Data Challenge
Prévision des validations de passagers pour le réseau SNCF-Transilien

Isabella RIBEIRO RIGUE
Luigi MELLO RIGATO

Nantes, France
novembre, 2024

Sommaire

1 Introduction

1.1 Le sujet

Ce data challenge proposé par SNCF-Transilien a pour objectif de mieux anticiper le nombre de validations quotidiennes dans les gares d'Île-de-France. En tant que principal opérateur de trains de banlieue dans cette région, SNCF-Transilien fait circuler chaque jour plus de 6 200 trains, transportant des millions de voyageurs. Le nombre de validations de cartes à puce augmente régulièrement — environ 6% par an entre 2015 et 2019 —, ce qui rend crucial de mieux prévoir cette croissance pour adapter les services et améliorer la gestion des flux.

Le but de ce challenge est de construire un modèle capable de prédire, pour chaque gare, le nombre de validations journalières sur le moyen et long terme. Il s'agit d'un problème classique de prévision de séries temporelles, mais la difficulté réside ici dans la gestion d'un grand nombre de séries distinctes, correspondant à chacune des gares. Pour SNCF-Transilien, l'enjeu est de pouvoir anticiper les besoins futurs en termes de transport et d'ajuster l'offre de façon proactive afin de répondre efficacement à la demande croissante.

1.2 Les données

Pour ce challenge, nous avons reçu les jeux de données :

— train_x.csv : Ce fichier contient les caractéristiques (features) relatives aux validations quotidiennes dans 448 gares du réseau SNCF-Transilien, incluant toutes les stations des lignes RER A et RER B. Il couvre une période allant du 1er janvier 2015 au 31 décembre 2022, soit un total de 2 922 jours d'observation.

— train_y.csv : Ce fichier comprend un index (qui est la combinaison de la date et de l'identifiant de la station) ainsi que la variable cible y, représentant le nombre de validations. Cette variable est associée aux caractéristiques présentes dans train_x.csv.

— test_x.csv : Ce fichier contient les mêmes caractéristiques (features) que train_x.csv, mais pour une période plus récente, du 1er janvier 2023 au 30 juin 2023, soit 181 jours. Ce jeu de données est destiné à être utilisé pour prédire les valeurs de y, c'est-à-dire le nombre de validations journalières pour chaque station, en se basant sur le modèle entraîné.

En ce qui concerne les features, nous avons reçu :

— index : une caractéristique présente dans test_x.csv et train_y.csv, mais pas dans train_x.csv. Cette caractéristique est simplement la concaténation avec un « _ » de la date et de la station, décrite ci-dessous.

— date : la date à laquelle les validations ont été enregistrées, au format AAAA-MM-JJ.

— station : identifiant anonymisé de chaque gare codé par une chaîne de trois caractères.

— job : indicateur du Jour Ouvrable de Base (JOB) valant 1 pour les jours de semaine et 0 pour les week-ends.

— ferie : indicateur pour les jours fériés, avec 1 si le jour est férié et 0 sinon.

— vacances : indicateur pour les vacances scolaires, valant 1 si le jour fait partie des vacances scolaires et 0 sinon.

2 Outils

Dans cette section, nous présenterons les outils utilisés pour aider à développer le code en binôme.

2.1 Git

Tout d'abord, nous avons utilisé Github pour la version et le stockage du code. Pour ce faire, nous avons créé une Organisation appelée ML-SNCF-Transilien, dont nous sommes membres.

Ainsi, à chaque nouveau modèle testé, changement de paramètres de fonctionnement ou modification du traitement des données, nous enregistrons un nouveau commit afin que le chemin soit balisé sur le site.

2.2 Visual Studio Code

Comme IDE, nous utilisons Visual Studio Code, où nous intégrons Git dans son terminal ainsi que les environnements virtuels conda. Nous utilisons également les extensions Python et le Jupyter Extension Pack pour fonctionner sur VSCode.

Les principales dépendances du projet sont : Ipykernel, Pandas, Matplotlib.pyplot, Seaborn, Lightgbm, Sklearn.model_selection et Numpy.

2.3 Google Colab

Google Colab était un moyen d'exécuter du code avec les ressources informatiques de Google Console sans dépendre du traitement de notre ordinateur, car les codes prenaient trop de temps à s'exécuter. Souvent, l'ordinateur tombait en panne Pour ce faire, nous avons transmis le code à colab et les données à un dossier sur le disque dur, puis nous avons intégré les données dans le code à l'aide des commandes :

from google.colab import drive

drive.mount('/content/drive')

Cependant, lorsque nous avons exécuté le code, Colab a donné une estimation de 85 heures pour exécuter la fonction GridSearchCV, nous avons donc abandonné et nous en revenons donc à l'utilisation du VSCode.

2.4 Remarques

L'un d'entre nous a un ordinateur de 8 Go qui ne pouvait pas fonctionner du tout. L'autre avait un ordinateur de 16 Go qui mettait du temps à s'exécuter et qui, parfois, s'exécutait. Une alternative était d'utiliser l'ordinateur d'un ami qui avait 32 Go de RAM. Une autre alternative était d'utiliser l'ordinateur de l'École via le VPN, qui, malgré ses 16 Go, n'a jamais eu de problèmes. La partie la plus difficile a été d'installer tous les outils sur l'ordinateur de l'École.

3 Données

3.1 Exploration des données

Au sujet de la qualité des données, nous n'avons pas rencontré de valeurs manquantes dans les fichiers fournis. Toutefois, un petit ajustement a été nécessaire dans notre code pour avoir un index défaut pour les fichiers csv. En effet, alors que les fichiers test_x et train_y contenaient déjà un index basé sur la combinaison de la date et de la gare, le fichier train_x ne disposait pas d'un index défini.

En continuant, lors de l'exploration des données, nous avons effectué plusieurs visualisations (comme representé dans la figure 1 ci-dessous et la figure 3 dans les annexes) pour mieux comprendre la distribution des validations et le type de tendance dont il s'agit. En examinant les boxplots (figure 4 dans les annexes), nous avons observé que les données sont concentrées autour de certaines valeurs, mais qu'il existe également de nombreux points de données éparpillés, ce qui compliquait l'interprétation du graphique. Pour avoir une vue plus claire, nous avons tracé un histogramme. Celui-ci a révélé que la majorité des validations sont concentrées sur des valeurs relativement faibles, mais il y a encore une grande quantité de valeurs dispersées, en raison du nombre important de données collectées.

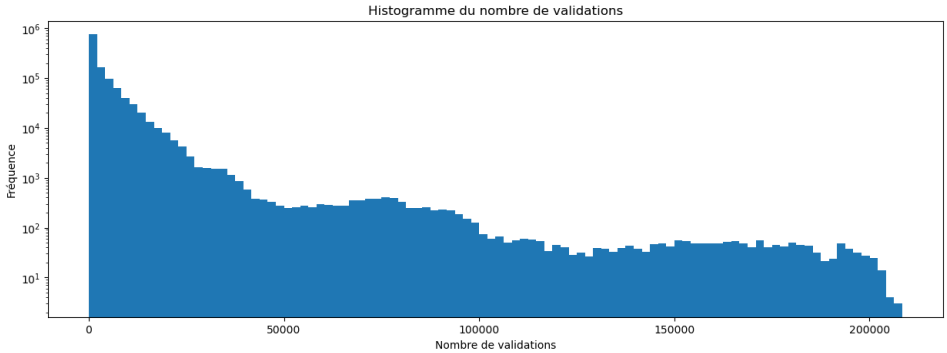

FIGURE 1 – Histogramme du nombre de validations

De plus, en visualisant les validations au fil du temps dans la figure ci-dessous (2), nous avons pu observer une certaine récurrence, principalement de manière hebdomadaire, ce qui suggère une tendance saisonnière ou un effet lié aux jours de la semaine.

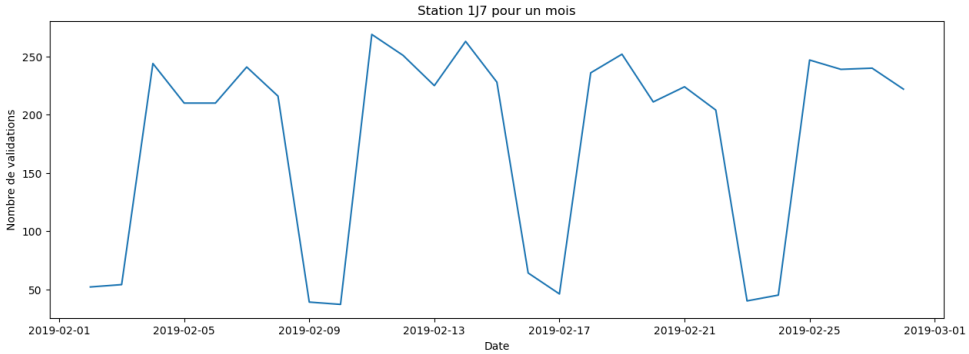

FIGURE 2 – Nombre de validations de la station 1J7 pour un mois

Enfin, nous avons également analysé quelques statistiques descriptives pour la variable

'y' (nombre de validations). Ces statistiques montrent que, bien que la plupart des valeurs de 'y' soient relativement faibles (avec une médiane de 1 108), l'écart-type élevé (9 683,39) et la valeur maximale très élevée (208 409) indiquent qu'il existe des valeurs extrêmes qui influencent fortement la distribution, confirmant ce qui a été montré dans les boxplots. Cela peut être dû à des événements rares, comme des jours particulièrement chargés dans certaines gares.

3.2 Manipulations

Dans cette partie, nous décrivons les différentes étapes de manipulations et de transformation des données que nous avons réalisées pour améliorer la qualité et la pertinence de notre modèle. Ces ajustements ont été cruciaux pour optimiser notre modèle et obtenir des prédictions plus précises.

3.2.1 Temporalité

Pour exploiter la dimension temporelle des données, nous avons d'abord converti les colonnes de date au format to_datetime(), ce qui a facilité leur manipulation et simplifié l'extraction d'informations temporelles.

En outre, comme mentionné précédemment, nous avons commencé par visualiser les données sur un mois afin de mieux observer les tendances et les cycles. Un examen sur une période plus longue rendait difficile l'observation des détails, tandis qu'un mois nous a permis de voir qu'il existe une certaine récurrence hebdomadaire, bien que ce comportement ne soit pas toujours strictement régulier. Cette analyse initiale a confirmé la présence de cycles, ce qui nous a incités à intégrer davantage d'informations temporelles.

Pour enrichir les analyses, nous avons développé la fonction create_features_date(), qui décompose chaque date en plusieurs composantes comme le jour, le mois, l'année, et le jour de la semaine. Cela nous a permis de mieux analyser ces informations de manière séparée et de les utiliser pour affiner le modèle. En testant notre modèle avec et sans ces nouvelles features temporelles, nous avons constaté une amélioration d'environ 16% de notre score, confirmant l'utilité de cette décomposition.

En parallèle, nous avons développé la fonction create_features_lags() pour introduire des lags temporels comme nouvelles features. Après plusieurs tests (et en tenant en compte les tendances saisonnières), nous avons retenu les lags de 1 jour, 7 jours et 1 mois, car ils ont obtenu les meilleurs résultats en termes de score et figuraient parmi les plus importants dans les analyses d'importance des features (détaillées plus bas). Ces lags se sont révélés cruciaux pour capturer la nature des données temporelles, apportant des informations essentielles pour un entraînement optimal du modèle. À savoir, en intégrant les lags, notre score a amelioré de plus de 40%, ce qui confirme leur rôle central dans la modélisation de séries temporelles.

3.2.2 Catégorique

Afin d'utiliser des modèles tels que XGBoost, LightGBM et d'autres, il a été nécessaire d'utiliser la features 'station' comme une catégorie. Pour ce faire, nous avons utilisé la méthode du pandas astype('category').

Pour d'autres modèles, tels que Random Forest, nous avons dû transformer la features 'station' en une valeur numérique. Pour ce faire, nous avons utilisé la fonction LabelEncoder(), qui transforme les valeurs comprises entre 0 et le nombre de classes moins 1.

3.2.3 Données manquantes

Les données originales étaient complètes, sans données manquantes. Cependant, depuis que nous avons décidé d'ajouter des décalages avec create_features_lags(), beaucoup de données de la semaine précédente, par exemple, les données respectives étaient manquantes.

Afin de tester la méthode Random Forest, nous avons testé l'imputation des données en utilisant la méthode pandas.DataFrame.fillna(method='ffill', inplace=True), qui remplit les données vides en propageant la dernière observation valide. Nous utilisons également l'imputation avec la moyenne, via pandas.DataFrame.fillna(X_train.mean(), inplace=True), par exemple pour les données d'entraînement. Quant à LightGBM et XGBoost, ils gèrent les données manquantes automatiquement, en interne.

Enfin, nous avons aussi essayé de supprimer les lignes pour lesquelles il manquait des données, mais cela a aggravé notre score. Nous obtenions normalement un score proche de 100, mais en faisant cela, nous avons obtenu un score de 39464.

3.2.4 Feature importances

Tout au long du processus de modélisation, nous avons accordé une attention particulière à l'analyse des feature importances, pour chaque modèle et chaque configuration testée. En examinant ces graphiques (figures 5, 6 et 7 dans les annexes, par exemple), nous avons pu identifier quelles variables étaient réellement pertinentes et ajuster notre modèle en conséquence. Cela nous a permis de retirer les features superflues, en visant réduire le risque de surapprentissage et améliorer la généralisation du modèle.

En plus, au début il n'était pas évident de déterminer combien de lags inclure dans notre modèle. Alors, grâce à l'analyse des feature importances, nous avons pu ajuster et sélectionner les lags les plus pertinents. Pour le modèle LightGBM, les lags de 1 jour, 1 semaine et 1 mois se sont révélés les plus utiles. En revanche, pour le XGBoost, le lag de 1 semaine a montré une importance nettement supérieure aux autres, en cohérence avec les récurrences hebdomadaires observées dans les données. Les autres lags, notamment ceux de 1 mois et 1 an, n'apportaient pas autant d'information pour ce modèle, et ont donc été retirés.

4 Analyses

Dans cette section, nous allons présenter les modèles que nous avons utilisés ainsi que les résultats obtenus lors de leur entraînement. Pour chaque modèle, nous avons effectué un grid search afin de trouver les paramètres les plus optimisés possibles, tout en tenant compte des limitations de nos ordinateurs. Bien que nous ayons testé plusieurs modèles différents, certains n'ont pas donné de bons résultats. Finalement, nous avons décidé de concentrer nos efforts sur l'étude et les tests des modèles de boosting, qui ont montré les meilleures performances pour ce problème spécifique. Finalmente, nous avons fait un tableau (voir table 1) regroupant tous les scores que nous avons eu, il est dans les annexes.

4.1 XGBoost

Nous avons utilisé XGBoost (Extreme Gradient Boosting) pour plusieurs tentatives, car il donnait généralement de bons résultats. Cependant, il était très lourd à exécuter et prenait beaucoup de temps lorsque nous utilisions plus de 500 arbres. En conséquence, il était difficile de faire la recherche sur le GridSearchCV, et donc de progresser dans le choix des paramètres. De plus, il n'utilise pas toutes les features dans la composition des décisions de l'arbre, contrairement à LightGBM, par exemple, comme il est possible de remarquer en comparant les figures 5 et 6 dans les annexes.

En outre, XGBoost nous a donné plusieurs prédictions négatives, ce qui est courant avec ce modèle, d'après nos recherches. Nous avons donc essayé d'y remédier en fixant le minimum qu'il pouvait donner à zéro, par exemple, cela a amelioré les résultats mais n'etait pas encore l'ideal (voir table 1).

Le meilleur score qu'on a obtenu avec XGBoost a été 104.05 avec la configuration : (n_estimators=700, max_depth=10, subsample=1, learning_rate=0.1), selon la table 1.

4.2 LightGBM

L'utilisation du LightGBM (Light Gradient Boosting Machine) a été le meilleur choix et nous a donné les meilleurs résultats. Ce modèle fonctionne relativement rapidement, ce qui nous a permis d'utiliser un plus grand nombre d'arbres, comme 10000, ce qui n'a fait qu'améliorer nos résultats. Aussi, avec cela il a été possible de progresser dans le choix des valeurs plus rapidement avec GridSearchCV et d'expérimenter avec davantage de paramètres combinés ensemble, pour une recherche plus complète.

Comme mentionné, augmenter le nombre d'arbres a effectivement amélioré la prédiction du modèle. Cependant, lorsque nous avons augmenté ce nombre de manière excessive (jusqu'à 14 000 arbres, par exemple), la performance s'est détériorée, probablement en raison d'un overfitting. Bien que l'augmentation du nombre d'arbres ait apporté des améliorations, les gains n'étaient pas très significatifs. Dans le contexte de notre classement (ranking data challenge), chaque petit gain avait un impact, mais en dehors de cette contrainte, nous aurions pu accepter une légère baisse de performance pour réduire le coût computationnel du modèle.

De plus, LightGBM utilise toutes les fonctions d'analyse des décisions de l'arbre, ce qui nous a permis de réaliser que la fonction de décalage mensuel, que nous avions supprimée dans XGBoost, pouvait améliorer notre score. Un autre avantage de LightGBM par rapport à XGBoost est qu'il n'a pas tendance de donner de valeurs négatives. Pour bien assurer ça, nous avons intégré la technique de Poisson dans le modèle LightGBM

en définissant le paramètre objective='poisson', afin de limiter les valeurs négatives et d'améliorer notre résultat.

Enfin, on a obtenu le meilleur score de 93.64, avec la configuration : (objective='poisson', max_bin=1000, max_depth=10, learning_rate=0.1, n_estimators=10000), selon la table 1.

4.3 Autres modèles

Nous avons également testé d'autres modèles mais leurs performances n'ont pas été satisfaisantes.

Tout d'abord, nous avons essayé le modèle Random Forest. Bien que nous ayons effectué un grid search pour optimiser les paramètres, le score obtenu (182.03) a été bien supérieur à celui des autres modèles testés. Peut être c'est à cause de la simplicité relative de Random Forest par rapport aux modèles de boosting, mais également à la gestion des valeurs manquantes. En effet, Random Forest ne supporte pas les valeurs NaN, et pour traiter les lags, nous avons dû effectuer une imputation, il se peut que ça a compromis la qualité du modèle.

Nous avons également exploré le modèle Prophet de Facebook, que nous avons choisi en faisant un benchmarking. Cependant, bien que ce modèle soit réputé pour les séries temporelles, il n'a pas donné de bons résultats dans notre cas, avec un score de 237.58. Ainsi on observe que, parfois, il peut être plus avantageux de développer un modèle personnalisé plutôt que de se reposer sur des solutions préexistantes, même si elles sont populaires.

Enfin, nous avons testé le Poisson Regressor, pensant que ce modèle pourrait être utile pour éviter la prédiction de valeurs négatives. Par contre, les résultats ont été incohérents et les prédictions ne correspondaient pas aux attentes. Alors, après ces essais, nous avons privilegié d'integrer la technique de régression de Poisson dans le modèle LightGBM (objective='poisson'), ce qui s'est avéré être beaucoup plus efficace pour notre tâche de prédiction, comme mentionné precédement.

5 Conclusion

Ce data challenge nous a appris plusieurs choses importantes. LightGBM s'est révélé être le modèle le plus adapté et performant pour notre problème (avec le meilleur score de 93.64), notamment en prenant en compte les capacités limitées de nos ordinateurs. Cela souligne l'importance de choisir un modèle non seulement en fonction de la nature des données, mais aussi en fonction des ressources informatiques disponibles : un modèle trop complexe pourrait rapidement dépasser les capacités de calcul, rendant le projet difficilement réalisable.

Un autre point fondamental a été la recherche des meilleurs paramètres pour le modèle. Réaliser un grid search avec cross validation permet d'entraîner un modèle de la meilleure façon possible, en prenant le temps de tester et d'ajuster les hyperparamètres, on peut avoir un impact significatif sur la qualité des prédictions.

La gestion des données manquantes a également été un point clé. Supprimer des valeurs manquantes n'est pas toujours la meilleure option, parce que, dans certains cas, les imputer permet d'obtenir de bien meilleurs résultats que les supprimer. De plus, avec les données temporelles, les lags se sont révélés essentiels pour capturer la dynamique des séries et améliorer la précision des prévisions.

En conclusion, ce projet a montré que réussir un travail de data science demande une bonne maîtrise des outils, un choix réfléchi des méthodes adaptées aux contraintes techniques et une collaboration efficace pour s'adapter aux objectifs du projet.

6 Annexes

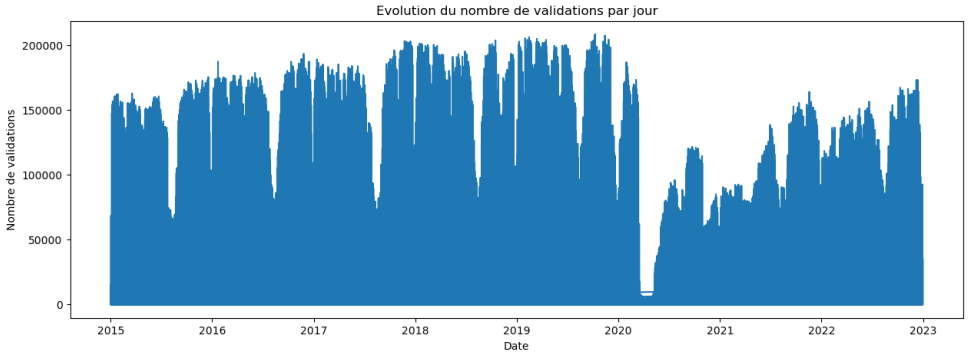

FIGURE 3 – Evolution du nombre de validations par jour - comprend toutes les années

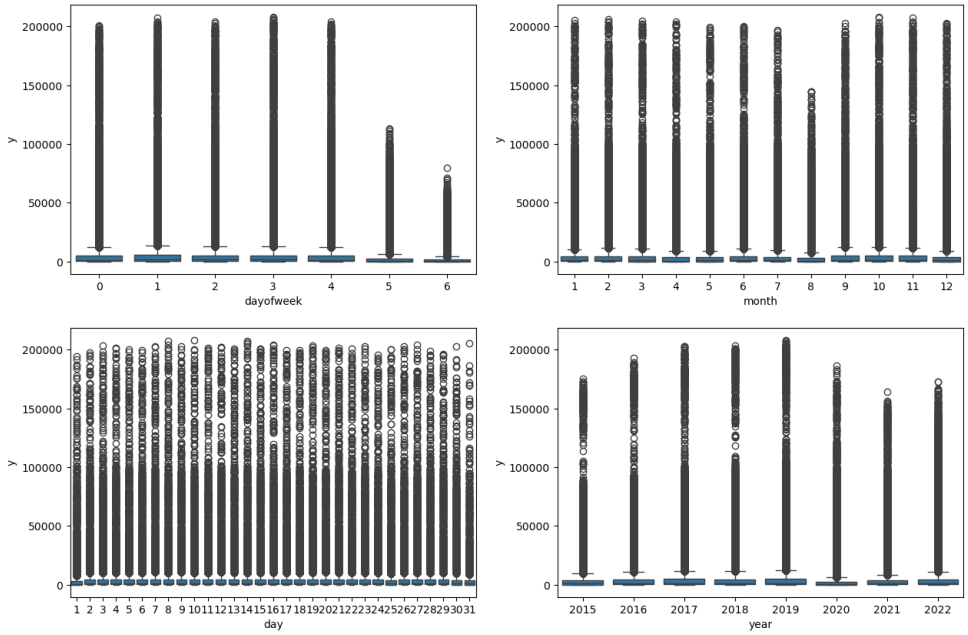

FIGURE 4 – Boxplots

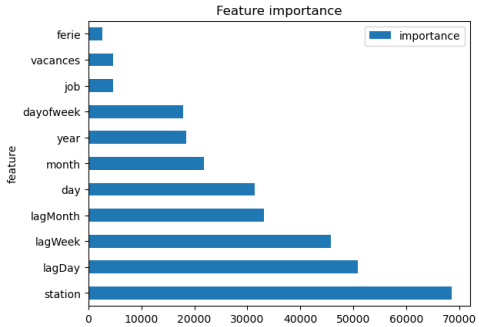

FIGURE 5 – Exemple Feature Importances LightGBM

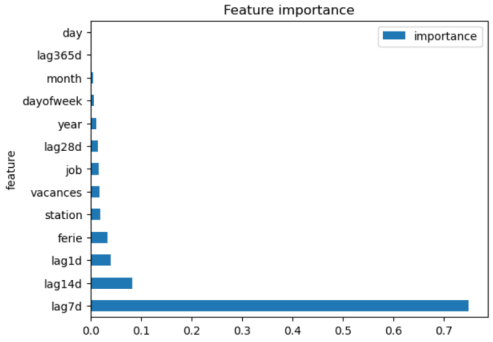

FIGURE 6 – Exemple Feature Importances XGBoost avec lags inutilisés

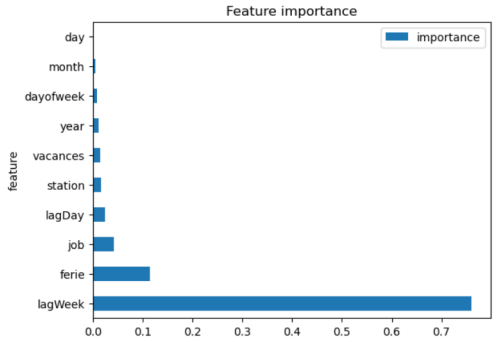

FIGURE 7 – Exemple Feature Importances XGBoost juste avec les lags pertinents

Modèle	**Paramètres**	**Score**
LightGBM avec les meilleurs hyperparamètres	objective='poisson', random_state=42, max_bin=1000, max_depth=10, learning_rate=0.1, n_estimators=10000	93.64
LightGBM avec plus n_estimators mais moins score	objective='poisson', random_state=42, learning_rate=0.1, n_estimators=14000, max_bin=10000	94.31
LightGBM avec de nombreux paramètres testés	min_data_in_leaf=50, num_leaves=200, objective='poisson', random_state=42, max_bin=1000, n_estimators=800, learning_rate=0.1, max_depth=10	98.06
XGBoost avec lags et les meilleurs hyperparamètres	max_depth=10, subsample=0.7, n_estimator=700, learning_rate=0.1	104.74
XGBoost avec lags et valeurs négatives mises à 0	learning_rate=0.05, n_estimators=700	106.59
XGBoost avec lags	learning_rate=0.05, n_estimators=700	166.93
Random Forest avec lags	n_estimators=500, max_depth=10, min_samples_split=2, bootstrap=True, random_state=42	182.03
XGBoost sans lags	n_estimators=1000, learning_rate=0.01	225.65
Prophet	–	237.58
XGBoost sans lignes avec NaN	n_estimators=700, max_depth=10, subsample=1, learning_rate=0.1	39464.87

TABLE 1 – Comparaison des modèles avec leurs hyperparamètres et scores.