Epitech MSC-Pro

Rapport sur la création d'un outil de recherche d'itinéraire

Jonathan KHALIFA, Bastien ANGLES,

Fabrice SUMSA, Valentin Noel et Axel Migy

Janvier 2022

Table des matières

Introduction

Notre projet s'inscrit dans le cadre de la mise en place d'une solution permettant aux utilisateurs d'effectuer une recherche de trajet entre les gares ferroviaires de France.

L'objectif est d'avoir une plateforme simple d'utilisation qui permet aux clients de pouvoir rechercher par message vocal, fichier audio ou par écrit, un itinéraire optimisé entre deux gares. Nous avions donc comme principal taches de prendre en compte les requêtes du client, de les traiter, déterminer si sa requête est valide ou pas, afin de pouvoir lui proposer l'itinéraire le plus optimal.

Le résultat lui sera alors affiché de manière claire et ludique.

Organisation de l'équipe

Nous sommes une équipe tech constitué de plusieurs développeurs. Nous avons un chef de projet tout aussi technique que le reste de l'équipe. Ayant plusieurs technologies à notre corde, nous avons pu nous mettre ensemble pour partager nos savoirs faire afin de mener à bien ce projet.

Nous avions mis en place une méthodologie basée sur du pseudo-agile afin d'avancer au mieux sur le projet. Le projet a été conduit comme un projet R&D avec une première phase de recherche, ce qui nous a permis de nous familiariser avec des technos. La phase de recherche une fois finie nous a permis de réunir l'ensemble et ainsi donc posé une architecture solide.

Pour le suivi de projet, un Trello (https://trello.com/b/X3s2fpPJ/ia-projet) a été mise en place afin de suivre de près les tâches et s'assurer de l'avancée du projet dans les meilleures conditions. Nous utilisions également teams comme plateforme d'échanges et ainsi nous avions un point hebdomadaire qui s'y tenais.

Le projet a été découpé en plusieurs modules. Le front-end, back-end, speech to text, spam-filter, nlp, pathfinding. Les Modules relatifs à de l'IA ont été conçus et testés sur des jupyter notebook avant d'être progressivement implémentés dans le back-end de notre application.

Architecture du Projet

1 - Architecture

Pour mener à bien ce projet, nous avons choisi une architecture web classique.

Nous avons une base de données en cloud de type document pour y stocker toutes nos données. Données brut, segments du graph, hyper-paramètres, tout y est stocké. Ce choix s'est fait par souci de bonnes pratiques et dans un but d'optimisation des performances.

Celle-ci est directement reliée à un back-end api. Ce back-end contiendra toute la logique de notre application, y compris nos algorithmes IA.

Le front-end communiquera avec le back-end afin de récupérer les données sous format json, de les traiter puis les afficher à l'écran de l'utilisateur.

2 - Choix des technologies

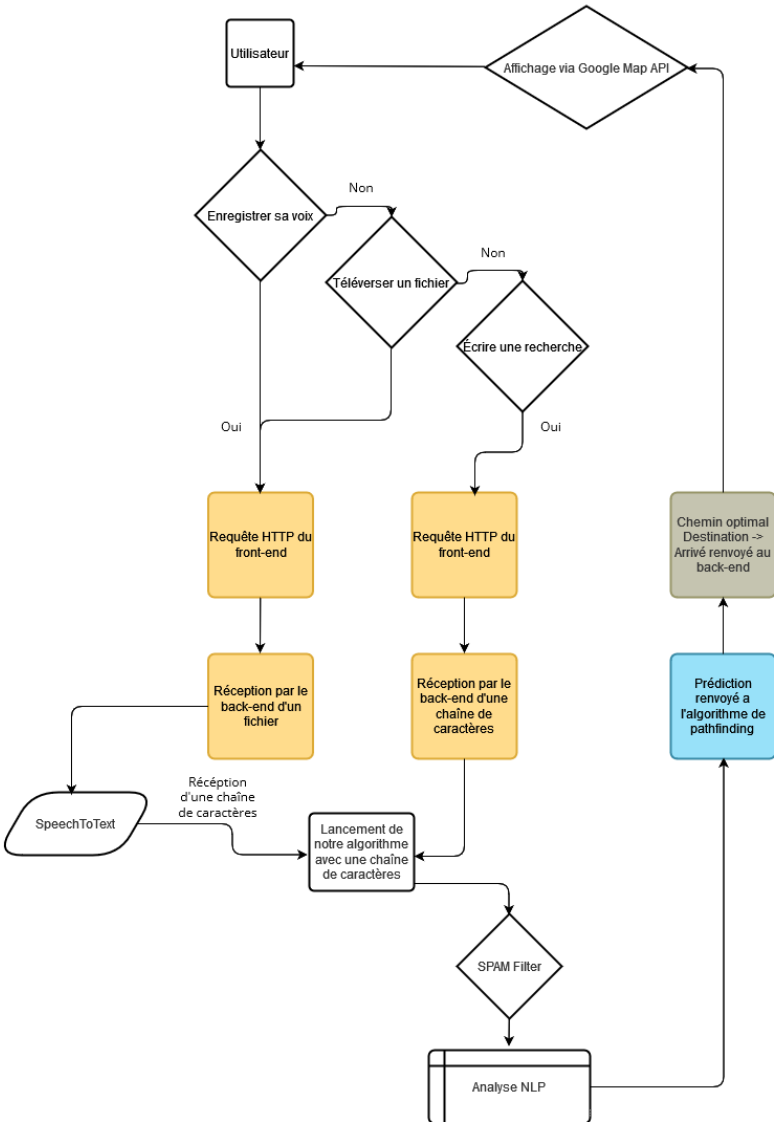
Pour pouvoir proposer un affichage de qualité nous avons réfléchi aux technologies existantes et avons choisi de faire un front-end en HTML sans framework car nous n'avions pas énormément de contenu à afficher, nous avons usé des principes de Single Page Application que nous détaillerons plus dans la partie adéquate.

Le back a été fait en Flask, un framework Python. Cela nous a permis de transformer rapidement eet sans encombre nos algos Python, en API pleinement opérationnelle.

La base de données s'est faite via Mongo Atlas pour l'optimisation de nos appels ainsi que pour un traitement plus rapide.

3 - Workflow

Ci-dessous un workflow décrivant l'enchainement de l'application.



Après chargement de la page, l'utilisateur aura trois possibilités pour soumettre sa demande.

Utiliser son micro ou uploader un fichier audio. Ceux-ci seront traités par le speech to text avant d'être relayés au spam-filter.

Ou bien écrire sa demande vie un text-box, qui sera directement relayé au spam-filter.

Le spam-filter va analyser le message et déterminer s'il s'agit d'une demande valide et relayera la demande au module NLP. Ou au contraire jugera que la demande n'est pas valide, et à ce moment affichera un message d'erreur personnalisé (diner, hôtel, autre moyen de transport, etc…).

Une fois le message arrivé au NLP, celui-ci va se charger de trouver un sens au message, et en déduire quelle est la ville ou gare de départ, et celle d'arrivée. Une fois chose faite, C'est le module de pathfinding qui prend la suite.

Le pathfinding se chargera de trouver l'itinéraire le plus optimal afin que le résultat puisse être affichée aussi intuitivement que possible par le front-end.

Nous rentrerons plus en détail pour chacune de ces étapes au fil de ce rapport.

Speech To Text

1 - Le concept

Afin de permettre aux utilisateurs de pouvoir utiliser leur micro ou de soumettre un fichier audio, nous avions donc décider d'intégrer le speech to text "stt". Le principe de ce dernier est de prendre en compte un fichier audio et le retranscrire en texte dans la langue de base.

2 - Plusieurs options possibles

Il existe plusieurs services qui permettent d'intégrer ces fonctionnalités notamment :

- Google Cloud Speech API
- Wit.ai
- Microsoft Bing Voice Recognition
- Houndify API
- IBM Speech to Text
- Snowboy Hotword Detection

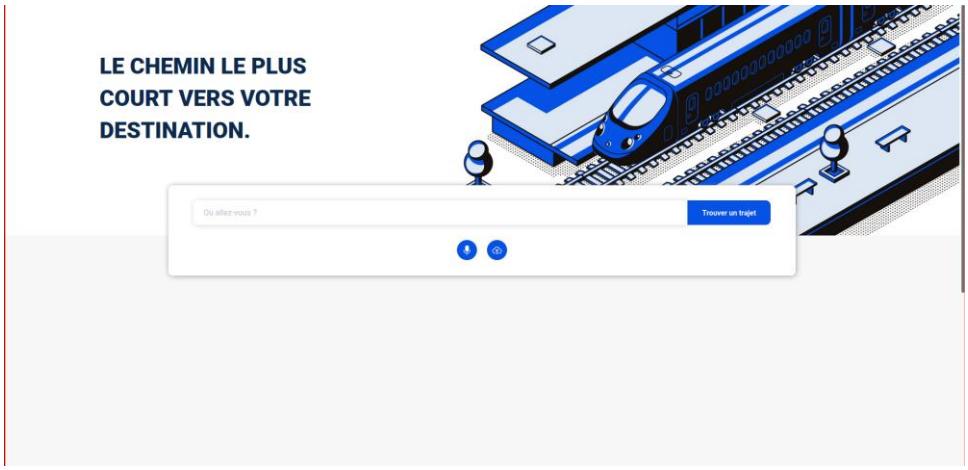
3 - La solution retenue

Ces différents services sont tous aussi fonctionnels les uns que les autres. Après avoir fait le tour nous avions décidé de rester sur le service de google aux vues de notre contexte car il est celui qui s'adapte le mieux au langage français et fourni également des paramètres qui permettent de filtrer le message malgré les bruits en fond sonores.

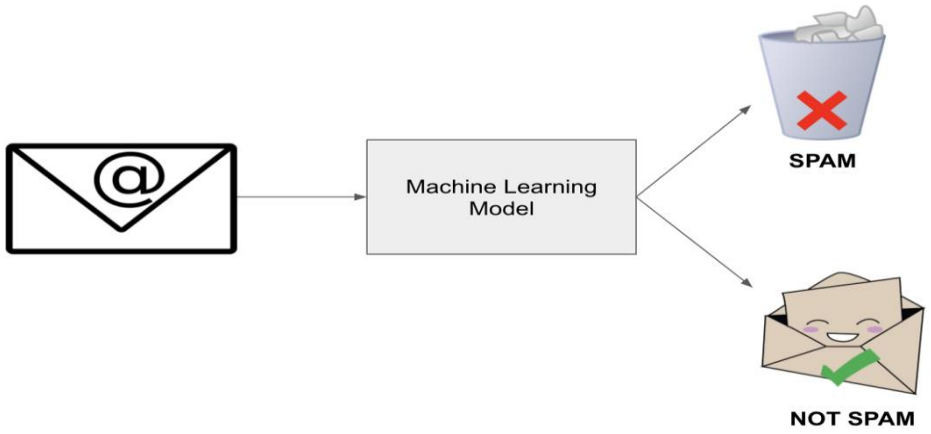
4 - Intégration

L'intégration fut en un 1er temps effectué sur un notebook avec deux différentes fonctions. La première favorisant l'utilisation d'un micro, et la deuxième l'importation d'un fichier audio.

Ensuite le module joins fut intégrer au niveau du front.


LE CHEMIN LE PLUS
COURT VERS VOTRE
DESTINATION.
Où allez-vous ?
Trouver un trajet

Spam-Filter

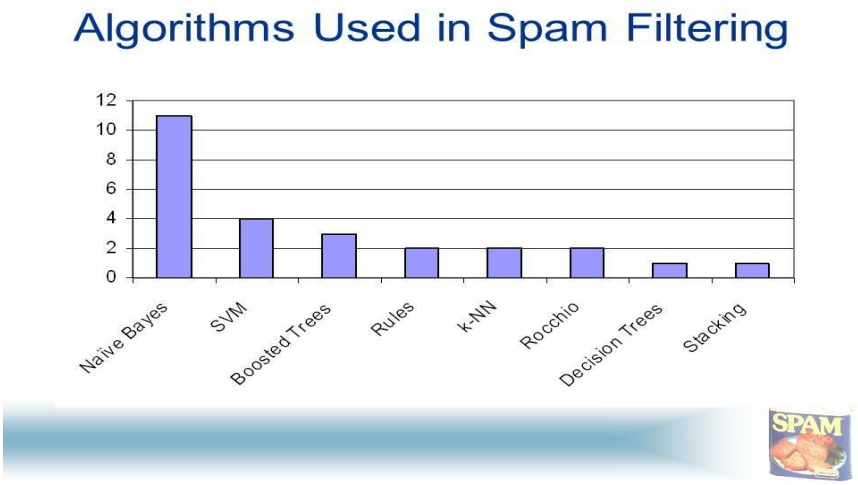


1 - Le concept

Le spam-filter, souvent assimilés au filtrage d'emails, est un procédé qui permet de différencier des textes ou phrases selon des critères bien définies. De cette manière, le spam-filter pourra décider si le texte que nous lui présentons correspond à ce que nous attendons, ou au contraire n'est pas valide. Dans notre cas, classifier les messages dont il s'agit bien d'une demande pour un billet de train.

Cette première étape va filtrer les demandes utilisateur pour et seul les requêtes valides seront traitées par les modules suivants et donc retourner un itinéraire cohérent à la demande de l'utilisateur. Les requêtes non valides afficheront un message d'erreur en demandant à l'utilisateur de reformuler sa demande.

2 - Plusieurs options possibles



- Naive Bayes, une méthode qui va se baser sur les probabilités qu'un mot désirable/indésirable apparaisse dans le texte. Le filtre ne connaît pas à l'avance ces probabilités, c'est pourquoi il lui faut un temps d'apprentissage pour les évaluer. Nous conduirons cet apprentissage via un jeu de

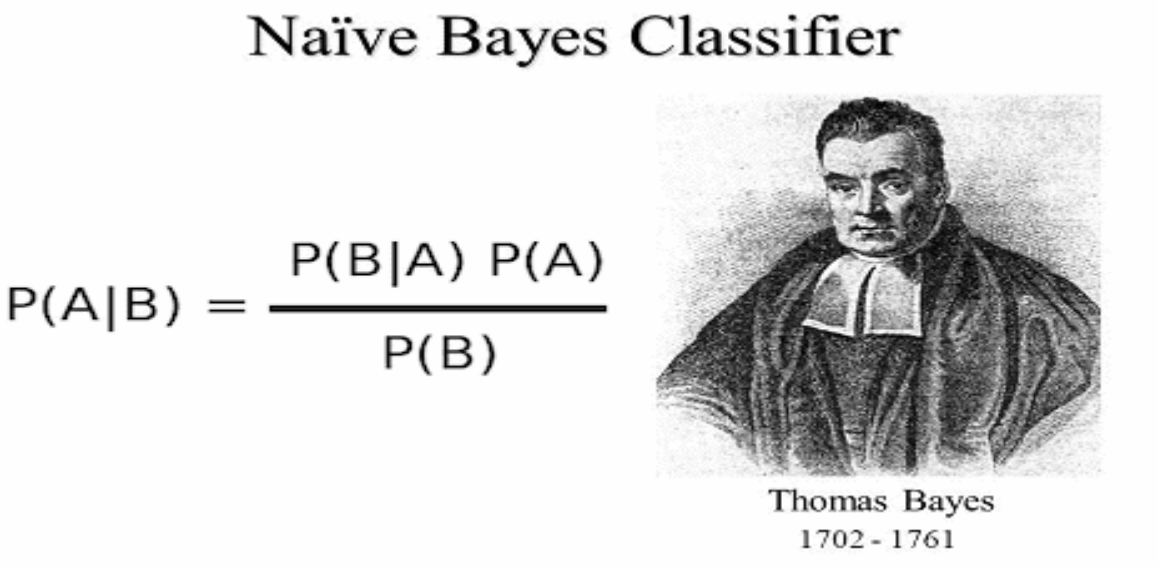
données que nous construirons. Chaque mot du message contribue à la probabilité que le message soit un spam. Cette contribution est calculée en utilisant le théorème de Bayes (https://fr.wikipedia.org/wiki/Théorème_de_Bayes) Une fois que le calcul pour le message en entier est terminé, on compare sa probabilité d'être un spam à une valeur arbitraire (95 % par exemple) pour marquer ou non le message comme spam. Les filtres bayésiens évitent particulièrement bien les faux positifs, c'est-à-dire de classer les messages légitimes comme des spams, critère important dans le cadre de nos demandes voyageurs.

- Autre solution, utiliser un simple jeu de règles prédéfinies qui rejettera d'office les demandes où l'on y trouve certain mots clés par exemple. Cette technique est cependant moins efficace. On peut s'imagine par exemple une série de mots comme liste noire qui définiraient le message comme spam, voir même une combinaison de mots. Les limites d'un tel algo sont généralement vite atteintes car les conditions de ces règles sont souvent simples à contourner.

- Pour ne pas se limiter qu'a deux choix, nous avons aussi exploré du côté des modèles KNN, qui permettra de calculer la distance d'un nouveau message, avec ceux qui ont servi à l'entrainement, puis de lui attribuer le label de celui qui s'en rapproche le plus.

3 - La solution retenue

C'est Naive bayes qui a été retenu. De loin le modèle le plus populaire pour son efficacité. Ce modèle de prédilection, couplé avec quelques règles prédéfinies en fera un filtre hybride et robuste.



4 - Création du dataset

Pour mener à bien cet exercice, il nous a fallu créer un dataset de messages et de les labeliser.

Tout d'abord nous avons recyclé le dataset de discours politiques que l'on nous avait fourni lors d'un bootstrap. Nous avons segmenté les discours afin d'obtenir à la fois plus de messages, mais aussi des messages de taille cohérente avec ceux d'un client. Des 20000 messages que nous avons créés, 50% ont été labelisés en tant que spam, et 50% comme ham.

Ensuite nous avons supprimé tous les noms de ville et pays présents dans nos messages car nous ne voulons pas qu'ils interfèrent dans nos probabilités.

Ensuite nous avons injecté aléatoirement dans nos messages ham des mots que nous jugeons pertinents pour un message ham. Puis avons fait de même avec un bag of « bad » words pour les messages spam. Ces derniers ont été injectés en plus grande quantité pour que les bad words aient un poids plus lourd.

Enfin le dataset a été divisé en deux pour avoir un dataset d'entrainement, et un de test sur un split 80/20. Chaque dataset est équilibré avec 50% de spam et 50% de ham.

5 - La phase d'entrainement

Dans notre cas, l'entrainement s'est fait en deux étapes. La première consiste a calculer les constantes suivantes à partir du contenu du dataset:

- p_spam : proba que le message soit un spam
- p_ham : proba que le message soit un ham
- n_spam : nombre de messages spam
- n_ham : nombre de messages ham
- n_vocabulary : nombre de mots différents dans le dataset
- alpha : laplace smoothing
- n_words_per_ham_message : nombre de mots par message ham
- n_words_per_spam_message : nombre de mots par message spam

La deuxième étape consistait à calculer les paramètres de notre modèle à partir des constantes que nous avons calculés, et ainsi connaitre à partir de la fréquence des mots dans nos messages, les probabilités de qu'ils soient destinés à un ham ou spam.

Nous avons enregistré ces paramètres en DB cloud afin de pouvoir s'en servir plus aisément dans l'exécution de notre algorithme.

6 - L'utilisation du modèle

Grace aux paramètres obtenus lors de l'entrainement du modèle, Il est maintenant possible de poursuivre sur l'algo qui sera exécuté pour faire la prédiction. Lorsque le module speech-to-text envoi un nouveau message client au spam-filter, voici le déroulement des actions qui seront exécutées :

- La langue dans laquelle a été formulée la demande du client est déchiffrée. S'il est autre que du Français, il sera automatiquement considéré comme spam.
- Vérification que le message comporte bien deux lieux ayant une correspondance avec les villes où se trouve au minimum une gare ferroviaire pour chaque lieu. A défaut d'au moins deux lieux valides, le message sera considéré comme spam.
- Nettoyage et pre-processing du message (suppressions des mots inutiles, mettre tout en lowercase, suppression de la ponctuation et autres caractères indésirables etc…)
- Comparaison de chaque mot du message avec la matrice de probabilités stockées en DB afin d'avoir une probabilité globale du message. Le message pourra enfin être jugé comme spam ou ham en fonction de des résultats obtenus. Dans le cas où la proba de ham est identique à la proba

de spam (0.5 et 0.5), le message sera alors considéré comme ham. Ce cas particulier arrive lorsqu'un message est uniquement constitué de deux noms de ville comme par exemple « Paris, Marseille », ou éventuellement si ce sont des mots que notre algo voit pour la première fois.

Le message pourra, à condition d'être un ham, être relayé au module suivant qui consiste à déterminer quelle est la ville de départ, et celle d'arrivée.

7 - Evaluation du modèle

Pour évaluer notre modèle nous avons utilisé la métrique d'accuracy. Pour y parvenir, nous avons confronté notre modèle a un nouveau jeu de données qu'il n'a jamais vu auparavant. Celui-ci est constitué de 3600 messages préalablement labelisés. Pour chacun de ces messages, notre algorithme a prédit s'il s'agissait d'un spam ou d'un ham. Nous avons comparé ces prédictions aux labels et avons comptabilisé chaque correspondance juste. Finalement le nombre de prédictions justes par rapport au nombre total de prédictions nous a donné le pourcentage d'accuracy. Nous avons obtenu un résultat de 92% de réponses justes, ce qui nous semble acceptable et suffisant pour poursuivre au module suivant

NLP

Le NLP (Natural Language Processing) est une **branche de l'intelligence artificielle** qui s'occupe en particulier du traitement du langage écrit aussi appelé avec le nom français TALN (traitement automatique du langage naturel). En bref, il regroupe tout ce qui est lié au langage humain et à son traitement par des outils informatiques. Le NLP peut être divisé en 2 grandes parties, le **NLU** (Natural Language Understanding) et le **NLG** (Natural Language Generation).

Le premier est toute la partie « compréhension » du texte, prendre un texte en entrée et pouvoir en ressortir des données.

Le second, est générer du texte à partir de données, pouvoir construire des phrases cohérentes de manière automatique.

Nous serons plus intéressés par le premier cas, qui nous permettra de lui passer une phrase une fois celle-ci considérée comme valide par notre algorithme de spam-filter.

Pour ce faire, il existe plusieurs façons de mettre en place un nlp. En ce qui nous concerne, nous avons décidé de mettre en place différents algorithmes afin de choisir le plus optimal.

1 – L'algorithme retenu : Spacy Matcher

Cet algorithme s'appuie sur la libraire Spacy et son système de Pattern-Matcher. Nous avons retenu cette solution à la vue de ces performances mais elle présente tout de même quelques limites que nous détaillerons.

1.1 - Objectif de l'algorithme

L'approche est simple : récupérer une phrase et extraire une **ville de départ**, une **ville d'arrivée** et éventuellement des **villes étapes**.

Pour réaliser cette analyse, Spacy va utiliser s'appuyer sur des **patterns** qui sont simplement des structures de phrases à identifier dans la phrase. Ces patterns sont définis par nos soins et vont influencer la pertinence de l'algorithme de recherche.

1.2 - Les mots clés

Les mots-clés sont une composante essentielle de nos patterns : ce sont des mots récurrents qui permettent d'identifier l'intention d'une phrase.

Par exemple, pour déterminer l'action de *partir d'un point*, nous avons défini les mots-clés suivants :

```
['partir','être','venir','train','trajet']
['rendre','aller','arriver']
```

Pour simplifier, nous allons chercher dans la phrase, des successions de mots du type "Je pars...", "Je suis...", "Je viens...", et nous allons attendre naturellement une ville à la suite de ce mot-clé. Cette approche simpliste sera étayée par la suite, pour préciser comment nous pouvons faire la différence entre "Je suis à Toulouse" et "Je suis un développeur".

Dans l'exemple précédent, une série de verbes et noms ont été présentés, mais pour préciser notre algorithme, nous utilisons également des mots de liaisons qui seront tout aussi importants du type :

```
['de','depuis']
['a','à','au','aux','en','vers','pour']
```

Ces deux groupements de mots-clés constituent le cœur de nos patterns et donc de notre algorithme.

1.3 - Les patterns

Le pattern MOT-CLÉ - DÉTERMINANT - VILLE

Ce premier modèle est l'un des plus simples. Il va rechercher dans une phrase une succession d'**un mot-clé** + **une préposition + une ville**.

Les mots-clés sont définis par nos soins, donc la phrase devra impérativement en contenir un pour être reconnue par Spacy. En revanche, Spacy permet de rechercher un des mots-clés sous diverses variantes. Par exemple, il va rechercher le mot-clé "partir" de façon stricte donc "partir" mais également sous ses diverses formes conjuguées : "partirons", "partais", "part", etc. Cette approche permet d'étendre notre champ de recherche de manière considérable.

Dans un second temps, Spacy va rechercher un déterminant. Dans ce cas, la librairie est totalement autonome. Nous lui demandons simplement de trouver à la suite du mot-clé, un des déterminants de langue française évidemment.

Si, Spacy valide les deux premières conditions, il va ensuite chercher immédiatement une ville. Encore une fois, Spacy est autonome. Il connait les villes, et plus globalement **les localisations**. Dans ce dernier cas, Spacy connait les localisations mais n'est pas limité aux villes françaises qui, dans notre cas, sont uniquement celles qui nous intéressent. Pour compléter l'analyse, l'algorithme vérifie que la ville trouvée par Spacy est bien une ville réelle en France. Pour ce faire, nous utilisons un dataset du gouvernement listant l'ensemble des villes françaises.
https://www.data.gouv.fr/en/datasets/regions-departements-villes-et-villages-de-france-et-doutre-mer/

Cette double vérification n'est pas limitée à ce pattern. Nous aborderons par la suite, d'autres patterns utilisés par l'algorithme et dans l'ensemble, il sera question de trouver une ville où ce contrôle sera systématiquement appliqué.

Comme mentionné, ce pattern permet d'extraire une ville. En réalité, il est à double sens : selon le groupe de mots-clés qui lui ait donné, la ville extraite sera identifiée comme ville de départ ou ville d'arrivée.

Le pattern VILLE - DÉTERMINANT - VILLE

Ce nouveau pattern permet d'identifier des tournures de phrases du type "Je vais de Paris à Toulouse". Dans ces cas, les villes ne sont plus introduites systématiquement par un verbe d'action qui permet de déterminer l'intention mais par une action globale.

Dans ce modèle **ville + déterminant + ville**, le déterminant est clairement défini par nos soins, contrairement au pattern précédent. En effet, une phrase du type "Je vais de Paris **le** Toulouse" n'a aucun sens.

Ainsi, grâce à nos deux groupes de déterminants, départ et arrivée, et selon le déterminant identifié dans le pattern, quel est la ville de départ et la ville d'arrivée.

Exemple, "Je vais de Paris **à** Toulouse", Paris est la ville de départ et Toulouse la ville d'arrivée. Au contraire, "Je vais à Toulouse **depuis** Paris", Toulouse est la ville de départ et Paris la ville d'arrivée. D'où, l'important d'identifié le déterminant utilisé entre les deux villes.

Le pattern VILLE – PONCTUATION – VILLE

Ce pattern est proche du précédent à la différence que la préposition entre les deux villes est remplacée par une ponctuation. La ponctuation est déterminée de manière autonome par Spacy : ce peut être un tiret, une virgule, etc. Dans ce cas, il est inutile de s'intéresser à la position des deux villes pour déterminer le départ à l'arrivée. Nous considérons que dans ce cas de figure très simple, la première ville est toujours la ville de départ. Exemple, "Je fais le trajet Paris – Toulouse", "Trouve-moi un billet Paris, Toulouse".

Pour être le plus précis possible, ce pattern prend également en charge, un ensemble du type VILLE VILLE, c'est-à-dire deux villes qui se suivent sans ponctuations, sans déterminants. En effet, dans ce pattern, la ponctuation qui fait la liaison entre les deux villes est optionnelle.

Le pattern SPECIFIC

Ce pattern se distingue un peu des autres En effet, ici, on va chercher des combinaisons de mots de manière strictes. Ce pattern est surtout utile pour combler des lacunes de Spacy ou gérer des cas très particuliers qui peuvent difficilement être définis autrement. Il permet d'extraire des phrases du type "Départ Toulouse", "Fin Paris". Nous avons fait le choix de l'intégrer pour pousser notre algorithme, mais dans la pratique, les identifications de ce pattern sont très basiques et la limite de la faute de français.

Le pattern STEPS

Ce dernier pattern permet d'extraire des villes étapes. Ici, il n'est plus du tout question d'identifier des villes de départ et d'arrivée : seulement les villes étapes sont visées.

Ce pattern est plus complexe que les autres à cause de certaines lacunes de Spacy. En effet, dans le cas présent, il aurait été possible d'utiliser Spacy en lui demande de chercher une combinaison du type "par Toulouse, Nice et Lyon". En considérant que le déterminant pour introduire les villes étapes est "par". Le pattern est totalement faisable avec Spacy, mais la librairie peine à reconnaitre les villes. En effet, certaines villes dont Paris notamment ne sont pas reconnues comme villes. Dans ce cas, il semble que Spacy considère Paris comme un dérivé du verbe "parier". Cette observation est d'autant plus étonnante lorsqu'on s'aperçoit que le problème ne se produit pas sur les autres patterns. Après plusieurs heures de recherches, nous avons constaté que la version 2 de Spacy était exempt de ce problème, contrairement à la 3 que nous utilisons. Toutefois, pour assurer le bon fonctionnement de notre algorithme, il nous est impossible de redescendre simplement à la version 2.

Pour contrer ce problème, les mots introduits par le déterminant "par" sont analysés un par un. Si le mot est identifié comme une ville alors elle est sauvegardée en tant qu'étape. En revanche, si la succession des villes est rompue par un autre élément qu'une virgule ou la conjonction "et", l'analyse s'arrête. Cette approche semble primaire mais s'adapte étonnamment bien à la plupart des phrases.

1.4 - Le matcher

Après avoir défini tous ces patterns, Spacy doit les interpréter. Ils sont alors intégrés dans un matcher. Le matcher est simplement un outil qui va analyser une phrase et tenter de trouver une combinaison de mots qui pourraient correspondre à l'un des patterns. Si un des patterns est validé, le matcher retourne la position de la chaine dans la phrase.

1.5 - Le pipe

Nous avons vu que des patterns étaient définis pour identifier des structures de phrases susceptibles

de contenir des villes. Ces patterns sont exécutés par un matcher. Maintenant, il s'agit comprendre comme le matcher est intégré dans le fonctionnement d'analyse de Spacy.

Le matcher est lui-même intégré au sein d'un **pipe**. Derrière ce terme se cache un concept assez simple. Pour analyser, la nature des mots et le sens d'une phrase, Spacy utilise une succession de fonctions d'analyses. L'ensemble de ces fonctions exécutées l'une après l'autre constitue le **pipe**. On pourrait l'assimiler à un "couloir" d'analyse. La phrase initiale est passée en paramètre d'une première fonction d'analyse. Cette première fonction va transformer, extraire, tirer des conclusions concernant la donnée en entrée pour ensuite transmettre son résultat à la fonction d'analyse suivante et ainsi de suite. A l'issue de ce "couloir" d'analyse, nous savons précisément comment la phrase est structurée, de quelles natures sont les mots qui la composent et quelles sont les relations qui les unis.

Dans notre cas, on va insérer à la fin de ce pipe, une fonction d'analyse supplémentaire qui va bénéficier de toutes les observations des fonctions précédentes, notamment une analyse qui nous intéresse beaucoup : la nature des mots contenus dans la phrase, une ville par exemple.

A la fin de ce long processus, nous pouvons déterminer précisément les villes de départ, d'arrivée et les étapes. Evidemment, si une phrase dont le contexte n'a rien avoir, aucunes villes ne serait trouvées.

1.6 - Performance de l'algorithme

Notre algorithme présente des performances convaincantes dans de nombreux cas de figures. Ci-dessous notre jeu de données et les résultats obtenus.

Phrase d'entrée	Ville de départ	Ville d'arrivée	Villes intermédiaires	Résultat
Je pars de Lyon pour arriver à Toulouse.	Lyon	Toulouse		Correct
Demain, j'irai à Lyon, mais aujourd'hui, je suis à Montpellier.	Montpellier	Lyon		Correct
Je voudrais aller à Montpellier depuis Toulouse.	Toulouse	Montpellier		Correct
Je vais aller de Paris à Lyon.	Paris	Lyon		Correct
Demain, je fais le trajet Lyon - Marseille.	Lyon	Marseille		Correct
Demain, je ferai le trajet de Paris à Marseille.	Paris	Marseille		Correct
Je compte prendre un train depuis Lille, et avec un peu de chance si la SNCF n'est pas en retard, j'arriverai à Toulouse.	Lille	Toulouse		Correct

Ville de départ Toulouse et ville d'arrivée Lille.	Toulouse	Lille		Correct
Trajet Paris à Marseille.	Paris	Marseille		Correct
Trajet Paris depuis Marseille.	Paris	Marseille		Correct
Aller de Toulouse à Lille demain.	Toulouse	Lille		Correct
Aller a Toulouse depuis Paris demain.	Paris	Toulouse		Correct
Départ Toulouse vers Perpignan.	Toulouse	Perpignan		Correct
Je suis à Toulouse, je voudrais aller demain à Perpignan.				Incorrect
Trajet de Lille à Toulouse ce soir.	Lille	Toulouse		Correct
Je voudrais aller de Toulouse à Paris après demain.	Toulouse	Paris		Correct
Trajet Lyon Toulouse aujourd'hui.	Lyon	Toulouse		Correct
Je voudrais me rendre à Toulouse demain, je suis à Nantes aujourd'hui.	Nantes	Toulouse		Correct
Trains disponibles pour aller à Marseille en venant de Lille.	Lille	Marseille		Correct
Lille Marseille.	Lille	Marseille		Correct
Lille en venant de Marseille.				Incorrect
Je vais de Toulouse à Marseille en passant par Perpignan Montpellier Nice.	Toulouse	Marseille	Perpignan, Montpellier, Nice	Correct

Sur 22 phrases, 19 sont analysées correctement. En revanche, 3 sont mal interprétées par l'algorithme : aucune ville n'est trouvée. Nos tests qui ont échoué sont toujours le résultat de villes introuvables : l'algorithme n'inverse jamais la ville de départ et la ville d'arrivée.

Pour les phrases "Je suis à Toulouse, je voudrais aller demain à Perpignan" et "Lille en venant de Marseille", on peut considérer que l'erreur est acceptable. En effet, les phrases sont mal tournées ou imprécises.

1.7 - Limite de l'algorithme

Notre algorithme est confronté à deux contraintes.

La première issue de notre propre interprétation. En effet, notre subjectivité a influencé le programme notamment lorsqu'il a fallu définir les mots-clés et les prépositions à cibler. Il est certain que des tournures de phrases tout à fait correctes ne seront pas identifiées. Cette observation est appuyée par notre jeu de test qui bien que diversifié reste limité. Certains cas de figures ont été omis sans aucun doute. Il parait difficile de s'en détacher complétement : tout algorithme d'intelligence artificielle est influencé par les biais de leurs auteurs.

La seconde limite relève la librairie Spacy. Comme énoncé dans le pattern Steps, Spacy est parfois sujet à des erreurs d'interprétation comme pour le terme Paris qui devrait être reconnue comme une ville mais qui est parfois classé comme une conjugaison du mot parier. Cet échec est d'autant plus frustrant quand l'erreur n'est pas reproduite dans une version antérieure de la librairie.

Malgré tout, à la suite de nos observations, nous constatons que notre algorithme NLP s'avère efficace dans la majorité des situations.

2 - Algo (option 2)

Ce modèle se base sur l'entité de l'annotation linguistique 'LOC' ou 'GPE' qui permet d'extraire un token correspondant à une ville ou un pays

Une fois ces tokens récupérer, il parcourt le Doc afin d'identifier via les classes Word & LinkedWord prédéfinie contenant une multitude de mots, à quelle propriété grammaticale / lexical il correspond, s'il trouve une propriété, il cherche à voir si le mot observé est présent dans la liste de mots de la propriété, en lui attribuant une force et une direction.

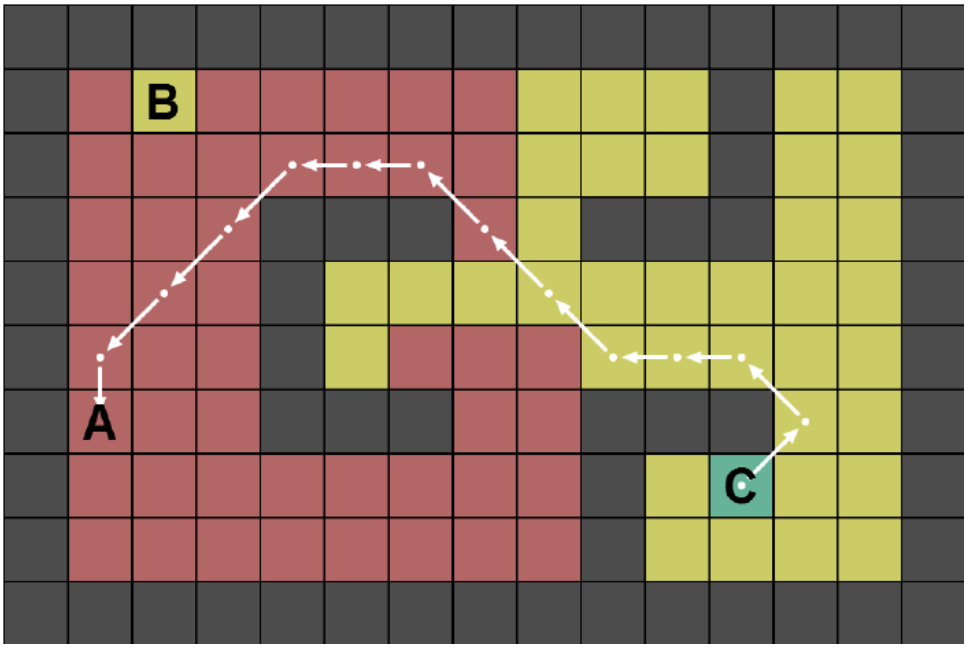
Une fois que toutes les propriétés grammaticales ont été identifiés (Part of Speech Tagging) , on construit un sens de la phrase via les propriétés de Direction et de Force implémenter par les classes Word et LinkedWord

3 - Algo (option 3)

Ce modèle est basé principalement sur la librairie spacy, un dictionnaire de verbes, et sur un calcul de distance entre les mots de ladite phrase.

Il prend en paramètre la phrase de l'utilisateur, la traite et en stock tous les verbes ainsi que les villes qu'il y aura trouvé. Il crée un tableau avec l'infinitif de tous ces verbes qu'il garde sur le côté. Un calcul basé sur la distance des verbes et des noms de ville est donc effectué pour trouver la ville de départ et celle d'arrivée.

Path-Finding



1 - Le concept

Le pathfinding se rattache plus généralement au domaine de la planification et de la recherche de solution. Il consiste à trouver comment se déplacer dans un environnement entre un point de départ et un point d'arrivée en prenant en compte différentes contraintes. Dans le cas de notre exercice, il s'agit de proposer à l'utilisateur l'itinéraire le plus optimal. Soit de trouver le chemin ferroviaire le plus rapide d'une ville à une autre.

2 - Plusieurs options possibles

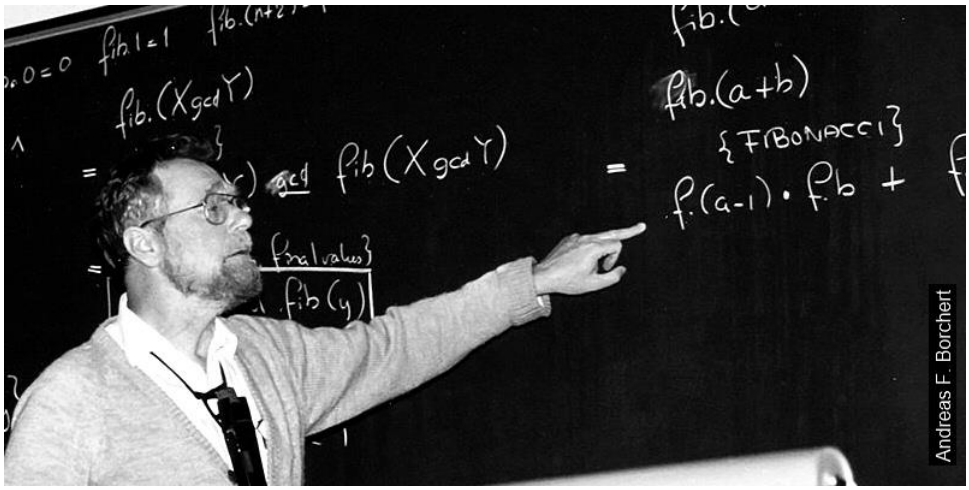
Deux algos sortent du lot lorsqu'il s'agit de pathfinding.

- Dijkstra : Cet algorithme sert à résoudre le problème du plus court chemin en théorie des graphes. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

- A* : C'est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final tous deux donnés. L'algorithme A* a été créé pour que la première solution trouvée soit l'une des meilleures, c'est pourquoi il est célèbre dans des applications comme les jeux vidéo privilégiant la vitesse de calcul sur l'exactitude des résultats.

3 – Solution retenue

L'objectif de départ était de proposer les deux solutions afin de pouvoir les benchmark l'un à l'autre. Seulement, par manque de temps nous avons été contraints de laisser de côté la solution A*, une seule solution sera implémentée et ce sera Dijkstra. Dommage, A* semblait être la meilleure option pour résoudre ce problème.



4 – Le dataset

Une fois que nous avons choisi l'algorithme, il reste plus qu'à lui donner de quoi manger.

Les données qui nous ont été fourni étaient volumineuses et éparses. Scindés en plusieurs fichiers, il a fallu dans un premier temps regrouper toutes les données en un seul dataframe pour y voir plus clair. Nous avons ensuite éliminé les colonnes inutiles puis procédé à un nettoyage classique.

Des dataframe plus petits ont été créer pour répondre aux besoins de chaque module puis insérées en DB cloud.

Dans le cas de notre problème de chemins le plus court, il nous a fallu cartographier toutes les gares sous forme de graph. Chaque gare correspond à un sommet de notre graph, et le lien entre deux sommets (s'il y en a un) à pour valeur non pas la distance entre eux deux, mais plutôt le temps que met le train pour faire ce trajet.

Il a fallu prendre en compte les lignes (tel une ligne de métro) qui contiennent plusieurs gares afin de réduire les correspondances.

Prenons l'exemple ci-dessous. La ligne comporte 4 gares (A, B C et D). Dans notre graph, nous aurions les sections suivantes :

A => B

A => C

A => D

B => C

B => D

C => D

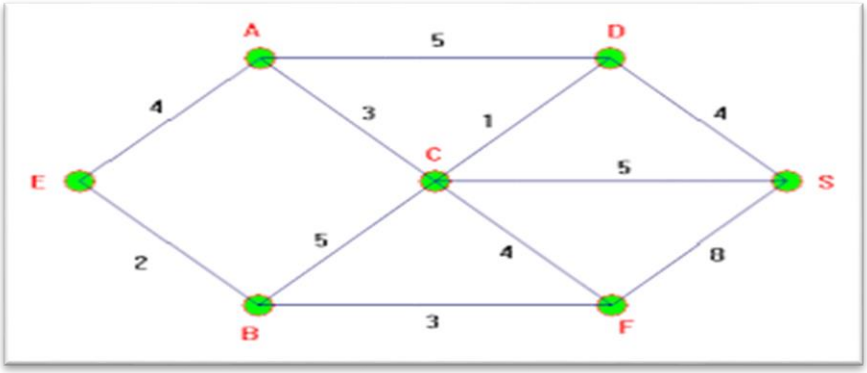
On pourra donc lui proposer un trajet direct A => D plutôt qu'un trajet A => B => C => D.

Au final, chaque sommet de notre graph va se retrouver avec une multitude de liens représentant toutes les combinaisons possibles et cohérentes de nos trajets.

5 – L'algorithme

Le fonctionnement de l'algo est le suivant.

Prenons comme gare de départ E puis comme gare d'arrivée S sur l'image ci-dessous.



E va commencer avec une valeur de 0. Depuis E nous prenons les sommets adjacents et gardons en mémoire le temps qu'il a fallu pour faire le trajet. Dans ce cas en A nous avons 4 puis en B nous avons 2. Nous poursuivons de la même manière sur tous les sommets visités. En D nous obtenons 4+5, en C nous retenons 2+5 et 4+3 qui donnent tous deux 7, en F nous obtenons 5. Lorsque plusieurs chemins mènent à un même sommet. Nous retenons en ce sommet le chemin le plus court avant de confronter ce sommet aux sommets adjacents non visités.

D, C et F ayant un lien direct avec notre gare d'arrivée. Nous pouvons enfin calculer le chemin le plus court.

Les chemins étudiés sont :

Phase 1

E à A = 4

E à B = 2

Phase 2

A à D = 5 donc E à D = 9

A à C = 3 donc E à C = 7

B à C = 5 donc E à C = 7

B à F = 3 donc E à F = 5

Phase 3

C à D = 1 donc E à D = 8

C à S = 5 donc E à S = 12

C à F = 4 donc E à F resta à 5 (phase 2, 5 < 7+4)

D à C = 1 donc E à C reste à 7 (phase 2, 7 < 9+1)

D à S = 4 donc E à S = 12

F à C = 4 donc E à C reste à 7 (phase2, 7 < 5+4)

F à S = 8 donc E à S reste à 12 (phase3, 12 < 5+8)

Nous trouvons deux chemins les plus courts avec la même valeur de 12 :

E à A à C à D à S et E à A à C à S

Le chemin à privilégier est le second car il a le moins de correspondances parmi les deux.

Contrairement à certains autres algos qui vont brute force et calculer tous les chemins possibles, l'avantage de Dijkstra est que certains chemins vont au fur et à mesure être abandonnés ce qui rend l'exécution beaucoup plus rapide.

Pour rendre notre application plus user friendly, il nous a fallu prendre en compte un cas particulier où la requête de l'utilisateur est minimaliste et imprécise.

Dans l'hypothèse où l'utilisateur donne que le nom des villes de départ et de destination, sans donner de précisions sur le nom de la gare. L'algo va faire un pathfinding sur chacune des gares présente dans la ville de départ, jumelée avec les villes de destination. En réalité sur chaque combinaison de gares possibles et proposera l'itinéraire qui est le plus rapide parmi tous les résultats obtenus.

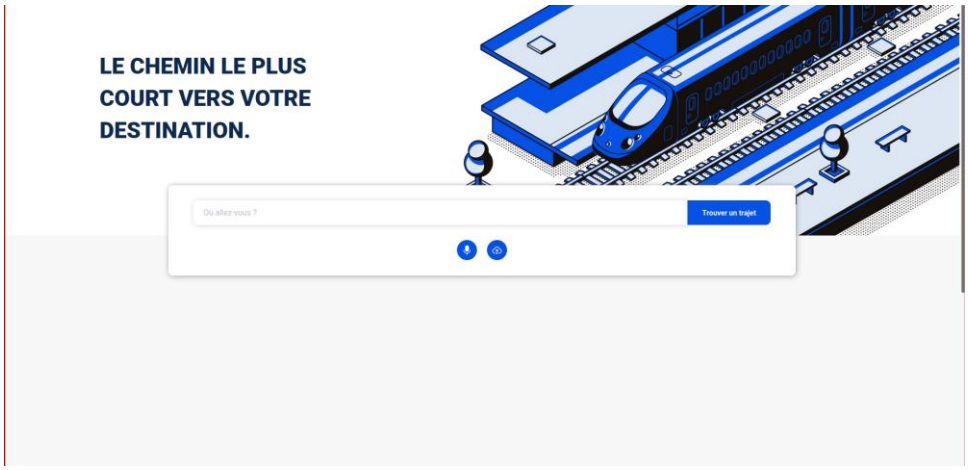
En output nous obtenons une liste des sommets visités pour ce chemin le plus court, puis le temps total du trajet.

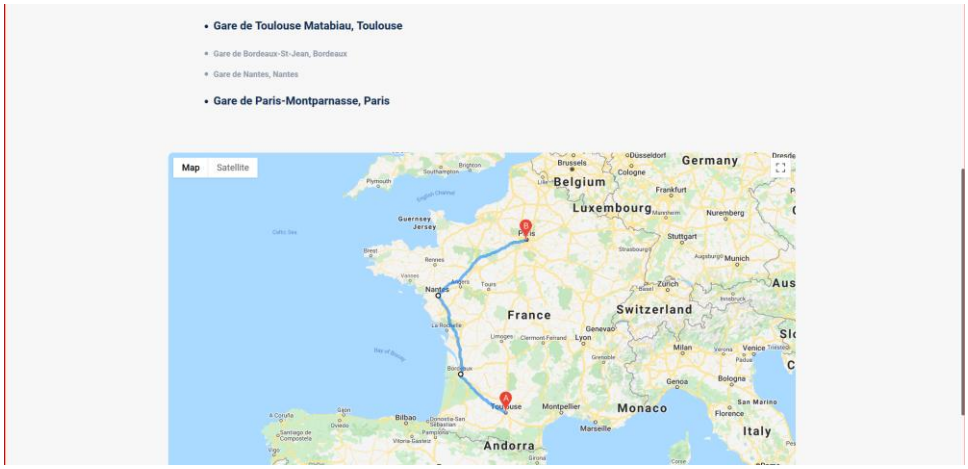
A ce stade la recommandation est terminée et l'application peut désormais afficher le résultat pour l'utilisateur.

6 – Evaluation du modèle

Pour évaluer notre modèle, nous avons pris plusieurs paires " départ - arrivé " puis comparé les résultats obtenus avec ceux d'autres groupes d'étudiants. Nous avons aussi comparé avec résultats obtenus en faisant des recherches sur lignes TER sur https://www.oui.sncf. Les temps de trajet comparés étaient cohérents. La trajectoire affichée sur la carte de France est aussi un repère pour déceler d'éventuelles incohérences.

Front-end

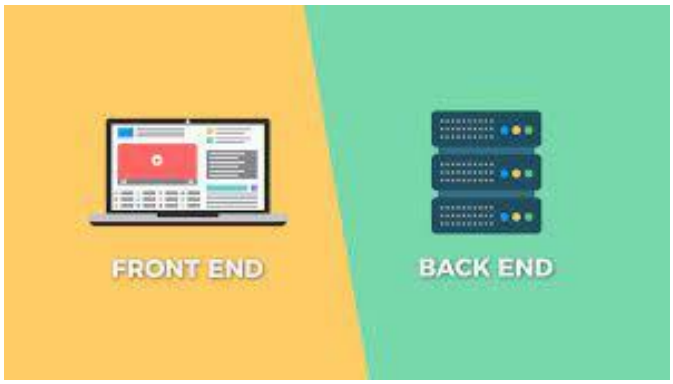




Afin de fournir un visuel adapté et intuitif, nous avons mis en place un one page, permettant à l'utilisateur de retrouver toutes les fonctionnalités sur une seule page. Ce dernier dispose donc d'un champ qu'il peut remplir à la main ou faire remplir par un fichier vocal.

Pour la mise en place du Front nous avons utilisé de l'html, Css, Js basé sur du serveur rendering.

Back-end



Les interactions sur notre site sont gérées grâce à la mise en place d'une Api, qui récupère tous les services et permet donc au front de pouvoir interagir avec ces derniers.

Nous avons utilisé Flask comme techno backend.

Résultats et Performances

1 - Résultats

Afin de juger du bon fonctionnement de notre application, nous avons pris deux métriques en compte.

L'accuracy que nous avons vu précédemment nous a permis de s'assurer de l'efficacité de notre spam-filter et de s'assurer qu'il ne soit ni trop ségrégatif avec les requêtes valides, ni trop permissif avec les requêtes non valides.

Le module NLP a été lui aussi comparé à un jeu de données labelisé pour s'assurer qu'il n'inverse pas les villes de départ avec les destinations.

Le pathfinder aurait idéalement dû subir le même sort avec un dataset de test labelisé avec des itinéraires optimaux et de comparer ceux-ci à ceux préconisés par notre solution. Ces données n'étaient pas à notre disposition et les recréer nous-même pouvait être source d'erreur. Après comparaison, les itinéraires suggérés nous ont semblé pertinents été sans incohérences.

2 - Performances

Le temps de chargement de page ainsi que la manière dont ont été stockés les données et la nature de celles-ci, ont minutieusement été réfléchis dans un but d'optimisation des performances.

Nous avons été sensibles aux performances à tous les niveaux, jusqu'au sein même de notre code avec des choix judicieux tels que faire l'usage de dataframes plutôt que de listes. Ou bien faire usage de transform() plutôt que de boucler sur une structure de données, etc...

L'exécution nos algorithmes nous semble optimal et offre à l'utilisateur une expérience ergonomique, rapide et efficace sans même négliger le design.

3 - Aller plus loin...

Coté pathfinder

La prise en compte du temps entre correspondances pour que l'itinéraire soit réaliste. Aucun voyageur ne pourrait faire une correspondance en deux minutes, même s'il s'appelle Bolt.

Lorsque le voyageur n'est pas explicit sur le nom des gares. Afficher la liste des gares présentes dans la ville demandée et forcer l'utilisateur à en choisir une pour affiner la recherche et n'avoir qu'à lancer le pathfinding sur une seule paire de gares. La recherche serait bien plus rapide et on exclurait les résultats ambiguës.

Avoir un deuxième modèle de comparaison (A*). Ça aurait été intéressant de voir lequel afficherait le plus rapidement un itinéraire cohérent.

Trouver une meilleure manière d'évaluer le modèle.

Coté Spam-filter

Nous pouvons imaginer de sauvegarder tous les messages clients en DB afin de pouvoir réajuster les paramètres de notre modèle à intervalles réguliers via un cron. De cette façon, notre spam-filter deviendrait de plus en plus précis au fil du temps.

Autre idée serait d'accepter les demandes clients qui n'ont qu'un lieu, puis utiliser la geoloc de l'adresse IP pour déterminer la ville de départ en supposant que le lieu renseigné serait la destination.

Ou bien encore, un modèle qui prend en compte le sens et la structure de la phrase. Car notre modèle traite les mots indépendamment et on perd tout le sens de la phrase ce qui peut être une source d'erreur dans les phrases compliquées.

L'application dans sa globalité

Héberger notre solution sur Heroku par exemple aurait été sympathique.

Rendre la solution polyglotte aurait été encore mieux.