ENSEIRB-MATMECA

Filière informatique, $2^{ème}$ année

Projet au Fil de l'Année

Rapport de PFA

Application hybride pour rediriger les utilisateurs lors des pannes de transport en commun sur Bordeaux Métropole

Pierre ROUX
Romain RAMBAUD
Victor SAINT GUILHEM
Mehdi BOUNAKHLA
Mohammed RIHANI
Nathan REAVAILLE

`Clients` Hassen BENSALEM, Raphaël CHERRIER
`Encadrant` Toufik AHMED

Mars 2016

Table des matières

1 Introduction

Ce projet a été réalisé pour l'entreprise Qucit, une startup bordelaise. Il consistait à développer une application mobile multi-plateforme permettant de rediriger les usagers en cas de panne de transport sur le réseau de Bordeaux Métropole.

Le projet a été réalisé entre le mois de Novembre 2015 et le mois d'Avril 2016. Nous avons dans un premier temps rédigé un cahier des spécifications, puis nous sommes passés au développement. Ce dernier a été globalement découpé en plusieurs grandes parties.

Une première étape consiste en la collection de données à partir de sources différents (Infotbc, Twitter, Qucit, ...) afin d'avoir une vision globale de l'état du réseau de la ville. Il faudra ensuite détecter les pannes dans le réseau de transport de bordeaux métropole.

La deuxième étape consiste à développer une application mobile simple multi-plateforme (basée sur la plateforme Polymer de Google) afin de permettre aux utilisateurs d'enregistrer leurs trajets habituels et de recevoir des notifications si ce trajet est affecté par une panne du réseau. Les notifications permettent non seulement d'informer l'utilisateur de l'existence d'un problème, mais aussi de lui proposer un itinéraire alternatif pour son déplacement.

Ce rapport est découpé en plusieurs grands chapitres. Dans un premier temps, nous présenterons notre gestion de projet. Puis nous enchaînerons sur la réalisation de ces différentes parties. Enfin, nous finirons par un bilan contenant la comparaison entre notre livrable et le cahier des charges.

2 La gestion de projet

Ce projet a été réalisé par une équipe de six personnes. De ce fait, nous avons dû mettre en place une certaine organisation afin de s'assurer de la bonne réalisation du projet.

2.1 Organisation de l'équipe

Comme indiqué dans notre cahier des spécifications, nous avons décidé d'utiliser la méthode SCRUM pour le suivi de notre projet. Cette décision a impacté notre façon d'aborder le projet sur toute sa longueur.

2.1.1 Répartition de l'équipe

En méthode SCRUM, la répartition des tâches se fait au fur et à mesure de l'avancement du projet, pendant les réunions de début de sprint (voir le cahier des spécifications pour le fonctionnement de la méthode SCRUM). Afin d'optimiser le temps, nous avons décidé en début de projet de répartir les membres du groupes plusieurs équipes :

Partie front-end : contenant le développement de l'interface utilisateur.

Partie back-end : contenant la collecte et le traitement des données, la gestion des itinéraires alternatifs et le développement de la partie serveur.

L'idée de cette séparation était surtout de séparer l'apprentissage des différentes technologies. En effet, pour la plupart des membres de l'équipe, l'ensemble des langages ou des frameworks qui allaient être utilisé n'étaient pas maîtrisés. Cette répartition nous a donc permis de gagner du temps tout au long du projet. Le fait de séparer le back-end et le front-end nous a aussi permis de les développer en parallèle.

2.1.2 Phases de travail

Globalement, le PFA s'est déroulé en plusieurs phases de travail :

Phase 1 : Cahier des charges : la première partie du projet a été consacrée à la rédaction du cahier des charges. Durant cette partie, les différents membres de l'équipe ont pu confronté leur différent point de vue avant de se mettre d'accord sur la solution à appliquer. Après plusieurs rencontres avec notre responsable pédagogique et le client, nous avons pu fixer les besoins fonctionnels et non fonctionnels du projet.

Phase 2 : Le développement : une fois le cahier des charges finalisé, l'équipe a commencé le développement de l'application. Nous nous sommes divisés le travail comme indiqué précédemment.

Phase 3 : Rédaction du rapport et de la documentation : en fin de projet, en parallèle avec la fin du développement, nous avons commencé à écrire le rapport et la documentation du projet. L'important était de fournir un rapport de qualité à notre encadrant, et une bonne documentation au client afin qu'il puisse exploiter notre code.

2.2 Organisation du travail

Comme indiqué précédemment nous avons utilisé la méthode SCRUM pour notre suivi de projet.

2.2.1 Moyen de communication

Afin de s'assurer une bonne communication au sein du groupe, nous avons dû utiliser différents de communication :

Réunions hebdomadaires : une à deux réunions hebdomadaires avec l'équipe nous a permis de s'assurer de l'avancement du projet, et de suivre les tâches qui étaient assignées à chacun. Ces réunions correspondent aux mêlées de la méthode SCRUM.

Réunions bi-hebdomadaires : une réunion toute les deux semaines avec notre client afin de conclure et de commencer nos sprints. Durant ces réunions, des présentations régulières du livrable ont été réalisée. Séparément, nous avons effectué des réunions à la même fréquence avec notre encadrant pédagogique.

2.2.2 Outils utilisés

Pour l'organisation de notre projet, nous avons utilisé plusieurs outils qui sont disponibles sur Internet :

Trello : Cet outil est assez régulièrement associé à la méthode SCRUM. Il permet une représentation claire et simple des tâches à réaliser sur le projet. En l'occurrence, nous avons rentré l'ensemble de nos user-story dessus, répartie entre différentes catégories : Product Backlog, Sprint Backlog, A tester etc... Cet outil était mis à jour à chaque réunion de fin de sprint.

Telegram : outil de chat en ligne très répandu. Nous avons décidé de l'utiliser pour sa facilité d'installation, sa mise à disposition sur de multiples plateformes, et ses salons de conversation simples d'utilisation.

Git Hub : afin de stocker notre serveur et permettre le développement collaboratif, nous avons utilisé un dépôt Git. Ce dernier a été passé en privé afin de ne pas laisser notre code accessible à tout le monde.

3 Détection des Pannes

3.1 Collecte

Dans le but de détecter les perturbations sur le réseau de la Cub, avec le plus de précision possible et avec une fiabilité correcte, nous avons essayé de collecter des données de sources différentes. Nous collectons des données sur les comptes de la TBC, qui gère 6 comptes Twitter sur lesquels elle informe régulièrement les usagers de l'état de circulation de certaines lignes du réseau.

Elle possède également un site web, www.infotbc.com, sur lequel figurent certaines informations lors des perturbations du réseau. Enfin, le site mobile de la Cub mobilinfotbc, qui recense tout ou une partie des problèmes de circulation des bus.

3.1.1 Collecte sur les sites de la TBC

Pour collecter ces données nous utilisons des scrapers : des robots qui vont parcourir les pages du sites, parser le code `html` de ces pages et extraire les informations qui nous intéressent. Pour coder ces scrapers nous avons utilisé la bibliothèque **Scrapy**. Nous recoltons ces données mais elle ne sont pas utilisées par la suite. Nous n'avons pas eu le temps de les traiter.

3.1.2 Collecte des Tweets

Pour collecter les informations depuis Twitter nous utilisons l'API fournie par le réseau social au travers d'un module python **Tweepy**. Nous collectons les Tweets sur 6 comptes Twitter de la TBC, un par ligne du réseau : les Tweets concernant les lignes de tram A, B et C et des lignes de bus lianes 1, lianes 3 et lianes 8.

Dans un premier temps nous avons collecté l'intégralité des Tweets postés sur les 6 comptes depuis leurs création. Cela représente, à la date où nous l'avons fait, environ 10 000 Tweets. Ces Tweets nous on servi à nourrir nos algorithmes d'apprentissage machine et d'analyser leur contenu pour nos algorithmes d'extraction d'informations.

L'application en elle même n'a pas besoin de ces données. Elle collecte toutes les 30 secondes le dernier Tweet de chacun des 6 comptes de la TBC. Ces Tweets sont ensuite, si ils sont nouveaux, c'est-à-dire que l'application ne les a pas déjà récupérés, stockés dans la base de données du serveur. Voici quelques exemples des Tweets qu'il est possible de voir sur ces différents comptes. Ce sont ces Tweets qui nous permettent de détecter les perturbations sur le réseau :

FIGURE 1 – Exemple tweet

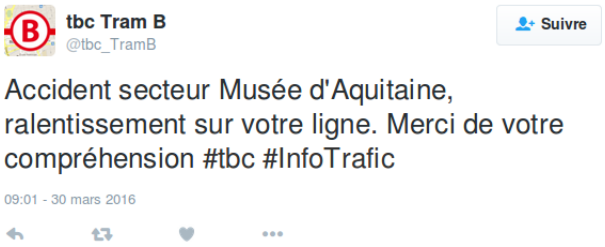


FIGURE 2 – Exemple tweet

FIGURE 3 – Exemple tweet

3.2 Analyse et Traitement des tweets

Analyser des tweets est un problème difficile, puisqu'il s'agit d'analyser du langage naturel. De plus c'est un/une employé(e) de la TBC qui écrit ces Tweets et nous ne somme même pas certains qu'il s'agisse de la même personne qui gère les différents comptes. Il peut donc y avoir de fortes différences syntaxiques entre les différents comptes. De plus, même sur des comptes officiels, les Tweets s'apparentent plus à du langage oral. Il y a des abréviations, des phrases mal construites, des fautes d'orthographes. Bref, quelques éléments qu'il est important de prendre en compte dans la suite de nos analyses.

Enfin, il y a une grande différence entre les comptes concernant les tram et les comptes concernant les bus, du fait de la nature différente du transport. Dans notre application le traitement des Tweets est généralement bien plus performant et précis dans l'analyse des tram car elle à été initialement conçu plus particulièrement pour ce mode de transport. C'est pourquoi dans les parties suivantes le cas général traite plus particulièrement du cas des tramways. Néanmoins le cas des bus est pris en compte.

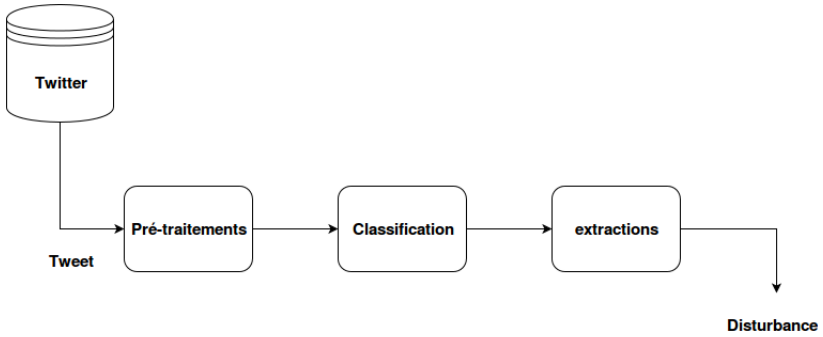


FIGURE 4 – Processus de traitement des tweets

3.3 Classification

La première étape de l'analyse des Tweets va être de les classer selon l'information que la personne qui l'a rédigé veut faire passer. Est-ce que ce Tweet annonce la panne d'un tram ? Un ralentissement sur une ligne de bus ? Un interruption du trafic ? La reprise du trafic sur une ligne ? Nous avons décidé de définir quatre classes de Tweets pour les trams.

Ces classes sont : `panne`, `ralentissement`, `reprise` et `non pertinent`. Elles sont définies comme suit :

— Un Tweet appartient à la classe `panne` s'il annonce une interruption du trafic sur une partie ou sur l'intégralité de la ligne, i.e. plus aucun tram ne circule ;

— Un Tweet appartient à la classe `ralentissement` s'il annonce un ralentissement du trafic sur une ligne et/ou une portion, i.e. les tram circulent mais avec une fréquence plus faible ;

— Une Tweet appartient à la classe `reprise` s'il annonce une reprise du trafic a la normale, suite a une panne ou un ralentissement ;
— Une Tweet est classé `non pertinent` s'il ne concerne pas l'état de circulation du réseau.

Voici quelques exemples :
ces Tweets, dans notre modèle appartiennent à la classe `panne` :
— Ligne Tram A interrompue entre Mériadeck et Stalingrad.
— Rame en panne à Jardin Botanique Ligne Tram A interrompue entre Galin et Stalingrad.
Les tweets :
— Retour à une fréquence normale sur l'ensemble de la Ligne A.
— Fin de l'intervention des pompiers, personne évacuée. La Ligne C circule normalement.
sont classé comme étant des `reprises`.
Les Tweets ci-dessous sont classé dans `ralentissement`
— Fin de l'intervention des pompiers, personne évacuée. La Ligne C circule normalement.
— FIN de MANIFESTATION : retards depuis STALINGRAD vers les antennes. Merci de patienter...
Les Tweets `non pertinent` sont, par exemple :
— Merci à tous pour votre participation à notre jeu "Je monte/Je retweete" ! RDV semaine prochaine pour l'annonce des gagnants ! #gagnerPassTBC.
— Toute l'équipe TBC vous présente ses meilleurs vœux de mobilité ! En route pour 2013.

Pour classer ces Tweets nous utilisons des algorithmes dit de "Machine Learning", plus particulièrement d'apprentissage supervisé. Ces algorithmes vont adapter leurs analyses et leurs comportements aux réponses, en se fondant sur l'analyse de données (des exemples). Ces données vont être classées par un "expert" selon le modèle définis plus haut. Le processus se passe en deux phases. Lors de la première phase (hors ligne, dite d'apprentissage), il s'agit de déterminer un modèle des données étiquetées. La seconde phase (en ligne, dite de test) consiste à prédire l'étiquette d'une nouvelle donnée, connaissant le modèle préalablement appris.

3.3.1 Premiers constats

Nous avons dans un premier temps essayé de classer les Tweets sans traitement préalable, en faisant l'hypothèse qu'un Tweet ne pouvait appartenir qu'à une seule classe. Mais cette hypothèse c'est avéré fausse. En effet, certains Tweets peuvent appartenir a plusieurs classes.

Par exemple, le Tweet :

`Reprise du trafic sur la ligne. Retards à prévoir sur votre ligne #tbc #InfoTrafic`

appartient à la fois à la classe `reprise` et à la classe `ralentissement`. Autre exemple, avec ce Tweet :

`Reprise du trafic entre Buttiniere et La Morlette. Ligne A interrompue entre La Morlettte et Dravemont. http://t.co/OQlL6kEw #tbc #tbcTramA`

qui peut être classé dans `panne` comme dans `reprise`.

D'autre part, il y a beaucoup d'éléments qui ne sont pertinents dans les Tweets. Les hashtags et les liens n'apportent aucune information sur la classe d'un Tweet par exemple, ils peuvent même gêner son processus de classification.

Enfin, il faut garder à l'esprit qu'il y a une personne derrière ces Tweets, et qu'elle est susceptible de faire des fautes d'orthographe ou de frappe.

Il est donc nécessaire d'effectuer un pré-traitement des Tweets avant de pouvoir les classer proprement.

3.3.2 Pré-Traitements

Découpage des Tweets Pour régler ce problème de tweet multiclasse, nous avons décidé de découper les tweets en phrases, au sens grammatical du mot. Nous utilisons une fonction de la librairie python `Pattern`. Cette fonction (Tokenize) fait la différence entre un point de fin de phrase et un point d'abréviation par exemple. Chaque partie va ensuite être classée séparément.

Nettoyage des tweets Nous supprimons les éléments génants contenues dans le tweets. Les hashtags, les liens et les `@`.

Correction orthographique Nous avons également implémenté un correcteur orthographique simple, qui fonctionne avec un dictionnaire. Il va, pour chaque mot du tweet calculer une distance avec les mots du dictionnaire. La correction sera le mot lui même s'il figure dans le dictionnaire, sinon le mot avec la distance la plus faible. Là distance que l'on considère est la distance de Levenshtein.
On appelle distance de Levenshtein entre deux mots M et P le coût minimal pour aller de M à P en effectuant les opérations élémentaires suivantes :

— substitution d'un caractère de M en un caractère de P ;
— ajout dans M d'un caractère de P ;
— suppression d'un caractère de M.

On associe ainsi à chacune de ces opérations un coût. Le coût est toujours égal à 1, sauf dans le cas d'une substitution de caractères identiques. Ceci nous permets de corriger les "petites" fautes d'orthographes ou de frappes, ce qui nous permet d'éviter des erreurs lors de l'extraction des nom d'arrêts par exemple.

Pour reprendre les exemples de la partie 3.3.1,

```
Reprise du trafic sur la ligne. Retards à prévoir sur votre ligne #tbc
#InfoTrafic
```

```
Reprise du trafic entre Buttiniere et La Morlette. Ligne A interrompue
entre La Morlettte et Dravemont. http://t.co/OQlL6kEw #tbc #tbcTramA
```

sont après traitement de la forme :

```
Reprise du trafic sur la ligne.
Retards à prévoir sur votre ligne
```

et

```
Reprise du trafic entre Buttiniere et La Morlette.
Ligne A interrompue entre La Morlettte et Dravemont.
```

3.3.3 Classifieur Bayésien Naïf

L'algorithme que nous avons utilisé pour cette classification s'appelle algorithme de classification naïve Bayésienne. L'avantage du classifieur bayésien naïf est qu'il requiert relativement peu de données d'entraînement pour estimer les paramètres nécessaires à la classification. Nous n'avons pas implémenté nous même cette algorithme, nous avons utilisé le module python **Sciki-Learn**.

La classification Bayésienne naïve de textes est une approche probabiliste de classification simple. Cette approche est basée sur un modèle probabiliste dérivant du théorème de Bayes qui fait l'hypothèse que les mots qui apparaissent dans un Tweet sont indépendants les uns des autres. Ce qui n'est pas vrai en pratique. Malgré leur modèle de conception « naïf » et ses hypothèses de base extrêmement simplistes, les classifieurs bayésiens naïfs ont fait preuve d'une efficacité plus que suffisante dans beaucoup de situations réelles complexes.

Description de l'algorithme De manière abstraite, le modèle probabiliste pour un classificateur Bayésien est un modèle conditionnel. Il se base sur la règle de Bayes qui s'énonce de la manière suivante :

$$P(A|B_1, B_2, \ldots, B_n) = \frac{P(B_1, B_2, \ldots, B_n|A) \cdot P(A)}{P(B_1, B_2, \ldots, B_n)} \tag{1}$$

La probabilité d'avoir l'événement A étant donné $B_1, B_2, \ldots, B_n$ est donnée par le rapport entre la probabilité d'avoir les événements $B_1, B_2, \ldots, B_n$ étant donné A et la probabilité que $B_1, B_2, \ldots, B_n$ se soient produits. Le dénominateur ne dépendant pas de A, on peut le considérer comme constant.

Le numérateur peut encore se décomposer de la manière suivante :

$$\begin{aligned} P(B_1, B_2, \ldots, B_n|A) \cdot P(A) &= P(A, B_1, B_2, \ldots, B_n) \\ &= P(A) \cdot P(B_1, A) \cdot (B_2, \ldots, B_n|A, B_1) \\ &= P(A) \cdot (B_1, A) \cdot (B_2|A, B_1) \cdot (B_3, \ldots, B_n|A, B_1, B_2) \end{aligned} \tag{2}$$

La décomposition se termine lorsque l'on a parcouru l'ensemble des classes $B_1, \ldots, B_n$.
Le caractère "naïf" de ce théorème vient du fait qu'on suppose l'indépendance des différentes classes $B_i, \ldots, B_j$, ce qui ce traduit par,

$$P(B_i|A, B_j) = P(B_i, A) \tag{3}$$

Cette hypothèse permet également d'écrire,

$$\begin{aligned} P(A, B_1, B_2, \ldots, B_n) &= P(A) \cdot (B_1|A) * P(B_2|A) \cdot \ldots \cdot (B_n|A) \\ &= P(A) \prod_{i=1}^{n} P(B_i|A) \end{aligned} \tag{4}$$

Supposons maintenant que nous disposons de n classes de Tweets. Déterminer à quelle classe C_i sera associé un Tweet T revient à calculer la probabilité d'appartenance du Tweet T à la classe C_i. En se basant sur le théorème vu ci-dessus, on peut calculer cette probabilité de la façon suivante :

$$P(C_i|T) = \frac{P(T|C_i) \cdot P(C_i)}{P(T)} \tag{5}$$

Dans cette formule, $P(C_i|T)$ représente la probabilité d'appartenance du Tweet T à la classe C_i qui peut être également déterminée en évaluant la fréquence d'apparition des mots du Tweet T qui sont associés à la catégorie C_i. $P(T|C_i)$ est la probabilité selon laquelle, pour une catégorie donnée, les mots du Tweet T sont associés à la catégorie C_i. $P(C_i)$ est la probabilité qui associe le Tweet T à la classe C_i indépendamment du document. Ainsi pour déterminer à quelle classe appartient T, il faut calculer $P(C_i|D)$ pour chacune des catégories. Étant donné que $P(T)$ reste constante pour toutes les classes, calculer $P(C_i|T)$ se résume juste au calcul de $P(T|C_i) \cdot P(C_i)$.

En considérant que le Tweet T est composé d'un ensemble de mots que nous noterons $\omega_1, \omega_2, \ldots, \omega_m$, calculer $P(T|C_i)$ revient à calculer le produit des probabilités d'apparition de chaque mot ω_i dans la catégorie C_i. Ce calcul se justifie par l'hypothèse selon laquelle tous les mots apparaissent indépendamment les un des autres dans un Tweet. Ce qui permet d'écrire :

$$P(T|C_i) = P(\omega_1|C_i) \cdot P(\omega_2|C_i) \cdot \ldots \cdot \ldots \cdot P(\omega_m|C_i) \tag{6}$$

Pour chacune des catégories $P(\omega_i|C_i)$ est le rapport entre le nombre de fois que le mot ω_i apparaît dans la classe C_i et le nombre total de mots que comprend la classe C_i. $P(C_i)$ est calculé en divisant le nombre total de mots pour la catégorie C_i par la somme du nombre total de mots dans toutes les catégories. D'où :

$$P(C_i|D) = P(\omega_1|C_i) \cdot P(\omega_2|C_i) \cdot \ldots \cdot P(\omega_m|C_i) \cdot P(C_i) \tag{7}$$

Ce calcul est effectué pour chaque catégorie et on considère la probabilité la plus élevée pour choisir la classe associée au Tweet T que l'on souhaite classer.

Le calcul ainsi présenté se justifie par l'hypothèse selon laquelle tous les mots apparaissent indépendamment les uns des autres dans le document. D'où le caractère naïf de la classification. En réalité, la probabilité d'apparition d'un mot est liée aux mots précédents.

Données d'apprentissage Pour réaliser les données d'apprentissage nous avons classé à la main environ 250 Tweets qui après traitement représentent 310 objets que nous avons classé en `panne`, `reprise`, `ralentissement` ou `non pertinent`, en essayant de garder le même nombre de Tweets dans chaque classe.

Distribution des Tweets		
Classe	Nombre	proportion
Panne	86	28%
Reprise	85	28%
Ralentissement	68	22%
Non pernitant	71	23%
total	310	100%

FIGURE 5 – Répartition des Tweets d'apprentissages

À noter qu'en pratique les Tweets ne sont pas parfaitement équi-répartis dans les classes de notre modèle. Les Tweets `non pertinent` sont moins fréquents que les 3 autres classes qui elles sont relativement bien réparties.

Données de test Nous avons également constitué une base de test pour nous permettre d'évaluer les performance de l'algorithme. Elle est constituée de 70 Tweets différents de ceux d'apprentissage.

Distribution des Tweets		
Classe	Nombre	proportion
Panne	52	56%
Reprise	18	21%
Ralentissement	10	11%
Non pernitant	12	13%
total	93	100%

FIGURE 6 – Répartition des Tweets de tests

Résultats Voici les résultats de l'algorithme que nous avons mis en place sur les données de test (i.e. sur des Tweets déjà découpés) :

Résultats		
	Erreurs	Success
Pourcentages	7,6%	92,4%
Nombre	7	86

FIGURE 7 – Résultat Classification

Classe	**Décision**			
	Panne	Reprise	Ralentissement	Non Pertinant
Panne	48	1	3	0
Reprise	0	17	1	0
Ralentissement	1	0	9	0
Non Pertinant	1	0	0	12

FIGURE 8 – Matrice de confusion

La figure 6 montre les résultats de la classification sur les Tweets découpés. Le classifieur fonctionne correctement. Il a un pourcentage d'erreur relativement faible autour des 7,6%.

Autre remarque, on peut voir sur la Matrice de confusion (Figure 7) que plus de la moitié des erreurs se situent entre les classes `panne` et `ralentissement`. Ceci est du à une certaine ambiguïté dans notre modèle. Nous avons défini une `panne` comme une interruption totale du trafic sur une ligne et un ralentissement comme une réduction des fréquences de passage sur la ligne. Or certain Tweets sont flous et difficiles à classer, même pour un humain. Le problème du point de vu d'un utilisateur est que si un ralentissement sur une ligne fait passer la fréquence de passage d'un tram toutes les 5 minutes à toutes les 10 minutes il s'agit bien d'un `ralentissement`. Mais si la fréquence passe à un tram toutes les 15 où 20 minutes, cela peu être considéré comme une `panne`. Or cette information est très rarement présente dans les Tweets. C'est une des limites que l'on rencontre.

Enfin, pour les algorithmes de machine learning, la quantités des données d'apprentissages en entrée a un impact important sur les résultats de l'algorithme. Ainsi, il est fort probable qu'un augmentation du nombres de Tweet dans ces données réduise le nombre d'erreur dans la classification.

3.3.4 Pour les bus

Nous avons dû faire quelques modifications pour le traitement des Tweets des comptes gérant les bus. Comme nous l'avons dit plus haut, les Tweets sont très différents. Les modifications apportées sont détaillé ci-dessous.

La classification Nous avons légèrement modifié le modèle de classification pour les bus. En effet cela n'a pas de sens de parler de `panne` pour une ligne de bus. Si un bus tombe en panne, si une rue est coupée dans le pire des cas le bus a la possibilité de faire

un détour, de doubler le bus en panne, ... bref la ligne n'est jamais interrompue. Nous avons uniquement des `ralentissement`. La classification est donc simplifiée.

Autre point, les Tweets sont beaucoup plus normés. Dans une grande majorité, un Tweet pour annoncer une panne ressemble à cela (à quelques légères modifications près) :

```
Départ 'Gare Routière' : 7h48 non assuré. Prochain départ = 8h04
```

Il y a également très rarement des Tweets de `reprise`.

Pré-traitement Au vue de la première analyse faite plus haut, il n'est plus nécessaire de découper les Tweets, comme c'était le cas pour les tram. Les Tweets concernant les bus ne sont donc pas découpés. Ils passent par contre par les mêmes autres pré-traitements que tram.

Résultats		
	Erreurs	Success
Pourcentages	14%	86%
Nombre	7	43

FIGURE 9 – Résultat Classification BUS

Résultats Nous obtenons des résultats inférieur pour les bus un taux de succès de 86% sur 50 Tweets testé avec 109 Tweets d'apprentissages. (cf Figure 9, 10 et 11)

Distribution des Tweets		
Classe	Nombre	proportion
Reprise	13	12%
Ralentissement	75	68%
Non pernitant	21	20%
total	109	100%

FIGURE 10 – Répartition des Tweets d'apprentissages BUS

3.4 Extraction d'information

Une fois le Tweet classé, il est nécessaire d'extraire des informations. La première chose que nous voulons savoir c'est : où se passe la `panne` ? la `reprise` ? Il faut donc extraire des nom d'arrêts.

Autre information importante sur notre traitement, c'est que nous avons remarqué que certains Tweets font état d'une perturbation du réseau à l'avance. Il faut donc repérer et extraire ces informations temporelles afin de ne pas avertir un utilisateur d'une panne qui aura lieu dans 2 jours par exemple.

Distribution des Tweets		
Classe	Nombre	proportion
Reprise	7	14%
Ralentissement	33	66%
Non pernitant	10	20%
total	93	100%

FIGURE 11 – Répartition des Tweets de tests BUS

3.4.1 Extraction des nom d'arrêts

Ce que l'on peut avoir après un rapide parcours des Tweets c'est que ce n'est pas toujours possible de connaître un intervalle précis d'arrêts. Dans le meilleur des cas il y a 2 noms d'arrêts bien orthographiés dans le Tweet. Parfois il n'y en a aucun, parfois nous n'avons qu'une direction.

Le cas parfait :

`Ligne Tram B interrompue entre` **Bougnard et France Alouette**

L'intervalle de la panne est bien indiqué, les arrêts sont correctement orthographiés.

Mais bien souvent, il n'y a qu'un seul ou pas d'arrêts. Autre problème : les noms d'arrêts composés comme Gare Saint-Jean sont souvent abrégés, il y a donc plusieurs orthographes possible d'un même arrêt dans les Tweets (nous avons pu compté pour certains arrêts plus de 9 orthographes différentes !)

Cette partie du traitement des Tweets a deux objectifs. Le premier est de déterminer, étant donné un Tweet, qu'un mot ou un ensemble de mot désigne un arrêt. Le second est de relier ce mots ou cet ensemble de mot à un nom d'arrêt "réel", c'est-à-dire un mot d'arrêt référencé dans notre base de données.

Trouver les noms d'arrêts Pour cela nous utilisons une fonctionnalité de la bibliothèque **Pattern** appelé taxonomy. Nous allons grâce à cette fonction pouvoir associer un ensemble de mots à une catégorie (un taxon), que l'on a appelé "ARRET". Nous avons donc constitué un lexique de mots-clé issue des noms d'arrêts. Grâce a cette fonction nous somme capables de dire si un mot est un "ARRET" ou pas. En faisant la même manipulation pour les voisins de ce mots, on se retrouve avec une liste de caractères désignant un arrêt.

De plus, dans quelques cas, les arrêts ne sont pas nommées directement, par exemple sur ce Tweets :

`Ligne A interrompue entre La Morlettte et les Antennes.`

"les Antennes" désigne ici les arrêts "Floirac Dravemont" et "La Gardette", les terminus de la ligne A. Nous avons implémenté une fonction de correspondance qui en fonction de la ligne, traduit "Antennes" en nom d'arrêts correspondants.

Correspondances Maintenant il s'agit de faire la correspondance avec les noms d'arrêts issue de la collecte sur OpenData. Prenons un exemple pour mettre en lumière le problème. Considérons l'arrêt de la ligne de tram A : Lycée Vaclav Havel. Dans notre base de données, cet arrêt figure sous le nom de : LYCEE VACLAV HAVEL. Or dans les Tweets il peut apparaître sous différentes orthographes, selon le bon vouloir de la personne qui écrit le Tweet.
Par exemple :

— Lycée V.Havel
— Lycée Vaclav Havel
— Lycée V. Havel
— L. Vaclav Havel,
— L. V. Havel,
— Lycée V. H.
— ...

Si l'on considère chaque nom d'arrêts des lignes que l'on considère comme une classe, nous pouvons réduire notre problème à un problème de classification. Nous utilisons ici le même algorithme que pour la classification des Tweets, le classifieur naïf Bayésien. Il est efficace dans ce cas, puisque les données d'apprentissages sont très peu nombreuses et le nombre de classes important.

Nous avons constitué des données d'apprentissages en regroupant les différentes orthographes des arrêts.

Un des avantages que nous avons dans ce cas c'est que lorsque la Cub interrompt une ligne de tram, elle est obligée de le faire entre des arrêts où les voies se croisent, comme Mériadeck, Floirac Dravemont ou Peixotto. Ainsi certain noms d'arrêts apparaissent beaucoup plus fréquemment que les autres.

Nous obtenons de très bon résultats. Nous avons effectué des tests sur 52 Tweets, avec 93 arrêts. Nous somme capable d'extraire correctement 95% des arrêts. Il reste encore un problème que nous n'avons pas réussi à régler c'est que l'algorithme trouve avec une probabilité de 4,3% des arrêts qui ne sont pas présents dans le Tweet.

3.4.2 Extraction des données temporelles

Une autre information importante dans les Tweets sont les expressions temporelles. Dans certains cas, les Tweets annoncent des perturbations du réseau en avance. Lors de grèves, de manifestations ou encore de travaux. Ainsi, il ne faut pas alerter l'utilisateur à la date où l'on reçoit le Tweet.

Cette extraction est délicate puisque, comme pour les arrêts, l'écriture des dates dans les Tweets ne respecte aucune norme. Il y a des abréviations des mois, beaucoup de formats de dates différents, des références à des moments de la journée, etc.

Nous avons mis en place un extraction qui à partir d'un Tweet retourne une liste d'objets python datetime représentant une date et un erreur. Nous avons utilisé pour cette partie le module **re** qui permet de manipuler des expressions régulières.

Nous avons mis en place toute une collection d'expressions régulières pour essayer d'extraire, avec la plus grande flexibilité possible tout un panel de possibilités, néanmoins

la solution que nous avons à l'heure actuelle n'est pas complètement satisfaisante dans le sens où ces expression sont codées "en dur" dans notre algorithme. Il se contente d'essayer de traiter tout les cas auxquels nous avons pensé. Ceci étant dit, nous avons quand même décortiqué des milliers de Tweets pour essayer de rendre cet algorithme le plus complet possible.

Quelques résultats Voici quelques résultats et les limites de ceux-ci.
Première exemple : un intervalle de temps,

```
Travaux journée 21 au 23/07: Interruption Tram B de Quinconces à
Berges-Garonne (relais bus assuré).
```

Nous somme capables de récupérer et de traiter ce motif. Nous retournons :

```
[datetime.datetime(2016, 7, 21, 5, 30), 10], [datetime.datetime(2016, 7,
22, 5, 30), 10], [datetime.datetime(2016, 7, 23, 5, 30), 10]]
```

Ici, l'erreur est égale à 10 puisque d'aucune heure a été trouvée, l'heure par défaut est le début du service des tram qui est 5h30.
`Autre exemple` : Une heure de la journée,

```
Ce soir TRAVAUX nocturnes : Tram B interrompu entre Doyen Brus et Pessac
Centre dès 21h.
```

Dans ce cas nous repérons deux choses. L'expression "Ce soir" et "dès 21h". Le second étant plus précis que le premier c'est lui qui est retenue. Nous retournons :

```
[[datetime.datetime(2016, 3, 10, 21, 0), 0]]
```

Le jour est celui où le Tweet est posté, et l'heure est bien 21h. L'erreur est égale à zéro puisque nous avons une heure précise.

Nous somme également capable de détecter les deux cas de figure précédents dans un même Tweet :

```
Travaux nocturnes : Interruption Tram C de Berges du Lac à Grand Parc les
28 et 29 Octobre dès 21h.
```

Nous retournons :

```
[datetime.datetime(2016, 28, 10, 21, 0), 0],[datetime.datetime(2016, 29,
10, 21, 0), 0]
```

Dernier exemple, avec des moments de la journée : nous repérons également des expressions qui sont fréquemment utilisées dans les Tweets comme "en soirée", "ce matin", "en début d'après-midi", "fin du service". Nous avons fixé arbitrairement une heure pour chacune de ces expressions avec un erreur de 2 (Ce qui signifie plus ou moins 2 heures).

Ainsi, pour le Tweet

`Intervention des pompiers sur la Ligne A. Reprise du trafic en début de soirée.`

nous retournons :

`[datetime.datetime(2016, 4, 3, 20, 0), 1]`

Nous avons fixé le "début de soirée" à 20h avec plus ou moins 1 heure. Néanmoins ces horaires restent indicatifs.

Nous rencontrons aussi quelques limites, nous ne traitons pas tous les cas. Par exemple dans certains Tweets les mois sont parfois abrégés (Le mois "Octobre" ce transforme en "Oct"), nous ne traitons pas les durées en heures. Par exemple l'expression "de 14h à 17h" ne sera pas traitée correctement, les minutes ne sont pas prises en compte.

3.5 Résultats

Nous avons pas eu la possibilité de tester notre application en situation réelle. Néanmoins voici pour 1 Tweet très classique, l'objet `disturbance` que nous générons :

En entrée, le Tweet :

`Ligne A interrompue entre La Morlettte et les Antennes.`

Nous obtenons en sortie :

```
{
     'class\_type': 'PANNE',
  'date_start': [datetime.datetime(2016, 4, 3, 15, 33, 51, 507552), 0],
  'direction': 2,
  'duration': 'Unknow',
  'interval': ['LA MORLETTE', 'BUTTINIERE'],
  'line': 'TBC TramA',
  'state': 'PROGRESS',
  'tweet': <Collecte.tweets\_tbc.Tweet object at 0x7f679e720b10>\
    }
```

3.6 Aller plus loin

Cette partie nous a pris beaucoup de temps, l'analyse des langages naturelle est un domaine complexe, nous n'avons pas eu le temps de complètement adapter nos algorithmes et notre raisonnement aux lignes de bus. Ce serait un des premiers point à améliorer dans les versions futures.

D'autres part, nous avons vu que les seules informations sur les Tweets ne suffisent souvent pas, en tout cas avec les algorithmes que nous avons mis en place, à obtenir dans tous les cas des informations correctes, cohérentes et précises.

Du point de vue de l'analyse des Tweets, plus particulièrement dans la classification, il est possible de mettre en place des algorithmes plus récents comme les arbres renforcés

ou les forêts aléatoires qui bien souvent permettent d'obtenir de meilleurs résultats. Ces algorithmes sont plus difficiles à mettre en place. Ils sont aussi présents dans la librairie **Scikit-Learn**.

Pour aller plus loin, il serait intéressant en plus de l'analyse de Twitter, d'analyser les données en temps réels fournies par data bordeaux métropole afin de pouvoir, lors de pannes et de ralentissement ou de reprise, "vérifier" ce qu'il en est réellement sur le réseau. Pouvoir être capable de quantifier un ralentissement par exemple peut être utile à l'utilisateur et de lever l'ambiguïté entre `panne` et `ralentissement`.

On pourrait par ailleurs essayer à partir des différentes données récoltées sur des pannes de prédire leur durée par exemple.

4 Graphe statique et gestion d'itinéraires alternatifs

Le but de cette partie sera de présenter le working package sur la création des itinéraires alternatifs, et le travail effectué par l'équipe pour amener à sa réalisation. Pour résumer, l'objectif de cette partie était de proposer des itinéraires alternatifs aux utilisateurs. Le déclenchement de la recherche d'itinéraires alternatifs devait se faire lors de la détection d'une panne, et à condition que celle ci appartienne à un utilisateur.

4.1 Analyse du problème

Ce module doit servir à répondre à une des problématiques essentielles du projet. En effet, lors de la présentation du projet par le client, ce dernier recherchait une expérience utilisateur complète sur cette application. Et au delà du fait de détecter les pannes du réseau sur les trajets quotidien de l'utilisateur, il souhaitait aussi que l'application propose des solutions alternatives afin de réaliser le trajet tout de même. Grâce aux différents types de transport en commun, il est normalement quasiment toujours possible de trouver un itinéraire alternatif pour arriver à un point particulier dans Bordeaux.

Le problème abordé a donc été le suivant : trouver un moyen d'avoir une représentation du réseau de TBC, sur lequel on puisse chercher des trajets allant d'un point A à un point B. L'autre contrainte majeure est que cette représentation du réseau doit soit prendre en compte les pannes des transports et être mise à jour en conséquence, soit doit nous permettre d'éviter ces zones.

Au début de la réalisation de cette phase, nous nous sommes tout d'abord dirigés vers Google Maps et ses APIs. Ce sont aujourd'hui des références en terme de recherche d'itinéraires et d'affichage ergonomique de ces derniers. Nous avons rencontré plusieurs problèmes sur l'utilisation de cette technologie :

— les faibles possibilités de contraintes proposées par l'API : il est en effet impossible à ce jour de préciser des points particuliers à éviter lors de la recherche d'un nouvel itinéraire. Les seules contraintes possibles sont l'évitement de péage, d'autoroute etc... Mais aucune qui concerne les transports en commun, et donc notre projet.
— le coût demandé par Google pour l'utilisation de ses APIs. Même si l'API nous avait offert les possibilités pour résoudre notre problème, ses faibles capacités d'utilisation lorsque l'on utilise un compte gratuit sont particulièrement faibles. Et les prix pour l'utilisation d'un compte premium sur ses API sont particulièrement chers, et pas forcément adaptés au client.

Après avoir décidé de ne pas utiliser Google Maps pour ce module, nous nous sommes tournés vers une toute autre approche. Grâce aux éléments récupérés par le module de collecte et de traitement des données, nous avons choisi de réaliser notre propre graphe pour représenter l'ensemble du réseau TBC. L'avantage d'une telle solution est qu'il permet d'avoir une vision d'ensemble de la totalité du réseau.

Enfin, grâce à cette approche, notre recherche d'itinéraire alternatif s'est retrouvée

réalisable sans l'utilisation des APIs de Google. Une modification du graphe sur réception des pannes par le module de collecte, puis une application d'un algorithme du plus court chemin lorsque le serveur le demande.

4.2 Le stockage des données

Dans un premier temps, avant de réaliser le graphe en lui même, il nous a fallu récupérer l'ensemble des données nécessaires à sa construction. Pour cela, nous avons utilisé les données disponibles sur bordeaux-metropole.fr. Et plus précisément, ce sont les données GTFS (que nous décrivons un peu plus bas) qu'on a utilisé pour ce module. Pour stocker ces informations en interne, nous avons utilisé un premier package Python nommé Pandas.

Ce dernier est réputé pour le stockage et l'analyse de données. Ce qui nous a intéressé dans notre cas était une structure de données particulière : les DataFrames. Ce sont des structures de données en deux dimensions. Les colonnes et les lignes sont labelisées, et acceptent de nombreuses sources d'entrée, notamment les fichiers *csv*, ce qui est le format de nos données. Voici le format de nos requêtes d'import de données dans ces DataFrames : `DataFrame = pandas.read_csv(chemin_vers_la_source_de_données), encoding='utf-8')`. Il est important de préciser le codage afin de ne pas avoir d'incohérence plus tard durant le traitement de notre graphe.

4.2.1 Format GTFS

La présentation de ce format a été tirée d'une page *Wikipédia* : General Transit Feed Specification (GTFS, traduction littérale : spécification générale pour les flux relatifs aux transports en commun) est un format informatique standardisé pour communiquer des horaires de transports en commun et les informations géographiques associées (topographie d'un réseau : emplacement des arrêts, tracé des lignes).

GTFS, originellement conçu par Bibiana McHugh, une responsable des systèmes d'information chez TriMet, l'autorité organisatrice des transports urbains de l'agglomération de Portland (Oregon), a été développé par Google et TriMet, et initialement dénommé Google Transit Feed Specification.

Les données sont codées dans plusieurs fichiers, dont :
— agency.txt regroupe les informations sur le service de transport (compagnies de transport, nom du réseau)
— calendar.txt et calendar_dates.txt qui contiennent le calendrier de circulation
— routes.txt présente le nom et la direction des routes (au sens d'une origine-destination)
— stops.txt liste tous les points d'arrêt et propose d'éventuelles informations
— trips.txt détaille les courses, sous la forme d'une table de liaison entre les services (agency), les routes et les régimes de circulation (calendar.txt et calendar_dates.txt)
— stops_times.txt présente les horaires des courses aux points d'arrêt

— transfers.txt présente les correspondances entre plusieurs points d'arrêt
— shapes.txt permet le tracé d'une route sur une carte
— frequencies.txt indique le temps entre deux courses d'une ligne (pour celles qui n'ont pas d'horaires fixes aux points d'arrêts)

4.3 La représentation du réseau

Ensuite, afin de réaliser la représentation du réseau TBC en interne sur notre serveur, nous avons utilisé le package Python Networkx. Il est spécialisé dans la modélisation et l'étude des graphes. On y retrouve des fonctionnalités très utiles, telles que l'intégration de la plupart des algorithmes liés au graphe, des générateurs, ou encore une licence Open Source BSD.

A partir des DataFrames et des fonctions proposées par Networkx, nous avons pu créer un graphe directionnel afin de représenter le réseau TBC.
Nous avons choisi un graphe directionnel afin d'avoir pour chaque portion de ligne les deux sens de circulation. De plus, pour certaines lignes de bus, il n'y a pas un simple aller-retour, mais un cercle qui est effectué par chaque navette. Sans ajouter les informations de direction dans le graphe, certaines routes alternatives pourraient être erronées.

Avant de construire le graphe, nous avons commencé par définir deux dictionnaires Python (des tableaux indexés par des clés) d'instances de classes d'objets *Stop* et *Trip* qui vont contenir l'ensemble des informations nécessaires à la construction de notre graphe et à la gestion des itinéraires alternatifs.

4.3.1 Classe Stop

Avant de présenter l'implémentation de cette première classe qui a pour objectif de stocker les informations qui concernent un arrêt, nous allons définir ce que c'est un *Stop* : Un *Stop* est un arrêt de Tram ou de Bus physique caractérisé par ses coordonnées GPS : le couple longitude et latitude.

Exemples significatifs :
— Si on prend un arrêt de Tram ou de bus, on aura un *Stop* pour chaque sens (physiquement le Tram ou le bus s'arrête à deux endroits différents).
— Si on prend les différents arrêts de bus Gambetta, on aura deux *Stops* en commun pour chaque ensemble de lignes de bus qui partagent le même arrêt de bus physique (un pour chaque sens).

Pour ce qui est des informations qu'on stocke dans un objet *Stop*, nous commençons par l'identifier par un identifiant unique sur l'ensemble des *Stops* Tram et Bus, lui attribuer un nom, des coordonnées GPS(longitude et latitude) ainsi que le type de transport qu'il supporte (0 pour Bus et 1 pour Tram) et les identifiants des *Trips* auxquels il appartient.

4.3.2 Classe Trip

Cette classe a pour but d'identifier un voyage qui est caractérisé par le triplet $(ligne, direction, terminus)$ et de stocker les informations concernant un voyage.

Entre les objets de classe *Stop* et ceux de classe *Trip*, les derniers ont été de loin les plus difficiles à construire. En effet, les données GTFS ne permettaient pas un accès direct à un *Trip* indépendamment de la notion du temps : ils différenciaient les *Trips* par leurs horaires de passage en plus du triplet que nous utilisons.

En effectuant des sélections sur nos dataframes pour supprimer des duplications, nous arrivons à avoir un représentant par triplet $(ligne, direction, terminus)$ pour chaque ensemble de voyages caractérisés par le même triplet mais passant par différents horaires.

Pour chacun de ses représentants, on crée un objet *Trip* qui contient les informations suivantes :

— Un identifiant : l'identifiant du représentant pour chaque groupe de voyages de même triplet $(ligne, direction, terminus)$
— Un identifiant de la route.
— Un identifiant de la direction (0 ou 1)
— Un trip_headsign : Le nom du terminus.
— La liste d'identifiants des *Stops* qui constituent le *Trip* (dans le bon ordre).
— Et finalement le type de transport : 0 pour Bus et 1 pour Tram.

Une fois qu'on a construit tous nos *Trips*, on rajoute un dernier *Trip* particulier qui correspond aux voyages à pied en attribuant aux champs décrits ci-dessus des valeurs spécifiques à ce type de *Trip* pour ne pas créer de confusion (liste vide pour la liste de *Stops*, -1 pour le type de transport...).

4.3.3 Construction du graphe

La construction du graphe s'est faite en deux temps : la construction des nœuds puis la construction des arêtes.

La construction des nœuds du graphe a été quasi immédiate après le travail effectué sur les objets *Stops*. Il fallait seulement faire attention à uniformiser le type des identifiants de ces derniers étant donné que la fonction de lecture des fichiers *csv* de la bibliothèque *Pandas* interprétait les colonnes différemment en fonction de la nature des données qu'elles contiennent (parfois elle considérait que c'est des entiers, et d'autres fois que c'est des caractères).

La partie la plus importante consistait en la construction des arêtes. Nous avons classifié les arêtes en deux catégories pour les implémenter une par une :

— Les arêtes principales issues des identifiants de chaque *Trip* : Cette partie était relativement facile puisque ces identifiants étaient dans le bon ordre dans le champ

qui les contenait.

— Les arêtes issues des correspondances : Une première approche était de relier les *Stops* qui avait exactement le même nom. Une approche qui fonctionnerait pour la correspondance de Tram Hôtel de ville par exemple mais qui échouerait pour les correspondances de Bus de Place de la Victoire qui contient des *Stops* de noms différents (Victoire La Marne, Victoire A.Briand...). La deuxième approche sur laquelle on est finalement restés a été de d'exploiter les coordonnées GPS des arrêts. Grâce aux positions longitude et latitude fournies par les données GTFS, nous étions capables de calculer la distance entre deux *Stops* en utilisant la formule de Harvesine qu'on détaille ci-dessous. (mais il fallait tout de même faire attention à ne pas rajouter des arêtes indésirables comme une liaison entre deux *Stops* du même arrêt mais de sens opposé).

Formule de Haversine (code python) :

```
def distance_between_two_stops(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points on the earth
    (specified in decimal degrees)
    @lon1   : the fisrt point's longitude
    @lat1   : the first point's latitude
    @lon2   : the second point's longitude
    @lat2   : the second point's latitude
    @Return : the harvesine formula
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    return km
```

Poids des arêtes : La construction du graphe telle qu'elle a été décrite jusqu'ici ne prend pas en compte le moyen de transport utilisé entre deux *Stops*, et pour le compléter de façon à ce qu'il puisse subvenir à nos besoins, il fallait qu'on pondère les arêtes et y stocker le *Trip* qui a amené à construire ces dernières.

4.3.4 Recherche d'un nouvel itinéraire

A partir du moment où un graphe représentant correctement le réseau TBC est à disposition, il est possible d'appliquer l'algorithme de notre choix pour trouver un chemin

alternatif. Cette recherche se lance sur requête du serveur. Cela arrive lorsque ce dernier détecte une panne et que cette panne impacte des trajets d'utilisateur.

Lorsqu'une notification de ce genre est reçu, il faut modifier le graphe en conséquence et supprimer les arrêtes concernées.

Avant de commencer la recherche d'un chemin alternatif, on a commencé par écrire des fonctions qui construisaient un nouveau graphe à partir du graphe initial et d'une panne. Grâce au poids des arêtes on arrivait, à repérer les arêtes à supprimer du graphe, ensuite on appliquait la fonction *shortest_path* que nous offrait la bibliothèque *Networkx* pour trouver le plus court chemin entre deux points du graphe en le retournant sous la forme d'une liste de nœuds (en considérant que le plus court chemin est celui qui contient le moins de *Stops*) et puis finalement utiliser les arêtes pour préciser les *Trips* (ligne de Bus, Tram ou à pied) empruntés par ce chemin.

5 Serveur

5.1 Flask

Afin de permettre l'implémentation du serveur, nous avons utilisé le micro-framework Flask. Nous l'avons choisi d'une part pour sa simplicité et d'autre part car l'un des membres du groupe avait déjà travaillé avec ce framework.

Cependant, Flask étant encore une fois très basique, il ne présente pas d'architecture par défaut. Nous avons néanmoins voulu organiser notre code selon le modèle Modèle-Vue-Contrôleur. Pour permettre cela avec Python, nous avons du placer tous nos fichiers de configuration à la racine du projet et ensuite diviser les sources.

Dans les contrôleurs nous retrouvons toutes les méthodes de l'API qui sont accessibles via une route définie par Flask. Par exemple, pour se connecter, le client doit requêter le serveur avec le chemin `/users/login` depuis la racine en utilisant une requête `HTTP-GET`. Des paramètres peuvent aussi être indiqués dans le chemin de la requête. D'autres méthodes auxiliaires ont aussi été implémentées, mais ne sont pas accessibles depuis le client (tant que l'on a pas précisé de route). Pour gérer les login logout nous avons utilisé l'extension `Flask-Principal` avec `Flask-Login`.

Les modèles contiennent la description de la base de données, les différentes tables ainsi que les différentes méthodes permettant d'interagir avec elle.

S'agissant ici d'une WebApp, nous n'avions pas de vues à implémenter, les seuls contenus envoyé à l'application étant des fichiers JSON ou des messages d'erreur.

Aussi, pour intégrer les modules inhérents à la collecte et au traitement des donnes de Twitter et de Bordeaux Métropole, nous avons créé des bibliothèques qu'il suffit ensuite d'importer.

5.2 Celery

Si le serveur permet de gérer les requêtes des clients, il doit également permettre de consulter de façon asynchrone les API de Twitter et de Bordeaux Métropole. Pour cela, nous avons utilisé une application intégrée à `Flask`, `Celery`.

Il faut cependant configurer ce module, ce qui est fait dans le fichier de configuration à la racine du projet, qui permet notamment de régler la fréquence de requête (en l'occurrence à 30 secondes).

5.3 La base de données

Cette base de données sert au serveur à stocker toutes les données des utilisateurs. Ces données sont manipulées par l'application `flask`. Il s'agit des informations sur l'utilisa-

teur, de son émail dont il se sert pour se connecter, de son mot de passe, nom, prénom, ainsi que tous ses trajets. Mais elle est aussi utilisée par l'application `Celery` pour stocker les tweets et les disturbances. Le schéma de cette base de données est présent en figure 12.

Nous avons utilisé `SQLAlchemy`, via une extension de `flask` pour la réaliser.

5.4 Déploiement sur machine de production

Le déploiement sur la machine de production est réalisé tel qu'il suffise de récupérer les derniers commits à partir du dépôts git source. En effet, le code du projet est déjà configuré pour être déployé automatiquement.

Sur la machine de production, nous avons néanmoins rajouté des scripts d'initialisation pour Celery, le serveur de batch, et pour Flask, le serveur d'API. Cela se matérialise sous la forme de scripts python, appelés par des scripts bash, eux mêmes lancés par des services systemd. Lorsque ces services systemd sont lancés, on peut observer les logs suivants qui décrivent l'état des services cités précédemment :

```
[qucit@qucit-pfa29 ~]$ systemctl status qucit-server-api

  qucit-server-api.service - Qucit Server API
   Loaded: loaded (/etc/systemd/system/qucit-server-api.service; enabled;)
   Active: active (running) since Sun 2016-04-03 19:25:05 EDT; 2min 5s ago
 Main PID: 25265 (run_api_server.)
   CGroup: /system.slice/qucit-server-api.service
           25265 /bin/bash /home/qucit/run_api_server.sh
           25273 python /home/qucit/pfa29_server/App/runserver.py
           25283 /home/qucit/pfa29_env/bin/python
                   /home/qucit/pfa29_server/App/runserver.py

[qucit@qucit-pfa29 ~]$ systemctl status qucit-server-celery

  qucit-server-celery.service - Qucit Server Worker
   Loaded: loaded (/etc/systemd/system/qucit-server-celery.service; enabled;)
   Active: active (running) since Sun 2016-04-03 19:24:58 EDT; 2min 20s ago
 Main PID: 25240 (run_celery_serv)
   CGroup: /system.slice/qucit-server-celery.service
           25240 /bin/bash /home/qucit/run_celery_server.sh
           25248 python /home/qucit/pfa29_server/App/runcelery.py worker -B
           25258 python /home/qucit/pfa29_server/App/runcelery.py worker -B
           25259 python /home/qucit/pfa29_server/App/runcelery.py worker -B
```

On remarque l'état `enabled` pour les deux services qui permet de spécifier, dans le script de configuration `systemd`, que chacun de ces deux services doit être lancé au dé-

marrage de la machine. Sans quoi, il faudrait passer par une commande dans le terminal pour pouvoir les lancer proprement.

Le fait d'utiliser `systemd` pour lancer les services du projet permet de s'assurer que leurs processus ne sont lancés qu'après que les autres services dont ils dépendent aient également été lancés. Cela permet par exemple de s'assurer que la connectivité réseau est en place avant de lancer des applications qui vont en avoir besoin, en particulier le service de Worker avec Celery.

5.5 Améliorations

Nous avions pensé à plusieurs façons d'amélioration de déploiement sur machine de production :

— Utiliser un reverse-proxy tel que `nginx` avec un gestionnaire `WSGI` tel que `gunicorn` pour mieux gérer la charge. Cependant, étant donné que le projet en est à la première version et que c'est un prototype, ce n'était pas une priorité.

— Utiliser un déploiement par container avec `docker` pour pouvoir réaliser des installations sur n'importe quel système hôte. En effet avec un système de container, le projet définit les couches de container dont il a besoin et il construit une image finale qui peut être importée depuis n'importe quelle autre machine qui exécute une distribution `docker`. Il est même possible de réutiliser des images docker officielles de composants internes du projet : par exemple, on aurait pu utiliser un container rabbitmq, une container python/flask/celery, une container pour la base de données, le tout relié dan un ensemble fermé grâce à `docker-compose`.

— Mettre en place des tests d'intégration avec un service comme `travis-ci` afin de tester lors de chaque commit sur le dépôt git que les tests unitaires et les tests d'intégration finissent toujours avec succès. Cependant cette partie aurait été assez ardue à réaliser car le serveur consiste en un ensemble de briques qui peuvent prendre assez de temps à s'installer, même automatiquement. De plus, il aurait fallu aussi tester les accès aux sources de données, qui peuvent parfois être indisponibles.

La mise en application de ces items aurait tout de même permis de suivre le concept de développement plus connu sous le nom d'intégration continue, mais encore une fois ils ne constituaient pas une priorité essentielle dans le projet. Cependant, il est vrai que si nous avions eu une première version de l'application du serveur plus tôt, cela aurait assurément facilité le développement qui s'en serait suivi.

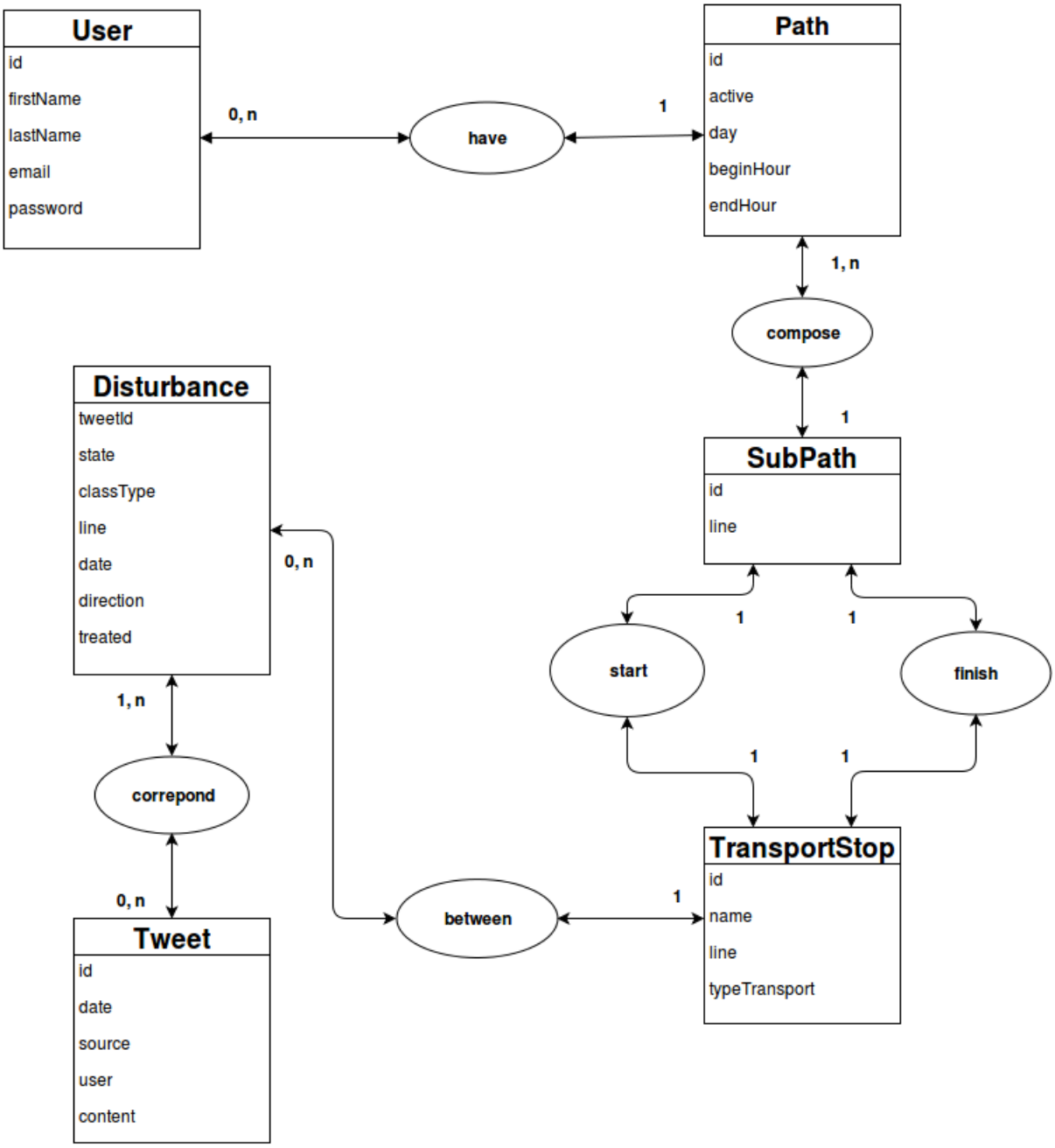


FIGURE 12 – Schéma Base de Données du Serveur

6 L'interface utilisateur

6.1 Contexte

L'objectif du projet au niveau de l'interface utilisateur (code client) était de développer une application mobile multi-platforme en utilisant des technologies web et notamment le nouveau framework de Google : `Polymer`. Nous étions ainsi mené à développer une application hybride. Nous avons donc construis l'application mobile en utilisant les technologies web (dont essentiellement `Polymer`) et en présentant le tout (web application) dans une Web View avec un outil (`Apache Cordova`) qui en génère une application native.

6.2 Environnement

Le développement de l'interface s'est réalisé en deux temps : le développement de la web application en utilisant la technologie web `Polymer`, et avec une approche en parallèlle, le portage multi-platforme avec `Cordova` pour vérifier à fur et à mesure la réalisation (développement et débogage en parallèle).

Dans cette section, nous allons présenter l'environnement de développement tout en suivant le cheminement de pensée qui a mené à la conception de l'interface utilisateur.

6.2.1 Polymer / Materiel Design : création de nouvelles expériences via le Web

`Polymer` est un framework web open-source, développé par Google permettant de créer rapidement des applications web à l'aide de composants préexistants.

Comme d'autres frameworks tels que `X-Tag` (Mozilla), Polymer repose sur les Web Components. Il s'agit d'une nouvelle spécification standardisée récemment par le W3C permettant la création de composants HTML indépendants et réutilisables.

Les Web Components visent à changer la façon dont nous construisons des applications Web en permettant aux développeurs d'étendre le vocabulaire HTML en créant leurs propres éléments HTML réutilisables. Ce moyen simple en apparence donne la possibilité de construire des application web en se basant sur des composants complexes déjà créés.

Les frameworks de l'interface utilisateur à base de composants ont été le moyen standard de construire des applications natives complexes pendant des années, et de nombreux frameworks web comme `angulare` et `Vaadin` nous donnent la possibilité de composer notre interface de blocs réutilisables. La grande différence entre ces approches et les composants Web est que les composants Web amènent la composition à un niveau de DOM afin que les éléments personnalisés peuvent être utilisés avec tout framework comme tout élément HTML standard.

Un des grands avantages des Web Components est qu'ils nous permettent de construire des applications en se basant sur des modules existants, au lieu de réinventer la roue. La plupart des applications utilisent un ensemble assez standardisé des contrôles d'interface utilisateur - des composants tels que les tableaux de données, sélecteurs de date, les champs de saisie, etc. En utilisant un composant existant, non seulement on économise du temps dans la construction de l'application, mais on n'a pas besoin de passer du temps en maintenance à l'avenir. Car on peut profiter de l'expansion des Web components et ainsi des avancements de la communauté qui contribuent aux corrections de bugs et aux améliorations des performances du même composant - résultant en un meilleur composant pour tous les utilisateurs.

`Polymer` encapsule trois langages différents dans ses librairies (HTML, CSS et JavaScript). Les éléments intégrés dans les librairies ont pour objectif de favoriser la création sur navigateur internet d'expériences similaires à celles des applications mobiles avec les « Web apps ». `Polymer` met l'accent également sur la rapidité de développement et la ré-utilisation du code.

Par ailleurs, `Polymer` s'appuie sur les codes propres au `Material Design`, le langage visuel introduit par Google à l'occasion du lancement d'Android Lollypop (Android 5.0). Il vise à unifier l'expérience utilisateur avec une charte graphique commune. Disponible sur Google.com/design, ce nouveau langage visuel est une réinterprétation du flat design, mais qui apporte des nouveautés non négligeables, son précédent nom de code était « Quantum Paper » (on en trouve d'ailleurs quelques références dans la documentation).

Paper Elements

`Paper Elements` fait partie des catégories d'éléments mis à disposition par l'équipe Polymer dans ses librairies. Elle implémente des éléments du nouveau langage visuel d'Android (`Material Design`).

L'utilisation de ces éléments est intéressante dans notre projet, dans le sens où une application basée sur cette catégorie d'éléments (`Paper Elements`) est conforme aux spécifications de la charte graphique d'Android et donc est assez bien adaptée aux mobiles (et notamment Android), du moins au niveau visuel.

6.2.2 Apache Cordova

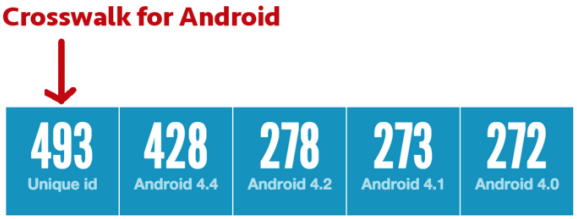
Une application créée à l'aide des éléments `Polymer` est essentiellement une collection de fichiers HTML, CSS et JavaScript. Cela signifie qu'elle a besoin d'un navigateur pour pouvoir s'exécuter. Cependant, elle peut fonctionner dans une `Web View`, un élément d'interface utilisateur native qui se comporte comme un navigateur chromeless (chromeless browsser).

`Apache Cordova` est un framework de développement mobile open-source qui permet de générer une application native contenant une Web View et permet de spécifier la page HTML à afficher. Ainsi, il permet d'exploiter les technologies Web pour développer des applications multi-plateformes, évitant l'utilisation des langages natifs propres aux différentes plates-formes mobiles. Les applications s'exécutent dans des wrappers ciblés pour chaque plate-forme, elles s'appuient sur des API conformes aux standards permettant l'accès aux capteurs de chaque appareil, aux données ainsi qu'à l'état du réseau.

Ainsi, Cordova nous a permis de faire le portage multi-plateforme de notre web application et de générer in fine des exécutables pour mobile (Android ou iOS principalement) avec des options configurables (notamment pour le type d'OS destination, sa version, son architecture processeur...).

6.2.3 Association Polymer / Cordova : Crosswalk à la rescousse

Le projet Crosswalk comprend des Web Views beaucoup plus récentes et mises à jour supportées par HTML5 que les Web Views par défauts des versions d'Android 4.x .
HTML5test.com donne des scores au caractéristiques pour Crosswalk pour une Web View d'Android s'exécutant sur un Nexus 5 sous Android 4.4.4 .



Crosswalk permet d'intégrer la Web View de Chrome notamment (chromium webview) au mobile et l'utilise au lieu de la Web View par défaut, si celle-ci n'est pas supportée.

Vu que la Web View d'Android des versions antérieures à la version 4.4 (KitKat) ne supporte pas Polymer, mais chrome pour Android le supporte, le projet Crosswalk doit nécessairement être intégré au projet Cordova pour supporter les anciennes versions d'Android.

Ceci a bien évidemment un coup, parce que finalement en intégrant Crosswalk on intègre une sorte de navigateur chromeless au projet (Dans son article sur les applications mobiles hybrides, John Bristowe décrit une Web View comme un : “chromeless browser window that's typically configured to run fullscreen.”), ce qui affecte la taille de l'exécutable pour mobile. Et donc affecte naturellement la fluidité de l'application. (La taille de notre APK par exemple (exécutable pour Android) à augmenter de près de 100% après l'intégration de Crosswalk et est passée d'une vingtaine de Mo à 40 Mo à peu près).

Ceci dit, Crosswalk n'est nécessaire que pour les versions d'Android antérieures à la version 4.4 (KitKat), et qui commencent à dater d'ailleurs, puisque la 4.4 a été sortie en 2013 et a été succédée par la version 5.0 (Lollipop) juste un an après et par la version 6.0 (Marshmallow) qui a apparue il y a quelques mois.

Donc intégrer Crosswalk à notre projet finalement nous a assuré d'avoir un exécutable qui marche sur toutes les plateformes d'Android et d'iOS. On a ainsi généré un exécutable générique, où est intégré Crosswalk entre autre (donc lourd), mais aussi un exécutable beaucoup moins lourd au niveau de la mémoire (sans Crosswalk), qui est destiné aux versions récentes d'Android.

6.3 Réalisations

6.3.1 La web application

Il a été question ainsi, en premier temps, de réaliser une web application en utilisant essentiellement Polymer, et notamment ses éléments implémentant Material Design (Paper Elements), pour avoir une application mobile de qualité.

En utilisant Polymer dans la construction de notre web application, nous avons aussi tiré profit des fonctionnalités du data binding qu'il fournit pour rendre notre code JavaScript plus simple et moins sujet aux erreurs.

Dans la partie qui suit, nous allons présenter de façon non exhaustive quelques uns des éléments que nous avons crée pour notre application (réutilisables), et comment nous avons utilisé le data binding pour abstraire notre réalisation et la séparer de l'utilisation.

6.3.2 Éléments de l'application

Cette partie traite quelques uns des éléments personnalisés (Custom Elements) conçus tant pour la mise en page (layout) que pour les fonctionnalités de notre application.

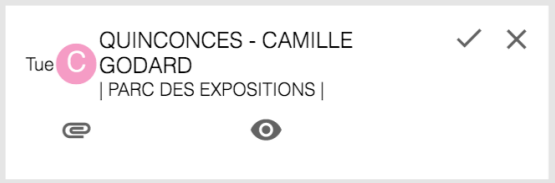
Notre application peut être comparée à une sorte d'alarme : l'utilisateur, une fois inscrit et connecté, saisi ses trajets avec le jour et la plage horaire (ce qui correspond à ajouter un réveil dans l'alarme), et a la possibilité de gérer chaque trajet et notamment d'activer ou de désactiver chaque trajet.

Si un de ses trajets activés est affecté par une panne, et le jour et la plage horaire de ce trajet correspondent au moment de la panne, l'utilisateur est notifié de son trajet affecté, et l'application lui offre la possibilité d'avoir un trajet alternatif en temps réel, suivant les préférences de transport choisies.

Nous avons ainsi conçu des éléments personnalisés (Custom Elements) représentant les différents modules de l'application (représentation d'un trajet : $post-card$, d'une page de profil utilisateur : $user-page$, d'une page de connexion : $user-login$...), et également un élément personnalisé représentant le module centrale le l'application ($post-list$) qui fait la liaison entre les différents modules.

C'est le cœur même de l'application au niveau du code client, il communique avec le serveur et liste les trajets d'un utilisateur donné. Il fait de plus la distinction entre les trajets activés et les non activés et permet à l'utilisateur de gérer ces trajets et de profiter des fonctionnalités de l'application.

Trajet non activé sans correspondance



Représenté par l'élément personnalisé $post-card$, un trajet contient l'information sur le trajet : ligne (C dans l'exemple), station de départ (Quinconces), station d'arrivée (Camille Godard), direction (Parc des expositions), jour (Mardi).

De plus, un trajet donne la possibilité de l'activer (bouton check), de le supprimer (bouton en haut à droite), d'ajouter une correspondance (bouton en bas à gauche) et de le visualiser (bouton en bas au centre).

Après création, l'utilisation de cet élément personnalisé est comparable à la l'utilisation d'une instance d'une classe (en programmation orientée objet). Il suffit de le déclarer ($<post-card></post-card>$) et d'initialiser ses attributs (i.e l'instancier).

Les valeurs d'initialisations d'un trajet sont représentés dans un tableau en format Json comme suit :

```
{
    "pid": 5,
    "subPaths": [{"line": "C",
                  "direction" : "PARC DES EXPOSITIONS",
                  "startStop": "QUINCONCES",
                  "finishStop": "CAMILLE GODARD"}],
    "day" : "Tue",
    "beginHour": "13h30",
```

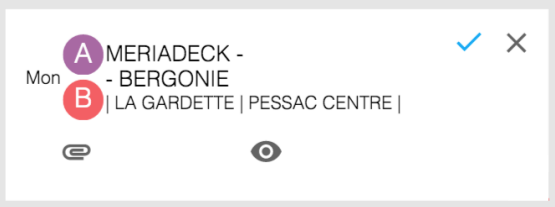
```
    "endHour": "14h",
    "active": false,
    "breakdown": false,
    "connection": false
}
```

L'utilisation de cet élément ($post-card$) dans $post-list$ (élément personnalisé qui fait la liaison entres les différents modules au niveau du code client) est faite en exploitant le data binding fourni par `Polymer` à l'aide du $dom-repeat$. Cet outil est comparable à une boucle for et permet ainsi de lister tous les trajets (en affinant l'ordre selon les besoins) d'un utilisateur après les avoir imprimer depuis un fichier Json propre à chaque utilisateur (contenant les tableaux de tous les trajets de cet utilisateur).

Ainsi, une fois que l'utilisateur est authentifié, l'application (et plus spécifiquement l'élément $post-list$) récupère le lien du fichier Json contenant ses trajets depuis le back end, et le liste.

Ainsi et de façon générale, toutes les interactions avec le back end se font à partir de fichiers Json (inscription, ajout de trajets, de correspondances, activation/désactivation de trajets...).

Trajet activé avec une seule correspondance



Dans cet exemple, le trajet comprend la ligne A puis la ligne B (dans l'ordre), la station de départ (de la ligne A) est Mériadeck, la station d'arrivée (de la ligne B) est Bergonié, et les directions pour les lignes A et B sont respectivement La Gardette et Pessac centre.

La station de correspondance n'est pas spécifiée pour des raisons de clarté de la présentation d'un trajet, mais celle-ci est facilement récupérable par l'utilisateur. Dans l'exemple fournie, il suffit de prendre la ligne A et de s'arrêter dans une station de correspondance avec la ligne B pour finir le trajet.

Le tableau correspondant au informations du trajet de l'exemple se présente comme suit (il y a un sous trajet de plus dans $subPaths$) :

```
{
    "pid": 3,
    "subPaths": [{"line": "A",
                  "direction" : "LA GARDETTE",
                  "startStop": "MERIADECK",
                  "finishStop": "BERGONIE"},
```

```
                {"line": "B",
                "direction" : "PESSAC CENTRE",
                "startStop": "MERIADECK",
                "finishStop": "BERGONIE"}],
    "day" : "Mon",
    "beginHour": "13h30",
    "endHour": "14h",
    "active": true,
    "breakdown": false,
    "connection": true
}
```


Trajet non activé affecté par une panne

Un trajet affecté par une panne est représenté ainsi de façon similaire à un trajet courant, et contient en plus un warning pour transmettre l'information de la panne (en haut a droite), et une proposition de calcul de trajet alternatif selon les préférences de l'utilisateur.

Le tableau (Json) correspondant à l'exemple est similaire aux précédents avec le champ *breakdown* qui est mis à true.

6.3.3 Représentation du réseau TBC au niveau du client

Notre application donne la possibilité à l'utilisateur d'ajouter des trajets et des correspondances à ses trajets. Nous étions ainsi amené à trouver un moyen pour donner la main à l'utilisateur (pour transcrire les informations du trajet à ajouter par exemple : station de dépars, d'arrivée...), tout en contrôlant ce qu'il saisie (car après tout, l'utilisateur pourrait saisir des trajets inexistants, si on lui laissait la main sans contrôle).

Notre choix s'est porté sur des listes qu'on propose à chaque fois selon le besoin, à l'utilisateur qui doit choisir un item parmi plusieurs. Par exemple, pour choisir la ligne, on lui donne la main sur la liste de toutes les lignes...

Pour pouvoir contrôler ce que l'utilisateur peut être mener à saisir, nous avons eu à faire le choix entre deux possibilités :

— Faire un contrôle dynamique, en requêtant le serveur à chaque fois que l'utilisateur saisie des données
— Faire un contrôle statique, au niveau du client en ayant une représentation du Réseau de la TBC

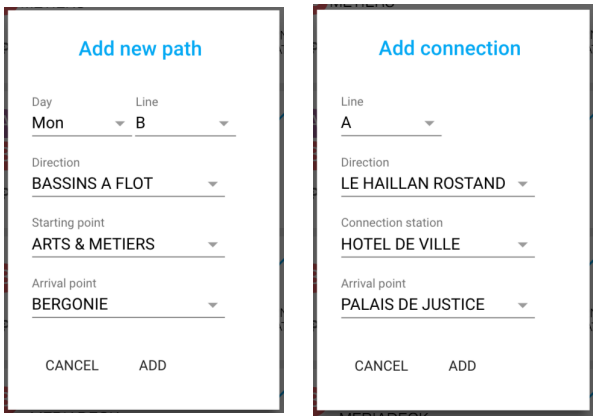
Nous avons jugé que le premier choix était inapproprié dans le sens où il mettrait éventuellement le serveur en surcharge, si le nombre d'utilisateurs est important.

Nous avons ainsi opté pour la deuxième solution : avoir une représentation du réseau de la TBC au niveau du client (l'exécutable). Cette solution était de plus bénéfique pour la représentation des lignes dans un trajet (avec ou sans correspondances). Car il fallait avoir les information sur les lignes (couleur de la ligne, couleur du texte...).

La représentation du réseau de la TBC au niveau du client est sous format Json, contenant un tableau d'objets *line*. Chaque objet *line* contient :
— *id* identificateur (joue le rôle d'indice),
— *lid* l'identificateur de la ligne,
— *line* nom de la ligne,
— *route_color* couleur de la ligne,
— *route_text_color* couleur du texte de la ligne,
— *stops* tableau des stations de la lignes,

Cette représentation nous a non seulement permis de contrôler les saisies des utilisateurs, mais également d'affiner les choix que nous leur proposons, et notamment pour l'ajout d'un nouveau trajet ou l'ajout d'une correspondance dans un trajet.

Pour l'ajout d'un nouveau trajet par exemple, l'utilisateur choisi en premier lieu la ligne (parmi toutes les lignes du réseau), puis la direction (parmi les directions de la ligne choisie), puis les stations de départ et d'arrivée (parmi les station de la ligne choisie). (L'ajout d'une correspondance est fait de façon similaire).



Vous pouvez remarquer que nous ne donnons pas à l'utilisateur la possibilité d'ajouter une correspondance lors de l'ajout d'un nouveau trajet, ceci est bien évidemment volontaire. Pour qu'un utilisateur ajoute un trajet avec une ou plus de correspondances, il doit ajouter ce dernier sous forme de sous trajets : il ajoute le premier sous trajet, puis dans sa page de trajets, il ajoute la première correspondance au premier sous trajet et ainsi

de suite.

Cette façon de faire est récursive et permet d'ajouter autant de correspondances que l'on veut, en ayant des pages (d'ajout de trajets et de correspondances) pas trop chargées. Cependant, nous avons fais le choix de limiter le nombre de correspondances possibles à deux (cette valeur est configurable bien sur).

6.3.4 Migration Polymer 0.5-1.0

Au début de la phase de développement du projet, nous avions commencé à utiliser Polymer en version 0.5 car les tutoriels et la documentation disponibles étaient orientés pour cette version en particulier. En effet Polymer est un projet de Google qui ne fait que commencer, donc les plupart des ressources disponibles sont largement éparpillées sur des blogs de spécialistes des technologies web.

De plus, la documentation en français est quasiment inexistante. (Vous pouvez taper Polymer sur Google, il n'y a qu'une seule page de documentation en Français, et elle traite de la version 0.5, Google la classe 3e dans la recherche après la documentation officielle en anglais et le Git). Ainsi il est vrai que la version 1.0 était déjà sortie (depuis quelques mois) mais la documentation officielle n'était pas forcément à jour en la matière. A cela faillait-il rajouter le fait que Polymer n'étant pas une solution tout en un, chaque explication pouvait potentiellement utiliser des solutions différentes.

Après une période d'adaptation, nous avons donc décidé d'utiliser la version 1.0 de Polymer qui apportait de nombreux avantages, tels qu'un catalogue de composants bien plus conséquent, ainsi que la correction de nombreux bugs. Il a donc fallu migrer le code de l'application en prenant en compte les changements au niveau des noms de propriétés, des noms des fonctions, mais surtout du changement de l'architecture du projet.

6.3.5 Optimisation de l'application

Nous avons pensé à plusieurs améliorations que nous n'avons pas eu le temps d'implémenter complètement (du moins pour la date du rendu) :

— L'intégration d'une représentation du réseau de la TBC au niveau du code client nous à imposé un minimum d'optimisation dans le data binding, pour que ça n'affecte pas de façon trop visible la rapidité du chargement de l'application. Bien que le json qui contient la représentation du réseau de la TBC n'est pas trop conséquent (environ 70 Ko), l'interaction avec ces données n'est pas des moins lourdes. Après optimisation du data binding (à l'aide des $dom-repeat$ et $dom-if$) on a pu arrivé à un temps de chargement raisonnable, pour les interactions avec la représentation du réseau notamment.

— Mise en place d'un cache. Afin d'augmenter la rapidité du premier chargement de l'application et d'éviter de faire des requêtes réseau à chaque fois que l'utilisateur arrive sur l'application, il était judicieux de mettre en place un cache des don-

nées. Cela a impliqué d'utiliser une sorte de base de données locale à l'application pour stocker des données temporaires. Ces données sont censés être rajoutées dans l'application dès qu'une requête réseau aurait été lancée auprès du serveur, à la condition qu'elles n'existent pas déjà dans ce même cache. Il aurait également été possible de rajouter un temps de validité pour chaque entrée dans le cache, afin qu'un enregistrement devienne invalide après un certain temps. Malheureusement, faute de temps nous n'avons pas pu finir l'implémentation du cache.

De plus, la façon dont nous avons conçu l'interface de données avec l'API aurait rendu ce mécanisme d'autant plus efficace car les ressources représentant les objets à communiquer s'imbriquent les unes dans les autres. Donc lorsque qu'une ressource aurait été récupérée du serveur, il aurait été possible de la mettre à jour, mais aussi les ressources imbriquées qu'elle possède.

Potentiellement, un tel mécanisme aurait pu à la fois améliorer la performance réseau de l'application et le rendu visuel car elle serait devenue plus réactive.

— Un versionnage de l'interface de données pour être capable de supporter différentes versions de l'application.

7 Bilan et recette du cahier des charges

Comme indiqué en partie 2, ce projet s'est déroulé suivant la méthode SCRUM. De fait, les évolutions des spécifications se sont souvent faites d'un commun accord avec le client, au fur et à mesure de l'étude, sans que cela ne perturbe le développement.

Conformément au cahier des spécifications, notre livrable est en mesure de répondre à 16 user-stories sur 33. Les autres user-stories ont été soient reportées à une prochaine version de l'application, soit partiellement implémentées. Beaucoup de ces user-stories manquantes sont liées aux problèmes rencontrés pour installer le serveur et donc communiquer avec l'application.

Concernant la partie inhérente à la collecte et aux traitement des données de Bordeaux Métropole, la majeure partie des tâches ont été implémentées :

— En tant que serveur, je dois fournir une représentation à jour de l'ensemble du réseau de transport TBC.
— En tant que serveur, je dois détecter, traiter et stocker les pannes du réseau TBC.
— En tant que serveur, je doit fournir, au module de recherche d'itinéraires alternatifs, toutes les données nécessaires à son algorithme.
— En tant que serveur, je doit notifier le module de Gestion utilisateur de chaque panne détectée sur le réseau TBC.

Néanmoins, les user-stories suivantes n'ont pas été mises en place :

— En tant que serveur, je dois fournir une représentation des stations de Vcub et de la disponibilité des vélos aux stations.
— En tant que serveur, je dois détecter qu'un changement dans les structures de données des sources de données rend la modélisation du réseau et des pannes non conformes.
— En tant que serveur, je dois détecter que le traitement des messages de pannes est défectueux.

En effet, la gestion des Vcub aurait du se faire en utilisant l'API Qucit. Il se trouve que nous n'avons pas pu bénéficier de cette API qui constituait un pré-traitement des données de Bordeaux Métropole. De fait et d'un commun accord avec Qucit, nous avons repoussé l'utilisation du Vcub dans nos algorithmes et dans les trajets à une prochaine version de l'application.
Concernant la conformité des informations, cela a été jugé comme non nécessaire par le responsable pédagogique du projet et il est donc normal que ce n'ait pas été implémenté.

C'est le module de gestion utilisateur qui connaît le plus de dérogations au cahier des charges. En effet, cette partie devait permettre de gérer l'inscription, la connexion et plus globalement la gestion des comptes utilisateurs. Pour cela il aurait fallu être en mesure d'assurer la connexion de l'application avec le serveur. Si une interface a été implémentée, elle n'a pas pu être testée faute d'avoir réussi à effectuer l'installation du serveur.

Le module des itinéraires alternatifs a lui vu peu de modifications. En effet, si l'implémentation a pris plusieurs directions au fur et à mesure du projet (voir la partie concernée), les spécifications elles ont pu être respectées. Exception faite de l'user-story inhérente à la communication d'un itinéraire alternatif à l'application :

— En tant qu'algorithme, je souhaite récupérer les trajets/utilisateurs concernés par une panne.
— En tant qu'algorithme, je souhaite avoir une représentation en temps réel du réseau de transports en commun de Bordeaux.
— En tant qu'algorithme, je souhaite créer au moins un itinéraire alternatif pour ces trajets, en tenant compte des préférences utilisateur.

Enfin, pour ce qui est de l'application, la majeure partie des user-stories ont également été implémentées :

— En tant qu'utilisateur, je souhaite pouvoir me créer un compte sur l'application.
— En tant qu'utilisateur, je souhaite pouvoir me connecter à l'application si je possède déjà un compte.
— En tant qu'utilisateur, je veux pouvoir me déconnecter.
— En tant qu'utilisateur, je veux pouvoir modifier mes informations.
— En tant qu'utilisateur, je veux pouvoir personnaliser mes préférences de transport
— En tant qu'utilisateur, je veux pouvoir créer mes différents trajets quotidiens.
— En tant qu'utilisateur, je veux pouvoir visualiser l'ensemble de mes trajets.
— En tant qu'utilisateur, je veux pouvoir activer/désactiver mes trajets.
— En tant qu'utilisateur, je veux pouvoir modifier et supprimer mes différents trajets quotidiens.

Les fonctionnalités qui n'ont pas été implémentées ont donc été celles liées à la communication des notifications de pannes et des itinéraires alternatifs, du fait que la communication avec le serveur n'a pas pu être testée.

De manière générale, la plupart des user-story non implémentées ne l'ont pas été principalement du fait des difficultés rencontrées en fin de projet pour l'installation du serveur. Nous avons passé énormément de temps à apprendre de nouvelles technologies, et les problèmes rencontrés en fin de projet n'ont pas pu être corrigé. Néanmoins, la plupart des fonctions ont été développées côté serveur et client afin d'assurer le fonctionnement de ces user-story.

8 Conclusion

Au vu du bilan fait Partie 7, nous sommes un peu déçus de ne pas avoir une application totalement finie. Le back-end comme le front-end sont bien avancés mais les 2 parties ne communiquent pas encore.

Néanmoins ce projet était un projet difficile à la fois algorithmiquement et technologiquement. La partie front-end nous a fait utiliser un Framework très récent, avec peu de documentation sur l'utilisation que nous en avons faite Cordova par exemple. Du côté back-end également avec de la collecte de données de formats divers, des scrapers, des algorithmes de "machine learning" et de traitement de langages naturels qui étaient totalement nouveaux pour nous, des algorithmes de manipulations de graphes, l'implémentation d'une API, l'installation du serveur et avec cela une pléthore de nouveaux modules Python qu'il a fallut apprendre à utiliser. On ne compte plus le nombre de pages de documentation qu'il nous a fallu lire pour en arriver là. Le travail en équipe n'a pas été une chose facile et travailler dans un groupe de cette taille était quelque chose de nouveau pour chacun d'entre nous.

Cependant nous avons appris bien plus de choses que nous ne l'aurions imaginé au moment de la sélection des projets au début de l'année. En effet, nous savons désormais structurer un projet en différents parties indépendantes, déterminer les rôles des parties importantes, choisir les meilleurs briques logicielles qui existent pour un environnement donné, ici pour la plate forme Python en particulier. Le rôle des projets open-source nous est maintenant plus évident car nous n'aurions jamais pu avoir accès à un tel panel d'outils performants et complets, même s'ils peuvent êtres complexes à assimiler et à utiliser. Nous nous sommes rendus compte de la complexité des projets en équipe et de la nécessité de savoir se partager les parties pour arriver à travailler de concert.