Rapport Project INF442-7

Hacking the Paris Metro

Sylvanus Mahe
Tzu-Yi Chiu

21 mai 2019

Table des matières

1 Introduction

Le projet sur lequel nous avons travaillé dans le cadre du cours sur les algorithmes pour le traitement de données en c++ porte sur la RATP. Il s'agissait d'un projet plus ou moins ouvert dont l'orientation dépendait principalement de nos connaissances et des données dont nous disposions. Dans les lignes qui suivent nous détaillerons des différentes étapes de notre travail (de la collecte des données à l'extraction d'information).

2 Collecte des données

2.1 Données théoriques

Les informations concernant les stations, les horaires prévus de trains et bus et les routes de chaque ligne se touvent aisément sur le site de la RATP. Les informations sont stockées dans des fichiers différents sous forme de tableaux, ce qui nous a incité à utiliser le package **pandas** en python pour bien les ranger, mélanger et visualiser de manière efficace.

2.2 Données en temps-réel

Pour accéder aux données de la RATP, il nous fallait faire une demande auprès du service **opendata** pour activer le droit à l'API. Après avoir reçu l'accord de la RATP, nous avons pu finalement envoyer des requêtes de type SOAP au serveur en utilisant le logiciel Postman pour obtenir des réponses dont nous pouvions tirer quelques informations intéressantes. Il y a 3 principaux services : **getLines**, **getStations**, et **getMissionsNext**.

Nous nous sommes intéressés particulièrement au service getMissionsNext qui, pour une gare donnée, nous renvoyait un fichier .xml comportant les informations des 5 prochains trains passant par cette gare, par exemple le temps de passage, la direction du train et le sens de circulation. Pour traiter ces réponses, nous avions adopté la démarche suivante :

1. **Extraction d'informations des fichiers xml**

   Comme nous voulions récupérer seulement les informations des trains en temps-réel, pour une gare donnée, nous devions réussir à exploiter la réponse xml (constituée d'énormément de balises), c'est-à-dire en tirer le nom de la gare, le quai, la direction, le sens et l'heure de passage seulement pour le train à quai. Pour ce faire, nous avons codé en c++ un programme **current_train_per_station** qui reçoit comme argument le nom d'un fichier xml et stocke ces informations dans un fichier csv.

2. **Utilisation Postman**

   Le logiciel Postman possède un très gros avantage : on peut créer une

collection pour regrouper des requêtes différentes et les envoyer successivement de manière automatisée. On peut même choisir le nombre d'itérations et le décalage entre chaque requête. Ayant cet outil en main, pour simplifier la démarche, nous avions commencé par travailler sur RERA et RERB. Nous avions écrit les requêtes de getMissionsNext pour toutes les gares de chaque ligne et les avons respectivement enregistré dans une collection. Le décalage entre chaque requête est choisi de sorte que, pour une gare donnée, nous avions ses informations au moins pour chaque minute. Comme nous avons 46 requêtes pour RERA et 47 pour RERB, le décalage est autour d'une seconde.

C'est à la fin de l'envoi de toutes les requêtes que nous pouvions exporter le résultat sous forme de .json après avoir au préalable réalisé des **tests** avec *une syntaxe particulière* dans la partie des requêtes. Ainsi nous devions transformer le résultat du fichier json pour en tirer les résponses xml pour toutes les gares de la collection.

3. Extraction d'informations des fichiers json

Nous avions écrit un programme c++ **parse_json** qui prend comme argument un fichier json et qui enregistre, pour toutes les gares, la réponse sous forme de xml dans un répertoire temporel. Ensuite nous avions écrit un script en Bash qui combine ce programme avec **current_train_per_station** exécutant sur tous les fichiers xml dans le répertoire temporel et stockant les informations nécessaires dans un fichier .csv.

4. Automatisation de l'envoi des requêtes et l'extraction des fichiers

L'utilisation de Postman ne pose aucun souci tant que le nombre d'itérations reste relativement petit (pas plus que 50 requêtes pour chacune des 46 gares). Or, dès que le nombre d'itérations dépasse un certain seuil, Postman subit une interruption spontanée et aucun résultat n'est mémorisé. Heureusement il existe une alternative qui s'utilise en ligne de commande : **newman**. Newman propose aussi toutes les options nécessaires sur le nombre d'itérations et le décalage entre chaque requête. Pourtant nous avions toujours constaté sa difficulté d'exporter le résultat lorsque ce dernier est de taille trop grosse. Nous avons donc choisi d'automatiser l'envoi des requêtes en faisant moins d'itérations et en bouclant plusieurs fois grâce à un script en Bash. Après plusieurs essais, nous avons finalement pu laissé l'ordinateur tourner pendant quelques jours sans avoir de bug et récupérer les informations des trains à quai.

La collecte des données était un succès, même si à cause du manque de temps, nous n'avons pu collecter les données en temps-réel que pour un peu plus que 3 jours. Par ailleurs, il y a environ la moitié des gares qui n'exposent pas d'information en temps-réel en mettant en commentaire *Les informations ne sont pas disponibles pour le moment, veuillez nous excuser pour cette gène occasionnelle*, plus nous avons parfois rencontré des problèmes de connexion de Wi-Fi à l'école (la RATP nous a

accordé l'accès à l'API seulement pour certaines adresses IP). Tout ceci a rendu les données incomplètes.

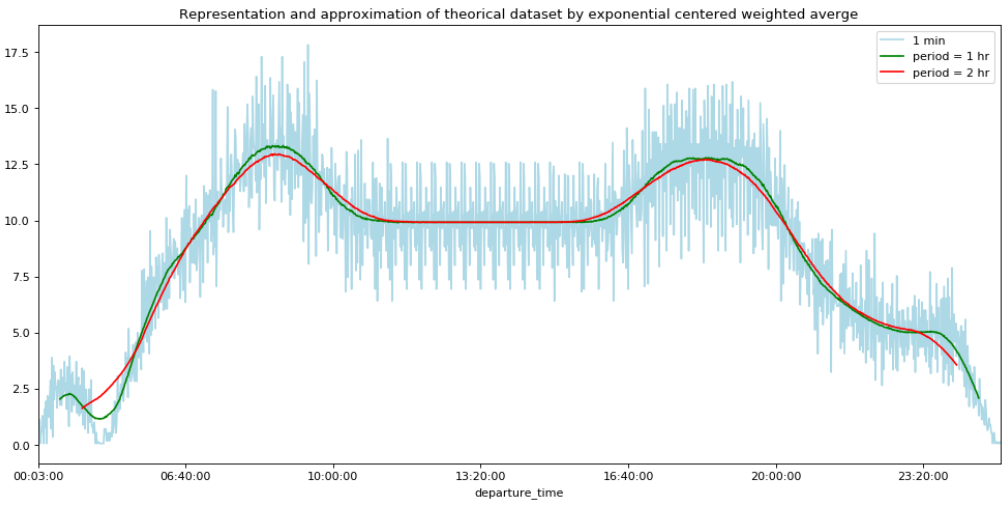
3 Traitement et Analyse des données

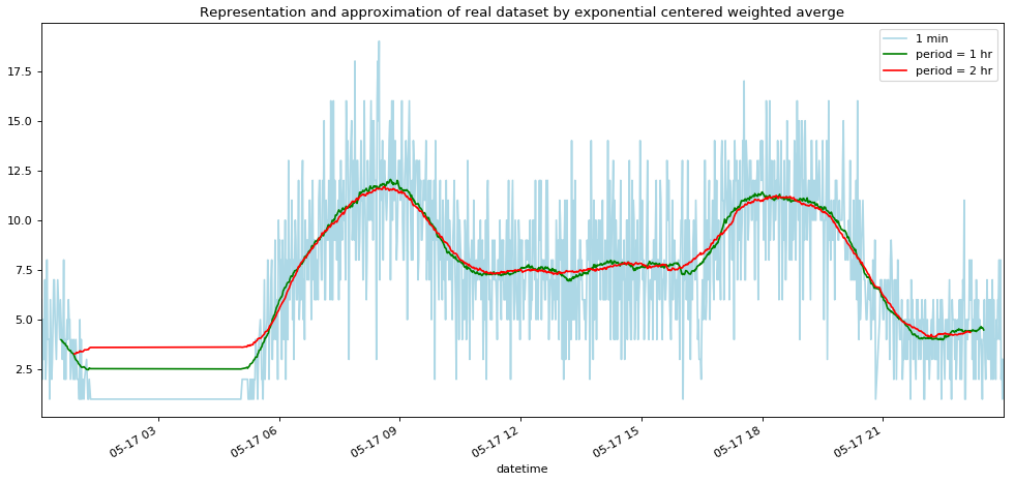
Une fois les données collectées nous pouvions entamer la deuxième grande étape de notre projet, à savoir le traitement et l'analyse de celles-ci. Pour ce faire, il fallait comprendre les différents liens entre les nombreux fichiers que nous possédions. Pour ce qui est des données théoriques, elles étaient reparties par classes et comportaient pour certaines des attributs communs. Pour la visualisation en python nous avons entrepris de les fusionner toutes pour en avoir une vue globale. A l'aide de la librairie pandas nous avons pu afficher ces données, les nettoyer, et enfin les fusionner.

Les nouvelles données qui en résultèrent fut très utiles à la compréhension des données historiques (données en temps réel). En effet, elles nous ont évité de nous perdre la masse d'information historique qu'il était possible d'avoir car on connaissait les attributs dont nous n'avions pas besoin

En croisant les données historique aux données théoriques nous pouvions observer plusieurs similitudes, notamment au niveau des fréquences moyennes de passage au cours d'une journée.

Bien que comportant des irrégularités les représentations graphiques des fréquences de passage pour les données théoriques et historiques présentent la même allure (régime très élevé aux heures de pointes (6h-10h ; 17h-20h) et relativement modéré (11h-16h) et voire même bas aux autres heures) comme on peut le voir sur les graphiques ci-dessous.





On peut aussi noter que l'amplitude des fluctuations au niveau des fréquences dans les données historiques est plus importante. Cela pour bien se justifier par les nombreux retards et pannes sur la ligne (ici RERA) L'approximation utilisée à savoir les moyennes pondérées exponentielles centrées a l'avantage de supprimer les fluctuations transitoires de façon a souligner les tendances à plus long termes, mettant ainsi en évidence les régimes mentionnés plus haut.

Une piste intéressante pour nous était de prédire le retard ou non d'un train. Pour ce faire, nous avons exploité les données fusionnées pour y contruire une nouvelle colonne qui indique si oui ou non le train est en retard. L'idée ici est en croisant les horaires théoriques et historiques, d'identifier les trains dont la date de départ apparaît dans les données théoriques mais pas dans les données historiques. Ces trains sont donc considérés comme en retard.

À partir de cette nouvelle information et des heures historiques, il est possible de calculer la durée de retard. C'est ce que nous avons fait grâce à la fonction **delay_compute** (cf. jupyter notebook : **INF442_PI.ipynb**).

4 Applications : Régression logistique

4.1 Retard/Non retard

Nous avons implémenté un algorithme de régression logistique en c++ en utilisant la librairie **eigen**. Cet algorithme permet de prédire le retard ou non d'un train, étant donnée une gare et une heure spécifiques. Pour entraîner notre algorithme, nous avons utilisé les données précedentes en veillant à ce qu'il y ait deux jeux de données distincts. Un pour l'apprentissage et l'autre pour les tests. Après l'avoir éxecuté, on pouvait constacter que le taux de réussite était relativement faible, environ 10 pourcents en moyenne. Ce taux faible est principalement dû au fait que les données ne sont pas linéairement séparables. Notre modèle de classification s'avère donc insuffisant pour un jeu de données aussi complexe. Ci-dessous une illustration de la répartition de notre nuages de points à classifier.

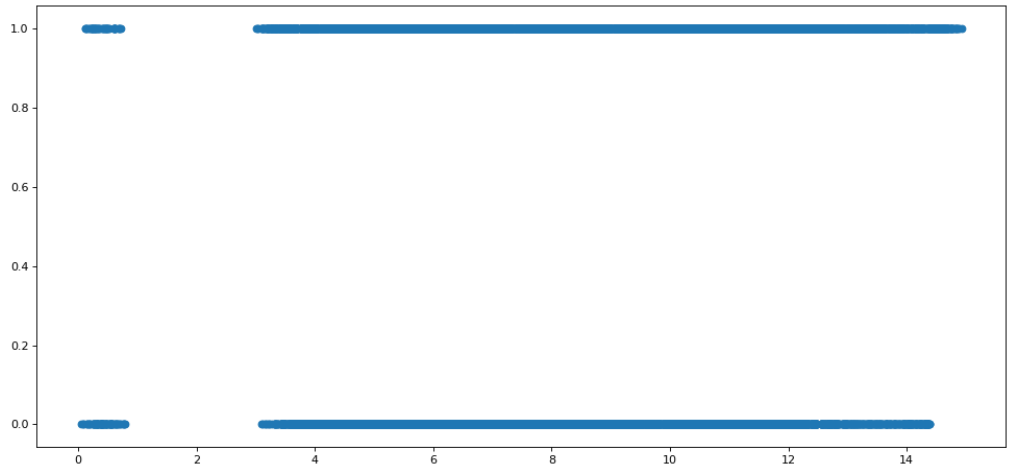


FIGURE 1 – Label en fonction du temps de départ

4.2 Retard plus/moins de 10 mins

Ici, nous avons adopté la même approche que précedemment en séparant les données de test des données d'apprentissage. Après l'exécution de la régression logistique, on observe le même taux de réussite, bien que les fluctuations suivant le label aient été réduites avec un plus gros retard. Ce résultat confirme le précédent, c'est-à-dire que les données ne sont pas linéairement séparables.

5 Bilan

L'étude de ce projet nous a permis de nous familiariser avec les outils des sciences de données et de nous confronter à un cas pratique d'analyse de données, d'autant plus que ces données sont utilisées par des millions de personnes. De la collect à l'analyse, puis à l'extraction d'information méconnue, nous avons pu vivre toutes les étapes du travail. Aussi, nous avons pu toucher aux difficultés liées à l'analyse de données incomplètes ainsi qu'à la sélection d'un modèle adapté à celles-ci.