Final project

Arnaud Savary et Matthieu Vos

May 2020

Contents

1 Présentation du projet et des données

Le but de ce projet est de mettre en oeuvre les techniques que nous avons apprises durant le semestre et d'en découvrir de nouvelles. Pour ce faire, nous allons reprendre le concours KDD de 2018, dans lequel nous devons prédire la qualité de l'air selon 3 polluants dans deux villes : Londres et Beijing.

Nous avons à notre disposition des données historiques sur la qualité de l'air et la météo dans ces deux villes du premier janvier 2017 à mars 2018. Ces données sont disponibles dans divers fichiers, dont voici l'architecture :

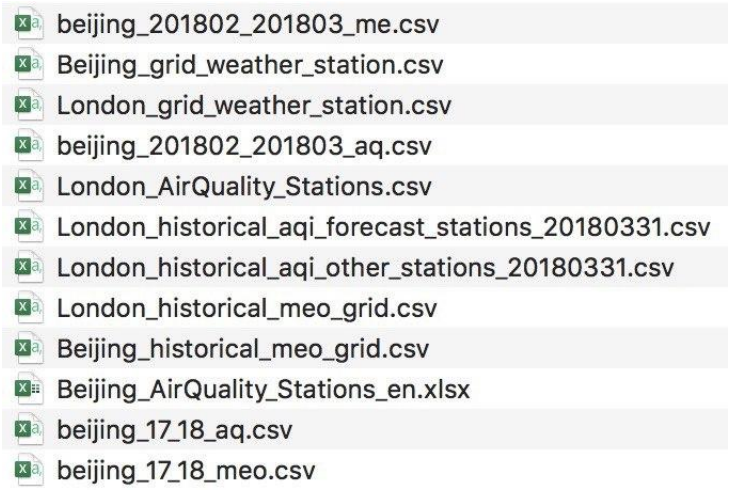


Figure 1: Architecture des fichiers originaux

Dans leur état actuel, ces données sont malheureusement inutilisables; nous allons devoir les remanier afin de produire des fichiers CSV propres et complets.

2 Préparation des données

Nous avons commencé notre travail sur ce projet en lisant les différents fichiers donnés et en essayant de comprendre quelles étaient les données à notre disposition et le format dans lequel elles étaient disponibles.

2.1 Problèmes présents dans les données initiales

Dans un premier temps, il a été nécessaire de lister tous les problèmes que nous pouvions repérer afin de pouvoir les traiter un par un par la suite. Voici la liste que nous avons créé :

- Données présentes dans divers fichiers
- Noms de colonnes différents d'un fichier à l'autre

- Données inutiles (dans la qualité de l'air : NO2, SO2 et CO et dans la météo : weather)
- Données dupliquées
- Heure présente mais sans donnée
- Pas de donnée, ni même d'heure
- Les stations météorologiques ne sont pas les mêmes que celles donnant la qualité de l'air
- Stations inutiles (pour Londres uniquement)

Nous allons décrire maintenant les étapes les moins triviales pour résoudre ces problèmes.

2.2 Étapes de la préparation des données

Nous avons traité séparément les données météorologiques et sur la qualité de l'air dans deux fichiers : aq_data.py et meo_data.py

2.2.1 Qualité de l'air

Pour la qualité nous commençons par ouvrir les fichiers contenant les données (2 par ville), retirons les colonnes inutiles, normalisons les noms des colonnes et concaténons le tout en un dataframe.

Nous transformons ensuite les dates de la colonne time qui étaient des strings en des dates et extrayons plusieurs objets du dataframe :

- aq_dataset : notre dataset avec la colonne time comprennant des dates
- stations : la liste des noms de toutes les stations
- aq_stations : dictionnaire où pour chaque station nous avons sa qualité d'air
- aq_stations_merged : dataframe avec la qualité de l'air de chaque station

Avec toutes ces informations, nous pouvons débuter le traitement. Pour commencer, nous parcourons toutes les heures afin de supprimer tous les doubles. Nous créons ensuite un dictionnaire qui à chaque station associe la liste des stations les plus proches, que nous utilisons pour remplir les lignes où nous avions l'heure mais pas les données. Il nous suffit de prendre les valeurs de la station où les données ne manquent pas la plus proche. Après cela, nous parcourons la liste des heures manquantes et faisons une approximation pour remplir ces dernières. Pour cela, nous prenons la donnée la plus proche avant et après le trou et calculons la moyenne pondérée par la distance de ces deux valeurs. Dans le cas des stations de Beijing, nous retirons enfin "_aq" des noms

des colonnes car c'est une information inutile et qu'elle complexifiera les split sur "_".

Il ne nous reste plus qu'à sauver ce dataframe dans un fichier CSV que nous appelons "CITY_aq_data.csv" avec CITY valant ld ou bj en fonction de la ville.

2.2.2 Météo

Dans le cas des données météorologiques, nous avons décidé de ne pas utiliser tous les fichiers : nous n'avons utilisé que les meo_grid. Dans le cas de Beijing uniquement, nous disposions de données supplémentaires appelées "observed weather". Il était cependant indiqué que ces données étaient moins fiables que celle de grid et nous avons décidé de ne pas fausser les données de grid en voulant inclure la météo observée.

Nous avons commencé le traitement des données météorologiques en ouvrant les fichiers CSV contenant meo grid, en éliminant leurs colonnes inutiles (longitude et lattitude) et uniformisant les noms des colonnes.

Nous allons utiliser dans les colonnes les stations de la qualité de l'air, nous avons donc dû trouver pour chacune le point dans la grille le plus proche et y associer ses données.

Nous supprimons tous les doubles en itérant sur chaque heure, puis complétons les données manquantes de la même manière que nous le faisions pour les données sur la qualité de l'air (interpolation des données juste avant et juste après les plus proches).

Nous concaténons les données de toutes les villes dans un dataframe, que nous sauvons alors dans un fichier CSV que nous appelons "CITY_meo_data.csv" avec CITY valant ld ou bj en fonction de la ville.

2.3 Données préparées

Une fois les 4 fichiers CSV créés, nous devons encore identifier le training set, qui commence le premier janvier 2017 et permettra d'entraîner nos algorithmes et le test set, qui commence le 21 mars 2018, dure 48 heures et permettra de tester nos algorithmes. Pour chacun, nous créons un fichier CSV ne contenant que les données nécessaires.

Nous nous retrouvons donc au final avec 12 fichiers, que vous pouvez retrouver sur la figure 2.

Nous pouvons maintenant aborder la structure interne d'un de ces fichiers : la première colonne indexe le temps et les suivantes donnent chaque information (les différents gaz pour la qualité de l'air et les différents critères météorologiques pour la météo) pour chaque station. Sur la figure 3, vous retrouverez par exemple l'organisation du fichier ld_aq_train_data.csv.

La préparation des données aura donc représenté une grosse charge de travail et est coûteuse en temps de calcul : un peu plus de 6 minutes. Malheureusement, elle est absolument primordiale et permet de travailler correctement ensuite.

Afin d'éviter de perdre 6 minutes à chaque tentative, nous avons décidé d'utiliser un Jupyter notebook, qui nous permet de lancer spécifiquement des

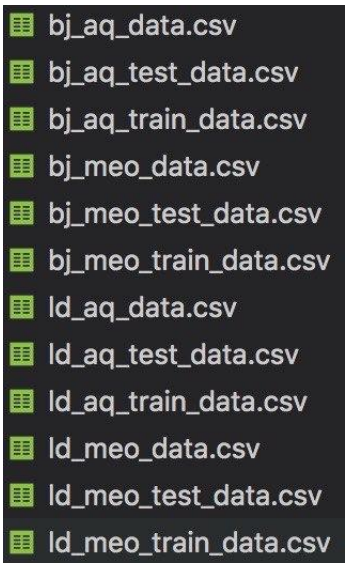
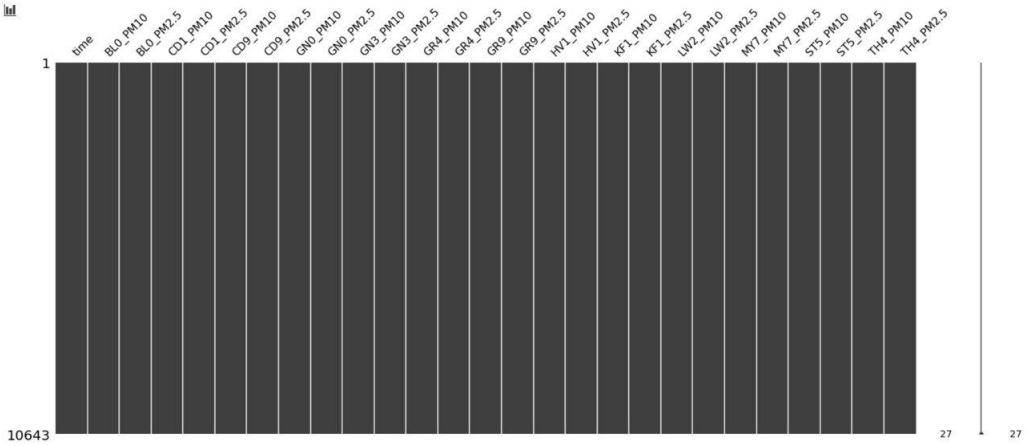


Figure 2: Architecture des fichiers finaux



portions des codes. Ceci combiné avec la sauvegarde des données préparées dans des fichiers permet d'éviter de devoir repréparer les données à chaque fois et ne relançant plus la portion de code appelant notre fonction prepare_data().

3 Approches

Nous allons maintenant présenter les différentes approches que nous avons développées ainsi que les résultats obtenus.

3.1 Valeur moyenne

La première méthode que nous avons décidé d'utiliser est assez simple à réaliser et produit des résultats mitigés. Nous voulions un modèle simple, qui permette de s'assurer que les méthodes suivantes basées sur du machine learning et demandant beaucoup plus de puissance de calcul avaient bien un intérêt.

Nous avons donc choisi de prendre la valeur moyenne de chaque polluant dans chaque station et de l'appliquer comme prédiction pour les 48 heures. Les moyennes sont calculées sur les données d'entraînement, et appliquées ensuite sur les 48 heures suivant la fin de ces données.

Nous obtenons ainsi les résultats suivants :

```
PM 2.5 RMSE in London : 5.02876965376741
PM 10 RMSE in London : 9.841977487421524
PM 2.5 RMSE in Beijing : 33.16772024063834
PM 10 RMSE in Beijing : 38.322853268318134
O3 RMSE in Beijing : 37.908385847992605

PM 2.5 SMAPE in London : 33.95350449726899
PM 10 SMAPE in London : 28.180610934159038
PM 2.5 SMAPE in Beijing : 34.694738106038095
PM 10 SMAPE in Beijing : 24.126770866510928
O3 SMAPE in Beijing : 84.00246992656014
```

Grâce aux RMSE, nous pouvons savoir de combien notre valeur est fausse en moyenne, et grâce aux SMAPE nous pouvons voir le pourcentage de l'erreur moyenne. Nous remarquons que les SMAPE sont très élevés avec cette méthode qui n'est donc évidemment pas bonne.

Le graphique de la figure 4 permet de visualiser les prédictions de notre modèle sur une station de Londres et de Beijing, toutes deux choisies aléatoirement.

Comme prévu, les prédictions ne suivent pas du tout les courbes réelles, mais sont de simples droites sur la valeur la plus probable. Cette méthode a tout de même un avantage : elle s'exécute en une demi seconde (contre plusieurs heures pour les suivantes).

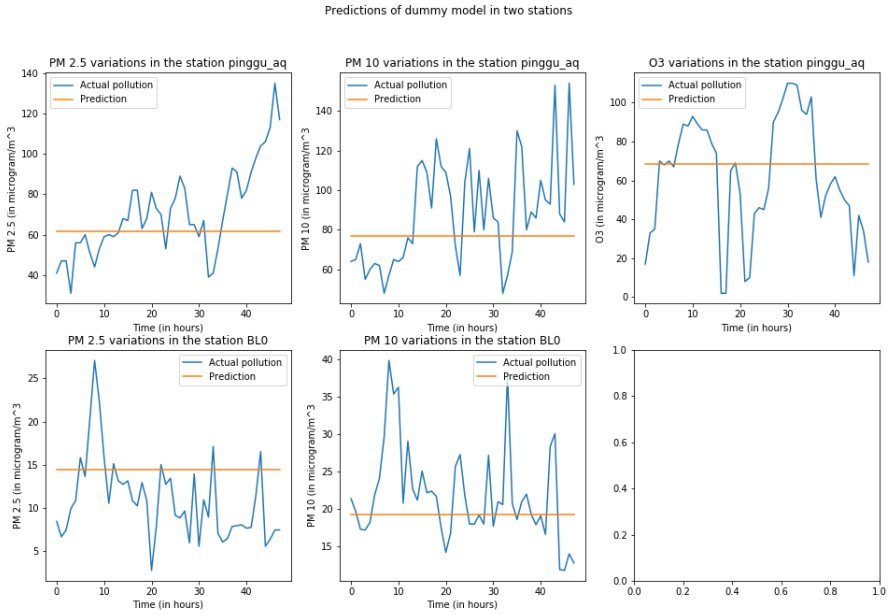


Figure 4: Dummy's method results in two cities

3.2 SVR

Le deuxième algorithme appliqué est l'algorithme SVR dit Support Vector Regression. C'est la combinaison d'un SVM, Support Vector Machines, et d'une régression linéraires. Afin d'utiliser SVR, nous aurons également besoin de standardiser nos données. Nous utiliserons ensuite la fonction SVR de la libraire scikit-learn de python avec nos paramètres.

3.2.1 SVM

Un SVM est surtout utilisé pour la classification. Le but d'un SVM est de définir la limite entre deux classes d'un graphique. Cette limite est appelé l'hyperplan. Cet hyperplan possède une marge entre les deux classes. Elle délimite la distance entre l'hyperplan et la première valeur de chaque classe. Une fois la séparation faite, un SVM est capable de prévoir à quel classe une donnée devrait appartenir. Idéalement, la marge devrait être la plus grande possible entre les deux classes.

3.2.2 Régression linéaire

Une régression linéaire permet de trouver la meilleure relation linéaire entre deux variables. Avec un X dépendant donné, on aimerait prédire notre variable Y indépendante. Ainsi une régression linéaire peut trouver la meilleure

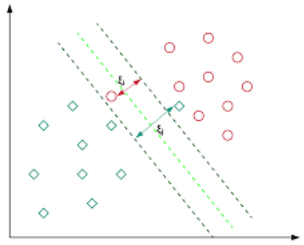


Figure 5: Exemple de SVM

représentation, Y_{pred} , d'un ensemble de donnée, Y, à partir d'un X.

Équation d'une régression linéaire :

$Y' = A + B \cdot X$ où

Y' est la valeur prédite

A est l'intercepte

B est le coefficient

X est la valeur d'input

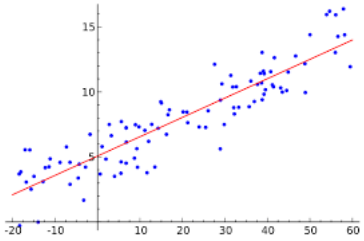


Figure 6: Exemple de régression linéaire

Durant le processus d'apprentissage, la régression linéaire tente de trouver les meilleurs valeurs A et B pour minimiser la fonction de coût. En général, la fonction de coût est le MSE, mean square error. Cette fonction est minimisé grâce à un processus nommé gradient descent.

3.2.3 La théorie derrière SVR

Nos données ne sont donc pas adaptée pour une régression linéaire. En effet, nos données ont beaucoup de variations et essayer de les approximer à l'aide d'une ligne ne nous apporterait rien (cf. dummy prediction). Un SVM ne marcherait pas non plus car on ne veut pas classer nos valeurs en deux catégories. Dans notre problème, nous voulons prédire la valeur suivante que va prendre un gaz polluant.

L'idée de SVR est de combiner les deux. Nous voulons approximer les valeurs

avec précision dans un certain seuil. Une régression linéaire à l'intérieur d'une marge d'un hyperplan. Nous avons donc une marge allant de $+\varepsilon$ à $-\varepsilon$ autour de notre hyperplan. Nous pouvons ensuite définir notre régression linéaire $Y = A \cdot X + B$. L'objectif est de réduire l'erreur en maximisant la marge. De plus, SVR est adaptable afin de prédire des valeurs dans un seuil non-linéaire. Ce qui est parfait pour notre système.

3.2.4 Standardisation des données

Nos données ont des valeurs variant beaucoup entre deux classes. L'ordre de grandeur de la pression n'est pas le même que celui de la température. Ainsi, une standardisation de nos données est nécessaire afin de ne pas avoir une pondération inéquitable de nos classes. Une standardisation permet d'avoir une distribution normale en faisant en sorte que la déviation à la moyenne soit égal à 1. Dans la librairie scikit-learn se trouve déjà des methodes permettant d'implémenter une standardisation de donnée : MinMaxScaler, RobustScaler, StandardScaler et Normalizer. Nous avons rapidement essayé MinMaxScaler, mais nous obtenions de mauvais résultat. En nous renseignant plus sur ces méthodes, nous avons compris que RobustScaler aurait produit un overfitting et Normalizer n'était pas recommandé en général. Nous avons donc décidé d'utiliser StandardScaler.

Nous l'avons utilisé sur nos valeurs d'input et d'output à l'aide de la méthode fit_transform() qui nous retourne nos valeurs transformée. Nous avons ensuite dû séparer nos données entre nos valeurs d'entraînement et nos valeurs cibles, train_X et train_y respectivement.

3.2.5 SVR dans python et les paramètres

Ayant nos valeurs d'entraînement et la base pour notre test, nous pouvons appeler notre fonction SVR. Celle-ci demande plusieurs paramètres en entrée que nous allons décrire içi.

kernel : permet de décrire le type de kernel utilisé dans notre algorithme. Il peut avoir comme valeur : 'linear', 'poly', 'rbf', 'sigmoid'. Nous avons utilisé la fonction rbf qui est la fonction basique radial kernel. Il est défini ainsi : $K(x, x') = e^{\frac{-||x-x||^2}{2\sigma^2}} = e^{-\gamma||x-x||^2}$

degree : décrit le degré de la fonction kernel polynomial. Il est ignoré si on utilise un autre kernel. default = 3

gamma : coefficient pour rbf poly et sigmoid. default = scale signifie que l'on utilise $1/(n\_features * X.var())$ comme valeur de gamma.

coef0 : terme indépendant de la fonction kernel. Il est utilisé seulement avec poly et sigmoid. default = 0

tol : tolérance pour le critère d'arrêt. default = 1e-3

C : Paramètre de régularisation. La force de la régularisation est inversement proportionnel à C. default = 1.0

epsilon : ε du modèle SVR. Il permet de préciser le tube dans lequel aucune pénalité est associé à la fonction de perte d'entraînement. Les points prédits se

trouvant à une distance maximum de ε de la valeur exact. default = 0.1

shrinking : utilisation ou non du rétrécissement heuristique. Permet d'activer l'option du rétrécissement qui consiste à réduire le temps d'entraînement en détectant les éléments bornés dès que possible dans les itérations de décompositions et en enlever certains. default = True

cache_size : précise la taille du cache du kernel en MB. default = 200

verbose : Permet l'affichage d'information. Selon la valeur, affiche l'évolution de l'entraînement de l'algorithme. default=False

max_iter : limite sur les iterations du solveur defautl = -1(no limit)

Les valeurs que nous avons utilisé et spécifié sont : C=2 , epsilon = 0.01, kernel ='rbf', gamma=0.1, tol = 0.01 , verbose = 0, shrinking= True, max_iter = 10000. Nous les avons obtenues en faisant des tests sur une station avec un gaz pour avoir un bon RMSE avec un temps de calcul qui n'était pas trop conséquent.

Ces valeurs ont permit de créer le modèle de l'algorithme que nous avons train avec nos données en utilisant la méthode fit(x,y[,samples_weight]) qui entraîne le modèle en accord avec nos données d'entraînement. Nous avons ensuite utilisé la méthode predict(X) qui effectue la régression sur l'échantillon X afin d'obtenir nos valeurs cibles. Il faut ensuite faire l'inverse de notre standardisation précédente afin de retrouver nos valeurs.

3.2.6 Résultats

Nous obtenons ainsi les résultats suivants :

```
PM 2.5 RMSE in London : 0.4220443237793854
PM 10 RMSE in London : 1.6220341083887175
PM 2.5 RMSE in Beijing : 1.269228563828763
PM 10 RMSE in Beijing : 3.6947014496240707
O3 RMSE in Beijing : 1.25146979038381

PM 2.5 SMAPE in London : 1.990906551863266
PM 10 SMAPE in London : 1.5318067698360578
PM 2.5 SMAPE in Beijing : 1.1227008310468576
PM 10 SMAPE in Beijing : 1.422936151229508
O3 SMAPE in Beijing : 6.778798472005728
```

Le graphique de la figure 7 permet de visualiser les prédictions de notre modèle sur une station de Londres et de Beijing, toutes deux choisies aléatoirement.

Nous remarquons sur nos graphiques et par nos RMSE que notre modèle SVR fonctionne très bien. Grâce au SMPAE, nous pouvons remarquer une erreur moyenne de 1%. Ce qui est un plutôt bon critère.Nous pouvons aussi relever la difficulté supérieur à estimer la quantité d'O3. Notre modèle suis bien la courbe de valeurs réels et nous n'avons pas de overfitting.

Notre temps d'exécution pour SVR est de 8174 secondes. On a donc des résultats significativement meilleurs que pour la méthode Dummy, mais un temps considérablement plus long également.

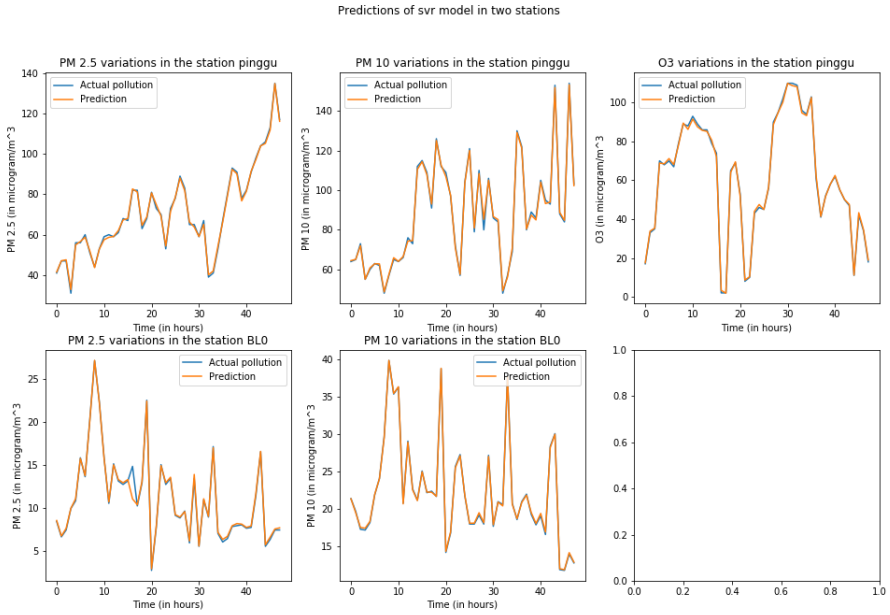


Figure 7: SVR's method results in two cities

3.3 GBRT

Le dernier algorithme appliqué est GBRT, Gradient Boosting Regression Tree. Il utilise un ensemble d'arbre de décision pour prédire une valeur cible. Pour l'application de l'algorithme, nous avons utilisé la fonction GradientBoostingRegressor de la librairie scikit-learn de python.

3.3.1 L'algorithme GBRT

La façon dont fonctionne GBRT est par la création de plusieurs arbre de décision, réduisant le résidu à chaque itération. Le résidu est la différence entre la valeur actuel et la valeur prédite, residual = actual value - predicted value.

1. Au départ du problème, GBRT commence avec une feuille contenant la valeur moyenne de la variable que l'on veut prédire. Elle servira de base afin de trouver la bonne solution pour les étapes suivantes.

2. À partir de là, l'algorithme calcule le résidu pour chaque valeur de nos données en ayant notre valeur moyenne à la place de la valeur prédite. Chacune de nos valeurs cible possède maintenant un résidu.

3. L'algorithme construit ensuite le premier arbre de décision afin de prédire notre résidu. L'arbre possède des branchements en fonction de chacune de nos valeurs de fonctionnalités. Les feuilles de cet arbre contiennent de nouveaux résidus. Si il y a plus de résidu que de feuille, plusieurs résidu se trouveront

dans la même feuille. Dans ce cas, l'algorithme calcul la moyenne des résidu et place le résultat dans la feuille.

4. Ainsi pour prédire une valeur, chaque fonctionnalité passe dans l'arbre de décision jusqu'à arriver à une feuille. Le résidu présent dans cette feuille est utilisé afin de prédire notre valeur cible. La valeur prédite est obtenu à l'aide du calcul suivant :

$V = A + L \cdot R$

où V est notre valeur prédite, A la valeur moyenne calculée au début de notre algorithme, L le taux d'apprentissage, et R le résidu prédit par l'arbre

5. À partir de cette valeur prédite, V, l'algorithme calcule un nouveau résidu pour chaque valeur selon la formule vu plus haut. Ces résidus seront utilisé comme feuille dans le prochain arbre de décision comme à l'étape 3.

6. Les étapes 3 à 5 sont répétées jusqu'à avoir le nombre d'itération prévu par l'hyperparamètre, nombre d'estimateur.

7. Une fois entrainé, l'algorithme utilise tout les arbres de l'ensemble afin de faire une dernière prédiction sur la valeur de la variable cible. La prédiction se fait ainsi :

$V = A + L \cdot \sum R_i$

où R_i est le résidu prédit par le ième arbre de précision.

3.3.2 La fonction GradientBoostingRegressor

Nous utilisons la même méthode que pour SVR pour préparer nos données et les standardiser. Ayant nos valeurs d'entraînement et la base pour notre test, nous pouvons appeler notre fonction GradientBoostingRegressor. Celle-ci demande plusieurs paramètres en entrée que nous allons décrire içi.

loss : fonction de perte à optimiser. Il peut prendre les valeurs : 'ls', 'lad', 'huber', 'quantile' .valeur par défault = ls qui signifie régression des moindres carrés

learning_rate : défini le taux d'apprentissage. default = 0.1

n_estimators : défini le nombre de cycle sur les étapes 3 à 5. Une plus grande valeur permet des résultats plus précis.

subsample : défini la fraction d'échantillons utilisé par les individual base learners. Si il est inférieur à 1 alors on a faire à un gradient boosting stochastique. default = 1

criterion : défini la fonction mesurant la qualité de la division. Il peut prendre les valeurs : 'friedman_mse', 'mse', 'mae' avec par défaut 'friedman_mse' qui signifie l'erreur absolue moyenne avec l'amélioration de Friedman

min_samples_split : le nombre minimum d'échantillon requis pour diviser un noeud interne. default = 2

min_sample_leaf : le nombre minimum d'échantillon afin d'être au niveau d'une feuille. Un division est considéré, à n'importe quel niveau, s'il laisse au moins min_sampple_leaf d'échantillons d'entraînement dans chacune des branches gauche ou droite.

min_weight_fraction_leaf : la fraction minimum de poids de la somme total des poids requis pour être au niveau d'une feuille

max_depth : profondeur maximal d'un estimateur de régression. default = 3

min_impurity_decrease : un noeud sera divisé si la division implique une baisse de l'impureté plus grande ou égale à cette valeur. la baisse d'impureté est calculé ainsi :

$\frac{N_t}{N} \cdot (impurity - N_{t_R}/N_t \cdot right\_impurity - N_{t_R}/N_t \cdot left\_impurity)$

où N est le nombre total d'échantillon, N_t est le nombre d'échantillon au noeud courant et N_{t_R} et N_{t_L} le nombre d'échantillon dans l'enfant de droite et de gauche respectivement.

min_impurity_split : seuil pour l'arrêt précoce dans l'agrandissement de l'arbre. Un noeud se divisera si l'impureté est au-dessus de ce seuil, sinon c'est une feuille. default = None

init : Défini un objet estimateur utilisé pour calculé les prédictions initiales. default = none. Utilise un DummyEstimator par défaut. pour précvoir la valeur moyenne cible.

random_state : Contrôle la graine aléatoire donnée a chaque arbre d'estimation à chaque itération. De plus, il contrôle la permutation aléatoire des fonctionnalités à chaque division. Il contrôle aussi la division aléatoire des données d'entraînement pour obtenir un set de validation if n_iter_no_change n'est pas None.

max_features : peut prendre les valeurs : 'auto', 'sqrt', 'log2' étant par défaut = None. Définis le nombre de fonctionnalités à considérer en cherchant la meilleure division :

Si égale à auto, alors $max\_features = n\_features$

Si égale à sqrt, alors $max\_features = \sqrt{n\_features}$

Si égale à log, alors $max\_features = \log_n(n\_features)$

Si égale à None, alors $max\_features = n\_features$

Si $max\_features < n\_features$, alors on a une réduction de la variance et une augmentation dans le biais

alpha : alpha-quantile pour la fonction de perte huber et la fonction de perte quantile.

verbose : Permet l'affichage d'information. Selon la valeur, affiche l'évolution de l'entraînement de l'algorithme. default=0

max_leaf_nodes : Crée des arbres avec max_leaf_nodes avec la meilleur-première mode. Les meilleurs noeuds sont défini relativement à la réduction d'impureté. dafault = None (no limit)

warm_start : Si égale à Vrai, alors réutilise la solution de l'appel précédent pour former et ajouter plus d'estimateur à l'ensemble. Sinon, efface la solution précédente. default = False

validation_fraction : La proportion de donnée d'entrainement à utilisé pour la partie de validation pour un arrêt plus rapide. Utilisé seulement si n_iter_no_change donné comme un entier.

n_iter_no_change : Il est utilisé afin de décider si l'arrêt rapide doit être utilisé pour arrêter l'entraînement quand le score de validation ne s'améliore pas. Si il est donnée comme un entier, alors l'algo ignore validation_fraction de taille d'entraînement comme des données de validation et arrête l'entraînement

quand le score de validation ne s'améliore pas dans toute les précédentes nombre d'itérations.

tol : Tolérance pour l'arrêt soudain. Quand la perte n'est pas amélioré par au moins tol pour n_iter_no_change itérations, l'entraînement s'arrête. default = 1e-4

ccp_alpha : Paramètre complexe pour le l'élagage minimal à coût-complexe. Le sous-arbre avec le plus grand coût complexe qui est plus petit que ccp_alpha est choisis. Par défault, aucun élagage n'est appliqué

Les valeurs que nous avons utilisé et spécifié sont : n_estimators=500 ,max_depth = 10, min_samples_split = 5, learning_rate = 0.1, loss = 'ls'. Nous les avons obtenues en faisant des tests sur une station avec un gaz pour avoir un bon RMSE avec un temps de calcul qui n'était pas trop conséquent. Malheureusement, durant la dernière évaluation devant nous donner les bon résultats, l'ordinateur compilant le code a eu un problème technique ce qui nous a fait perdre les données de cette évaluation. Par manque de temps, nous n'avons pas pu la relancer avec les mêmes paramètres. Nous avons donc une précision qui n'est pas optimale décrite dans ce rapport. Cependant, nous avons relancé l'évaluation afin d'avoir un meilleur résultat à présenter lors de la présentation. Les valeurs utilisés pour ce rapport sont les même à l'exception de n_estimators qui a été changé à 30.

Ces valeurs ont permit de créer le modèle de l'algorithme que nous avons train avec nos données en utilisant la méthode fit(x,y[,samples_weight]) qui entraîne le modèle en accord avec nos données d'entraînement. Nous avons ensuite utilisé la méthode predict(X) qui effectue la régression sur l'échantillon X afin d'obtenir nos valeurs cibles. Il faut ensuite faire l'inverse de notre standardisation précédente afin de retrouver nos valeurs.

3.3.3 Résultats

Nous obtenons ainsi les résultats suivants :

```
PM 2.5 RMSE in London : 1.7583065305741379
PM 10 RMSE in London : 2.789441757402706
PM 2.5 RMSE in Beijing : 6.467225258525666
PM 10 RMSE in Beijing : 13.771993821034526
O3 RMSE in Beijing : 5.489118837874595

PM 2.5 SMAPE in London : 13.581715396443464
PM 10 SMAPE in London : 9.41619611506955
PM 2.5 SMAPE in Beijing : 6.3345879964829415
PM 10 SMAPE in Beijing : 9.718536096489848
O3 SMAPE in Beijing : 28.015620648116343
```

Le graphique de la figure 8 permet de visualiser les prédictions de notre modèle sur une station de Londres et de Beijing, toutes deux choisies aléatoirement.

Nous remarquons une moins bonne qualité de RMSE et de SMAPE que pour SVR en grande partie dût aux paramètres utilisés pour notre dernière

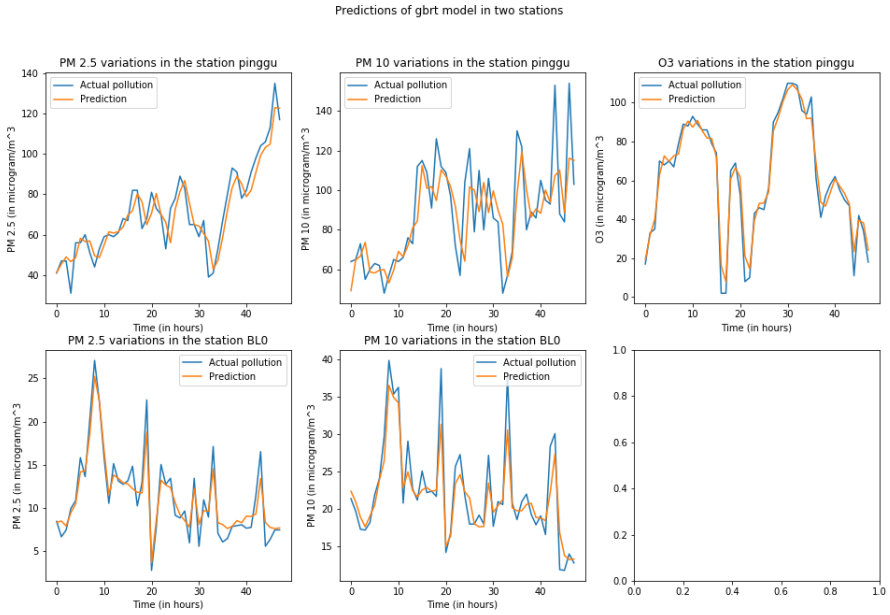


Figure 8: GBRT's method results in two cities

exécution. Nous avions les bons paramètres dans notre avant-dernière exécution, mais comme mentionné précédemment, cette exécution a rencontré un problème. Ces données sont donc irrécupérables. Nous pouvons cependant remarquer une plutôt bonne approximation graphique avec de tels paramètres, ce qui présage de bons résultats pour une exécution avec les paramètres adéquats, que nous présenterons pour la présentation de jeudi. De même que pour SVR, nous pouvons relever la valeur plus élevée de SMAPE pour le gaz O3. Notre temps d'exécution est de 9241 secondes, mais il faut prendre en compte que nous avions une exécution volontairement courte par manque de temps. Il est aussi important de noter que l'exécution de GBRT a été faite sur un ordinateur significativement moins puissant que pour SVR, le temps peut ainsi changer. Des résultats plus précis seront fournis durant la présentation.