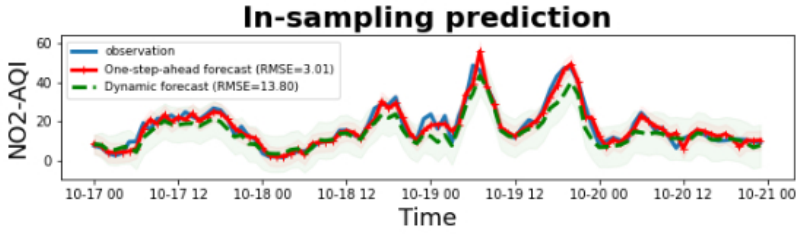

FIGURE 6 – Résultat de prédition sans normalisation - NO2

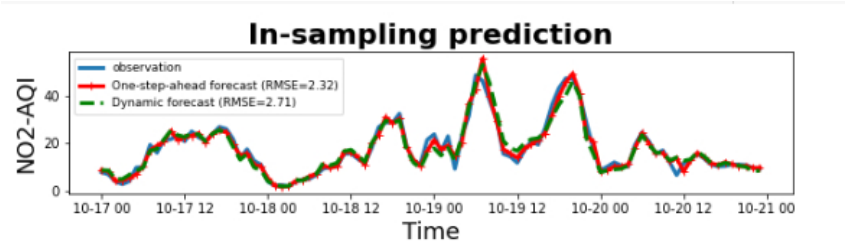

FIGURE 7 – Résultat de prédition avec normalisation - NO2

IL est évident que la prédiction avec normalisation est meilleure que celle sans normalisation, d'où l'importance de toujours penser à standardiser les données.

Après avoir remplacé les valeurs manquantes et avoir normalisé les données, on peut généralement commencer à appliquer les algorithmes de Machine Learning. Mais le cas est un peu particulier pour les séries temporelles et les algorithmes statistiques comme ARIMA, SARIMAX, VAR, etc. Il faut qu'on vérifie tout d'abord que la série temporelle est stationnaire, c'est'-à-dire dont la variance, la moyenne, la tendance ne varie pas dans le temps, et ensuite vérifier si les autres conditions nécessaires sont satisfaisantes pour qu'on puisse mettre en œuvre le modèle et avoir un bon résultat. On va maintenant présenter comment on peut réussir à appliquer les modèles SARIMAX, VAR respectivement.

Tous les développements et toutes les explications suivantes se baseront sur le fait qu'on a pris les polluants (NO2, NOx, CO) du capteur pollution 1 et le facteur VH (la vitesse du vent) du capteur pollution 2. Car ce sont des caractéristiques qui ont le plus d'impact sur la pollution de l'air et qui sont corrélés les uns avec les autres.

4 Mise en œuvre - SARIMAX

4.1 Introduction

SARIMAX (Seasonal AutoRegressive Integrated Moving Average eXogenous model) sont de modèles saisonniers, est une généralisation des modèles (AR, MA, ARMA, ARIMA ou ARIMAX) qui peut intégrer la saisonnalité, l'intégration et/ou les variables exogènes en même temps. Donc en fixant les valeurs de certains ordres à 0, ou en ne fournissant pas certaines informations, le modèle peut être converti en (AR, MA, ARMA, ARIMA ou ARIMAX).

La saisonnalité se produit lorsque certains schémas ne sont pas cohérent, mais apparaissent périodiquement. Pour tenir compte d'un tel schéma, nous devons inclure dans le modèle les valeurs enregistrées pendant la période périodique précédente.

Par rapport à l'ARIMAX qui exige 3 ordres non-saisonnières (p, d, q), SARIMAX nécessite quatres ordres supplémentaires (P, D, Q, s). (P, D, Q) ne sont que des versions saisonnières des ordres de l'ARIMA. En d'autres terme, on a :

- un ordre auto-régressif saisonnier désigné par P.
- un ordre d'intégration saisonnière désigné par D.
- un ordre de moyenne mobile saisonnière désigné par Q.
- un ordre de la durée du cycle saisonal désigné par s (seasonal lags). Par exemple, si on dispose de données horaires et que la longueur du cycle est de 24, le schéma saisonnier apparaît une fois toutes les 24 heures.

L'ordre (P, Q) détermine le nombre de décalages saisoniers que l'on considère. Par exemple, si P = 2 et s = 24, alors nous incluons les valeurs de 1 et 2 décalages saisoniers, ce qui correspond à 24 et 48 décalages. Ensuite, si p = 1, on inclurait $X_{t-1}, X_{t-24}, X_{t-25}, X_{t-48}, X_{t-49}$

car pour chacune des deux décalages saisoniers, on doit inclure autant de p des valeurs passées pertinentes pour elle.

On voit maintenant à quoi ressemble l'équation d'un modèle d'ordre SARIMAX (1,0,0) et d'un ordre saisonnier (2,0,0,24) :

$$X_t = C + \phi_1 X_{t-1} + \phi_{24} X_{t-24} + \phi_{25} X_{t-25} + \phi_{48} X_{t-48} + \phi_{49} X_{t-49} + \epsilon_t$$

Toutefois, les valeurs pour ϕ_{25} et ϕ_{49} sont limitées. Elles doivent être égales à $\phi_1\phi_{24}$ et $\phi_1\phi_{48}$ respectivement.

$$X_t = C + \phi_1 X_{t-1} + \phi_{24} X_{t-24} + \phi_1\phi_{24} X_{t-25} + \phi_{48} X_{t-48} + \phi_1\phi_{48} X_{t-49} + \epsilon_t$$

Ainsi, nous pouvons réécrire l'équation pour obtenir ce qui suit :

$$X_t = C + \phi_1 X_{t-1} + \phi_{24}(X_{t-24} + \phi_1 X_{t-25}) + \phi_{48}(X_{t-48} + \phi_1 X_{t-49}) + \epsilon_t$$

Par souci de cohérence, on va utiliser une notation distincte Φ_1 et Φ_2 pour les coefficients de saisonnalité également :

$$X_t = C + \phi_1 X_{t-1} + \Phi_1(X_{t-24} + \phi_1 X_{t-25}) + \Phi_2(X_{t-48} + \phi_1 X_{t-49}) + \epsilon_t$$

Maintenant, si on décide d'inclure les résidus en indiquant (q,Q), on doit savoir que les ordres saisonniers ne s'influencent pas directement les uns avec les autres. On voit maintenant à quoi ressemble l'équation d'un modèle SARIMAX (1,0,2)(2,0,1,24) :

$$X_t = C + \phi_1 X_{t-1} + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \Phi_1(X_{t-24} + \phi_1 X_{t-25}) + \Phi_2(X_{t-48} + \phi_1 X_{t-49}) + \theta_{24}\epsilon_{t-24} + \theta_{25}\epsilon_{t-25} + \theta_{26}\epsilon_{t-26} + \epsilon_t$$

De plus, les coefficients θ_{25} et θ_{26} sont également limités et sont égaux à $\theta_1\theta_{24}$ et $\theta_2\theta_{24}$ respectivement, donc on peut obtenir finalement :

$$X_t = C + \phi_1 X_{t-1} + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \Phi_1(X_{t-24} + \phi_1 X_{t-25}) + \Phi_2(X_{t-48} + \phi_1 X_{t-49}) + \Theta_1(\epsilon_{t-24} + \theta_1\epsilon_{t-25} + \theta_2\epsilon_{t-26}) + \epsilon_t$$

Après bien compris le principe mathématique de modèle SARIMAX, notre objectif est de trouver toutes les ordres de SARIMAX (p, d, q)x(P, D, Q, s) de modèle pouvant bien expliquer les caractéristiques de données.

4.2 Détrendisation exponentielle & différentiation

Les ensembles de données de séries temporelles peuvent contenir des tendances et des variations saisonnières : les tendances peuvent entraîner une moyenne variable dans le temps, tandis que la saisonnalité peut entraîner une variance changeante dans le temps, qu'il est donc nécessaire de supprimer avant la modélisation.

Pour le cas où notre série n'est pas stationnaire, si elle présente une tendance exponentielle, on peut d'abord supprimer cette tendance en appliquant la fonction log sur les données. Si elle n'est enocre pas stationnaire, on peut la rendre stationnaire par différenciation.

La différenciation peut aider à stabiliser la moyenne des séries temporelles en supprimant les changements de niveau d'une série temporelles, et donc en éliminant la tendance et la saisonnalité. Elle s'effectue en soustrayant l'observation précédente de l'observation actuelle :

$$Difference(t) = Observation(t) - Observation(t-1)$$

De plus, la reconversion est nécessaire lorsque de prédiction, qui doit être reconvertie à l'échelle originale. Ce processus peut être inversé en ajoutant à la valeur de la différence l'observation de l'étape précédente.

$$Reconverted(t) = Difference(t) + Observation(t-1)$$

4.3 Vérification de stationnarité

Avant d'aller plus loin dans notre analyse, nos séries doivent être rendues stationnaires afin qu'elles soient plus faciles à analyser et qu'il soit possible de les prévoir, ce qui signifie qu'elles doivent présenter des propriétés statistiques constantes dans le temps. La stationnarité dont on parle est la stationnarité faible avec des moyens et des écarts constants, qui peut être inspectée de deux façons différentes :

4.3.1 Graphique de la moyenne et de la variance mobile

La première façon est de tracer la figure de la moyenne ainsi que la variance mobile en utilisant une 'fenêtre glissante' :

```
# Plotting rolling means and variances by setting the window equal to
    seasonal lags
plot_rolling_means_variance(Pol_BA_df_NO2, window=24)
```

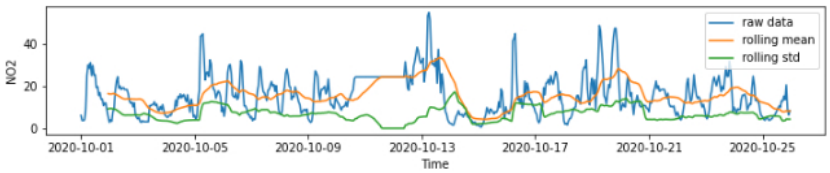

FIGURE 8 – Moyenne et variance mobile sans normalisation - NO2

Comme la série normalisée présente la moyenne et la variance en tendance constante, On peut conclure que la série est stationnaire.

4.3.2 Test d'AdFuller

Une autre façon est d'utiliser la méthode statistique AdFuller test, en fonction de p-value retournée, on peut décider si on rejette ou pas l'hypothèse nulle (suppose que la série n'est pas stationnaire) avec un certain niveau de confiance. Si le p-value est inférieure à 0.05, on peut rejeter l'hypothèse nulle en disant que la série est stationnaire.

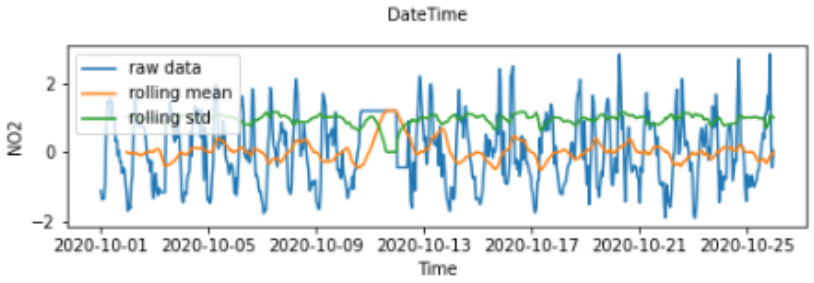

FIGURE 9 – Moyenne et variance mobile avec normalisation - NO2

```
# Applying ADFuller test to check if the series is stationary
is_stationnary(Pol_BA_df_NO2)
```

```
Augmented Dickey-Fuller Test on NO2
Null Hypothesis: Data has unit root. Non-Stationary.
Test statistic = -6.184
P-value = 0.000
Critical values :
        1%: -3.441 ==> The data is  stationary with 99% confidence
        5%: -2.866 ==> The data is  stationary with 95% confidence
        10%: -2.569 ==> The data is  stationary with 90% confidence
```

FIGURE 10 – Vérification de stationnarité - NO2

On peut constater dans la figure que le p-value = 0, donc la série est stationnaire.

4.4 Décomposition - Extrait de la 'saisonnalité'

Pour une série temporelle, on peut la décomposer en trois composants : tendance, saisonnalité, résiduel pour examiner son pattern et décider son décalage saisonnier (seasonal lags) servant à nous aider à fixer l'ordre et l'ordre saisonnière du modèle SARIMAX.

On peut constater que la série NO2 présente une saisonnalité journalière, c'est-à-dire que le seasonal lags est équivalent à 24h. On peut aussi extraire seulement le composant saisonnalité en le zoomant sur un jour pour voir dans les détails comment la donnée s'évolue dans le temps.

```
# Extract the seasonality
seasonal = decompose_series(Pol_BA_df, return_seasonal=True)

# Visualize the seasonality for one day
seasonal['2020-10-01'].plot(subplots=True, layout=(3,3), figsize=(15,5),
    colormap='Dark2')
```

Sur le graphique saisonnier ci-dessus, on peut voir que la concentration de NO2 augmente pendant la période de 6h-8h, 13h-14h et 17h-18h. Le NOx et le CO suivent le même schéma. Comme on le sait tous, il y a souvent trop de personnes qui vont au travail en

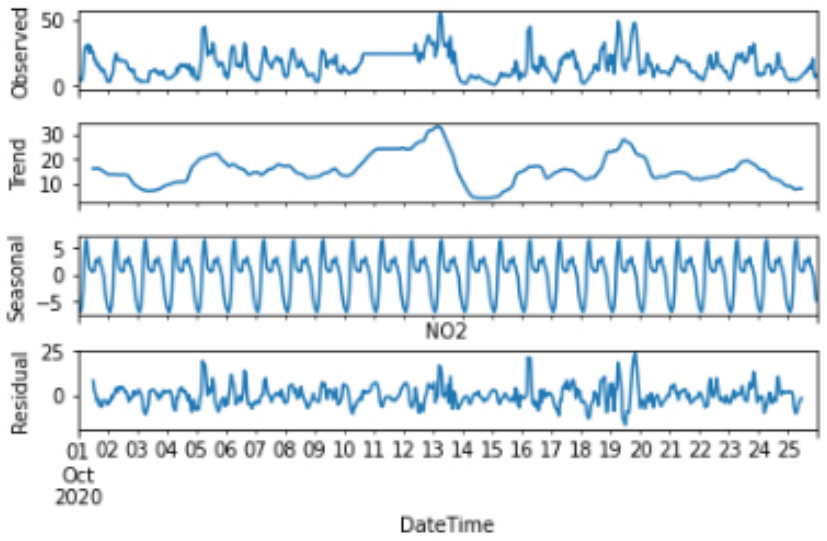

FIGURE 11 – Décomposition de saisonnalité - NO2

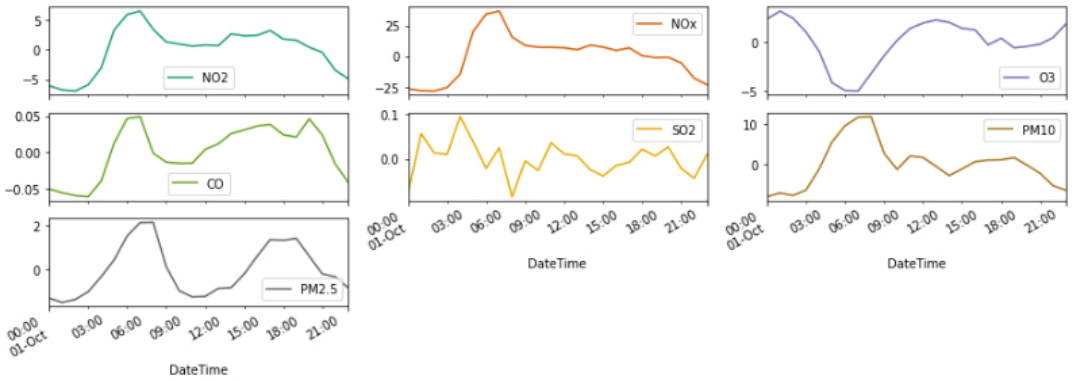

FIGURE 12 – Focus sur la serie saisonnalité

voiture ou en voiture pendant ces deux périodes, ce qui produit beaucoup de polluantion en NOx, CO mais aussi PM_s (particuls fins). On peut donc supposer que le trafic a une influence importante sur la qualité de l'air, c'est exactement le cas car nos capteurs sont installés au niveau de la rue dans les rues 'a fort trafic.

Jusqu'à présent, on a fini toutes les étapes de preprocessing de données et on peut passer au phase pour la séléction de modèle. Notre but est de trouver la meilleure combinaison des ordres et des ordres saisonnières (p, d, q)x(P, D, Q, s) pour notre modèle qui s'adapte mieux aux données.D'après la décomposition et analyse de seasonalité, on a déjà fixé que notre série présente une saisonnalité jounalière, ce qui signifique que s = 24.

4.5 Sélection de modèle

4.5.1 Autocorrélation principale et partielle & Principe parcimonial

Basé sur ACF(Fonction d'autocorrélation) et PACF(onction d'autocorrélation partielle) plot, on peut fixer (d, D) par one-lagged differencing, (p, P) par la figure PACF et (q, Q) par la figure ACF :

La première étape est de visualisez l'acf et le pacf pour voir si la structure d'autocorrélation des séries est constante.

```
# Visualize the acf and pacf to see if the autocorrelation structure of
    series is constant
acf_pacf_plot(Pol_BA_df_NO2, lags=50)
```

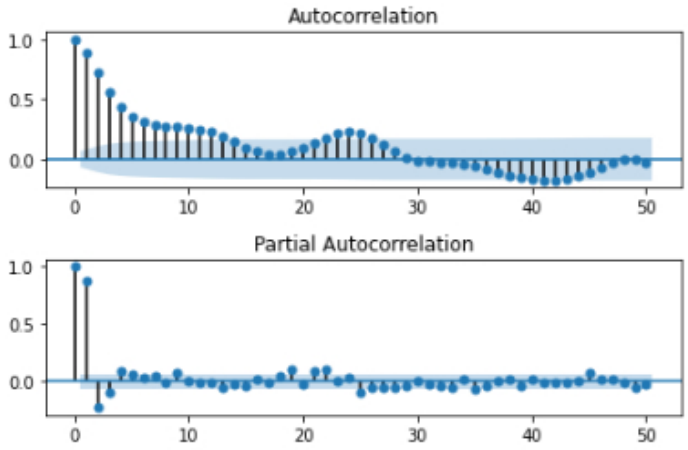

FIGURE 13 – ACF & PACF sur la saisonnalité - étape 1

Comme on peut voir dans la [FIGURE 13], ACF plot présente que les pics d'autocorrélation ne diminuent pas immédiatement jusqu'à zéro et présentent une structure d'autocorrélation variante car on n'a pas encore éliminé son seasonal pattern jusqu'à présent.

La deuxième étape est d'effectuer one-step différenciation (série de y(t) - y(t-1)) pour trouver l'ordre de différenciation d.

```
# One-step differencing to find the differencing order d
acf_pacf_plot(transform_diff_log(Pol_BA_df_NO2), lags=50)
```

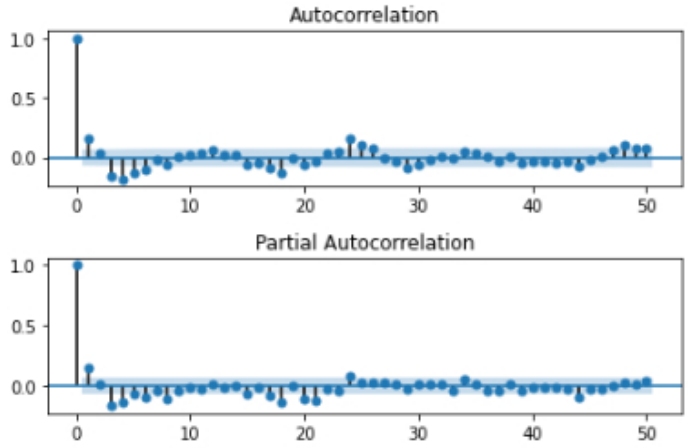

FIGURE 14 – ACF & PACF sur la saisonnalité - étape 2

Comme ce qui a été montré dans la [FIGURE 14], ACF et PACF plot présentent tous les deux une chute instantanée du pic au lag-1, de sorte qu'on n'a pas besoin de différencier davantage la série et de considérer que d pourrait être 0 ou 1.

La troisième étape consiste à appliquer la différenciation saisonnière pour trouver l'ordre de différenciation saisonnière D.

```
# Applying seasonal differencing to find the seasonal differencing order D
acf_pacf_plot(transform_diff_log(transform_diff_log(Pol_BA_df_NO2), lags=24),
    lags=50)
```

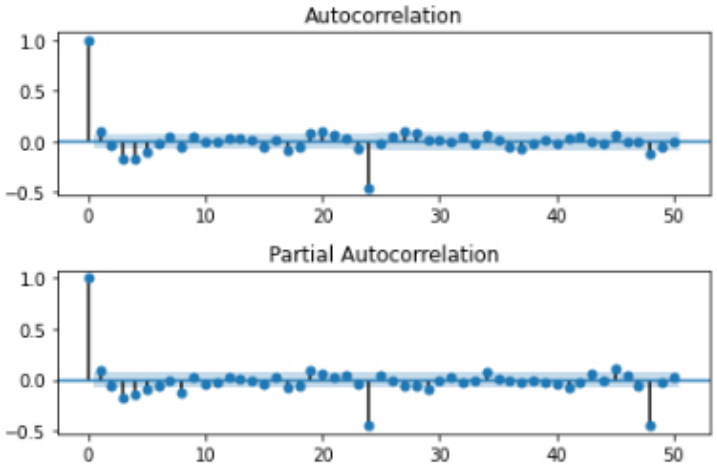

FIGURE 15 – ACF & PACF sur la saisonnalité - étape 3

On peut voir dans la [FIGURE 15], ACF et PACF ne nous donnent que les pics saisonniers à lag-24 et lag-48, on peut donc considérer que D pourrait être $[1, 2]$.

D'après ACF plot, il y a une chute instantanée au lag-1 et seasonal lag-1. on peut dire que q pourrait être $[0, 1]$ et Q pourrait être $[0, 1]$.

De plus, d'après PACF plot, il y a une chute instantanée au lag-1 et seasonal lag-1, seasonal lag-2. on peut dire que q pourrait être $[0, 1]$ et Q pourrait être $[0, 1, 2]$

Il faut toujours garder à l'esprit que notre modèle doit être parcimonieux, ce qui signifie $p + d + q + P + D + Q \leq 6$, afin de réduire la complexité et d'éviter le sur-estimation.

4.5.2 Élimination de volatilité et saisonnalité

Comme notre donnée présente encore une fluctuation dans certains points même après la normalisation (Voir FIGURE 9) et notre série garde encore son seasonal pattern (Voir FIGURE 13), il est donc indispensable de les éliminer avant d'alimenter les données au modèle. Dans notre cas, les données sont saisonnières sur 24 heures, en soustrayant la série avec une différence de 24 heures, on obtient une série plus 'plate'.

```
# Remove Increasing Volatility and Seasonality
Pol_BA_df_copy = remove_volatility_seasonality(Pol_BA_df_copy)
```

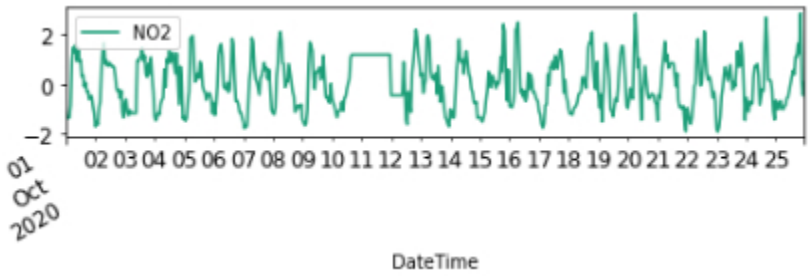

FIGURE 16 – Élimination de volatilité et saisonnalité

On peut voire que notre donnée est strictement stationnaire maintenant.

4.5.3 Fractionnement des données

Généralement, on va diviser notre données en 3 parties : ensemble d'entraînement qui sert à alimenter notre modèle, ensemble de validation qui sert à nous guider à améliorer notre modèle au cours d'entraînement, ensemble de test qui sert à évaluer la capacité de généralisation de notre modèle sur les nouvelles données. Pour notre cas, on ne fractionne notre données qu'en ensemble d'entraînement et ensemble de test.

```
# Split the stationary data into training set (before 21th Octover) and
    testing set(rest)
Pol_BA_df_NO2_train, Pol_BA_df_NO2_test =
    train_test_split(Pol_BA_df_NO2_trans, start_at='2020-10-01 00:00:00',
    end_at='2020-10-20 23:30:00')
```

4.5.4 Grid search par AIC

Pour trouver les meilleurs valeurs des paramètres pour le modèle (Hyperparameter tuning), on peut le faire de manière manuelle par des fonctions créées par nous même, soit automatiquement par la fonction proposée par la librairie Python pmdarima.

AIC (Akaike Information criterion) est une mesure de la qualité d'un modèle statistique, permet de pénaliser les modèles en fonction du nombre de paramètres afin de satisfaire le critère de parcimonie.

Basé sur les ordres (p,d,q) et les ordres saisonnières (P,D,Q) qu'on a trouvés auparavant, on peut itérer toute la combinaison d'ordre possible, entraîner plusieurs fois le modèle pour chaque combinaison, et comparer leurs AIC en choissant le modèle avec un AIC minimal.

[**Manuellement**]

```
# Get the best parameters manually
order, seasonal_order, AIC= grid_search_AIC_sarimax(Pol_BA_df_NO2, d=[0,1],
    D=[1,2], seasonal_lags=24, return_aic=True)
```

On a finalement obtenu les meilleurs paramètres $(1,0,1)x(2,1,1,24)$, qui satisfont le principe de parcimonie.En général, il est trop compliqué et risqué en utilisant la méthode manuelle pour calculer les paramètres optimisés pour SARIMAX, bien qu'on limite l'ensemble de valeurs pour certains ordres, ce calcul est assez long (plusieurs minutes).

[**Automatiquement**]

```
# Get the best parameters automatically
auto_sarimax_results = pm.auto_arima(Pol_BA_df_NO2_train, start_p=0, d=0,
    start_q=0, start_P=0, D=1, start_Q=0, max_p=1, max_q=1, max_P=3, max_Q=2,
    seasonal=True,
                # The max order of the differencing and seasonal differencing
                max_d=1, max_D=2,
                # The period for seasonal differencing
                m=24,
                # Satisfy parsimony principle
                max_order=6,
                stepwise=True, suppress_warnings=True, error_action='ignore')
order = auto_sarimax_results.order
seasonal_order = auto_sarimax_results.seasonal_order
```

Les meilleurs paramètres qu'on a trouvé sont $(1,0,0)x(3,1,0,24)$. On va continuer notre développement basé sur ces paramètres. En général, ce calcul est moins long que la manière manuelle (queleques minutes), mais il nous donnne exactement les hyperparamètres les plus adaptés.

4.6 Entraînement, Prédiction, Évaluation

4.6.1 One-step forecast & Dymamic forecast

Pour ARIMAX ou SARIMAX, il y a deux façons de faire le forecast : 'One-step forecast' est de prendre les valeurs précédentes pour prédire la valeur suivante successivement. 'Dymamic forecast' est de prévoir plusieurs valeurs à l'avance à partir de l'instant en prenant uniquement les valeurs avant t. Généralement, la méthode 'One-step forecast' intérprète mieux que la méthode 'Dymamic forecast'.

4.6.2 In-sampling prédiction & Out-of-sampling prédiction

Avec SARIMAX, on peut effectuer l'évaluation du modèle sur l'ensemble d'entraînement par la prédiction 'In-sampling' pour nous guider à ajuster les paramètres et aussi l'évaluation sur l'ensemble de test pour tester la robustesse de notre modèle final.

4.6.3 Variable endogène & Variable exogène

Avec SARIMAX, on peut choisir de prendre en compte ou pas des valeurs exogènes pour construire le modèle. Généralement, si on importe un facteur exogène, notre modèle sera plus capable d'interpréter notre donnée.

L'évalution est faite à l'aide de l'erreur quadratique moyenne (RMSE) sur une période (courte ou longue) pour mesurer l'écart entre les valeurs réelles et la prédiction.

4.6.4 Modèle sans variable exogène - SARIMA

La première étape réalisée consiste à trouver un bon modèle sans considérer de valeur exogène :

```
# Run SARIMAX model
model_SARIMAX = fit_sarimax(Pol_BA_df_NO2_train, order=order,
    seasonal_order=seasonal_order, trend='c')

# Make prediction and visualize the result
predict_plot_sarimax(model_SARIMAX, Pol_BA_df_NO2_train, start=-96)

# Out-of-sampling predictions without exogenous factor
predict_plot_sarimax(model_SARIMAX, Pol_BA_df_NO2_test,
    df_to_predict=Pol_BA_df_NO2_test, zoom=True, zoom_start_at='2020-10-21
    00:00:00', zoom_end_at='2020-10-21 23:00:00')
```

On peut constater à partir du résultat que la prédiction One-step-ahead a bien saisi les caractéristiques des données réelles, mais montre encore quelques différences de décalage temporel entre la valeur prédite et la valeur réelle.

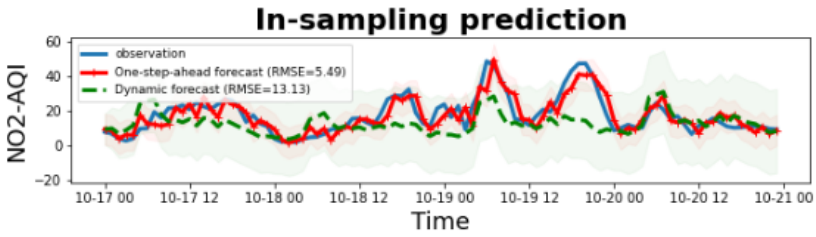

FIGURE 17 – Prédiction in-sampling sans facteur exogène - NO2

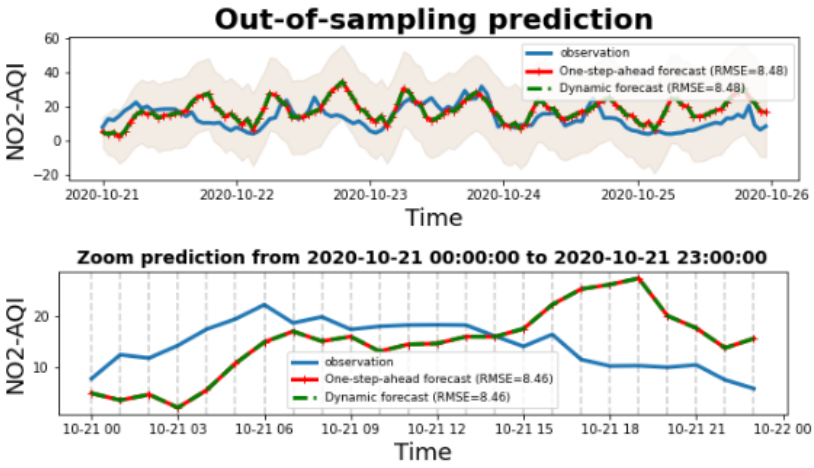

FIGURE 18 – Prédiction out-of-sampling sans facteur exogène - NO2

La prédiction dynamique pourrait bien suivre la tendance quotidienne des données réelles, la sous-estimation des pics est probablement due au fait qu'on n'a pas encore pris en compte le facteur exogène (qui suit la même tendance que la donnée prédite).

4.6.5 Analyse de corrélation

Avant d'ajouter le facteur exogène dans notre modèle, il faut qu'on examine quel facteur sera le plus corrélé avec la variable à prédire. A l'aide de l'analyse de corrélation linéaire ou non-linéaire par la méthode 'Pearson' et 'Spearman' respectivement, on peut visualiser le niveau de corrélation en affichant une carte thermique (Heatmap).

```
# Analyze the correlation
correlation_analyse(Pol_BA_df_NO2, Pol_BA_df[['NOx', 'O3', 'CO', 'SO2',
    'PM10', 'PM2.5']], figsize=(8,6))
```

La matrice d'analyse de corrélation linéaire ci-dessus montre qu'il existe une forte corrélation linéaire entre NO2 et NOx(0,93), CO(0,74) indépendamment.

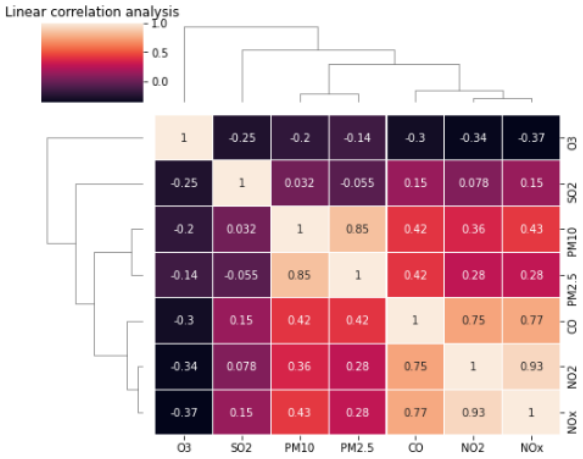

FIGURE 19 – Analyse de corrélation entre différents pollution - Heatmap

4.6.6 Modèle avec variable exogène - SARIMAX

On a donc d'abord pris NOx comme facteur exogène pour essayer de trouver un meilleur modèle, en prenant les mêmes paramètres $(1,0,0)x(3,1,0,24)$. que le modèle précédent sans facteur exogène . Avec le modèle qu'on a obtenu, on a lancé la prédition pour comparer sa performance sur le même période :

```
# Run SARIMAX model by considering NOx as exogenous factor
model_SARIMAX_exo_NOx = fit_sarimax(Pol_BA_df_NO2_train,
    exog=Pol_BA_df_NOx_train, order=order, seasonal_order=seasonal_order,
    trend='c')

# Make in-sampling prediction
prediction_plot_sarimax(df_original=Pol_BA_df['NO2'].loc['2020-10-17
    00:00:00':'2020-10-20 23:00:00'], df_forecast=[prediction_osa_1,
    prediction_dyn_1], figsize=(10,2))

# Make out-of-sampling prediction and zoom the result
prediction_plot_sarimax(df_original=Pol_BA_df['NO2'].loc['2020-10-21
    00:00:00':'2020-10-25 23:00:00'], df_forecast=prediction_dyn_out_1,
    figsize=(10,2), zoom=True, zoom_start_at='2020-10-21 00:00:00',
    zoom_end_at='2020-10-21 23:00:00')
```

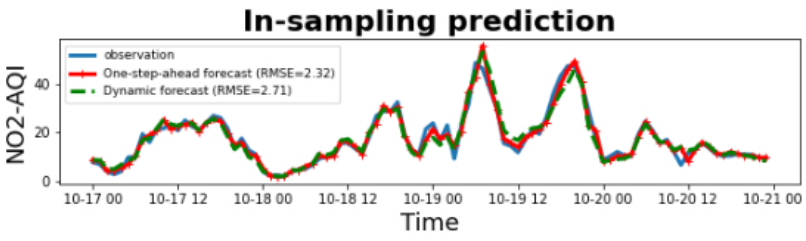

FIGURE 20 – Prédiction in-sampling avec facteur exogène NOx - NO2

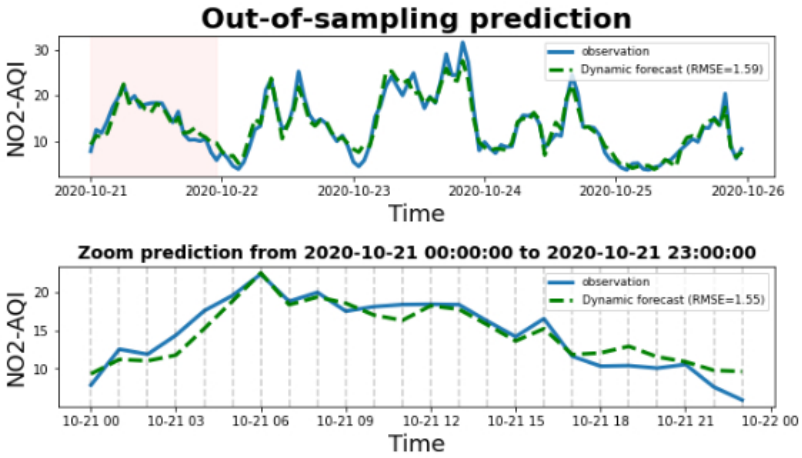

FIGURE 21 – Prédiction out-of-sampling sans facteur exogène NOx - NO2

On peut constater que le modèle qu'on a obtenu pour NO2 suit mieux la tendance, la fluctuation et les pics sur une longue période, que le modèle sans prise en compte de facteur exogène.

On a ensuite pris CO comme facteurs exogènes en faisant la même chose que pour NO2 :

```
# Run SARIMAX model by considering CO as exogenous factor
model_SARIMAX_exo_CO = fit_sarimax(Pol_BA_df_NO2_train,
    exog=Pol_BA_df_CO_train, order=order, seasonal_order=seasonal_order,
    trend='c')

# Make in-sampling prediction
prediction_plot_sarimax(df_original=Pol_BA_df['NO2'].loc['2020-10-17
    00:00:00':'2020-10-20 23:00:00'], df_forecast=[prediction_osa_2,
    prediction_dyn_2], figsize=(10,2))

# Make out-of-sampling prediction and visualize the result (from 21th
    December until 26th December)
prediction_dyn_out_2 = predict_sarimax(model=model_SARIMAX_exo_CO,
    start_at_out_sampling='2020-10-21 00:00:00',
    end_at_out_sampling='2020-10-25 23:00:00', exog=Pol_BA_df_CO_test)
```

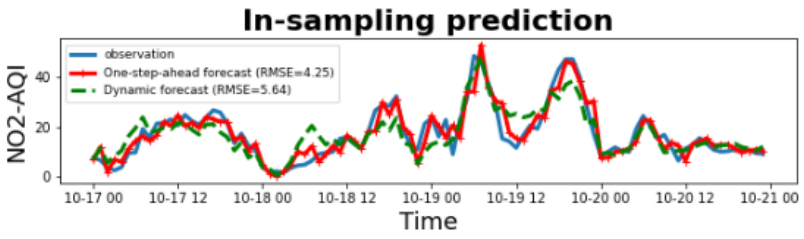

FIGURE 22 – Prédiction in-sampling avec facteur exogène CO - NO2

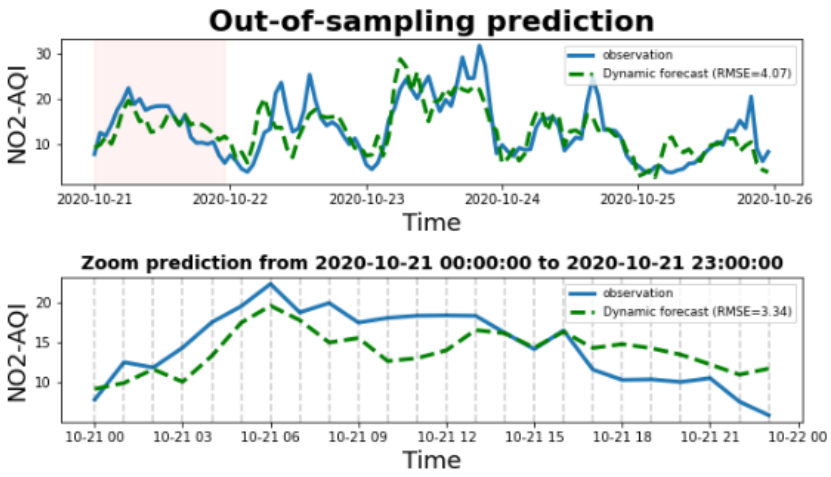

FIGURE 23 – Prédiction in-sampling avec facteur exogène CO - NO2

On peut constater que le modèle fonctionne moins bien si on prend le CO comme facteur exogène au lieu de NOx après avoir normalisé les données. Jusqu'à maintenant, on a bien trouvé un bon modèle pour prédire NO2 et on a aussi bien exploré la magie de SARIMAX pour la prédiction de série temporelle.

Comme les NOx et le CO influencent tous les deux NO2, il est intéressant pour nous de les prendre en même temps et de voir leur impact ensemble. La limite pour SARIMAX est qu'on ne peut que prendre un seul facteur endogène et un seul facteur exogène au maximum chaque fois. Imagine si on rencontre un cas où on veut prédire l'impact de variable exogène (un ou plusieurs) sur variables endogènes (un ou plusieurs) en même temps, SARIMAX ne s'adaptera plus. Mais grâce au VAR, on peut réaliser notre but.

5 Mise en œuvre de modèle multivarié- VAR

5.1 Introduction

VAR est un algorithme multivariée qui peut être utilisé lorsque deux ou plusieurs séries temporelles s'influencent mutuellement et que cette relation est bidirectionnelle.

5.1.1 Principe mathématique

Dans le modèle VAR, chaque variable est modélisée comme une combinaison linéaire de ses propres valeurs passées et des valeurs passées d'autres variables du système, ce qui signifie qu'on peut prédire la série avec ses propres valeurs passées ainsi que d'autres séries du système. On a autant d'équations que de variables dans le système : si vous avez 5 séries temporelles qui s'influencent mutuellement, on aura un système de 5 équations.

Pour un modèle VAR bivariant d'ordre 1 / VAR(1) sans facteur exogène, on peut construire un système ci-desssous :

$$y_{1,t} = \alpha_{10} + \beta_{11} * y_{1,t-1} + \beta_{12} * y_{2,t-1} + \epsilon_{1,t}$$

$$y_{2,t} = \alpha_{20} + \beta_{21} * y_{1,t-1} + \beta_{22} * y_{2,t-1} + \epsilon_{2,t}$$

Ce système peut être écrit en forme matricielle :

$$y_t = \alpha + \beta * y_{t-1} + \epsilon_t$$

avec $y_t = \begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix}$, $\alpha = \begin{bmatrix} \alpha_{10} \\ \alpha_{20} \end{bmatrix}$, $\beta = \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{bmatrix}$, $y_{t-1} = \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \end{bmatrix}$, $\epsilon_t = \begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix}$

Pour un modèle VAR bivariant d'ordre 2 / VAR(2) sans facteur exogène :

$$y_{1,t} = \alpha_1 + \beta_{11} * y_{1,t-1} + \beta_{12} * y_{2,t-1} + \gamma_{11} * y_{1,t-2} + \gamma_{12} * y_{2,t-2} + \epsilon_{1,t}$$

$$y_{2,t} = \alpha_2 + \beta_{21} * y_{1,t-1} + \beta_{22} * y_{2,t-1} + \gamma_{21} * y_{1,t-2} + \gamma_{22} * y_{2,t-2} + \epsilon_{2,t}$$

Ce système peut être écrit en forme matricielle :

$$y_t = \alpha + \beta * y_{t-1} + \gamma * y_{t-2} + \epsilon_t$$

avec $y_t = \begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix}$, $\alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}$, $\beta = \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{bmatrix}$, $y_{t-1} = \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \end{bmatrix}$, $\gamma = \begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix}$,
$y_{t-2} = \begin{bmatrix} y_{1,t-2} \\ y_{2,t-2} \end{bmatrix}$, $\epsilon_t = \begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix}$

Pour un VAR(1) avec un seul facteur exogène(on peut aussi prendre plusieurs), le système est devenu :

$$y_t = \alpha + \beta * y_{t-1} + \theta * x_t + \epsilon_t$$

avec $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$, $\epsilon_t = \begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix}$

On peut voir que chaque facteur (variable de série temporelle) devient un facteur endogène dans le système, c'est-à-dire que toutes les variables sont endogènes au niveau de l'équation. Chaque coefficient pouvant être estimé par la méthode de moindre carrée (OLS) équation par équation. De la même manière, on peut construire le système VAR mathématique qui contient un nombre quelconque de variables endogènes pour n'importe quel ordre, avec un nombre quelconque de variables exogènes.

De plus, VAR est en "Prédiction Dynamique", le système utilisera ses propres prédictions pour effectuer de nouvelles prédictions[4].

5.1.2 Malédiction de la dimension & Choix d'ordre

Pour un modèle VAR d'ordre p / VAR(p), un grand p entraînera la perte de degrés de liberté, un petit p d'ordre fera que l'autocorrélation des termes d'erreur apparemment significatives et conduira à des sous-estimations. Pour un VAR(p) avec g variables, on aura au moins $g * (g * p + 1)$ paramètres. Il est donc indispensable de choisir un bon longueur du décalage ou un bon ordre VAR afin de trouver la balance entre la complexité et la capacité de généralisation du modèle.

On peut trouver le bon ordre en minimisant la valeur du principe AIC (Akaike Information Criteria) ou SIC(Schwarz Information Criteria)) :

$$MAIC = In(|\hat{E}|) + 2 * K/(T - P)$$

$$MSIC = In(|\hat{E}|) + [K/(T - P)] * In(T - P)$$

avec K le nombre de coefficients à estimer dans toutes les équations dans VAR, $T - P$ est nombre d'observations totales utilisées pour l'estimation et $|\hat{E}$ est la covariance.

Ayant ces bases théoriques, on peut commencer à voir comment mettre en œuvre le module VAR. Il faut toujours tester la stationnarité de série temporelle en priorité, on a déjà vérifié que toutes les séries temporelles NO2, NOx et CO sont bien stationnaires (voir section 4.1).

5.2 Test de causalité de Granger

Lorsqu'un VAR comporte de nombreux décalages temporelles de variables, il sera difficile de voir quel ensemble de variables ont des effets significatifs sur chaque variable dépendante et lesquels n'en ont pas.

Les tests de causalité de Granger visent à répondre à des questions telles que "Le changement en y1 entraîne-t-il des changements en y2 ?". Si c'est le cas, tous les termes de y1 devraient être significatifs dans l'équation y2, on dit alors y1 "Granger-Causes" y2.

On prend l'example de VAR(2) cité précédemment, si y1 ne "Granger-Causes" pas y2, alors les coeffients β_{21}, γ_{21} sont tous nulls, alors le système est devenu

$$y_{1,t} = \alpha_1 + \beta_{11} * y_{1,t-1} + \beta_{12} * y_{2,t-1} + \gamma_{11} * y_{1,t-2} + \gamma_{12} * y_{2,t-2} + \epsilon_{1,t}$$

$$y_{2,t} = \alpha_2 + \beta_{22} * y_{2,t-1} + \gamma_{22} * y_{2,t-2} + \epsilon_{2,t}$$

avec $y_{2,t}$ est un processus AR(2), aucun intérêt de prédire $y_{2,t}$ par VAR. Donc, il est très important d'assurer que les séries temporelles sont en "Granger-Causes" interchangeable.

Cette évaluation est réalisée en supposant que l'hypothèse nulle est que y_1 ne cause pas l'autre variable y_2. Avec les p-values retournés, si la valeur est inférieure au niveau de signification à 0,05, on peut sans risque rejeter l'hypothèse nulle et conclure que le y_1 cause y_2.

```
# Check for causality by returning p-value table
grangers_causation_matrix(Pol_BA_df_trans[['NO2', 'NOx', 'CO']])
```

	NO2_x	NOx_x	CO_x
NO2_y	1.0000	0.0262	0.0607
NOx_y	0.0004	1.0000	0.0409
CO_y	0.0000	0.0000	1.0000

FIGURE 24 – Granger causality test- (NO2, NOx, CO)

Les lignes dans le tableau ci-dessus sont la variable de réponse, les colonnes sont des prédicteurs et les valeurs sont les p-values. On peut voir que la p-value de couple (NO2_x, NOx_y) est inférieure au niveau de signification de 0.05, ce qui nous permet de dire que NO2_x "Granger cause" NOx_y. Il en va de même pour les autres combinaisons sauf (CO_x, NO2_y), car la valeur dépasse légèrement à 0.05. Mais on condidère quand même que CO "Granger cause" NO2, on peut donc conclure que les trois séries temporelles sont "Granger cause" l'une de l'autre de manière interchangeable.

On va maintenant sélectionner le bon ordre (p) du modèle VAR en ajustant itérativement les ordres croissants du modèle VAR et en choisissant l'ordre qui donne un modèle avec une valeur d'AIC minimum.

5.3 Sélection d'ordre

Cette fois-ci, on prend NO2 et CO comme deux facteurs endogènes et NOx comme un facteur exogène, on recherche l'ordre p par la fonction select_order, proposée par VAR :

```
# Initialize VAR model
model_VAR = VAR(endog=df_train[['NO2','CO']], exog=df_train['NOx'])

# Select the best order iteratively
order_VAR = model_VAR.select_order(maxlags=12)

# Output the result
order_VAR.summary()
```

VAR Order Selection (* highlights the minimums)

	AIC	BIC	FPE	HQIC
0	-3.294	-3.258	0.03712	-3.280
1	-3.753	-3.682*	0.02346	-3.725
2	-3.771	-3.664	0.02303	-3.729*
3	-3.757	-3.616	0.02334	-3.702
4	-3.743	-3.566	0.02367	-3.674
5	-3.748	-3.535	0.02356	-3.664
6	-3.762	-3.514	0.02324	-3.664
7	-3.785*	-3.501	0.02271*	-3.673
8	-3.780	-3.460	0.02283	-3.654
9	-3.767	-3.412	0.02313	-3.627
10	-3.761	-3.371	0.02326	-3.608
11	-3.754	-3.328	0.02343	-3.586
12	-3.754	-3.293	0.02344	-3.572

FIGURE 25 – Séléction d'ordre - VAR

Dans la [FIGURE 27] , les chiffres indiqués en étoile sont le minimum pour chaque critère (AIC, BIC, FPE, HQIC) mais on n'utilise que AIC pour notre cas. On voit que la valeur AIC commence à diminuer à partir de lag-0 jusqu'au décalage 2, et puis augmente au décalage 3. Pour réduire la complexité du système, on prend le décalage 2 au lieu du décalage 7 avec le mimimum AIC, donc le meilleur ordre pour le modèle est égale à 2.

On peut aussi se donner la formule mathématique du système pour un modèle VAR bivariant (NO2, CO) d'ordre 2 / VAR(2) avec un seul facteur exogène (NOx) :

$$NO2_t = \alpha_1 + \beta_{11} * NO2_{t-1} + \beta_{12} * CO_{t-1} + \gamma_{11} * NO2_{t-2} + \gamma_{12} * CO_{t-2} + \theta_1 * NOx_t + \epsilon_{1,t}$$

$$CO_t = \alpha_2 + \beta_{21} * CO_{t-1} + \beta_{22} * NO2_{t-1} + \gamma_{21} * CO_{t-2} + \gamma_{22} * NO2_{t-2} + \theta_2 * NOx_t + \epsilon_{2,t}$$

Notre objectif est de determiner tous les coefficients. Avec l'ordre obtenu, on passe à l'étape suivante pour entraîner le modèle.

5.4 Entraînement

```
# Fitting the model with order p=2
model_VAR_1 = model_VAR_1.fit(maxlags=2, method='ols', ic='aic', verbose=True)
```

```
# Print the model coeffients details
model_VAR_1.summary()
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Thu, 07, Jan, 2021
Time:                     15:56:06
--------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                   -3.68200
Nobs:                     478.000    HQIC:                  -3.74552
Log likelihood:          -439.491    FPE:                  0.0226710
AIC:                     -3.78667    Det(Omega_mle):       0.0221124
--------------------------------------------------------------------
Results for equation NO2
==============================================================================
            coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const          0.002669         0.012588            0.212           0.832
exog0          0.823328         0.021264           38.719           0.000
L1.NO2         0.177479         0.033169            5.351           0.000
L1.CO          0.030142         0.022446            1.343           0.179
L2.NO2        -0.043887         0.027368           -1.604           0.109
==============================================================================

Results for equation CO
==============================================================================
            coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const          0.008630         0.022619            0.382           0.703
exog0          0.007613         0.025073            0.304           0.761
L1.NO2         0.661237         0.042354           15.612           0.000
L1.CO         -0.178857         0.066065           -2.707           0.007
L2.NO2         0.454080         0.044707           10.157           0.000
==============================================================================

Correlation matrix of residuals
            NO2        CO
NO2    1.000000  0.132499
CO     0.132499  1.000000
```

FIGURE 26 – Résultat de la regression - NO2,NOx,CO

En appliquant la méthode OLS sur chaque équation, le résultat retourné nous permet de retrouver la valeur des paramètres. On n'attache de l'importance que sur la colonne 'prob' qui est basée sur loi de student, qui sert à ici tester si les prédicteurs sont corrélés avec la variable à expliquer (Hypothèse nulle est qu'il y a aucune corrélation). Si la valeur 'prob' (p-value) est inférieur à 0.05, on peut rejeter l'hypothèse nulle en disant que cette variable est significative pour la détermination de variable à expliquer.

Pour variable NO2 : $\alpha_1 = const = 0, \theta_1 = exog0 = 0.823, \beta_{11} = L1.NO2 = 0.177, \beta_{12} = L1.CO = 0, \gamma_{11} = L2.NO2 = 0, \gamma_{12} = 0$. Donc la formule finale pour $NO2$ est devenue :

$$NO2_t = 0.177 * NO2_{t-1} + 0.823 * NOx_t$$

Cela signifie $NO2$ est déterminé par la valeur en décalage 1 d'elle-même et la valeur actuelle de la variable exogène NOx, ayant aucune relation avec CO même si on a supposé que CO "Granger cause" NO2.

5.5 Corrélation des résidus

Il est important de vérifier la corrélation des résidus (erreurs), en s'assurant que le modèle est suffisamment capable d'expliquer les variances et les patterns des séries temporelles. Si jamais il existe des corrélations de résidus, cela signifie qu'il y a encore des patterns qui doivent encore être expliqués par le modèle et on peut effectuer trois actions pour résoudre ce problème : incrémentation de l'ordre, réduction des prédicteurs, inclusion plus de variables exogènes. Heureusement, le tableau "Matrice de corrélation des résidus" dans [FIGURE 26] montre qu'il n'y a pas de forte corrélation parmi les résidus.

5.6 Prédiction & Évaluation

Comme ce qu'on a fait avec SARIMAX, on fait la prédiction toujours de NO2 sur la même période pour faciliter la comparaison entre SARIMAX et VAR :

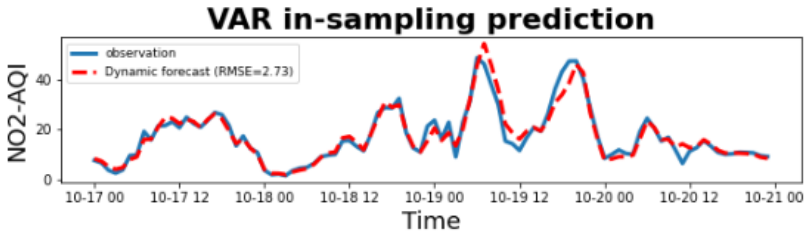

FIGURE 27 – Prédiction in-sampling - NO2,NOx,CO

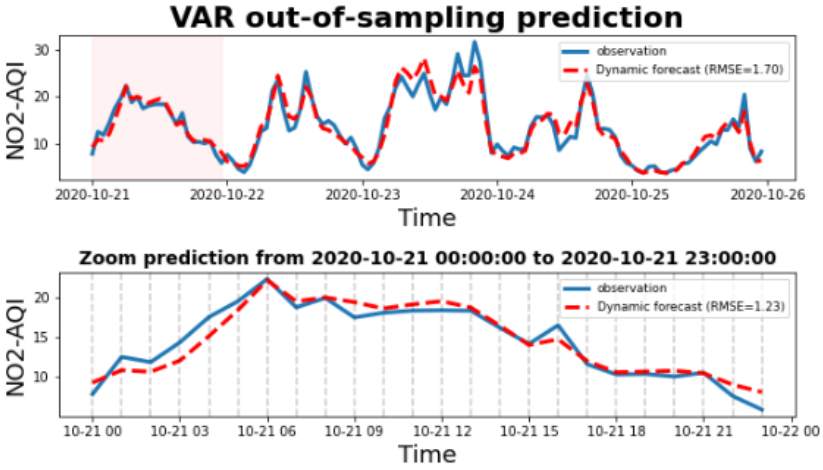

FIGURE 28 – Prédiction out-of-sampling - NO2,NOx,CO

Basé sur le résultat de prédiction [FIGURE 27 et FIGURE 28], on constate que le modèle VAR est moins bon sur l'ensemble d'entraînement que SARIMAX mais il présente une meilleure capacité de généralisation avec meilleure performance sur l'ensemble de test. De plus, l'avantage du modèle VAR est évident : plus simple, plus rapide et permet de prévoir plusieurs séries temporelles en même temps.

5.7 Intégration de données météo

Vu que le facteur de météo comme le vent, aura peut-être un impact sur la concentration de NO2, ça nous intéresse aussi de prendre ce facteur en compte, en le mettant comme un autre facteur exogène pour essayer de trouver un meilleur modèle de prédiction pour NO2.

5.7.1 Analyse des corrélations & traitement de données

On a d'abord analysé la corrélation parmi les variables NO2, NOx, CO, VH (vitesse du vent) :

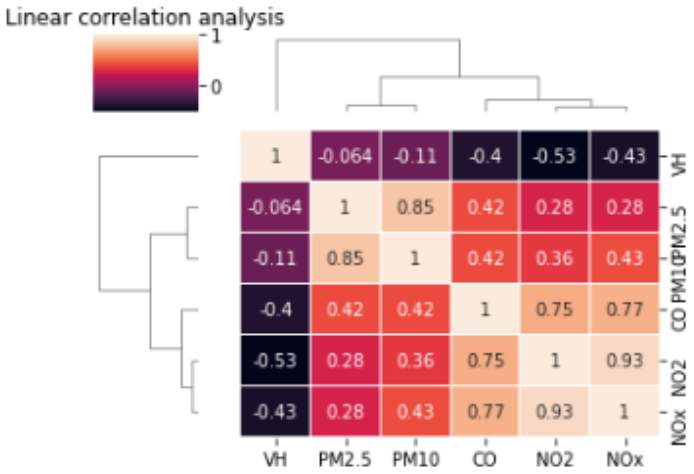

FIGURE 29 – Analyse de corrélation - NO2,NOx,CO,VH

On peut voir que la vitesse du vent (VH) a une influence négative relativement forte (-0.53) sur NO2(C'est logique car plus de vent, moins de pollution), ce qui montre qu'on peut continuer notre recherche. On a donc traité les données de météo comme ce qu'on a fait auparavant pour les données de pollution (voir section 2.3 - section 4.5.2) et finalement on obtient une série stationnaire :

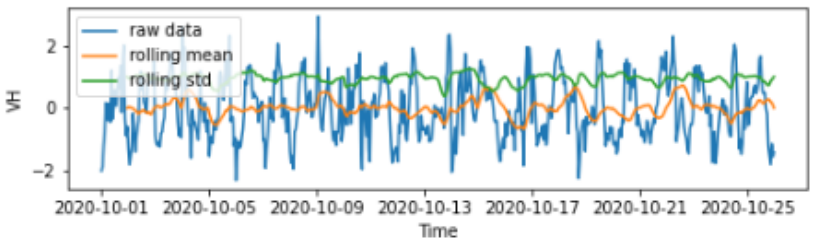

FIGURE 30 – Vitesse du vent - VH

5.7.2 Entraînement de modèle

En suivant la même procédure que précédemment, on va d'abord sélectionner le meilleur ordre, entraîner le modèle avec l'ordre obtenu p et afficher le résultat en résolvant le système par la méthode OLS (minimisation de l'erreur par le moindre carrés).

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Sun, 03, Jan, 2021
Time:                      15:41:31
----------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                   -3.71682
Nobs:                     598.000    HQIC:                  -3.77963
Log likelihood:          -540.966    FPE:                  0.0219348
AIC:                     -3.81968    Det(Omega_mle):       0.0214302
----------------------------------------------------------------------
Results for equation NO2
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const          0.003354         0.010564            0.317           0.751
exog0          0.827984         0.017569           47.128           0.000
exog1         -0.100450         0.011048           -9.092           0.000
L1.NO2         0.132854         0.027389            4.851           0.000
L1.CO          0.037148         0.018235            2.037           0.042
==============================================================================

Results for equation CO
==============================================================================
             coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const         -0.022441         0.022152           -1.013           0.311
exog0          0.008826         0.018322            0.482           0.630
exog1          0.004383         0.023521            0.186           0.852
L1.NO2         0.655367         0.039118           16.754           0.000
L1.CO          0.001965         0.024600            0.080           0.936
==============================================================================

Correlation matrix of residuals
            NO2        CO
NO2    1.000000  0.145192
CO     0.145192  1.000000
```

FIGURE 31 – Résultat de la regression - NO2,NOx,CO,VH

Basé sur ce résultat, on peut obtenir la formule mathématique pour NO2 :

$$NO2_t = 0.828 * NOx_t - 0.1 * VH_t + 0.133 * NO2_{t-1} + 0.037 * CO_{t-1}$$

Comme le coefficient de VH est égale à -0.1, cela montre que VH a un impact négatif sur NO2. Cette fois-ci, NO2 dépend non seulement la valeur en décalage 1 d'elle-même et la valeur actuelle de la variable exogène NOx, mais aussi la valeur de CO en décalage 1. De plus, les résidus ne sont pas corrélées, on a obtenu un bon modèle et on passe à la prédiction et l'évaluation du modèle.

5.7.3 Prédiction & Évaluation

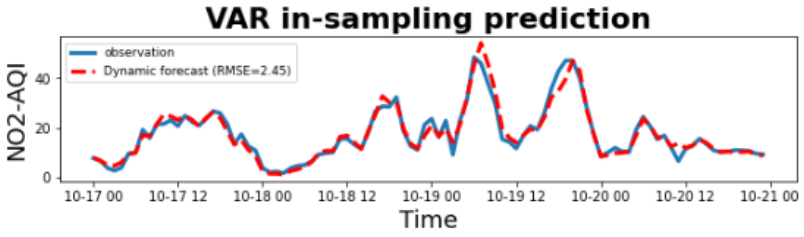

FIGURE 32 – Prédiction in-sampling - NO2,NOx,CO,VH

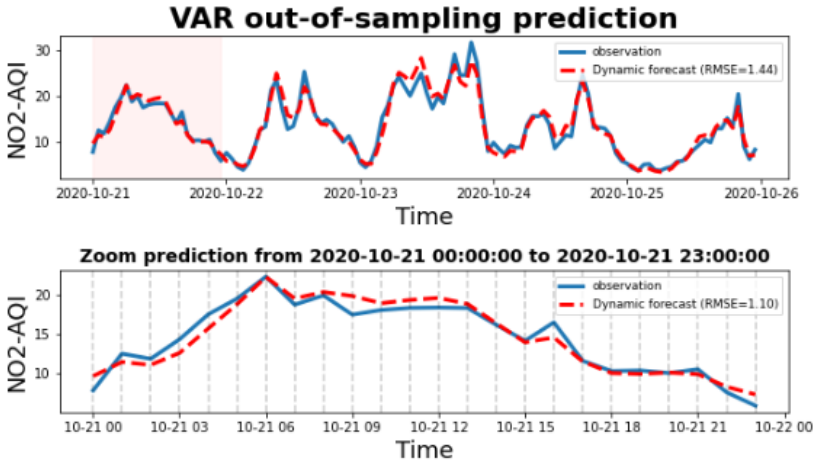

FIGURE 33 – Prédiction out-of-sampling - NO2,NOx,CO,VH

D'après le résultat de la prédiction, on peut voir qu'après avoir ajouté VH comme autre variable exogène, la performance du modèle VAR s'améliore pendant une longue période avec une erreur de prédiction la plus faible, ce qui est donc le meilleur modèle qu'on a pu trouver pour NO2.

Tout ce qui a été développé jusqu'ici concerne la prédiction sur un capteur (NO2), avec d'autres capteurs situés sur la même station locale. Ce qui nous intéresse maintenant est de prendre plusieurs capteurs en même temps pour répondre à la question 'Lorsqu'un capteur qui mesure la concentration de NO2 est tombé en panne, peut-on prédire quand même sur la base des données passées d'autres capteurs situés sur d'autres stations à proximité dans la ville ?'.

5.8 Prédiction inter-stations

On a déjà fait de la prédiction 'inter-capteurs' avant sur (NO2, NOx, CO, VH) mais pas inter-stations. On va maintenant essayer de prédire NO2 de capteur de pollution 1 (situé à la station 1) en prenant les données NO2 de capteur de pollution 2 (situé à la station 2) et de capteur pollution 3 (situé à la station 3). Comme ce qu'on a fait auparavant pour les données de pollution et de météo, on obtient tout d'abord de séries temporelles stationnaires :

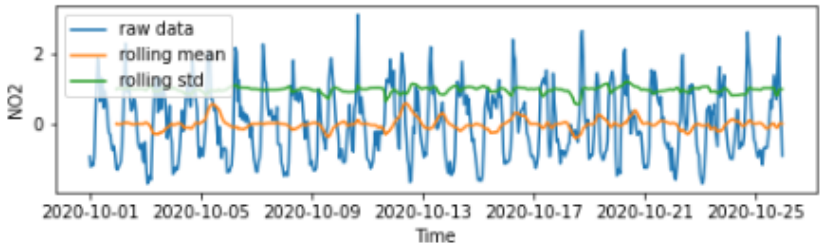

FIGURE 34 – NO2 - Capteur pollution 2

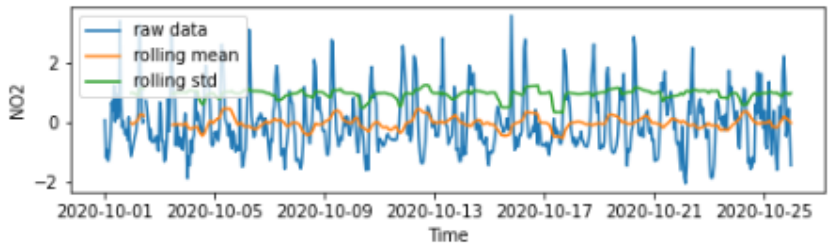

FIGURE 35 – NO2 - Capteur pollution 3

Vu que toutes les séries NO2 des trois capteurs sont stationnaires, on a aussi analysé la causalité parmi ces trois séries. Le résultat montre qu'ils 'Granger cause' l'une de l'autre de manière interchangeable.

	Senor1_NO2_x	Senor2_NO2_x	Senor3_NO2_x
Senor1_NO2_y	1.0	0.0005	0.0
Senor2_NO2_y	0.0	1.0000	0.0
Senor3_NO2_y	0.0	0.0000	1.0

FIGURE 36 – Granger causality NO2 - trois capteurs

On passe à l'étape pour la recherche de l'ordre, l'entraînement du modèles VAR et on obtient finalement lé resultat suivant :

$\text{Sensor1\_NO2}_t = 0.855 * Sensor1\_NO2_{t-1} - 0.254 * Sensor3\_NO2_{t-1} - 0.157 * Sensor1\_NO2_{t-2} - 0.125 * Sensor3\_NO2_{t-3}$

Cette formule nous a montré que la concentration NO2 du capteur pollution 1 est déterminé par l'enregistrement passé (décalage 1 et décalage 3) du capteur pollution 3 et

l'enregistrement passé (décalage 1 et décalage 2) de lui-même. Donc on a bien trouvé la solution pour résoudre le problème où le capteur 1 est tombé en panne brusquement, il est encore possible de prédire sa valeur NO2 en utilisant les données du capteur 3. On va donc passer à la prédiction et l'évaluation :

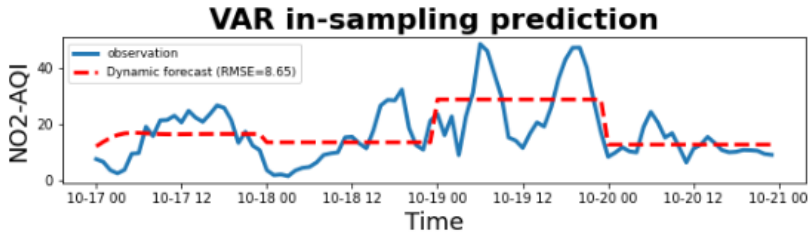

FIGURE 37 – Prédiction in-sampling NO2 - Intercapteurs

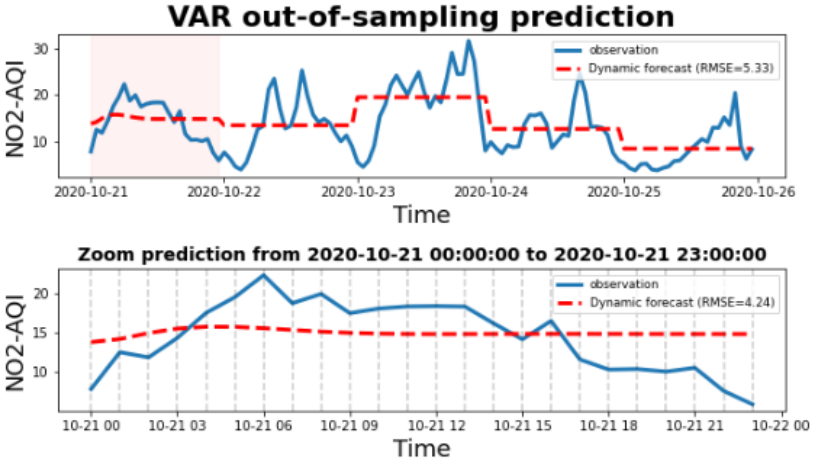

FIGURE 38 – Prédiction out-of-sampling NO2 - Intercapteurs

D'après les [FIGURE 37 et FIGURE 38], on peut voir que la courbe a bien capturé le changement de la moyenne journalière mais n'arrive pas à expliquer les fluctuations dans le temps. On peut supposer que les fluctuations viennent de l'impact des autres facteurs (CO, NOx, VH).

5.9 Enchainement des modèles

On va maintenant utiliser le résultat de prédiction qu'on vient d'obtenir en le mettant comme données d'entrée pour le modèle qu'on a entrainé auparavant en fonction de quatre facteurs (NO2,NOx,CO,VH), pour vérifier si on peut enchaîner ces deux modèles (modèle obtenu à Section 5.7.3 et à Section 5.8) et améliorer le résultat de prédiction. De plus, on peut aussi créer un pipeline en enchaînant toutes les étapes nécessaires pour aboutir à la prédiction de polluant NO2.

Après la concaténation de données, le fractionnement de données, la sélection d'un bon ordre, l'entraînement de modèle et la vérification de corrélation de résidus, on arrive à obtenir finalement un bon résultat de prédiction suivante :

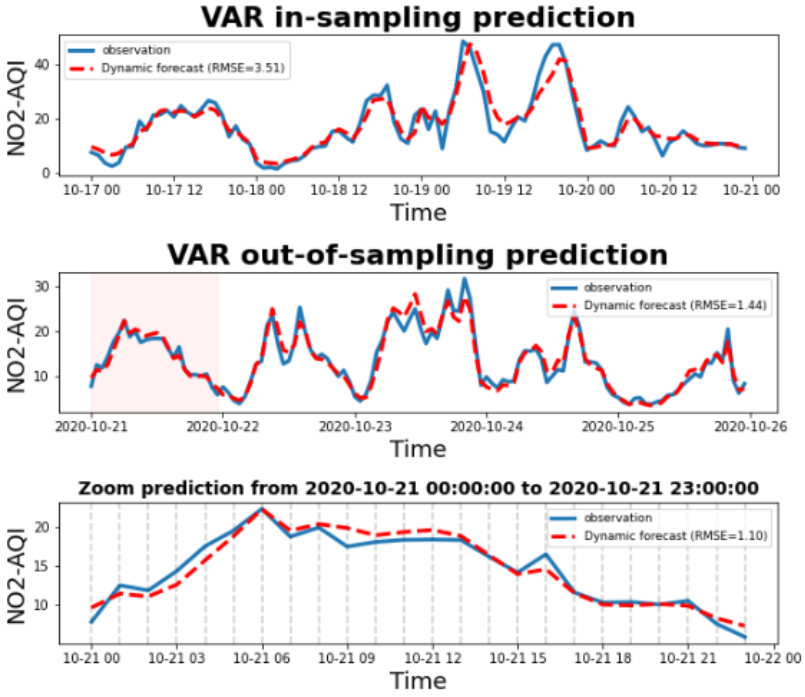

FIGURE 39 – Prédiction du modèle enchaîné - NO2,NOx,CO,VH

C'est surprenant qu'on retrouve un bon résultat de prédiction aussi bon qu'on a pu trouver auparavant en prenant quatre facteurs (NO2,NOx,CO,VH) en même temps (voir section 5.7.3). Par comparaison, on peut constater que la performance des modèles enchaînés sur l'ensemble d'entraînement est moins bonne, mais sa performance sur l'ensemble de test est aussi bonne, on peut conclure que les modèles enchaînés a une meilleure capacité de généralisation.

6 Tableau de bord

Jusqu'à ici, on a bien réussi à réaliser la prédiction par l'enchaînement des modèles. De même façons, on peut réussir à trouver pour chaque polluant un meilleur modèle basé sur toutes les données d'observation par SARIMAX ou VAR, et finalement on peut créer un tableau de bord pour afficher le résultat de prédiction pour tous les polluants (Voir FIGURE 40).

```
# Display the Dashbord
Air_Situation_Dashbord(Pol_BA, start_time=pd.to_datetime("2020-10-01
    00:00:00"), end_time=pd.to_datetime("2020-10-05 23:30:00"),
    figsize=(10,1), width=0.03, min_max_distance=0.3)
```

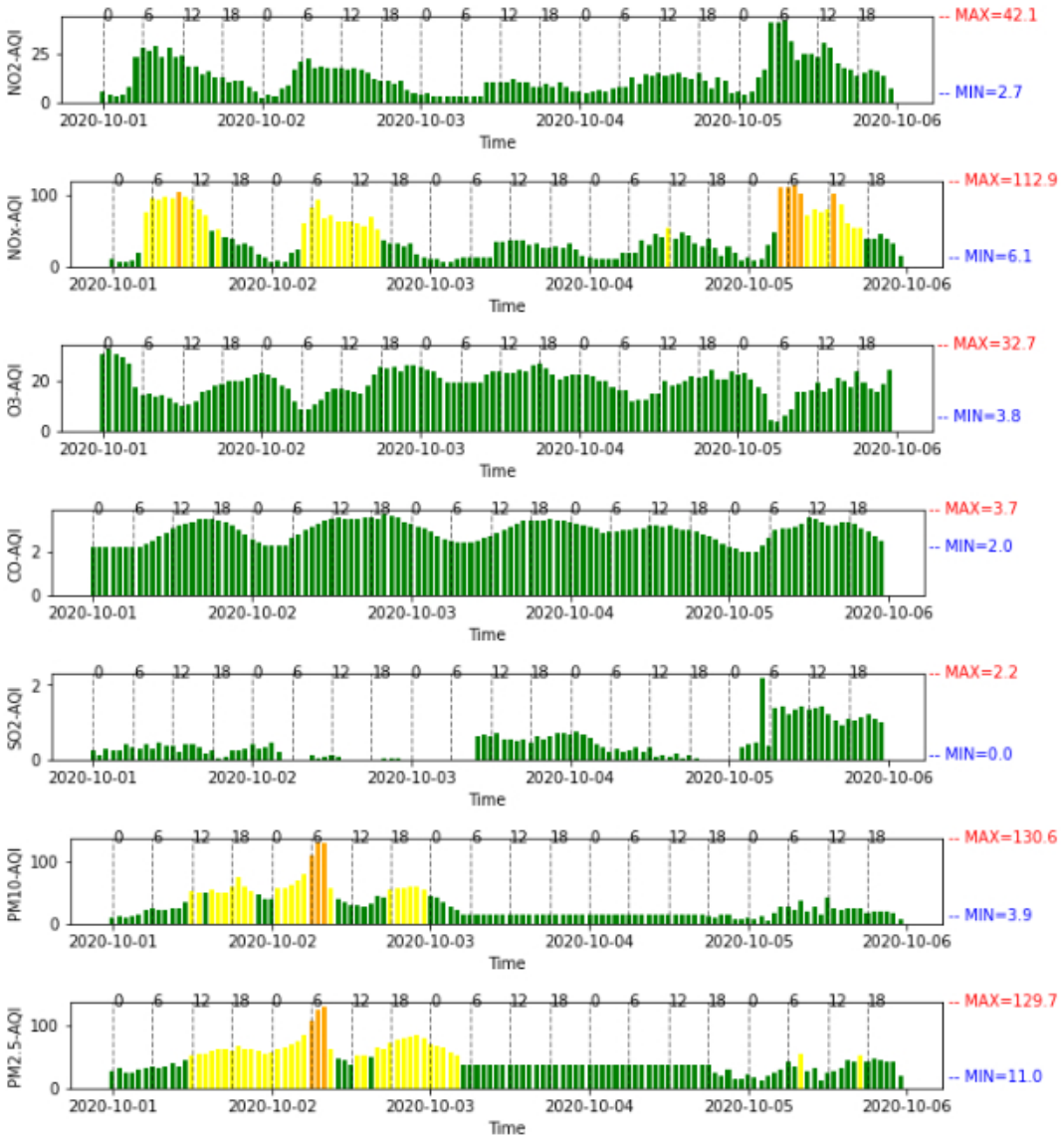

FIGURE 40 – Dashboard - Visualisation de la qualité de l'air mesurée en couleur

La dernière chose qui reste est de prédire le niveau d'alerte de pollution afin qu'on puisse rendre notre modèle opérationnel.

7 Prédire les niveaux d'alerte

Le niveau d'alerte est un seuil qui mesure si la concentration de polluant dépasse une valeur nuit à la santé de l'humain. Il y a de niveaux d'alerte définis officiellement[5], mais pour notre cas, on a définit nous-mêmes trois niveaux d'alerte différents pour NO2 :

Niveau 1 : la moyenne horaire glissante (en indiquant la taille de fenêtre) dépasse un "seuil_1 (en AQI)" prédéfini.
Niveau 2 : la moyenne horaire dépasse un "seuil_2 (en AQI)" pendant des trois heures consécutives.
Niveau 3 : la moyenne horaire dépasse un "Seuil_3 (en AQI)".

Basé sur ces trois niveaux définis, c'est possible pour nous de prendre le résultat de prédiction qu'on a obtenu et prédire à quel moment, la concentration de polluant arrive en un état dangereux (Voir FIGURE 41).

```
# Produire le resultat de la prediction d''alerte en donnant trois seuils
forecast_warning(prediction_final['NO2'], level_1=20, level_2=20, level_3=28)
```

```
                Summary of Warning Forecasting Results
========================================================================
Level 1(AQI) - Hourly Rolling(window=3) Average Exceeds 20
Level 2(AQI) - Hourly Average Exceeds 20 for 3 consecutive hours
Level 3(AQI) - Hourly Average Exceeds 28
------------------------------------------------------------------------
Results for Level_1
========================================================================

Time: 2020-10-21 07:00:00       Value: 19.570659032967388
Time: 2020-10-21 08:00:00       Value: 20.39020297585575
Time: 2020-10-22 10:00:00       Value: 20.60967632371122
Time: 2020-10-22 11:00:00       Value: 15.845978106952025
Time: 2020-10-23 07:00:00       Value: 25.09274357686707
Time: 2020-10-23 08:00:00       Value: 24.304509501484077
Time: 2020-10-23 09:00:00       Value: 23.17488834892463
Time: 2020-10-23 10:00:00       Value: 25.5070190062735
Time: 2020-10-23 11:00:00       Value: 28.24532554225462
Time: 2020-10-23 12:00:00       Value: 23.976792747239934
Time: 2020-10-23 13:00:00       Value: 19.943152447808963
Time: 2020-10-23 14:00:00       Value: 20.56206131013247
Time: 2020-10-23 16:00:00       Value: 23.481287258266935
Time: 2020-10-23 17:00:00       Value: 26.706036319556862
Time: 2020-10-23 18:00:00       Value: 23.14490836891262
Time: 2020-10-23 19:00:00       Value: 22.209978787956935
Time: 2020-10-23 20:00:00       Value: 27.685604486039487
Time: 2020-10-23 21:00:00       Value: 24.836814922301365
Time: 2020-10-23 22:00:00       Value: 15.27867217513822
Time: 2020-10-24 17:00:00       Value: 19.51907743254917

Results for Level_2
========================================================================
Time: 2020-10-23 08:00:00       Value: 24.304509501484077
Time: 2020-10-23 09:00:00       Value: 23.17488834892463
Time: 2020-10-23 10:00:00       Value: 25.5070190062735
Time: 2020-10-23 11:00:00       Value: 28.24532554225462
Time: 2020-10-23 12:00:00       Value: 23.976792747239934
Time: 2020-10-23 18:00:00       Value: 23.14490836891262
Time: 2020-10-23 19:00:00       Value: 22.209978787956935
Time: 2020-10-23 20:00:00       Value: 27.685604486039487
Time: 2020-10-23 21:00:00       Value: 24.836814922301365

Results for Level_3
========================================================================
Time: 2020-10-23 11:00:00       Value: 28.24532554225462
========================================================================
```

FIGURE 41 – Prédiction de niveau d'alerte du polluant

8 Conclusion

Étant donné tout ce qui est présenté, on a déjà bien maîtrisé comment analyser, et prédire les valeurs d'une série temporelle à l'aide de module SARIMA et VAR, en bien distinguant leur avantages et inconvénients. La prédiction par VAR est généralement meilleure qu'un modèle traditionnel, et plus simple à interpréter à condition d'avoir plusieurs variables corrélées. De plus, on a bien constaté l'importance de pre-processing, une étape qui prend beaucoup de temps mais essentielle à faire. Finalement, il faut toujours s'assurer que notre série temporelle est stationnaire, sinon, tous les algorithmes statistiques n'auront aucun sens. Le VAR est intéressant dans le cas d'un réseau de capteurs, dans la ville ou le bâtiment car il y a généralement plusieurs capteurs entre eux, du même type ou sur des variables hétérogènes.

Références

[1] https ://aqicn.org/scale/fr/

[2] https ://en.wikipedia.org/wiki/Air_quality_index

[3] https ://www.opendata.dk/city-of-copenhagen/luftforurening

[4] https ://www.researchgate.net/publication/238195024_Introducing_VAR_and_SVAR_predictions_in_system_dynamics_models

[5] https ://www.airparif.asso.fr/alertes/historique

Appendices

```
# ======================*BLOC - Importing packpages*======================
# Common packages
import numpy as np
import pandas as pd
pd.options.mode.chained_assignment = None
import matplotlib.pyplot as plt
import seaborn as sns
import itertools
import datetime
import warnings
import requests

# Statistic module packages
from statsmodels.tools.sm_exceptions import ConvergenceWarning
warnings.simplefilter('ignore', ConvergenceWarning)
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.stats.diagnostic import acorr_ljungbox
from statsmodels.tsa.vector_ar.var_model import VAR
from statsmodels.tsa.stattools import grangercausalitytests
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from statsmodels.stats.stattools import durbin_watson

# Scikit_learn and RNN module packages
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

# Install inexistant packages
import importlib
if importlib.util.find_spec("missingno") is None:
  !pip install --user --upgrade missingno
if importlib.util.find_spec("pmdarima") is None:
  !pip uninstall numpy
  !pip uninstall pmdarima
  !pip install numpy
  !pip install pmdarima

import missingno as msno
import pmdarima as pm
```

```
# ======================*END - Importing packpages*======================


# ======================*BLOC - General functions*======================
def inspect_data(df=None, visualize_missing_value=False,
    display_form='matrix', freq=None, figsize=(6,4)):
  """Function to display the information of data
  - Paramters:
    df: DataFrame or Series
    visualize_missing_value: is or not to visualize the missing value, default
        False
    display_form: visualization chart to use, 'matrix'(defaul), 'bar',
        'heatmap'
    figsize: The size of the figure to display
    freq: Specify a periodicity for time-series data, default 'D'
  - Return value: None
  """
  print(df.info())
  print(">>> The first five lines of data: \n", df.head())
  if visualize_missing_value == True:
    print(">>> Missing value visualization:")
    if display_form == 'matrix':
      if freq is not None:
        msno.matrix(df, freq=freq, figsize=figsize)
      else:
        msno.matrix(df, figsize=figsize)
    if display_form == 'bar':
      msno.bar(df, figsize=figsize)
    if display_form == 'heatmap':
      msno.heatmap(df, figsize=figsize)
  plt.show()
  return


def plot_data(df=None, figsize=(12,8), layout=(3,3)):
  '''Function to plot the time series data
  - Paramters:
    df: DataFrame or Series
    figsize: size of figure
    layout: layout of figure plot decided by the number of time-series
  - Return value: None
  '''
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  df.plot(subplots=True, figsize=figsize, layout=layout, colormap='Dark2',
      fontsize=12)
  plt.show()
```

```
  return


def fill_missing_values(df=None, method='ffill', interpolate_mode='linear',
    fill_value=0, figsize=(8,4)):
  """Function to fill the missing value of time series
  - Paramters:
    df: DataFrame or Series
    method: could be ['ffill'(default), 'bfill', 'value', 'interpolate']
    interpolate_mode: compulsory when methode='interpolate', could be
        ['linear'(defaut), 'polynomial', 'akima', 'pad', 'nearest', 'zero',
        'quadratic', 'cubic', 'spline']
    fill_value: compulsory when methode='value', 0 by default
    figsize: size of figure
  - Return value:
    df_fill: series with missing value filled
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Filled by foreward or backward value
  if method in ['ffill', 'bfill']:
    df_fill = df.fillna(method=method)
  # Filled by given a value
  if method == 'value':
    df_fill = df.fillna(fill_value)
  # Fiiled by method 'interpolation'
  if method == 'interpolate':
    df_fill = df.interpolate(interpolate_mode, order=2)
  return df_fill

def conversion_to_ppb(df=None):
  """Function to convert polluant concentration in unit g/m3 to ppb
    -  g /m3 is micrograms of gaseous pollutant per cubic meter of ambient air
    - ppb (v) is parts per billion by volume
    - NO2 1 ppb = 1.88 g/m3
    - NO 1 ppb = 1.25 g/m3
    - O3 1 ppb = 2.00 g/m3
    - SO2: 1 ppb = 2.62 g/m3
    - CO 1 ppm = 1 mg/m3
  - Parameters:
    df: DataFrame or Series
  - Return value:
    df: DataFrame converted
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
```

```
  # Get all the column names
  column_names = df.columns.values
  for i, column_name in enumerate(column_names):
    # Case NO2
    if 'NO2' in column_name:
      df[column_name] = df[column_name] / 1.88
    # Case NOx
    if 'NOx' in column_name:
      df[column_name] = df[column_name] / 1.25
    # Case O3
    if 'O3' in column_name:
      df[column_name] = df[column_name] / 2.00
    # Case SO2
    if 'SO2' in column_name:
      df[column_name] = df[column_name] / 2.62
  return df


def select_period_resample(df=None, columns=[], start_at=None, end_at=None,
    freq=None):
  """Function to extract a given period of Time Series
  - Paramters:
    df: DataFrame or Series
    columns: list of dataframe columns to choose, defalt all
    strat_time: start time point, format String or Timestampe
    end_time: end time point, format Timestamp
    freq: set resampling frequency of data
  - Return value:
    Series with period selected and resampled
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Transform datetime string to timestamp
  start_at = pd.to_datetime(start_at)
  end_at = pd.to_datetime(end_at)
  if len(columns) != 0: # Extract part of columns
    return df.loc[start_at:end_at][columns].resample(freq).mean()
  else: # Extract all the columns
    return df.loc[start_at:end_at].resample(freq).mean()


def decompose_series(df=None, model='additive', return_seasonal=False,
    return_trend=False, figsize=(8,4)):
  """Function to decompose the series into trend, seasonal and residual, to
      return seasonal and trend DataFrame if needed
  - Paramters:
    df: DataFrame or Series
```

```
    model : Type of seasonal component, {"additive"(default), "multiplicative"}
    return_seasonal; is or not to return seasonal dataframe of all the columns
    return_trend; is or not to return trend dataframe of all the columns
    figsize: size of figure
  - Return value:
    trend_df or seasonal_df
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Initialize dict to stock trend and seasoanl component
  my_dict = {}
  my_dict_trend = {}
  my_dict_seasonal = {}
  # Get all the column names
  column_names = df.columns
  for column_name in column_names:
    decomp = seasonal_decompose(df[column_name], model=model)
    my_dict[column_name] = decomp
    if return_seasonal==False and return_trend==False:
      decomp.plot()
      plt.title(column_name, fontsize=10, fontweight=4)
      plt.show()
  # Extract the trend component
  for column_name in column_names:
    my_dict_trend[column_name] = my_dict[column_name].trend
  # Extract the sesonal component
  for column_name in column_names:
    my_dict_seasonal[column_name] = my_dict[column_name].seasonal
  # Convert to a DataFtrame
  trend_df = pd.DataFrame.from_dict(my_dict_trend)
  seasonal_df = pd.DataFrame.from_dict(my_dict_seasonal)
  if return_seasonal==True and return_trend==False:
    return seasonal_df
  if return_seasonal==False and return_trend==True:
    return trend_df
  if return_seasonal==True and return_trend==True:
    return seasonal_df, trend_df
  if return_seasonal==False and return_trend==False:
    return


def plot_rolling_means_variance(df=None, window=None, figsize=(12,2)):
  '''Function to inpsect the stationarity of time series by visualizing
     rolling means and variances
  - Parameters:
    df: DataFrame or Series
    window: rolling size of window, equal to seasonal lags
```

```
    figsize: size of figures
  - Return value: None
    '''
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Get all the column names
  column_names = df.columns
  for i, column_name in enumerate(column_names):
    fig, ax = plt.subplots(figsize=figsize)
    ax.plot(df[column_name], label='raw data')
    ax.plot(df[column_name].rolling(window=window).mean(), label="rolling
        mean")
    ax.plot(df[column_name].rolling(window=window).std(), label="rolling std")
    ax.set_xlabel('Time')
    ax.set_ylabel(column_name)
    ax.legend()
  plt.show()
  return


def is_stationnary(df=None):
  """Function to check if the time series is stationary by Augmented
      Dickey-Fuller Test
  - Paramters:
    df: DataFrame or Series
  - Return value: None
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Get all the column names
  column_names = df.columns.values
  for i, column_name in enumerate(column_names):
    print("Augmented Dickey-Fuller Test on {}".format(column_name))
    # Calculate ADF value
    results_ADF = adfuller(df.iloc[:,i].dropna(), autolag='AIC')
    print('Null Hypothesis: Data has unit root. Non-Stationary.')
    print("Test statistic = {:.3f}".format(results_ADF[0]))
    print("P-value = {:.3f}".format(results_ADF[1]))
    print("Critical values :")
    for k, v in results_ADF[4].items():
        print("\t{}: {:.3f} ==> The data is {} stationary with {}%
            confidence".format(k, v, "not" if v<results_ADF[0] else "",
            100-int(k[:-1])))
    print('\n')
  return
```

```
def transform_diff_log(df=None, lags=1, rolling=False, windows=None,
    log=False, valid=False):
  """Function to transform a non-stationary series to a stationary one by
      differencing, or to remove the exponential trend
  - Paramters:
    df: DataFrame or Series
    lags: differences lags to take, 1 default
    rolling: rolling data to make it more smoothly
    windows: size of rolling window, compulsary if rolling=True
    log: logging the data if the raw data exhibits an exponential trend
    valid: is or not to verify the stationarity
  - Return value:
    df: stationary time series
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  if log:
    df = np.log(df).dropna()
  if rolling:
    df = df.rolling(window=windows).mean().dropna()
  if lags is not None:
    df = df.diff(lags).dropna()
  if valid:
    is_stationnary(df)
  return df


def acf_pacf_plot(df=None, lags=20, alpha=0.1, acf=True):
  """Funtion to visualize the ACF and PACF plot
  - Paramters:
    df: DataFrame or Series
    lags: numbers of lags of autocorrelation to be plotted
    alpha: set the width of confidence interval. if aplha=1, hidden blue bound
    acf: is or not to plot acf
  - Return value: None
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  if acf:
    fig, axes = plt.subplots(2,1)
    # Plot the ACF
    plot_acf(df, lags=lags, alpha=alpha, ax=axes[0])
    # Plot the PACF
    plot_pacf(df, lags=lags, alpha=alpha, ax=axes[1])
  else:
```

```
    fig, axes = plt.subplots()
    plot_pacf(df, lags=lags, alpha=alpha, ax=axes)
  plt.title(df.columns.values[0])
  plt.tight_layout()
  plt.show()
  return


def train_test_split(df=None, train_size=None, start_at=None, end_at=None):
  """Function to split a time series into training and testing set
  - Paramters:
    df: DataFrame or Series
    train_size: size of training set
    start_at: start point of training set, not to set if using train_size
    end_at: end point of training set, not to set if using train_size
  - Return value:
    training_set: training part of series
    testing_set: testing part of series
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  if train_size is not None:
    training_set = df.iloc[:np.int(len(df) * train_size)]
    testing_set = df.iloc[np.int(len(df) * train_size):]
  else:
    training_set = df.loc[start_at:end_at]
    testing_set = df.loc[end_at:]
  return training_set, testing_set


def data_labeled(df=None, freq=None, mode='Mean'):
  """Function to label the data based on the different thresholds of air
     pollution
  - Parameters:
    df: DataFrame or Series
    freq: data frequency by Hour('H') or by Day('D')
    mode: calculating hourly average(Mean, default) or maximum(Max) value
  - Remark:
    NO2, O3, SO2: hourly average/maximum value in g/m
    PM10, PM2.5: hourly average/maximum value or daily average adjusted in g/m
    CO: 8-hour rolling average/maximum in mg/m
  - Return value:
    df: DataFrame labeled
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
```

```
# Resample and aggregate the data
if mode == 'Max':
  df = df.resample(freq).max()
else:
  df = df.resample(freq).mean()
# Get all the column names
column_names = df.columns.values
for i, column_name in enumerate(column_names):
  # Case NO2
  if 'NO2' in column_name:
    df[str(column_name) + '_Class'] = None
    AQI_class1 = (50 - 0) / (53 - 0) *
        (df[column_name].iloc[np.where((df[column_name] >= 0) &
        (df[column_name] <= 53))].to_frame() - 0) + 0
    index_class1 = AQI_class1.index
    AQI_class2 = (100 - 51) / (100 - 54) *
        (df[column_name].iloc[np.where((df[column_name] >= 54) &
        (df[column_name] <= 100))].to_frame() - 54) + 51
    index_class2 = AQI_class2.index
    AQI_class3 = (150 - 101) / (360 - 101) *
        (df[column_name].iloc[np.where((df[column_name] >= 101) &
        (df[column_name] <= 360))].to_frame() - 101) + 101
    index_class3 = AQI_class3.index
    AQI_class4 = (200 - 151) / (649 - 361) *
        (df[column_name].iloc[np.where((df[column_name] >= 361) &
        (df[column_name] <= 649))].to_frame() - 361) + 151
    index_class4 = AQI_class4.index
    AQI_class5 = (300 - 201) / (1249 - 650) *
        (df[column_name].iloc[np.where((df[column_name] >= 650) &
        (df[column_name] <= 1249))].to_frame() - 650) + 201
    index_class5 = AQI_class5.index
    AQI_class6 = (500 - 301) / (2049 - 1250) *
        (df[column_name].iloc[np.where(df[column_name] >= 1250)].to_frame()
        - 1250) + 301
    index_class6 = AQI_class6.index
    df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
    df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
    df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
    df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
    df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
    df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
    df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
        AQI_class2[column_name], AQI_class3[column_name],
        AQI_class4[column_name], AQI_class5[column_name],
        AQI_class6[column_name]])
 # Case O3
  if 'O3' in column_name:
    df[str(column_name) + '_Class'] = None
```

```
    AQI_class1 = (50 - 0) / (54 - 0) *
        (df[column_name].iloc[np.where((df[column_name] >= 0) &
        (df[column_name] <= 54))].to_frame() - 0) + 0
    index_class1 = AQI_class1.index
    AQI_class2 = (100 - 51) / (70 - 55) *
        (df[column_name].iloc[np.where((df[column_name] >= 55) &
        (df[column_name] <= 70))].to_frame() - 55) + 51
    index_class2 = AQI_class2.index
    AQI_class3 = (150 - 101) / (85 - 71) *
        (df[column_name].iloc[np.where((df[column_name] >= 71) &
        (df[column_name] <= 85))].to_frame() - 71) + 101
    index_class3 = AQI_class3.index
    AQI_class4 = (200 - 151) / (106 - 86) *
        (df[column_name].iloc[np.where((df[column_name] >= 86) &
        (df[column_name] <= 105))].to_frame() - 86) + 151
    index_class4 = AQI_class4.index
    AQI_class5 = (300 - 201) / (200 - 106) *
        (df[column_name].iloc[np.where((df[column_name] >= 106) &
        (df[column_name] <= 200))].to_frame() - 106) + 201
    index_class5 = AQI_class5.index
    AQI_class6 = (500 - 301) / (300 - 201) *
        (df[column_name].iloc[np.where(df[column_name] >= 201)].to_frame() -
        201) + 201
    index_class6 = AQI_class6.index
    df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
    df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
    df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
    df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
    df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
    df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
    df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
        AQI_class2[column_name], AQI_class3[column_name],
        AQI_class4[column_name], AQI_class5[column_name],
        AQI_class6[column_name]])
# Case SO2
 if 'SO2' in column_name:
    df[str(column_name) + '_Class'] = None
    AQI_class1 = (50 - 0) / (35 - 0) *
        (df[column_name].iloc[np.where((df[column_name] >= 0) &
        (df[column_name] <= 35))].to_frame() - 0) + 0
    index_class1 = AQI_class1.index
    AQI_class2 = (100 - 51) / (75 - 36) *
        (df[column_name].iloc[np.where((df[column_name] >= 36) &
        (df[column_name] <= 75))].to_frame() - 36) + 51
    index_class2 = AQI_class2.index
    AQI_class3 = (150 - 101) / (185 - 76) *
        (df[column_name].iloc[np.where((df[column_name] >= 76) &
        (df[column_name] <= 185))].to_frame() - 76) + 101
```

```
  index_class3 = AQI_class3.index
  AQI_class4 = (200 - 151) / (304 - 186) *
      (df[column_name].iloc[np.where((df[column_name] >= 186) &
      (df[column_name] <= 304))].to_frame() - 186) + 151
  index_class4 = AQI_class4.index
  AQI_class5 = (300 - 201) / (604 - 305) *
      (df[column_name].iloc[np.where((df[column_name] >= 305) &
      (df[column_name] <= 604))].to_frame() - 305) + 201
  index_class5 = AQI_class5.index
  AQI_class6 = (500 - 301) / (1004 - 605) *
      (df[column_name].iloc[np.where(df[column_name] >= 605)].to_frame() -
      605) + 301
  index_class6 = AQI_class6.index
  df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
  df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
  df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
  df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
  df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
      AQI_class2[column_name], AQI_class3[column_name],
      AQI_class4[column_name], AQI_class5[column_name],
      AQI_class6[column_name]])
# Case PM10
if 'PM10' in column_name:
  df[str(column_name) + '_Class'] = None
  AQI_class1 = (50 - 0) / (54 - 0) *
      (df[column_name].iloc[np.where((df[column_name] >= 0) &
      (df[column_name] <= 54))].to_frame() - 0) + 0
  index_class1 = AQI_class1.index
  AQI_class2 = (100 - 51) / (154 - 55) *
      (df[column_name].iloc[np.where((df[column_name] >= 55) &
      (df[column_name] <= 154))].to_frame() - 55) + 51
  index_class2 = AQI_class2.index
  AQI_class3 = (150 - 101) / (254 - 155) *
      (df[column_name].iloc[np.where((df[column_name] >= 155) &
      (df[column_name] <= 1254))].to_frame() - 155) + 101
  index_class3 = AQI_class3.index
  AQI_class4 = (200 - 151) / (354 - 255) *
      (df[column_name].iloc[np.where((df[column_name] >= 255) &
      (df[column_name] <= 354))].to_frame() - 255) + 151
  index_class4 = AQI_class4.index
  AQI_class5 = (300 - 201) / (424 - 355) *
      (df[column_name].iloc[np.where((df[column_name] >= 355) &
      (df[column_name] <= 424))].to_frame() - 424) + 201
  index_class5 = AQI_class5.index
```

```
  AQI_class6 = (500 - 301) / (604 - 425) *
      (df[column_name].iloc[np.where(df[column_name] >= 425)].to_frame() -
      425) + 301
  index_class6 = AQI_class6.index
  df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
  df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
  df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
  df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
  df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
      AQI_class2[column_name], AQI_class3[column_name],
      AQI_class4[column_name], AQI_class5[column_name],
      AQI_class6[column_name]])
# Case PM2.5
if 'PM2.5' in column_name:
  df[str(column_name) + '_Class'] = None
  AQI_class1 = (50 - 0) / (12 - 0) *
      (df[column_name].iloc[np.where((df[column_name] >= 0) &
      (df[column_name] <= 12))].to_frame() - 0) + 0
  index_class1 = AQI_class1.index
  AQI_class2 = (100 - 51) / (35.4 - 12.1) *
      (df[column_name].iloc[np.where((df[column_name] >= 12.1) &
      (df[column_name] <= 35.4))].to_frame() - 12.1) + 51
  index_class2 = AQI_class2.index
  AQI_class3 = (150 - 101) / (55.4 - 35.5) *
      (df[column_name].iloc[np.where((df[column_name] >= 35.5) &
      (df[column_name] <= 55.4))].to_frame() - 35.5) + 101
  index_class3 = AQI_class3.index
  AQI_class4 = (200 - 151) / (150.4 - 55.5) *
      (df[column_name].iloc[np.where((df[column_name] >= 55.5) &
      (df[column_name] <= 150.4))].to_frame() - 55.5) + 151
  index_class4 = AQI_class4.index
  AQI_class5 = (300 - 201) / (250.4 - 150.5) *
      (df[column_name].iloc[np.where((df[column_name] >= 150.5) &
      (df[column_name] <= 250.4))].to_frame() - 150.5) + 201
  index_class5 = AQI_class5.index
  AQI_class6 = (500 - 301) / (500.4 - 250.5) *
      (df[column_name].iloc[np.where(df[column_name] >= 250.5)].to_frame()
      - 250.5) + 301
  index_class6 = AQI_class6.index
  df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
  df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
  df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
  df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
```

```
  df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
      AQI_class2[column_name], AQI_class3[column_name],
      AQI_class4[column_name], AQI_class5[column_name],
      AQI_class6[column_name]])
# Case NOx
if 'NOx' in column_name:
  df[str(column_name) + '_Class'] = None
  AQI_class1 = (50 - 0) / (53 - 0) *
      (df[column_name].iloc[np.where((df[column_name] >= 0) &
      (df[column_name] <= 53))].to_frame() - 0) + 0
  index_class1 = AQI_class1.index
  AQI_class2 = (100 - 51) / (100 - 54) *
      (df[column_name].iloc[np.where((df[column_name] >= 54) &
      (df[column_name] <= 100))].to_frame() - 54) + 51
  index_class2 = AQI_class2.index
  AQI_class3 = (150 - 101) / (360 - 101) *
      (df[column_name].iloc[np.where((df[column_name] >= 101) &
      (df[column_name] <= 360))].to_frame() - 101) + 101
  index_class3 = AQI_class3.index
  AQI_class4 = (200 - 151) / (649 - 361) *
      (df[column_name].iloc[np.where((df[column_name] >= 361) &
      (df[column_name] <= 649))].to_frame() - 361) + 151
  index_class4 = AQI_class4.index
  AQI_class5 = (300 - 201) / (1249 - 650) *
      (df[column_name].iloc[np.where((df[column_name] >= 650) &
      (df[column_name] <= 1249))].to_frame() - 650) + 201
  index_class5 = AQI_class5.index
  AQI_class6 = (500 - 301) / (2049 - 1250) *
      (df[column_name].iloc[np.where(df[column_name] >= 1250)].to_frame()
      - 1250) + 301
  index_class6 = AQI_class6.index
  df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
  df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
  df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
  df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
  df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
  df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
      AQI_class2[column_name], AQI_class3[column_name],
      AQI_class4[column_name], AQI_class5[column_name],
      AQI_class6[column_name]])
# Case CO
if 'CO' in column_name:
  # 8-hour rolling average/maximum in mg/m
  df_CO = df.copy()
  df_CO = df_CO.rolling(window=8).mean().fillna(method='bfill')
  df[column_name] = df_CO[column_name]
  df[str(column_name) + '_Class'] = None
```

```
        AQI_class1 = (50 - 0) / (4.4 - 0) *
            (df[column_name].iloc[np.where((df[column_name] >= 0) &
            (df[column_name] <= 4.4))].to_frame() - 0) + 0
        index_class1 = AQI_class1.index
        AQI_class2 = (100 - 51) / (9.4 - 4.5) *
            (df[column_name].iloc[np.where((df[column_name] >= 4.5) &
            (df[column_name] <= 9.4))].to_frame() - 4.5) + 51
        index_class2 = AQI_class2.index
        AQI_class3 = (150 - 101) / (12.4 - 9.5) *
            (df[column_name].iloc[np.where((df[column_name] >= 9.5) &
            (df[column_name] <= 12.4))].to_frame() - 9.5) + 101
        index_class3 = AQI_class3.index
        AQI_class4 = (200 - 151) / (15.4 - 12.5) *
            (df[column_name].iloc[np.where((df[column_name] >= 12.5) &
            (df[column_name] <= 15.4))].to_frame() - 12.5) + 151
        index_class4 = AQI_class4.index
        AQI_class5 = (300 - 201) / (30.4 - 15.5) *
            (df[column_name].iloc[np.where((df[column_name] >= 15.5) &
            (df[column_name] <= 30.4))].to_frame() - 15.5) + 201
        index_class5 = AQI_class5.index
        AQI_class6 = (500 - 301) / (50.4 - 30.5) *
            (df[column_name].iloc[np.where(df[column_name] >= 30.5)].to_frame()
            - 30.5) + 301
        index_class6 = AQI_class6.index
        df[str(column_name) + '_Class'].loc[index_class1] = 'Good'
        df[str(column_name) + '_Class'].loc[index_class2] = 'Moderate'
        df[str(column_name) + '_Class'].loc[index_class3] = 'Unhealthy for SG'
        df[str(column_name) + '_Class'].loc[index_class4] = 'Unhealthy'
        df[str(column_name) + '_Class'].loc[index_class5] = 'Very Unhealthy'
        df[str(column_name) + '_Class'].loc[index_class6] = 'Hazardous'
        df[str(column_name) + '_AQI'] = pd.concat([AQI_class1[column_name],
            AQI_class2[column_name], AQI_class3[column_name],
            AQI_class4[column_name], AQI_class5[column_name],
            AQI_class6[column_name]])
    # Fill the missing value
    df = df.fillna(method='ffill')
    return df


def Air_Situation_Dashbord(df=None, start_time=None, end_time=None,
    figsize=(10, 4), width=0.5, freq='H', mode='Mean', min_max_distance=0.1,
    return_AQI=False, title=None):
    """Function to display the dashbord of pollution concentration(sigle or
        more) for a seleted period
    - Parameters:
        df: DataFrame or Series
        start_time: start time point to display(String or Timestampe), not set
            when using whole data
```

```
  end_time: end time point to display(String or Timestampe), not set when
      using whole data
  figsize: size of figure, default (10,4)
  width: control the width between each bar, default 0.5
  freq: frequency of resampling, used in function 'data_labeled'
  mode: calculating hourly average(Mean, default) or maximum(Max) value
  min_max_distance: used to modeify the position of MIN and MAX value
  return_AQI: is or not to return labeled DataFrame
  title: title to set
- Retrun value
  df: labeled DataFrame / None
"""
# Transform Series objet to DataFrame object
if not isinstance(df, pd.DataFrame):
  df = df.to_frame()
# Transform datetime string to timestamp
start_time = pd.to_datetime(start_time)
end_time = pd.to_datetime(end_time)
# Label data
if start_time is not None and end_time is not None:
  df = data_labeled(df[start_time:end_time], freq=freq, mode=mode)
else:
  df = data_labeled(df[start_time:end_time], freq=freq, mode=mode)
# Get all the column names
column_names = df.columns
for i, column_name in enumerate(column_names):
  if column_name in ['NO2', 'NOx', 'O3', 'SO2', 'PM10', 'PM2.5', 'CO']:
    df_color = df.copy().dropna()
      # Caclulate the MIN and MAX value of polluation concentration in
          seleted period
    min_value = df_color[str(column_name) + '_AQI'].min()
    max_value = df_color[str(column_name) + '_AQI'].max()
      # Transform the polluation indice to color correpondent
    df_color['Color'] = None
    index_green = df_color.iloc[np.where(df_color[str(column_name)+'_Class']
        == 'Good')].index
    df_color['Color'].loc[index_green] = 'Green'
    index_yellow =
        df_color.iloc[np.where(df_color[str(column_name)+'_Class'] ==
        'Moderate')].index
    df_color['Color'].loc[index_yellow] = 'Yellow'
    index_orange =
        df_color.iloc[np.where(df_color[str(column_name)+'_Class'] ==
        'Unhealthy for SG')].index
    df_color['Color'].loc[index_orange] = 'Orange'
    index_red = df_color.iloc[np.where(df_color[str(column_name)+'_Class']
        == 'Unhealthy')].index
    df_color['Color'].loc[index_red] = 'Red'
```

```
        index_purple =
            df_color.iloc[np.where(df_color[str(column_name)+'_Class'] == 'Very
            Unhealthy')].index
        df_color['Color'].loc[index_purple] = 'Purple'
        index_maroon =
            df_color.iloc[np.where(df_color[str(column_name)+'_Class'] ==
            'Hazardous')].index
        df_color['Color'].loc[index_maroon] = 'Maroon'
          # Set and plot the figure
        fig, ax = plt.subplots(figsize = figsize)
        ax.bar(x=df_color.index, height=df_color[str(column_name)+'_AQI'],
            width=width, color=df_color['Color'])
        ax.set_xlabel('Time')
        ax.set_ylabel(str(column_name) + "-AQI")
        ax.text(df.index[-1]+datetime.timedelta(days=min_max_distance),
            max_value, '-- MAX={:.1f}'.format(max_value), color='red')
        ax.text(df.index[-1]+datetime.timedelta(days=min_max_distance),
            min_value, '-- MIN={:.1f}'.format(min_value), color='blue')
        # Set the separating vertical lines
        for i in range(len(df.index)):
          if np.mod(i,6) == 0:
            ax.axvline(df.index[i], linewidth=1, linestyle='--', color='black',
                alpha=0.5)
            if i >= 24:
              i_temp = np.mod(i, 24)
              ax.text(df.index[i], max_value, i_temp)
            else:
              ax.text(df.index[i], max_value, i)
    plt.title(title, fontsize=18, loc='center')
    plt.show()
    if return_AQI:
      return df
    else:
      return


def evaluate(y_true=None, y_pred=None):
    """Evaluate MSE and RMS of true value and predict value
    - Parameters:
      y_true: Truth (correct) target values, DataFrame, Series or Array
      y_pred: Estimated target values same type with y_true
    - Return value: None
    """
    mse = mean_squared_error(y_true=y_true, y_pred=y_pred, squared=True)
    rmse = mean_squared_error(y_true=y_true, y_pred=y_pred, squared=False)
    print("MSE Value:{}".format(mse))
    print("RMSE Value:{}".format(rmse))
    return
```

```
def is_incointegrated(P=None,Q=None):
  """Function to test if the linear combination(P - Coeff * Q) of two random
     series P and Q is not a random walk.
  - Paramters:
    P, Q: DataFrame or Series
  - Return value:
    The incointegrated series if they are incointegrated
  """
  # Transform DataFrame object to Series objet
  if not isinstance(P, pd.Series):
    # P = P.iloc[:,0]
    P = pd.Series(P)
  if not isinstance(Q, pd.Series):
    Q = Q.iloc[:,0]
  # Regress P on Q to get the regression coefficient Coeff
  Q = sm.add_constant(Q)
  Coeff = sm.OLS(P,Q).fit().params[1]
  # Compute ADF
  inter_series = P - Coeff * Q.iloc[:,1]
  p_value = adfuller(inter_series)[1]
  if p_value < 0.05:
    print("Two series are incointegrated with with 95% confidence")
    return inter_series
  else:
    print("Two series are not incointegrated")
    return


def correlation_analyse(df1=None, df2=None, method='pearson', figsize=(5, 4)):
  """Function to calculate and display the correlation of two Time Series
  - Paramters:
    df1: DataFrame or Series
    df2: DataFrame or Series
    method: test the linear correlaiton by 'pearson'(default) or non-linear
        correlaiton by 'spearman'
    figsize: size of plot figure
  - Return value: None
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df1, pd.DataFrame):
    df1 = df1.to_frame()
  if not isinstance(df2, pd.DataFrame):
    df2 = df2.to_frame()
  # Concatenate dataframe
  if df2 is None:
    df = df1
```

```
  else:
    df = pd.concat([df1,df2], axis=1)
  # Calculate correlation coefficient
  correlation = df.corr(method=method)
  fig = sns.clustermap(correlation, annot=True, linewidth=0.1,
      figsize=figsize)
  plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
  plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=90)
  if method=='pearson':
    plt.title('Linear correlation analysis', fontsize=12, fontweight=5,
        loc='center')
  else:
    plt.title('Non-linear correlation analysis', fontsize=12, fontweight=5,
        loc='center')
  plt.tight_layout()
  plt.show()
  return

def extract_by_hour(df=None, start_day=None, end_day=None, start_time=None,
    end_time=None):
  '''Function to extract a specific time period of data for each day
  - Parameters:
    df: DataFrame or Series
    start_day: Timestamp or string
    end_day: Timestamp or string
    start_time: start time point, int
    end_time: end time point, int
  - Return values:
    df: DataFrame with data in extracting period
  '''
  # Transform datetime string to Timestampe
  start = pd.to_datetime(start_day)
  end = pd.to_datetime(end_day)
  time_stamp = []
  for i in pd.date_range(start=start_day, end=end_day):
    for j in range(start_time, end_time+1):
      time_stamp.append(i + datetime.timedelta(hours=j))
  df = df.loc[time_stamp]
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  return df


def pc_last_previous(df=None):
  '''Function to calculates the % change between the last value and the mean
      of previous values
  - Paramters:
```

```
    df: DataFrame or Series
  - Return value:
    percent_change
  '''
  # Seperate the last value and all previous values into variables
  previous_values = df[:-1]
  last_values = df[-1]
  # Calculate the % of difference between the last value and the mean of
      earlier values
  percent_change = (last_values - np.mean(previous_values)) /
      np.mean(previous_values)
  return percent_change


def pc_rolling_plot(df=None, window=5, show=True, figsize=(16,6)):
  '''Function to calculates the % change, to transform it within a certain
      size of rolling window or or to plot it
  - Paramters:
    df: DataFrame or Series
    window: size of rolling window, 5 default
    show: is or not to display the rolling data, default True
    figsize: size the plot figure
  - Return value:
    df_rolling: serie
  '''
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Calculate % change based on rolling window
  df_rolling = df.rolling(window=window).apply(pc_last_previous)
  if show:
    # Plot the raw data and rolling data
    fig, axs = plt.subplots(1,2,figsize=figsize)
    ax = df.plot(ax=axs[0], title="Raw data", xlabel="Time", ylabel="Mean")
    ax = df_rolling.plot(ax=axs[1], title="Rolling data", xlabel="Time",
        ylabel="Mean")
    plt.tight_layout()
    plt.show()
  return df_rolling

def forecast_warning(df, level_1=None, level_2=None, level_3=None, window=3,
    report=True):
  '''Function to forecast the warning time
  - Paramters:
    df: DataFrame or Series
    level_1: fist warning threshold
    level_2: second warning threshold
    level_3: third warning threshold
```

```
  report: print summary report
- Return value: None
'''
# Transform Series objet to DataFrame object
if not isinstance(df, pd.DataFrame):
  df = df.to_frame()
# Check the frequency and resample if necessary
if df.index.freqstr != 'H':
  df = df.resample('H').mean()
# Initialize dataframe
df['Level 1'] = None
df['Level 2'] = None
df['Level 3'] = None
# Test for level 1
df_rolling = df.rolling(window=window).mean().fillna(method='bfill')
df.loc[df.iloc[np.where(df_rolling.iloc[:,0] >= level_1)].index, 'Level 1']
    = 'Warning'
df.loc[df.iloc[np.where(df_rolling.iloc[:,0] < level_1)].index, 'Level 1']
    = 'Safe'
# Test for level 2
for ind in df.index[:-2]:
  ind_1 = ind
  ind_2 = ind+datetime.timedelta(hours=1)
  ind_3 = ind+datetime.timedelta(hours=2)
  if df.loc[ind_1].iloc[0] >= level_2 and df.loc[ind_2].iloc[0] >= level_2
      and df.loc[ind_3].iloc[0] >= level_2:
    df.loc[ind_3, 'Level 2'] = 'Warning'
  if df.loc[ind_1, 'Level 2'] != 'Warning':
    df.loc[ind_1, 'Level 2'] = 'Safe'
  if df.loc[ind_2, 'Level 2'] != 'Warning':
    df.loc[ind_2, 'Level 2'] = 'Safe'
  if df.loc[ind_3, 'Level 2'] != 'Warning':
    df.loc[ind_3, 'Level 2'] = 'Safe'
# Test for level 3
df.loc[df.iloc[np.where(df.iloc[:,0] >= level_3)].index, 'Level 3'] =
    'Warning'
df.loc[df.iloc[np.where(df.iloc[:,0] < level_3)].index, 'Level 3'] = 'Safe'
if report:
  print("\t\tSummary of Warning Forecasting Results\n",
     "=============================================================================\n",
     "Level 1(AQI) - Hourly Rolling(window=3) Average Exceeds
        {}\n".format(level_1),
     "Level 2(AQI) - Hourly Average Exceeds {} for 3 consecutive
        hours\n".format(level_2),
     "Level 3(AQI) - Hourly Average Exceeds {}\n".format(level_3),
     "-----------------------------------------------------------------------\n"
     " Results for Level_1\n",
     "=============================================================================\n"
```

```
      )
    result_level_1 = df.iloc[np.where(df['Level 1'] == 'Warning')].iloc[:,0]
    for time, value in zip(result_level_1.index, result_level_1.values):
      print(" Time: {}\t".format(time), "Value: {}".format(value))
    print("\n Results for Level_2\n",
      "==========================================================================")
    result_level_2 = df.iloc[np.where(df['Level 2'] == 'Warning')].iloc[:,0]
    for time, value in zip(result_level_2.index, result_level_2.values):
      print(" Time: {}\t".format(time), "Value: {}".format(value))
    print("\n Results for Level_3\n",
      "==========================================================================")
    result_level_3 = df.iloc[np.where(df['Level 3'] == 'Warning')].iloc[:,0]
    for time, value in zip(result_level_3.index, result_level_3.values):
      print(" Time: {}\t".format(time), "Value: {}".format(value))
    print("==========================================================================")
  return


def nomalize_data(df=None):
  """Function to normalize the time series
  - Paramters:
    df: DataFrame or Series
  - Return value:
    df: DataFrame normalized
    avgs: average of each column, used to convert the prediction result
    devs: deviation of each column, used to convert the prediction result
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  avgs = df.mean()
  devs = df.std()
  for col in df.columns:
    df[col] = (df[col] - avgs.loc[col]) / devs.loc[col]
  return df, avgs, devs

def remove_volatility_seasonality(df=None):
  """Function to remove volatility and easonality of time series
  - Paramters:
    df: DataFrame or Series
  - Return value:
    df: DataFrame with volatility removed
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Get all the column names
  column_names = df.columns.values
```

```
  # Remove volatility
  daily_volatility = df.groupby(df.index.day).std()
  for i, column_name in enumerate(column_names):
    df[column_name + '_daily_vol'] = df.index.map(lambda d:
        daily_volatility.loc[d.day, column_name])
    df[column_name] = df[column_name] / df[column_name + '_daily_vol']
  # Remove seasonality
  daily_avgs = df.groupby(df.index.day).mean()
  for i, column_name in enumerate(column_names):
    df[column_name + '_daily_avg'] = df.index.map(lambda d:
        daily_avgs.loc[d.day, column_name])
    df[column_name] = df[column_name] - df[column_name + '_daily_avg']
  return df

def Reconvert_prediction(df_to_convert=None, df_trans=None, avgs=None,
    devs=None):
  """Function to reconvert the prediction result
  - Paramters:
    df_to_convert: DataFrame or Series, prediction result
    df_trans: DataFrame or Series, time series with volatility and easonality
        removed
    avgs: average of each column
    devs: deviation of each column
  - Return value:
    df: DataFrame reconverted
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df_to_convert, pd.DataFrame):
    df_to_convert = df_to_convert.to_frame()
  if not isinstance(df_trans, pd.DataFrame):
    df_trans = df_trans.to_frame()
  # Get all the column names
  column_names = df_to_convert.columns.values
  for col in column_names:
    df_to_convert[col] = (df_to_convert[col] + df_trans[col + '_daily_avg']) *
        df_trans[col + '_daily_vol']
    df_to_convert[col] = df_to_convert[col] * devs.loc[col] + avgs.loc[col]
  return df_to_convert
# ======================*END - General functions*======================


# ====================*BLOC - SARIMAX Model functions*====================
def grid_search_AIC_sarimax(df=None, exog=None, p=range(0,2), d=range(0,2),
    q=(0,2), P=range(0,3), D=range(0,3), Q=range(0,3), seasonal_lags=None,
    return_aic=False):
  """Function to determine the best orders based on the possible values we
     found by ACF and PACF plot
  - Paramters:
```

```
  df: DataFrame or Series
  exog: Array of exogenous regressors
  p: range of p in list, default (0,2)
  d: range of d in list, default (0,2)
  q: range of q in list, default (0,2)
  P: range of P in list, default (0,3)
  D: range of D in list, default (0,3)
  Q: range of Q in list, default (0,3)
  seasonal_lags: seasonal lags of data
  return_aic: is or not to return AIC evaluation dataframe
- Return value:
  pdq: trend orders
  seasonal_pdq: seasonal orders
  order_aic: AIC search result
"""
# Transform Series objet to DataFrame object
if not isinstance(df, pd.DataFrame):
  df = df.to_frame()
p, d, q, P, D, Q = p, d, q, P, D, Q
# Get all the possible combination of orders
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], seasonal_lags) for x in
    list(itertools.product(P, D, Q))]
# Initialize the AIC evaluation dict
order_aic = {}
for param in pdq:
  for param_seasonal in seasonal_pdq:
    try:
      model = SARIMAX(df, exog=exog, order=param,
          seasonal_order=param_seasonal)
      results = model.fit()
      order_aic['Order'] = [param]
      order_aic['Seasonal order'] = [param_seasonal]
      order_aic['AIC'] = [results.aic]
    except:
      continue
# Transform AIC evaluation dict to dataframe
order_aic = pd.DataFrame.from_dict(order_aic)
# Affect the column name
order_aic.columns=['Order', 'Seasonal order', 'AIC']
# Get the minimum AIC tuple
order_aic_min = order_aic.loc[np.where(order_aic['AIC'] ==
    order_aic['AIC'].min())]
# Get the best orders and seasonal orders
pdq = order_aic_min.iloc[0]['Order']
seasonal_pdq = order_aic_min.iloc[0]['Seasonal order']
if return_aic:
  return pdq, seasonal_pdq, order_aic
```

```
  else:
    return pdq, seasonal_pdq


def fit_sarimax(df=None, exog=None, order=None, seasonal_order=None,
    trend=None, figsize=(10, 4), enforce_stationarity=True,
    enforce_invertibility=True):
  """Function of fitting SARIMAX to data
  - Paramters:
    df: DataFrame or Series
    exog: Array of exogenous regressors
    order: trend orders
    seasonal_order: seasonal orders
    trend: Parameter controlling the deterministic trend polynomial,
       {n, c , t , ct }
    figsize: size the plot figure
  - Return value:
    fitted model
  """
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  # Fit SARIMAX model
  model = SARIMAX(df, exog=exog, order=order, seasonal_order=seasonal_order,
      trend=trend, enforce_stationarity=enforce_stationarity,
      enforce_invertibility=enforce_invertibility).fit()
  print(model.summary())
  model.plot_diagnostics(figsize=figsize)
  plt.tight_layout()
  plt.show()
  return model


def residual_check_sarimax(df=None, lags=None, figsize=(8,2)):
  """Function to use Ljung Box test to inspect if the residuals are correlated
  - Parameters:
    df: DataFrame or Series
    lags: lags to take
    figsize: size of figure
  - Return value: None
  """
  # Perform the Ljung-Box test
  lb_test = acorr_ljungbox(df, lags=lags)
  p_value = pd.DataFrame(lb_test[1])
  # Plot the p-values
  fig, ax = plt.subplots(figsize=figsize)
  ax.plot(p_value, linewidth=3, linestyle='--', label="Residual
      std={:.2f}".format(df.std()))
```

```
  ax.set_xlabel('Lags', fontsize=12)
  ax.set_ylabel('P-value', fontsize=12)
  ax.set_title('Ljung Box test - residual autocorrelation', fontsize=15)
  ax.legend(loc='best')
  plt.show()
  return


def predict_plot_sarimax(model=None, df_original=None, start=None,
    df_to_predict=None, exog=None, return_predict=False, return_rmse=False,
    zoom=False, zoom_start_at=None, zoom_end_at=None, figsize=(10, 2)):
  """Function to make predictions on testing set and visualize the result at
      one time, for the case no nomalization on the data
  - Parameters:
    model: fitted model
    df_original: DataFrame or Series of original data
    start: Int, str, or datetime, set for in-sample forecast
    df_to_predict: DataFrame or Series to predict, set for out-of-sample
        forceast
    plot: is or not to plot both orgianal data and prediction result
    return_predict: is or not to return predict value in Dataframe
    return_rmse: is or not to return rmse evalution
    zoom: zoom the prediction result for a selected period
    zoom_start_at: str or datetime, start time point to zoom
    zoom_end_at: str or datetime, end time point to zoom
    figsize: size of figure, (10,2) default
  - Return value:
    predicted_values: result of predictions
    rmse: Root Mean Square Error of prediction
  """
  fig, ax = plt.subplots(figsize=figsize)
  if start is not None:
      # One-step-ahead prediction
    forecast_osa_in = model.get_prediction(start=start, exog=exog)
    mean_forecast_osa_in = forecast_osa_in.predicted_mean.to_frame()
    mean_forecast_osa_in.columns = [df_original.columns.values[0]]
    osa_ci_in = forecast_osa_in.conf_int()
      # Dynamic Prediction
    forecast_dyn_in = model.get_prediction(start=start, exog=exog,
        dynamic=True)
    mean_forecast_dyn_in = forecast_dyn_in.predicted_mean
    mean_forecast_dyn_in.columns = [df_original.columns.values[0]]
    dyn_ci_in = forecast_dyn_in.conf_int()
      # Calculate the Root Mean Square Error of prediction
    rmse_osa_in = np.sqrt(np.mean(np.square(df_original[start:].values -
        mean_forecast_osa_in.values)))
    rmse_dyn_in = np.sqrt(np.mean(np.square(df_original[start:].values -
        mean_forecast_dyn_in.values)))
```

```
  # Visualize the predcitions result
if df_original is not None:
    # Transform Series objet to DataFrame object
  if not isinstance(df_original, pd.DataFrame):
    df_original = df_original.to_frame()
  ax.plot(df_original[start:], linewidth=3, label='observation')
ax.plot(mean_forecast_osa_in.index, mean_forecast_osa_in.values, 'r-+',
    linewidth=3, label="One-step-ahead forecast
    (RMSE={:0.2f})".format(rmse_osa_in))
ax.fill_between(osa_ci_in.index, osa_ci_in.iloc[:,0], osa_ci_in.iloc[:,1],
    color='r', alpha=0.05)
ax.plot(mean_forecast_dyn_in.index, mean_forecast_dyn_in.values, 'g--',
    linewidth=3, label="Dynamic forecast
    (RMSE={:0.2f})".format(rmse_dyn_in))
ax.fill_between(dyn_ci_in.index, dyn_ci_in.iloc[:,0], dyn_ci_in.iloc[:,1],
    color='g', alpha=0.05)
ax.legend(loc='best', fontsize=9)
ax.set_title("In-sampling prediction", fontsize=22, fontweight="bold")
ax.set_xlabel('Time', fontsize=18)
ax.set_ylabel(str(df_original.columns.values[0])+'-AQI', fontsize=18)
 # Zoom the prediction result for a indicated period
if zoom:
  fig, ax_zoom = plt.subplots(figsize=figsize)
  zoom_start_at = pd.to_datetime(zoom_start_at)
  zoom_end_at = pd.to_datetime(zoom_end_at)
  if df_original is not None:
    # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
    df_original = df_original.loc[zoom_start_at:zoom_end_at]
    ax_zoom.plot(df_original, linewidth=3, label='observation')
    # Transform datetime string to Timestampe
  df_forecast_osa_in = mean_forecast_dyn_in.loc[zoom_start_at:zoom_end_at]
  df_forecast_dyn_in = mean_forecast_dyn_in.loc[zoom_start_at:zoom_end_at]
  rmse_osa_in = np.sqrt(np.mean(np.square(df_original.values -
      mean_forecast_dyn_in.values)))
  rmse_dyn_in = np.sqrt(np.mean(np.square(df_original.values -
      mean_forecast_dyn_in.values)))
  ax_zoom.plot(df_forecast_osa_in.index, df_forecast_osa_in.values, 'r-+',
      linewidth=3, label="One-step-ahead forecast
      (RMSE={:0.2f})".format(rmse_osa_in))
  ax_zoom.plot(df_forecast_dyn_in.index, df_forecast_dyn_in.values, 'g--',
      linewidth=3, label="Dynamic forecast
      (RMSE={:0.2f})".format(rmse_dyn_in))
  ax_zoom.legend(loc='best', fontsize=9)
  ax_zoom.set_title("Zoom prediction from " + str(zoom_start_at) + " to "
      + str(zoom_end_at), fontsize=14, fontweight="bold")
  ax_zoom.set_xlabel('Time', fontsize=18)
```

```
    ax_zoom.set_ylabel(df_forecast_dyn_in.columns.values[0]+'-AQI',
        fontsize=18)
    for index in df_forecast_dyn_in.index:
      ax_zoom.axvline(index, linestyle='--', color='k', alpha=0.2)
    ax.axvspan(zoom_start_at, zoom_end_at, color='red', alpha=0.05)
  plt.show()
  if return_predict == True and return_rmse==True:
    return mean_forecast_osa_in, mean_forecast_dyn_in, rmse_osa_in,
        rmse_dyn_in
  if return_predict == True and return_rmse==False:
    return mean_forecast_osa_in, mean_forecast_dyn_in
  if return_predict == False and return_rmse==False:
    return
  # Out-of-sample prediction and confidence bounds
if df_to_predict is not None:
  if not isinstance(df_to_predict, pd.DataFrame):
    df_to_predict = df_to_predict.to_frame()
    # One-step-ahead prediction
  forecast_osa_out =
      model.get_prediction(start=df_to_predict.index[0],end=df_to_predict.index[-1],
      exog=exog)
  mean_forecast_osa_out = forecast_osa_out.predicted_mean.to_frame()
  mean_forecast_osa_out.columns = [df_to_predict.columns.values[0]]
  osa_ci_out = forecast_osa_out.conf_int()
    # Dynamic Prediction
  forecast_dyn_out =
      model.get_prediction(start=df_to_predict.index[0],end=df_to_predict.index[-1],
      exog=exog, dynamic=True)
  mean_forecast_dyn_out = forecast_dyn_out.predicted_mean.to_frame()
  mean_forecast_dyn_out.columns = [df_to_predict.columns.values[0]]
  dyn_ci_out = forecast_dyn_out.conf_int()
    # Calculate the Root Mean Square Error of prediction
  rmse_osa_out = np.sqrt(np.mean(np.square(df_to_predict.values -
      mean_forecast_osa_out.values)))
  rmse_dyn_out = np.sqrt(np.mean(np.square(df_to_predict.values -
      mean_forecast_dyn_out.values)))
    # Visualize the predcitions result
  if df_original is not None:
    # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
    ax.plot(df_original[df_to_predict.index[0]:df_to_predict.index[-1]],
        linewidth=3, label='observation')
  ax.plot(mean_forecast_osa_out.index, mean_forecast_osa_out.values, 'r-+',
      linewidth=3, label="One-step-ahead forecast
      (RMSE={:0.2f})".format(rmse_osa_out))
  ax.fill_between(osa_ci_out.index, osa_ci_out.iloc[:,0],
      osa_ci_out.iloc[:,1], color='r', alpha=0.05)
```

```
ax.plot(mean_forecast_dyn_out.index, mean_forecast_dyn_out.values, 'g--',
    linewidth=3, label="Dynamic forecast
    (RMSE={:0.2f})".format(rmse_dyn_out))
ax.fill_between(dyn_ci_out.index, dyn_ci_out.iloc[:,0],
    dyn_ci_out.iloc[:,1], color='g', alpha=0.05)
ax.legend(loc='best', fontsize=9)
ax.set_title("Out-of-sampling prediction", fontsize=22, fontweight="bold")
ax.set_xlabel('Time', fontsize=18)
ax.set_ylabel(str(df_to_predict.columns.values[0])+'-AQI', fontsize=18)
plt.show()
# Zoom the prediction result for a indicated period
if zoom:
  fig, ax_zoom = plt.subplots(figsize=figsize)
  # Transform datetime string to Timestampe
  zoom_start_at = pd.to_datetime(zoom_start_at)
  zoom_end_at = pd.to_datetime(zoom_end_at)
  if df_original is not None:
    # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
    df_original = df_original.loc[zoom_start_at:zoom_end_at]
    ax_zoom.plot(df_original, linewidth=3, label='observation')
  df_forecast_osa_out =
      mean_forecast_osa_out.loc[zoom_start_at:zoom_end_at]
  df_forecast_dyn_out =
      mean_forecast_dyn_out.loc[zoom_start_at:zoom_end_at]
  rmse_osa_out = np.sqrt(np.mean(np.square(df_original.values -
      df_forecast_osa_out.values)))
  rmse_dyn_out = np.sqrt(np.mean(np.square(df_original.values -
      df_forecast_dyn_out.values)))
  ax_zoom.plot(df_forecast_osa_out.index, df_forecast_osa_out.values,
      'r-+', linewidth=3, label="One-step-ahead forecast
      (RMSE={:0.2f})".format(rmse_osa_out))
  ax_zoom.plot(df_forecast_dyn_out.index, df_forecast_dyn_out.values,
      'g--', linewidth=3, label="Dynamic forecast
      (RMSE={:0.2f})".format(rmse_dyn_out))
  ax_zoom.legend(loc='best', fontsize=9)
  ax_zoom.set_title("Zoom prediction from " + str(zoom_start_at) + " to "
      + str(zoom_end_at), fontsize=14, fontweight="bold")
  ax_zoom.set_xlabel('Time', fontsize=18)
  ax_zoom.set_ylabel(df_forecast_dyn_out.columns.values[0]+'-AQI',
      fontsize=18)
  for index in df_forecast_dyn_out.index:
    ax_zoom.axvline(index, linestyle='--', color='k', alpha=0.2)
  ax.axvspan(zoom_start_at, zoom_end_at, color='red', alpha=0.05)
if return_predict == True and return_rmse==True:
  return mean_forecast_osa_out, mean_forecast_dyn_out, rmse_osa_out,
      rmse_dyn_out
```

```
    if return_predict == True and return_rmse==False:
      return mean_forecast_osa_out, mean_forecast_dyn_out
    if return_predict == False and return_rmse==False:
      return


def predict_sarimax(model=None, start_at_in_sampling=None,
    start_at_out_sampling=None, end_at_out_sampling=None, exog=None):
  """Function to make predictions on testing set and return prediction
      result(do not return the confidence interval)
  - Parameters:
    model: fitted model
    start_at_in_sampling: Int, str, or datetime, set for in-sampling forecast
    start_at_out_sampling: str or datetime, set for out-sampling forecast
    end_at_out_sampling: str or datetime, set for out-sampling forecast
    exog: Series or DataFrame, exogenous variable
  - Return value:
    predicted_values: result of predictions
  """
  if start_at_in_sampling is not None: # In_sampling forecast
      # One-step-ahead prediction
    forecast_osa_in = model.get_prediction(start=start_at_in_sampling,
        exog=exog)
    mean_forecast_osa_in = forecast_osa_in.predicted_mean.to_frame()
    # osa_ci_in = forecast_osa_in.conf_int() # Get the confidence interval
      # Dynamic Prediction
    forecast_dyn_in = model.get_prediction(start=start_at_in_sampling,
        exog=exog, dynamic=True)
    mean_forecast_dyn_in = forecast_dyn_in.predicted_mean.to_frame()
    # dyn_ci_in = forecast_dyn_in.conf_int()
    # return mean_forecast_osa_in, osa_ci_in, mean_forecast_dyn_in, dyn_ci_in
    return mean_forecast_osa_in, mean_forecast_dyn_in
  if start_at_out_sampling is not None and end_at_out_sampling is not None:
    start_at_out_sampling = pd.to_datetime(start_at_out_sampling)
    end_at_out_sampling = pd.to_datetime(end_at_out_sampling)
      # Dynamic Prediction
    forecast_dyn_out = model.get_prediction(start=start_at_out_sampling,
        end=end_at_out_sampling, exog=exog, dynamic=True)
    mean_forecast_dyn_out = forecast_dyn_out.predicted_mean.to_frame()
    # dyn_ci_out = forecast_dyn_out.conf_int()
    # return mean_forecast_dyn_out, dyn_ci_out
    return mean_forecast_dyn_out


def prediction_plot_sarimax(df_original=None, df_forecast=None,
    figsize=(10,2), zoom=False, zoom_start_at=None, zoom_end_at=None):
    '''
    Function to visualize the prediction result
```

```
- Parameters:
  df_original: DataFrame or Series of original data
  df_forecast: DataFrame or Series to predict, set for out-of-sample
      forceast
  figsize: size of figure, (12,4) default
  zoom: zoom the prediction result for a selected period
  zoom_start_at: str or datetime, start time point to zoom
  zoom_end_at: str or datetime, end time point to zoom
- Return value: None
'''
fig, ax = plt.subplots(figsize=figsize)
 # In-sample prediction and confidence bounds
if len(df_forecast) == 2:
  df_forecast_osa_in = df_forecast[0]
  df_forecast_dyn_in = df_forecast[1]
    # Transform Series objet to DataFrame object
  if not isinstance(df_forecast_osa_in, pd.DataFrame):
    df_forecast_osa_in = df_forecast_osa_in.to_frame()
  if not isinstance(df_forecast_dyn_in, pd.DataFrame):
    df_forecast_dyn_in = df_forecast_dyn_in.to_frame()
  if df_original is not None:
      # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
      # Calculate the Root Mean Square Error of prediction
    rmse_osa_in = np.sqrt(np.mean(np.square(df_original.values -
        df_forecast_osa_in.values)))
    rmse_dyn_in = np.sqrt(np.mean(np.square(df_original.values -
        df_forecast_dyn_in.values)))
    ax.plot(df_original, linewidth=3, label='observation')
    ax.plot(df_forecast_osa_in.index, df_forecast_osa_in.iloc[:,0].values,
        'r-+', linewidth=3, label="One-step-ahead forecast
        (RMSE={:0.2f})".format(rmse_osa_in))
    ax.plot(df_forecast_dyn_in.index, df_forecast_dyn_in.iloc[:,0].values,
        'g--', linewidth=3, label="Dynamic forecast
        (RMSE={:0.2f})".format(rmse_dyn_in))
  else:
    ax.plot(df_forecast_osa_in.index, df_forecast_osa_in.iloc[:,0].values,
        'r-+', linewidth=3, label="One-step-ahead forecast")
    ax.plot(df_forecast_dyn_in.index, df_forecast_dyn_in.iloc[:,0].values,
        'g--', linewidth=3, label="Dynamic forecast")
  # ax.fill_between(df_forecast_osa_in.index,
      df_forecast_osa_in.iloc[:,1], df_forecast_osa_in.iloc[:,2],
      color='r', alpha=0.05)
  # ax.fill_between(df_forecast_dyn_in.index,
      df_forecast_dyn_in.iloc[:,1], df_forecast_dyn_in.iloc[:,2],
      color='g', alpha=0.05)
  ax.legend(loc='best', fontsize=9)
```

```
  ax.set_title("In-sampling prediction", fontsize=22, fontweight="bold")
  ax.set_xlabel('Time', fontsize=18)
  ax.set_ylabel(str(df_forecast_osa_in.columns.values[0])+'-AQI',
      fontsize=18)
  # Zoom the prediction result for a indicated period
  if zoom:
    fig, ax_zoom = plt.subplots(figsize=figsize)
      # Transform datetime string to Timestampe
    zoom_start_at = pd.to_datetime(zoom_start_at)
    zoom_end_at = pd.to_datetime(zoom_end_at)
    df_forecast_osa_in = df_forecast_osa_in.loc[zoom_start_at:zoom_end_at]
    df_forecast_dyn_in = df_forecast_dyn_in.loc[zoom_start_at:zoom_end_at]
    if df_original is not None:
      # Transform Series objet to DataFrame object
      if not isinstance(df_original, pd.DataFrame):
        df_original = df_original.to_frame()
      df_original = df_original.loc[zoom_start_at:zoom_end_at]
      # Calculate the Root Mean Square Error of prediction
      rmse_osa_in = np.sqrt(np.mean(np.square(df_original.values -
          df_forecast_osa_in.values)))
      rmse_dyn_in = np.sqrt(np.mean(np.square(df_original.values -
          df_forecast_dyn_in.values)))
      ax_zoom.plot(df_original, linewidth=3, label='observation')
      ax_zoom.plot(df_forecast_osa_in.index,
          df_forecast_osa_in.iloc[:,0].values, 'r-+', linewidth=3,
          label="One-step-ahead forecast
          (RMSE={:0.2f})".format(rmse_osa_in))
      ax_zoom.plot(df_forecast_dyn_in.index,
          df_forecast_dyn_in.iloc[:,0].values, 'g--', linewidth=3,
          label="Dynamic forecast (RMSE={:0.2f})".format(rmse_dyn_in))
    else:
      ax_zoom.plot(df_forecast_osa_in.index,
          df_forecast_osa_in.iloc[:,0].values, 'r-+', linewidth=3,
          label="One-step-ahead forecast")
      ax_zoom.plot(df_forecast_dyn_in.index,
          df_forecast_dyn_in.iloc[:,0].values, 'g--', linewidth=3,
          label="Dynamic forecast")
    ax_zoom.legend(loc='best', fontsize=9)
    ax_zoom.set_title("Zoom prediction from " + str(zoom_start_at) + " to
        " + str(zoom_end_at), fontsize=14, fontweight="bold")
    ax_zoom.set_xlabel('Time', fontsize=18)
    ax_zoom.set_ylabel(df_forecast_dyn_in.columns.values[0]+'-AQI',
        fontsize=18)
    for index in df_forecast_dyn_in.index:
      ax_zoom.axvline(index, linestyle='--', color='k', alpha=0.2)
    ax.axvspan(zoom_start_at, zoom_end_at, color='red', alpha=0.05)
else:
  df_forecast_dyn_out = df_forecast
```

```
  # Transform Series objet to DataFrame object
if not isinstance(df_forecast_dyn_out, pd.DataFrame):
  df_forecast_dyn_out = df_forecast_dyn_out.to_frame()
if df_original is not None:
    # Transform Series objet to DataFrame object
  if not isinstance(df_original, pd.DataFrame):
    df_original = df_original.to_frame()
  rmse_dyn_out = np.sqrt(np.mean(np.square(df_original.values -
      df_forecast_dyn_out.values)))
  ax.plot(df_original, linewidth=3, label='observation')
  ax.plot(df_forecast_dyn_out.index, df_forecast_dyn_out.values, 'g--',
      linewidth=3, label="Dynamic forecast
      (RMSE={:0.2f})".format(rmse_dyn_out))
else:
  ax.plot(df_forecast_dyn_out.index, df_forecast_dyn_out.values, 'g--',
      linewidth=3,label="Dynamic forecast")
# ax.fill_between(df_forecast_dyn_out.index,
    df_forecast_dyn_out.iloc[:,1], df_forecast_dyn_out[:,2], color='g',
    alpha=0.05)
ax.legend(loc='best', fontsize=9)
ax.set_title("Out-of-sampling prediction", fontsize=22,
    fontweight="bold")
ax.set_xlabel('Time', fontsize=18)
ax.set_ylabel(str(df_forecast_dyn_out.columns.values[0])+'-AQI',
    fontsize=18)
if zoom:
  fig, ax_zoom = plt.subplots(figsize=figsize)
    # Transform datetime string to Timestampe
  zoom_start_at = pd.to_datetime(zoom_start_at)
  zoom_end_at = pd.to_datetime(zoom_end_at)
  df_forecast_dyn_out =
      df_forecast_dyn_out.loc[zoom_start_at:zoom_end_at]
  if df_original is not None:
    # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
    df_original = df_original.loc[zoom_start_at:zoom_end_at]
    # Calculate the Root Mean Square Error of prediction
    rmse_dyn_out = np.sqrt(np.mean(np.square(df_original.values -
        df_forecast_dyn_out.values)))
    ax_zoom.plot(df_original, linewidth=3, label='observation')
    ax_zoom.plot(df_forecast_dyn_out.index,
        df_forecast_dyn_out.iloc[:,0].values, 'g--', linewidth=3,
        label="Dynamic forecast (RMSE={:0.2f})".format(rmse_dyn_out))
  else:
    ax_zoom.plot(df_forecast_dyn_out.index,
        df_forecast_dyn_out.iloc[:,0].values, 'g--', linewidth=3,
        label="Dynamic forecast")
```

```
        ax_zoom.legend(loc='best', fontsize=9)
        ax_zoom.set_title("Zoom prediction from " + str(zoom_start_at) + " to
            " + str(zoom_end_at), fontsize=14, fontweight="bold")
        ax_zoom.set_xlabel('Time', fontsize=18)
        ax_zoom.set_ylabel(df_forecast_dyn_out.columns.values[0]+'-AQI',
            fontsize=18)
        for index in df_forecast_dyn_out.index:
          ax_zoom.axvline(index, linestyle='--', color='k', alpha=0.2)
        ax.axvspan(zoom_start_at, zoom_end_at, color='red', alpha=0.05)
    plt.show()
    return
# ======================*END - SARIMAX Model functions*====================


# ======================*BLOC - VAR Model functions*======================
def grangers_causation_matrix(df=None, test='ssr_chi2test', maxlag=12,
    verbose=False):
    """Funtion to Check Granger Causality of all possible combinations of the
        time series
    - Parameters:
      df: Dataframe
      maxlag: the Granger causality test results are calculated for all lags
          up to maxlag
      verbose: print results if true
    - Return:
      df_p_value: test P-value matrix
    - Remark:
      The rows are the response variable, columns are predictors. The values
          in the table
      are the P-values. P-values lesser than the significance level (0.05),
          implies
      the Null Hypothesis that the coefficients of the corresponding past
          values is
      zero, that is, the X does not cause Y can be rejected.
    """
    # Transform Series objet to DataFrame object
    if not isinstance(df, pd.DataFrame):
      df = df.to_frame()
    df_p_value = pd.DataFrame(np.zeros((len(df.columns), len(df.columns))),
        columns=df.columns, index=df.columns)
    for column in df.columns:
        for row in df.columns:
            test_result = grangercausalitytests(df[[row, column]],
                maxlag=maxlag, verbose=False)
            p_values = [round(test_result[i+1][0][test][1],4) for i in
                range(maxlag)]
            if verbose:
              print(f'Y = {row}, X = {column}, P Values = {p_values}')
```

```
            min_p_value = np.min(p_values)
            df_p_value.loc[row, column] = min_p_value
    df_p_value.columns = [var + '_x' for var in df.columns]
    df_p_value.index = [var + '_y' for var in df.columns]
    return df_p_value


def cointegration_test(df=None, det_order=-1, k_ar_diff=5, alpha=0.05):
    """Perform Johanson's Cointegration Test and Report Summary
    - Parameters:
      df: DataFrame or Series
      det_order: ['-1 - no deterministic terms', '0 - constant term', '1 -
          linear trend']
      k_ar_diff: Number of lagged differences in the model
      alpha: significance level
    - Return value: None
    """
    # Transform Series objet to DataFrame object
    if not isinstance(df, pd.DataFrame):
      df = df.to_frame()
    result = coint_johansen(endog=df, det_order=det_order, k_ar_diff=k_ar_diff
        )
    d = {'0.90':0, '0.95':1, '0.99':2}
    traces = result.lr1 # Trace statistic
    cvts = result.cvt[:, d[str(1-alpha)]] # Critical values (90%, 95%, 99%)
        for trace statistic
    def adjust(val, length= 6):
      return str(val).ljust(length)
    # Prit the summary result
    print('Name  |  Test Statistic | Critical Value({}%) | Cointegrated
        \n'.format(int(100*(1-alpha))), '--'*30)
    for column, trace, cvt in zip(df.columns, traces, cvts):
        print(adjust(column), '| ', adjust(round(trace,2), 14), "|",
            adjust(cvt, 19), '|' , trace > cvt)


def prediction_plot_var(df_original=None, df_forecast=None, figsize=(10,2),
    zoom=False, zoom_start_at=None, zoom_end_at=None, title="VAR
    out-of-sampling prediction"):
  """Function to make predictions on testing set
  - Parameters:
    df_original: DataFrame or Series of original data
    df_forecast: DataFrame or Series to predict, out-of-sample forceast result
    figsize: size of figure, (10,2) default
    zoom: zoom the prediction result for a selected period
    zoom_start_at: str or datetime, start time point to zoom
    zoom_end_at: str or datetime, end time point to zoom
  - Return value: None
```

```
    """
    # Transform datetime string to Timestampe
    zoom_start_at = pd.to_datetime(zoom_start_at)
    zoom_end_at = pd.to_datetime(zoom_end_at)
    # Transform Series objet to DataFrame object
    if not isinstance(df_original, pd.DataFrame):
      df_original = df_original.to_frame()
    if not isinstance(df_forecast, pd.DataFrame):
      df_forecast = df_forecast.to_frame()
    # Calculate the Root Mean Square Error of prediction
    rmse = np.sqrt(np.mean(np.square(df_original.values - df_forecast.values)))
    fig, ax = plt.subplots(figsize=figsize)
    if df_original is not None:
      ax.plot(df_original, linewidth=3, label='observation')
    ax.plot(df_forecast.index, df_forecast.values, 'r--', linewidth=3,
        label="Dynamic forecast (RMSE={:0.2f})".format(rmse))
    ax.legend(loc='best', fontsize=9)
    ax.set_title(title, fontsize=22, fontweight="bold")
    ax.set_xlabel('Time', fontsize=18)
    ax.set_ylabel(df_original.columns.values[0]+'-AQI', fontsize=18)
    # Zoom the prediction result for a indicated period
    if zoom:
      df_original = df_original.loc[zoom_start_at:zoom_end_at]
      df_forecast = df_forecast.loc[zoom_start_at:zoom_end_at]
      fig, ax_zoom = plt.subplots(figsize=figsize)
      rmse = np.sqrt(np.mean(np.square(df_original.values - df_forecast.values)))
      if df_original is not None:
        ax_zoom.plot(df_original, linewidth=3, label='observation')
      ax_zoom.plot(df_forecast.index, df_forecast.values, 'r--', linewidth=3,
          label="Dynamic forecast (RMSE={:0.2f})".format(rmse))
      ax_zoom.legend(loc='best', fontsize=9)
      ax_zoom.set_title("Zoom prediction from " + str(zoom_start_at) + " to " +
          str(zoom_end_at), fontsize=14, fontweight="bold")
      ax_zoom.set_xlabel('Time', fontsize=18)
      ax_zoom.set_ylabel(df_original.columns.values[0]+'-AQI', fontsize=18)
      for index in df_forecast.index:
        ax_zoom.axvline(index, linestyle='--', color='k', alpha=0.2)
      ax.axvspan(zoom_start_at, zoom_end_at, color='red', alpha=0.05)
    return
# ======================*END - VAR Model functions*======================


# ======================*BLOC - RNN Model functions*======================
def preparation_data_rnn(df=None, lags=None):
    """Function to preprare taining and testing set for RNN model based on
       seasonal lags
    - Parameters:
      df: DataFrame or Series
```

```
    lags: seasonal lags
  - Retrun value:
    X: 1D array of inputing data
    y: 1D array of target value
  """
  X, y = [], []
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  for row in range(len(df) - lags):
  # Transform 2D data to 1D
    x_1d_array = []
    for i in zip(df.values[row:(row + lags)]):
      x_1d_array.append(i[0][0])
    X.append(x_1d_array)
    y_1d_array = []
    for i in zip(df.values[row + lags]):
      y_1d_array.append(i[0])
    y.append(y_1d_array)
  return np.array(X), np.array(y)


def convert_to_dataframe_rnn(Y=None, y=None):
  """Function to convert Array into Dataframe
  - Parameters:
    Y: Original testing set based on which implementing the split (DataFrame)
    y: Array data to convert
  - Return value:
    y: Dataframe converted
  """
  # Transform Series objet to DataFrame object
  if not isinstance(Y, pd.DataFrame):
    Y = Y.to_frame()
  # Convert y to DataFrame
  y = pd.DataFrame(y)
  # Get the right index of y
  y.index = Y.index[len(Y)-len(y):]
  # Set the columns name of y
  y.columns = Y.columns
  return y


# Fitting the RNN model
def fit_RNN(X=None, y=None, epochs=50, batch_size=None, verboses=2):
  '''Function to fit RNN model to data
  - Paramter:
    X: Input data, Array
    y: Target data, Array
```

```
    epochs: Number of epochs to train the model, default 50
    batch_size: Number of samples per gradient update
    verbose: Verbosity mode, verbose=2 is recommended when not running
        interactively
  - Return value:
    model: fitted model
  '''
  model = Sequential()
  model.add(Dense(4, activation='relu'))
  model.add(Dense(8, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mean_squared_error', optimizer='adam')
  model.fit(x=X, y=y, epochs=epochs, batch_size=batch_size, verbose=verboses)
  return model


def predict_rnn(df_original=None, df_test=None, df_predict=None, figsize=(12,
    4), plot=False, return_rmse=False, return_dataframe=False, zoom=False):
  '''Function to plot the prediction result
  - Parameters:
    df_original: DataFrame or Series of original data
    df_test: true value, Array or DataFrame
    df_predict: predicted value, Array or DataFrame
    figsize: size of plot figure
    plot: is or not to plot both orgianal data and prediction result
    return_rmse: is or not to return rmse of prediction
    zoom: is or not to zoom the prediction result part
  - Return value:
    return_rmse: Root Mean Square Error of prediction / df_test and
        df_predict(DataFrame)
  '''
  # Transform Series objet to DataFrame object
  if not isinstance(df_original, pd.DataFrame):
    df_original = df_original.to_frame()
  if not isinstance(df_test, pd.DataFrame):
    df_test = convert_to_dataframe_rnn(df_original, df_test)
  if not isinstance(df_predict, pd.DataFrame):
    df_predict = convert_to_dataframe_rnn(df_original, df_predict)
  # Calculate the Root Mean Square Error of prediction
  rmse = np.sqrt(np.mean(np.square(df_test.values - df_predict.values)))
  if plot:
    # Plot in-sample-prediction
    fig, ax = plt.subplots(figsize=figsize)
    ax.plot(df_original, linewidth=3, label='Observation')
    ax.plot(df_test.index, df_test.values, linewidth=3, label='Truth')
    ax.plot(df_predict.index, df_predict.values, linewidth=3,
        label="prediction (RMSE={:0.2f})".format(rmse))
    ax.set_xlabel('Time', fontsize=18)
```

```
    ax.set_ylabel(str(df_test.columns.values[0])+'/ppb', fontsize=18)
    ax.set_title('Prediction RNN', fontsize=22, fontweight="bold")
    plt.legend(loc='upper left')
    plt.show()
  if zoom:
    fig1, ax1 = plt.subplots(figsize=figsize)
    ax1.plot(df_test.index, df_test.values, linewidth=3, label='Truth')
    ax1.plot(df_predict.index, df_predict.values, linewidth=3,
        label="prediction (RMSE={:0.2f})".format(rmse))
    ax1.set_xlabel('Time', fontsize=18)
    ax1.set_ylabel(str(df_test.columns.values[0])+'/ppb', fontsize=18)
    ax1.set_title('Prediction RNN', fontsize=22, fontweight="bold")
    plt.legend(loc='upper left')
    plt.show()
  if return_rmse == True and return_dataframe == False:
    return rmse
  if return_rmse == False and return_dataframe == True:
    return df_test, df_predict
  if return_rmse == True and return_dataframe == True:
    return rmse, df_test, df_predict
  if return_rmse == False and return_dataframe == False:
    return


def predict_future_rnn(df=None, model=None, start=None, end=None, lags=None,
    figsize=(10,4), return_result=False, plot=False, width=0.03,
    title='Predictions', min_max_distance=0.1):
  '''Function to predict by RNN model for a future new period
  - Paramters:
    df: DataFrame or Series
    model: fitted model
    start: start time point to display(String or Timestampe)
    end: end time point to display(String or Timestampe)
    lags: seasonal lags
    figsize: size of figure, default (10,4)
    return_result: is or not to return prections values
    plot: is or not to plot dashbord
    width: control the width between each bar, default 0.03
    title: set tilte of plot
    min_max_distance: used to modeify the position of MIN and MAX value
  - Return value:
    predict_data: prediction values
  '''
  # Transform Series objet to DataFrame object
  if not isinstance(df, pd.DataFrame):
    df = df.to_frame()
  df = df.resample('H').mean()
  # Transform datetime string to Timestampe
```

```
  start = pd.to_datetime(start)
  end = pd.to_datetime(end)
  length = int((end-start)/datetime.timedelta(1/24) + 1)
  # Find the index and value of 'lags' previous data
  lags_data, predict_data = [], []
  for j in range(lags):
    index = start - datetime.timedelta((lags-j)/24)
    value = df.loc[index]
    lags_data.append(value[0])
  predict_value = model.predict(np.array([lags_data]))
  lags_data.append(predict_value[0,0])
  predict_data.append(predict_value)
  # Get all the prediction value
  for i in range(length-1):
    temp = []
    for j in range(lags):
      temp.append(lags_data[len(lags_data) - (lags-j)])
    predict_temp = model.predict(np.array([temp]))
    lags_data.append(predict_temp[0,0])
    predict_data.append(predict_temp)
  predict_data = np.array(predict_data).reshape(length,1)
  # Convert y to DataFrame
  predict_data = pd.DataFrame(predict_data)
  # Get the right index of y
  predict_data.index = pd.date_range(start=start, end=end, freq='H')
  # Set the columns name of y
  predict_data.columns = df.columns.values
  # Display Predict value Dashbord
  if plot:
    Air_Situation_Dashbord(predict_data, width=width, title=title,
        min_max_distance=min_max_distance)
  if return_result:
    return predict_data
  else:
    return
# ======================*END - RNN Model functions*======================
```