École pour l'Informatique et les nouvelles technologies (EPITECH)
Programme Master of Science

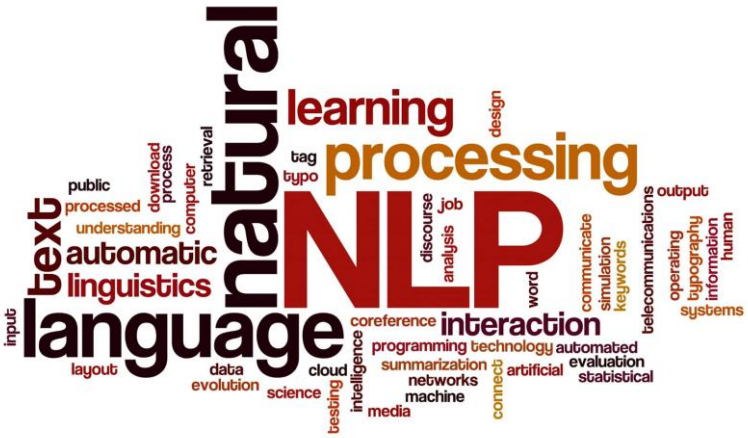
RAPPORT

Pour le projet Intelligence Artificielle – *M2 Semestre 9*

Sur le sujet NLP

Rédigé par

DAMODARANE Jean-Baptiste
ZHANG Victor
TALATIZI Kamel



De septembre 2024 à janvier 2025

Table des matières

1 . Contexte du Projet

1.1 Introduction

Le projet **Travel Order Resolver** s'inscrit dans le cadre du Master 2, semestre 9, et vise à développer une solution innovante combinant plusieurs technologies dans le domaine de l'intelligence artificielle. Ce projet représente un défi académique et technique, mobilisant des compétences en **traitement automatique du langage naturel (NLP)**, en **optimisation sur graphe**, et en **reconnaissance vocale**. L'objectif principal est de créer un programme capable de traiter des commandes de voyage sous forme de texte ou de voix, et de fournir un itinéraire optimal en réponse aux besoins exprimés par l'utilisateur.

L'ère numérique actuelle a conduit à une explosion des données textuelles et vocales, créant une opportunité pour des applications qui peuvent interpréter ces informations et offrir des solutions adaptées. Dans ce contexte, le **Travel Order Resolver** se concentre sur la compréhension du langage naturel, la classification des commandes valides, et l'optimisation des trajets entre différentes destinations.

1.2 Contexte et Motivation

Les systèmes de gestion des trajets, notamment dans le domaine des transports publics, souffrent souvent d'une interface utilisateur complexe et de capacités limitées en matière de traitement de requêtes en langage naturel. Les utilisateurs s'attendent à des interactions simples et naturelles, comme poser une question ou formuler une demande sous une forme non structurée, par exemple :

- *« Comment puis-je aller de Paris à Lyon demain matin ? »*
- *« Je souhaite voyager de Marseille à Nice . »*

Reconnaître l'intention derrière ces demandes, identifier les destinations et origines mentionnées, et proposer un trajet optimal constituent des tâches complexes nécessitant des techniques avancées en NLP.

Par ailleurs, l'optimisation des trajets sur un graphe de connexions ferroviaires, intégrant des contraintes comme le temps de trajet, les correspondances, ou les itinéraires intermédiaires, apporte une dimension algorithmique cruciale. Enfin, l'intégration d'une couche de reconnaissance vocale pour transcrire des commandes orales en texte enrichit l'expérience utilisateur et rend le système plus accessible.

1.3 Objectifs du Projet

Le projet s'articule autour des objectifs suivants :

1. **Développement d'un module de traitement automatique du langage naturel (NLP)** :
    - Identifier les commandes de voyage valides dans des textes.
    - Extraire les lieux d'origine et de destination.
    - Reconnaître des structures grammaticales variées et gérer les ambiguïtés linguistiques.
2. **Implémentation d'un système d'optimisation de trajets** :
    - Construire un graphe de connexions entre différentes destinations.
    - Identifier les trajets optimaux en minimisant le temps de voyage ou le nombre de correspondances.
3. **Ajout d'un module de reconnaissance vocale (optionnel)** :
    - Convertir les données vocales en texte compréhensible pour le module NLP.
    - Gérer les défis liés aux accents, à la ponctuation, et aux éventuelles erreurs de transcription.
4. **Intégration des différentes composantes** :
    - Concevoir une architecture modulaire permettant de relier les modules NLP, d'optimisation, et de reconnaissance vocale.
    - Garantir une compatibilité avec des formats standardisés (fichiers texte, flux audio).

1.4 Importance Académique

Le projet offre une occasion unique d'appliquer des connaissances théoriques à une problématique concrète et multidisciplinaire. Il englobe :

- **NLP** : Une discipline en pleine expansion avec des applications dans la traduction automatique, la recherche d'information, et les assistants virtuels.
- **Optimisation sur Graphe** : Une base fondamentale en informatique pour résoudre des problèmes liés aux trajets, réseaux, et planifications.
- **Reconnaissance Vocale** : Une technologie essentielle pour l'accessibilité et l'interaction naturelle avec les systèmes.

Ce projet met également en avant l'importance de la collaboration entre différentes technologies et outils, tels que les modèles pré-entraînés (CamemBERT, SpaCy), les algorithmes de recherche optimale (Dijkstra, A*), et les plateformes de traitement vocal.

1.5 Organisation du rapport

Le rapport propose une vue détaillée du projet, depuis sa conception jusqu'à sa finalisation, en débutant par le **Contexte du Projet**, qui présente les objectifs, les problématiques, et le cadre technique requis pour comprendre et réaliser ses composantes.

Ensuite, le rapport explore la **Modélisation de la Carte des Trajets**, qui constitue un élément clé pour optimiser les itinéraires. Cette partie présente d'abord les jeux de données utilisés pour construire la base de connaissances nécessaires. Elle explique ensuite comment ces données ont été organisées en un graphe, représentant les connexions entre différentes destinations. Des algorithmes de parcours y sont décrits pour trouver les trajets possibles, suivis des techniques d'optimisation employées pour minimiser le temps ou la distance.

La section suivante est consacrée à la **Reconnaissance Vocale et Transformation en Texte**, un module optionnel mais essentiel pour rendre l'application plus intuitive. Elle discute des technologies utilisées pour convertir les données vocales en texte et met en lumière les défis rencontrés lors de cette intégration, notamment en termes de précision et de compatibilité avec les étapes de traitement du langage naturel (NLP).

Le cœur du projet repose sur le **Traitement du Langage Naturel (NLP)**, une section qui examine en profondeur les techniques utilisées pour comprendre et analyser les commandes de voyage en langage naturel. Après une introduction aux objectifs du NLP, cette partie décrit les différences entre classification de texte et classification par tokens, ainsi que leur pertinence pour ce projet. La reconnaissance des entités nommées (NER) y est également expliquée, accompagnée d'une présentation des jeux de données conçus spécifiquement pour entraîner et tester les modèles. Cette section met en lumière les étapes de prétraitement des données, les ajustements effectués sur les modèles grâce au fine-tuning, et la construction d'un pipeline NLP robuste.

Les **Modèles et Techniques Utilisés** constituent une étape cruciale pour répondre aux objectifs du projet. Cette section analyse les performances de divers modèles de classification de texte et de classification par tokens. Les expérimentations menées sont ensuite présentées dans une section dédiée, **Expérimentations et Résultats**, qui détaille les tests effectués sur plusieurs modèles (SpaCy, CamemBERT, DistilBERT, CRF, BiLSTM-CRF, entre autres) et propose une analyse comparative des résultats obtenus. Une fois les expérimentations terminées, le rapport justifie le **Choix Final du Modèle Retenu**, en expliquant les critères qui ont guidé la sélection de la meilleure solution pour répondre aux exigences du projet.

Le **Développement Frontend** détaille les objectifs, l'architecture de l'interface utilisateur, les technologies employées et les principes de design (UI/UX) pour garantir une interaction claire et intuitive avec les utilisateurs. Parallèlement, le **Développement Backend** décrit l'architecture, les API pour la communication entre les composants, la gestion optimisée des données et les interactions avec les modules NLP et de pathfinding.

Le rapport se conclut par une **Conclusion** synthétique, où les résultats obtenus sont récapitulés, et où des pistes d'amélioration et des perspectives pour le développement futur du projet sont proposées.

2 . Modélisation de la Carte des Trajets (Préalable à NLP)

2.1 Jeu de Données

Dans cette partie, nous allons explorer les différents fichiers de données utilisés pour analyser les trajets en train en France. Ces fichiers contiennent des informations essentielles sur les gares, les trajets, et les villes, ce qui nous permet de mieux comprendre les réseaux ferroviaires et d'optimiser les parcours. Nous allons d'abord décrire les différents datasets, tels que **timetables.csv**, **gares-de-voyageurs.csv**, et **stations_villes.csv**, qui nous fournissent des informations sur les gares, les trajets et les villes de départ et d'arrivée.

Ces données sont cruciales pour la suite du projet, notamment pour le calcul du **plus court chemin** entre deux points, une fonctionnalité clé qui sera abordée dans la partie dédiée à la **NLP** (traitement du langage naturel). Grâce à ces fichiers, nous pouvons non seulement analyser les trajets entre les gares, mais aussi associer chaque gare à sa ville, ce qui nous aide à mieux comprendre le réseau ferroviaire et à concevoir des outils de planification de voyages plus efficaces. Cette analyse préliminaire des données servira donc de base pour la suite du projet, où l'objectif sera de trouver les itinéraires les plus rapides et les plus optimisés pour les voyageurs.

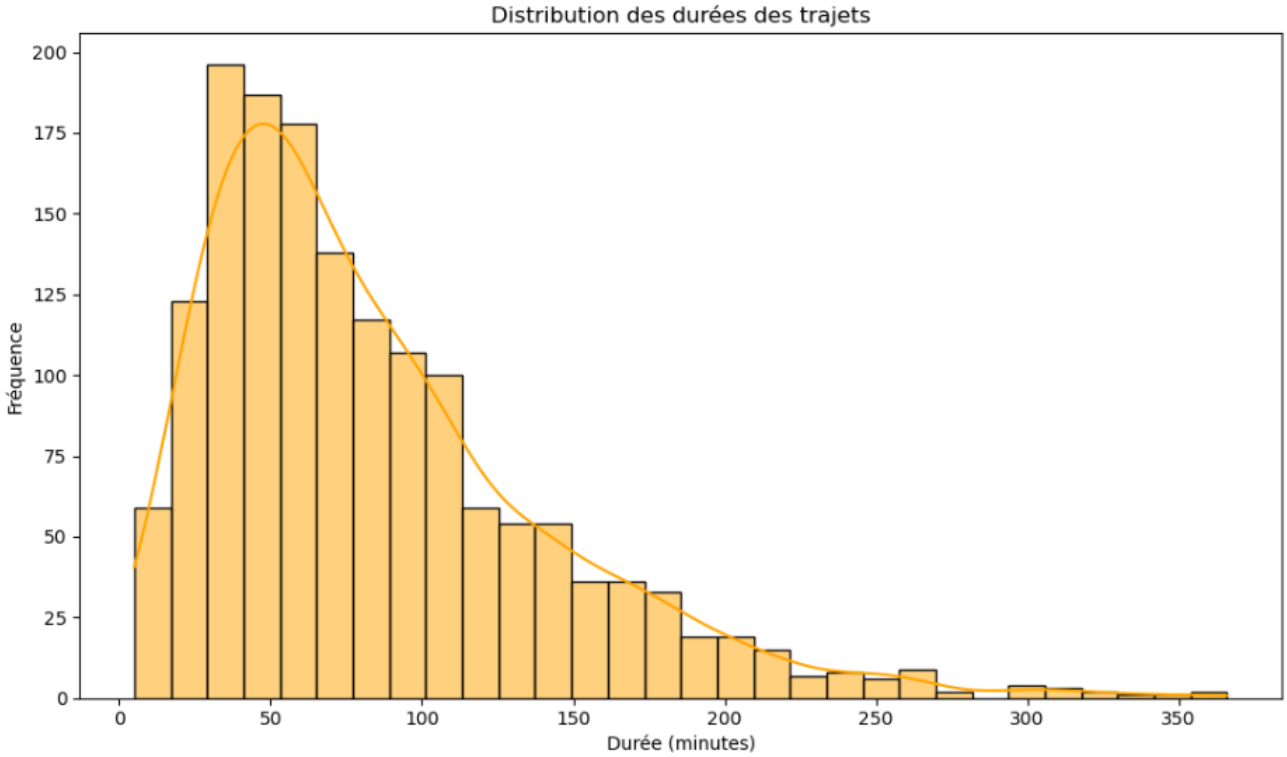
Informations sur les trajets en train

	trip_id	trajet	duree
0	OCESN003100F140147152	Gare de Le Havre - Gare de Paris-St-Lazare	138
1	OCESN003190F040047309	Gare de Dieppe - Gare de Paris-St-Lazare	145
2	OCESN003198F030037315	Gare de Paris-St-Lazare - Gare de Rouen-Rive-D...	97
3	OCESN003300F030037323	Gare de Cherbourg - Gare de Paris-St-Lazare	194
4	OCESN003313F380387526	Gare de Caen - Gare de Paris-St-Lazare	149

Le fichier **timetables.csv** contient des informations simples mais très utiles sur les trajets en train. Chaque ligne représente un trajet spécifique et est composée de trois colonnes. La première colonne, trip_id, est un identifiant unique pour chaque trajet. Cet identifiant suit un format précis qui permet de distinguer facilement les trajets entre eux. La deuxième colonne, trajet, indique la gare de départ et la gare d'arrivée, comme par exemple "Gare de Le Havre - Gare de Paris-St-Lazare". Enfin, la troisième colonne, duree, précise la durée totale du trajet, exprimée en minutes.

Ce fichier est particulièrement pratique pour analyser les temps de trajet entre différentes gares ou pour comprendre les itinéraires empruntés. Par exemple, il nous apprend que le train reliant Le Havre à Paris prend 138 minutes, alors que celui entre Dieppe et Paris dure 145 minutes. Ces informations permettent aussi de comparer les durées et d'identifier les trajets les plus rapides ou les plus longs.

Le fichier a été récupéré à partir de l'ENT d'Epitech. Il peut être utilisé pour créer des plannings, optimiser les correspondances ou encore prévoir des scénarios de transport. Ce dataset est simple dans sa structure, mais il constitue une base essentielle pour travailler sur des projets liés aux transports ferroviaires.



L'analyse montre une forte concentration de trajets courts (0-50 minutes), avec un pic de fréquence près de 200 trajets. Ensuite, la fréquence baisse progressivement et devient très faible pour les trajets supérieurs à 100 minutes.

- **Durée moyenne :** 84,34 minutes, indiquant que la majorité des trajets sont plutôt courts, mais il y a des trajets plus longs qui influencent la moyenne.
- **Durée médiane :** 70,00 minutes, ce qui signifie que la moitié des trajets durent moins de 70 minutes.
- **Écart-type élevé (57,10 minutes),** suggérant une grande variation des durées, allant de 5 à 366 minutes.

En conclusion, la majorité des trajets sont courts, mais il existe une certaine diversité, avec des trajets très longs. Cette variation reflète différents types de lignes, des trajets locaux aux trajets longue distance.

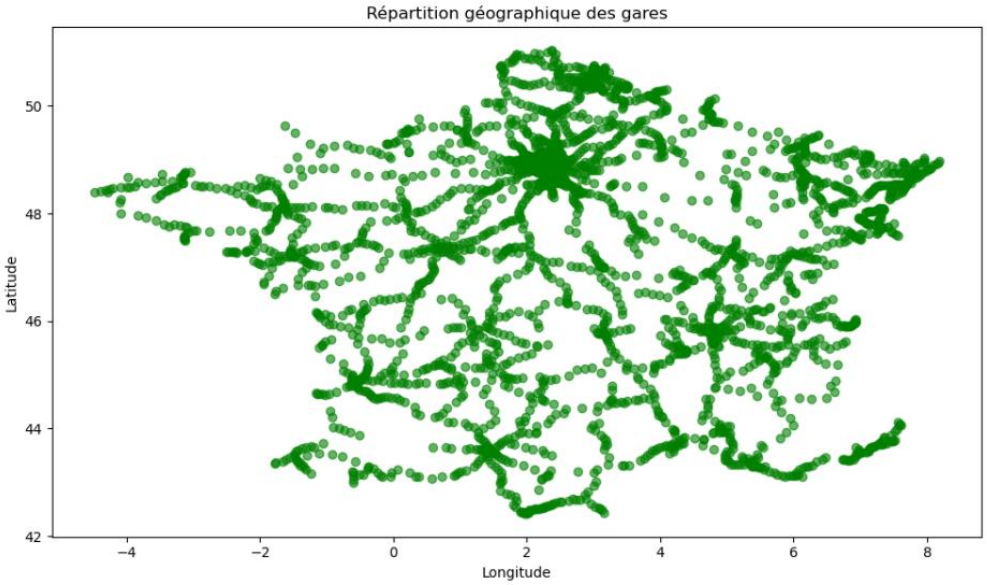
Détails sur les gares ferroviaires pour voyageurs

	Nom	Trigramme	Segment(s) DRG	Position géographique	Code commune	Code(s) UIC
0	Abancourt	ABT	C	49.6852237, 1.7743058	60001	87313759
1	Abbaretz	AAR	C	47.5546432, -1.5244159	44001	87481614
2	Abbeville	ABB	B	50.10221, 1.82449	80001	87317362
3	Ablon-sur-Seine	ABL	B	48.725468, 2.419151	94001	87545269
4	Achères Grand Cormier	GCR	B	48.9551835, 2.0919031	78551	87386052

Le fichier **gares-de-voyageurs.csv** est une liste détaillée des gares ferroviaires pour voyageurs en France. Chaque ligne du dataset correspond à une gare et contient plusieurs informations clés. La première colonne, Nom, donne le nom complet de la gare, comme "Abancourt" ou "Abbeville". Ensuite, la colonne **Trigramme** fournit un code abrégé à trois lettres qui permet d'identifier rapidement chaque gare, par exemple "ABT" pour Abancourt.

Une autre colonne importante est Position géographique, qui indique les coordonnées GPS exactes de chaque gare, sous la forme de latitude et longitude. Cela permet de localiser précisément la gare sur une carte. De plus, la colonne Code commune donne un identifiant unique pour la commune où la gare est située, et la colonne Code(s) UIC fournit un numéro standard utilisé à l'international pour identifier les gares.

Ce dataset est particulièrement utile pour des projets qui nécessitent des données géographiques ou logistiques sur les gares. Par exemple, il peut être utilisé pour tracer des cartes ferroviaires, analyser la répartition des gares en France, ou encore pour créer des applications de navigation et d'information pour les voyageurs. Les données proviennent directement du site officiel de la SNCF, garantissant leur fiabilité.



Le graphe avec les pointillés verts représente la répartition des gares ferroviaires à travers la France, dessinant ainsi une carte du pays.

- **Latitude minimale :** 42,42 (sud de la France)
- **Latitude maximale :** 51,03 (nord de la France)
- **Longitude minimale :** -4,48 (ouest de la France)
- **Longitude maximale :** 8,18 (est de la France)

Cela montre une couverture géographique étendue des gares, allant des régions du sud au nord, et de l'ouest à l'est. Avec un total de **2881 gares**, la densité de gares est plus élevée dans certaines zones urbaines et plus faible dans les zones rurales ou montagneuses. La distribution des gares est relativement homogène, bien que les grandes agglomérations comme Paris, Lyon ou Marseille aient une concentration plus dense. Cette carte permet de visualiser l'accessibilité du réseau ferroviaire et d'identifier les zones moins couvertes.

Informations détaillées sur les gares ferroviaires en France

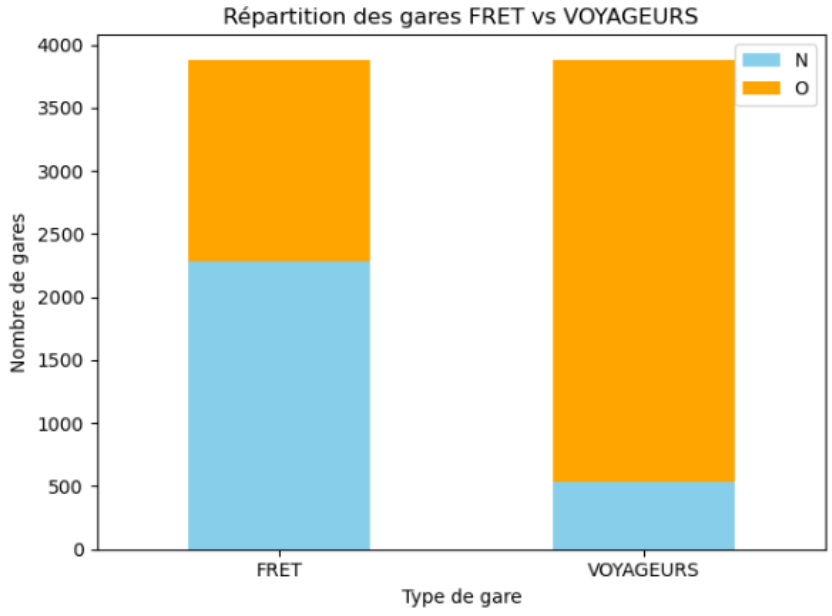
	CODE_UIC	LIBELLE	FRET	VOYAGEURS	CODE_LIGNE	RG_TRONCON	PK	COMMUNE	DEPARTEMEN	IDRESEAU	IDGAIA	X_L93	Y_L93	X_WGS84	Y_WGS84	C_GEO	Geo Point	Geo Shape
0	87009696	La Douzillère	N	O	594000	1	244+100	JOUE-LES-TOURS	INDRE-ET-LOIRE	4650	d9dc0092-6667-11e3-89ff-01f464e0362d	522803.9864	6.695782e+06	0.653001	47.338661	47.33866140621093, 0.6530013866824887	47.338661406198106, 0.6530013866824868	{"coordinates": [0.653001386682487, 47.3386614...
1	87142554	Châtillon-sur-Seine	O	N	839000	1	035+431	SAINTE-COLOMBE-SUR-SEINE	COTE-D'OR	9201	29d3de32-dfbc-11e3-a2ff-01a464e0362d	815990.5833	6.753363e+06	4.551565	47.870404	47.87040423667916, 4.551565119152175	47.87040423666789, 4.551565119152176	{"coordinates": [4.551565119152176, 47.8704042...
2	87382218	La Défense	N	O	973000	1	008+295	PUTEAUX	HAUTS-DE-SEINE	4648	c0d4c69a-f312-11e3-90ff-015864e0362d	644164.1873	6.866206e+06	2.238472	48.893437	48.89343723770491, 2.2384716845345993	48.89343723768303, 2.2384716845345984	{"coordinates": [2.238471684534598, 48.8934372...
3	87718122	Byans	N	O	871000	1	015+118	BYANS-SUR-DOUBS	DOUBS	3446	297c8c1e-dfbc-11e3-a2ff-01a464e0362d	916198.4419	6.672579e+06	5.852088	47.118329	47.11832925172662, 5.852088069196292	47.11832925171314, 5.852088069196295	{"coordinates": [5.852088069196295, 47.1183292...
4	87721829	Chamelet	N	O	775000	1	074+576	CHAMELET	RHONE	3578	d9cff2d6-6667-11e3-89ff-01f464e0362d	816664.1967	6.543552e+06	4.507016	45.981670	45.98167025804158, 4.507016244543598	45.98167025802479, 4.5070162445436	{"coordinates": [4.5070162445436, 45.981670258...

Le fichier **liste-des-gares.csv** est un dataset très détaillé qui rassemble de nombreuses informations sur les gares ferroviaires en France. Chaque ligne correspond à une gare spécifique, identifiée par un code unique, le CODE_UIC. Par exemple, "87009696" représente la gare "La Douzillère". La colonne LIBELLE contient le nom de la gare, ce qui permet de la reconnaître facilement.

On trouve également des informations sur le type d'activités de la gare : les colonnes FRET et VOYAGEURS indiquent si elle est utilisée pour le transport de marchandises (avec "O" pour Oui ou "N" pour Non) ou pour les passagers. Les données incluent aussi des détails techniques comme le CODE_LIGNE et le PK (point kilométrique), qui servent à localiser la gare sur une ligne ferroviaire.

Le fichier intègre des données géographiques très précises, avec les coordonnées en plusieurs formats, comme X_WGS84 et Y_WGS84, qui permettent de la placer sur une carte mondiale. Une colonne Geo Point fournit les coordonnées sous une forme lisible pour des outils de cartographie modernes. Enfin, d'autres champs donnent des informations sur la commune et le département où se trouve la gare.

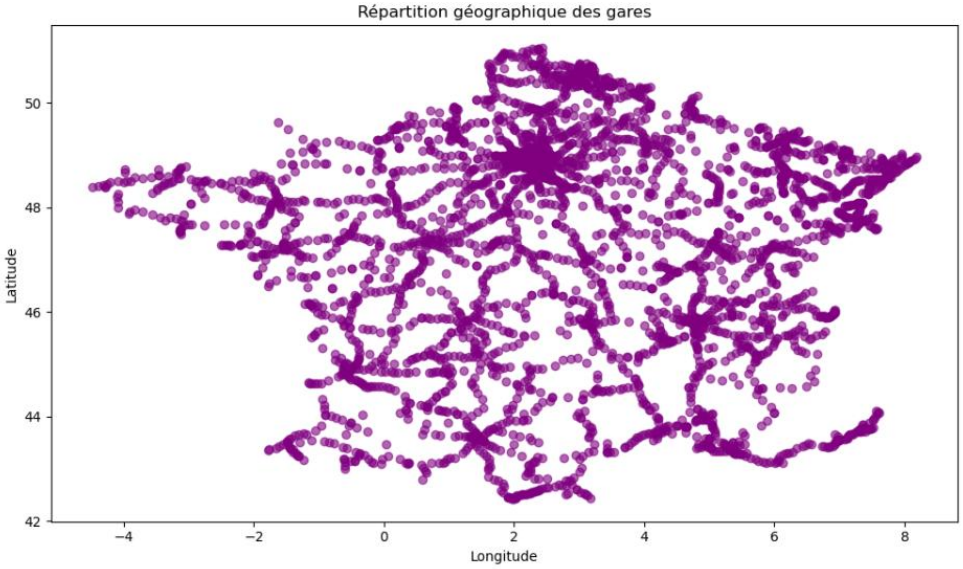
Ce dataset est une mine d'or pour des applications variées : il peut servir à créer des cartes interactives, à analyser le réseau ferroviaire français ou encore à construire des outils pour les voyageurs ou les gestionnaires des infrastructures. Les données viennent de la SNCF, ce qui garantit leur précision et leur fiabilité.



Le graphique montre la répartition des gares selon leur type d'activité, FRET (transport de marchandises) et VOYAGEURS (transport de passagers).

- **Gares FRET** :
  - **2280 gares** ne sont pas utilisées pour le transport de marchandises (indiquées par "N").
  - **1604 gares** sont utilisées pour le transport de marchandises (indiquées par "O").
- **Gares VOYAGEURS** :
  - **532 gares** ne sont pas utilisées pour les passagers (indiquées par "N").
  - **3352 gares** sont utilisées pour le transport de passagers (indiquées par "O").

Cette distribution montre que le réseau ferroviaire est largement dominé par les gares dédiées au transport de passagers, avec plus de **3300 gares VOYAGEURS** actives contre seulement **530** inactives dans ce domaine. En revanche, les gares FRET sont beaucoup moins nombreuses, bien que celles utilisées pour le transport de marchandises restent significatives. Ce résultat souligne l'importance du réseau ferroviaire pour les passagers, avec un plus grand nombre de gares adaptées à ce type de transport par rapport aux infrastructures dédiées au fret.



Le graphique montre la répartition géographique des **3884 gares** présentes en France, dont les coordonnées géographiques s'étendent entre :

- **Latitude minimale** : 42.42
- **Latitude maximale** : 51.05
- **Longitude minimale** : -4.48
- **Longitude maximale** : 8.18

Ces données sont identiques à celles obtenues précédemment.

Intégration des trajets et des villes associées aux gares

	trip_id	trajet	duree	gare_a	gare_b	gare_a_city	gare_b_city
0	OCESN003100F140147152	Gare de Le Havre - Gare de Paris-St-Lazare	138	Le Havre	Paris-St-Lazare	Le Havre	Paris-St-Lazare
1	OCESN003190F040047309	Gare de Dieppe - Gare de Paris-St-Lazare	145	Dieppe	Paris-St-Lazare	Dieppe	Paris-St-Lazare
2	OCESN003198F030037315	Gare de Paris-St-Lazare - Gare de Rouen-Rive-D...	97	Paris-St-Lazare	Rouen-Rive-Droite	Paris-St-Lazare	Rouen-Rive-Droite
3	OCESN003300F030037323	Gare de Cherbourg - Gare de Paris-St-Lazare	194	Cherbourg	Paris-St-Lazare	Cherbourg	Paris-St-Lazare
4	OCESN003313F380387526	Gare de Caen - Gare de Paris-St-Lazare	149	Caen	Paris-St-Lazare	Caen	Paris-St-Lazare

Le fichier **timetables_with_cities.csv** est créé en combinant plusieurs datasets originaux, dont timetables.csv, gares-de-voyageurs.csv et stations_villes.csv, en suivant un processus en plusieurs étapes.

D'abord, on extrait les gares de départ et d'arrivée pour chaque trajet dans timetables.csv. Ce fichier contient une colonne trajet qui décrit les parcours sous la forme "Gare A - Gare B". À l'aide d'une fonction d'extraction, on sépare ces deux gares et on les place dans une nouvelle colonne appelée stations. Puis, on décompose cette liste pour obtenir chaque gare individuellement.

Ensuite, on associe chaque gare aux villes correspondantes en utilisant les données du fichier gares-de-voyageurs.csv, qui contient les noms des gares ainsi que leurs coordonnées géographiques. Cela nous permet de savoir dans quelle ville se trouve chaque gare. Parfois, les noms des gares sont détaillés, comme "Gare de Le Havre". On les simplifie alors en ne gardant que le nom de la gare (par exemple, "Le Havre" au lieu de "Gare de Le Havre").

Après cela, on utilise le fichier stations_villes.csv, qui lie les gares à leurs villes, pour récupérer la ville associée à chaque gare. Grâce à cette étape, on ajoute les villes de départ et d'arrivée dans les colonnes gare_a_city et gare_b_city.

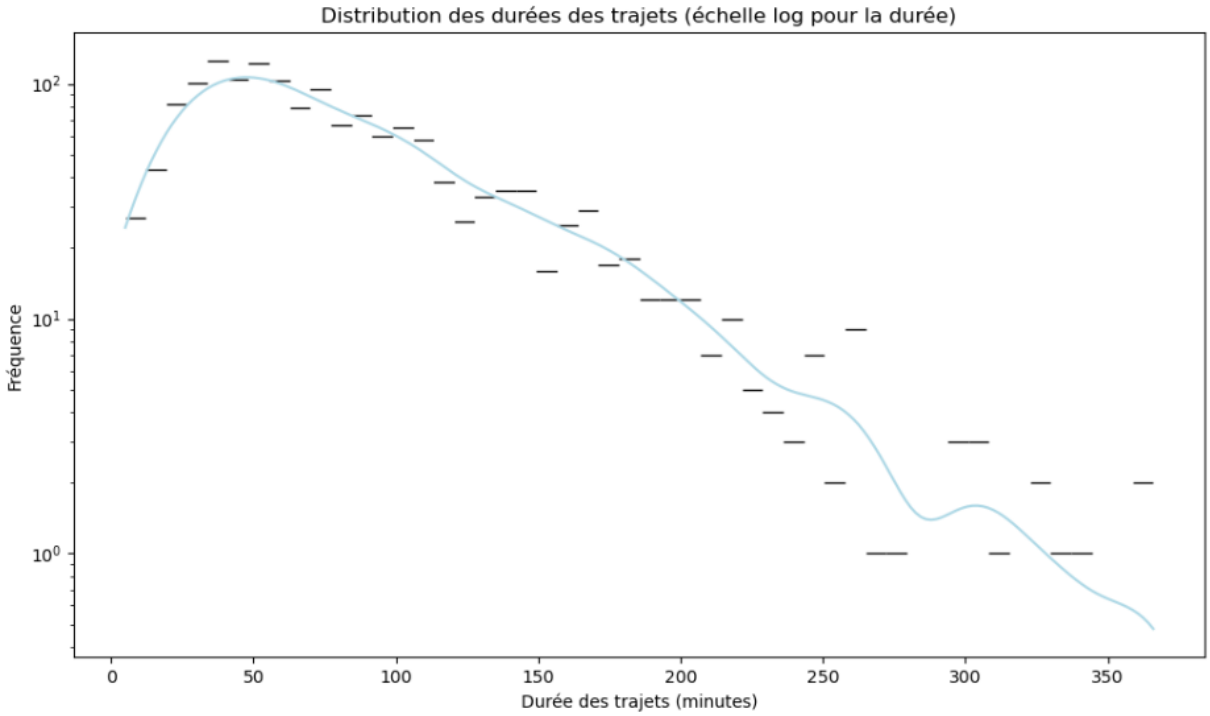
Enfin, on crée le fichier timetables_with_cities.csv, qui contient toutes les informations sur les trajets : identifiant du trajet, description, durée, gares de départ et d'arrivée, et leurs villes respectives. Ce fichier est ensuite enregistré en format CSV, prêt à être utilisé pour des analyses ou des applications de planification de voyages.

En résumé, ce fichier permet de visualiser les trajets en train non seulement entre les gares, mais aussi avec les villes de départ et d'arrivée, ce qui est utile pour des applications de planification de voyages ou pour l'analyse des réseaux ferroviaires.

Le graphe montre la répartition des durées des trajets en utilisant une échelle logarithmique pour mieux visualiser les variations. Les trajets varient entre :

- **Durée minimale** : 5 minutes
- **Durée maximale** : 366 minutes
- **Durée moyenne** : 84.34 minutes
- **Durée médiane** : 70.00 minutes

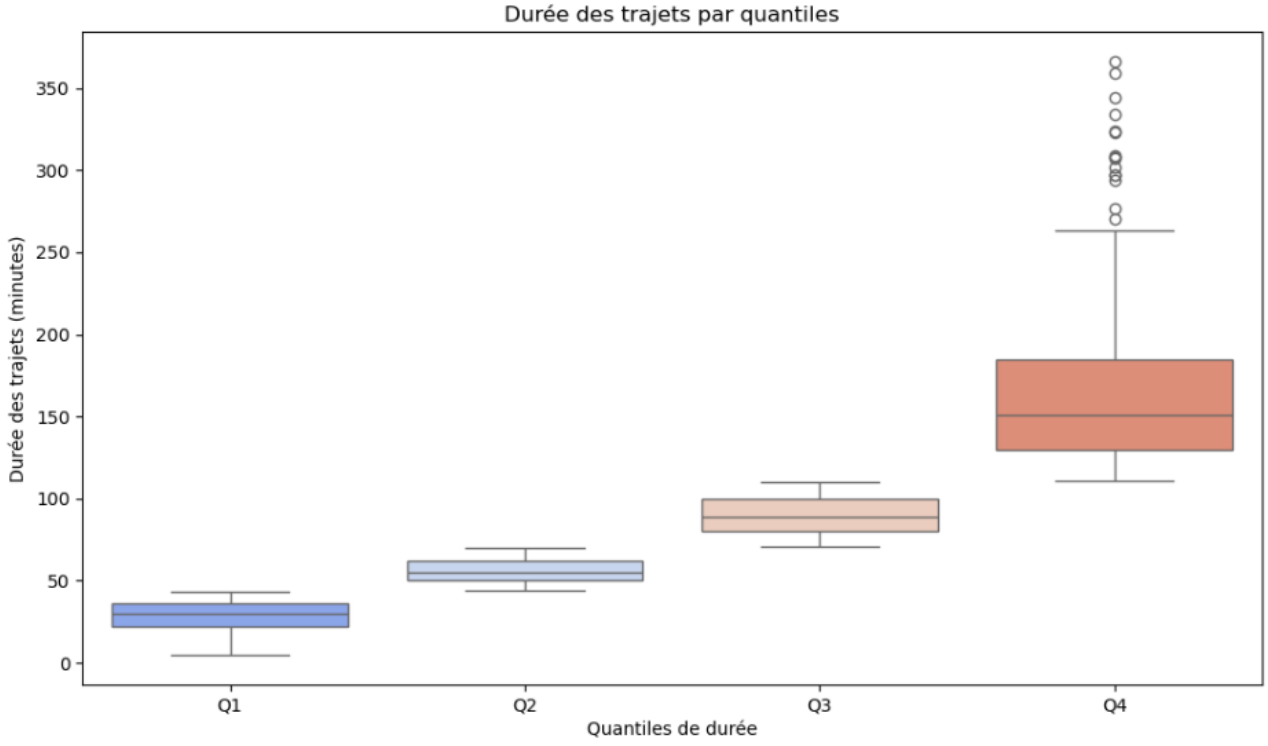
Cela montre une majorité de trajets relativement courts, avec quelques trajets plus longs, notamment dans les durées supérieures à 100 minutes.



Le graphe montre la **répartition des durées des trajets** en utilisant une échelle logarithmique pour mieux visualiser les variations. Les trajets varient entre :

- **Durée minimale** : 5 minutes
- **Durée maximale** : 366 minutes
- **Durée moyenne** : 84.34 minutes
- **Durée médiane** : 70.00 minutes

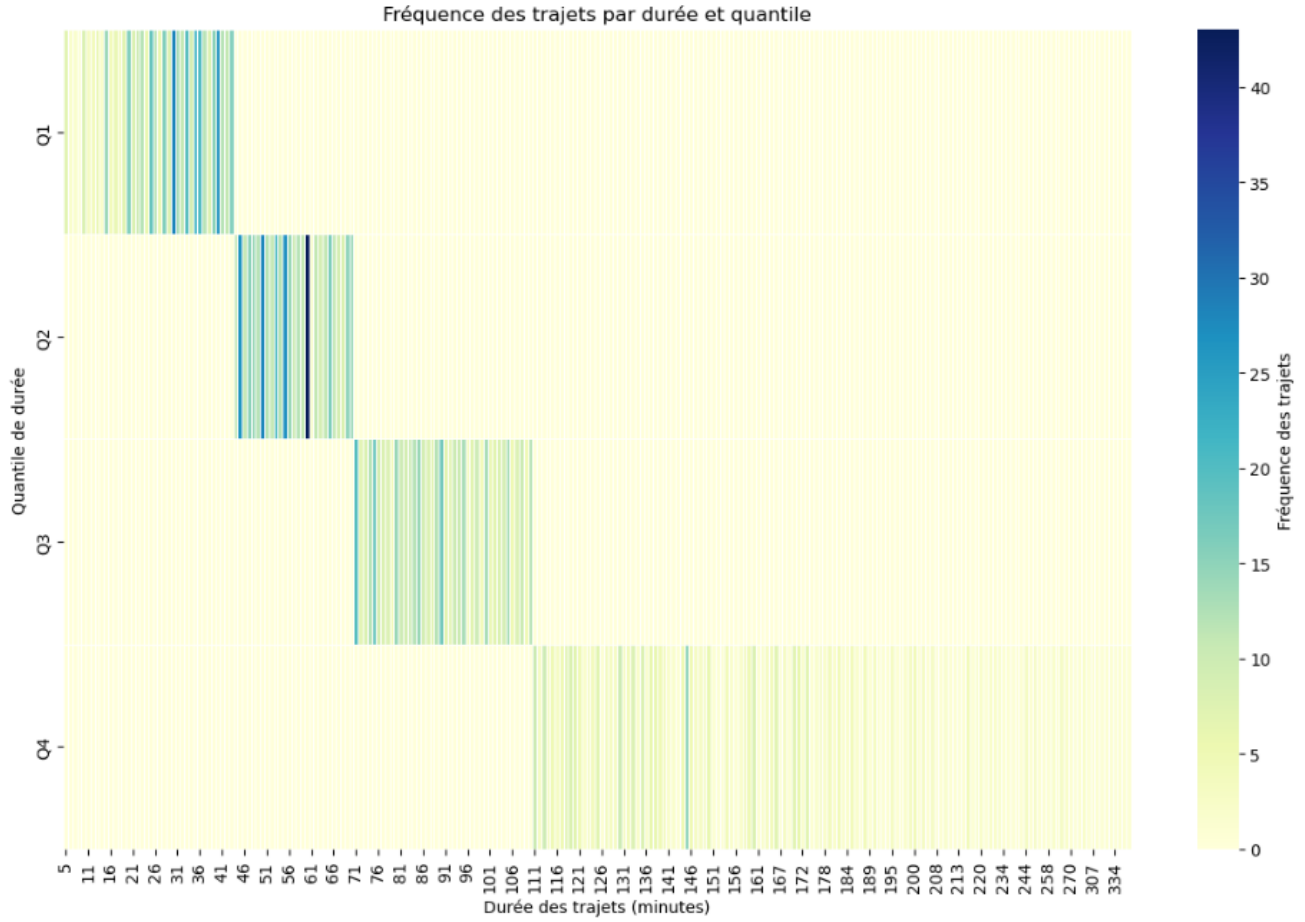
Cela montre une majorité de trajets relativement courts, avec quelques trajets plus longs, notamment dans les durées supérieures à 100 minutes.



Le graphe des **durées des trajets par quantiles** présente la répartition des durées dans quatre groupes distincts. Les moyennes des durées par quantile sont les suivantes :

- **Premier quantile (Q1)** : 28.96 minutes
- **Deuxième quantile (Q2)** : 56.16 minutes
- **Troisième quantile (Q3)** : 89.27 minutes
- **Quatrième quantile (Q4)** : 164.95 minutes

Cela montre une augmentation progressive des durées de trajet, avec une forte concentration de trajets courts dans les premiers quantiles.



La **heatmap de la fréquence des trajets** montre que la **fréquence maximale** de trajets est observée pour des trajets de **60 minutes**, avec **43 trajets** dans le quantile **Q2**. Ce pic indique une forte demande pour des trajets d'une durée d'environ 1 heure, suggérant que cette plage horaire est privilégiée pour de nombreux trajets.

Association des gares et des villes

	station	city
0	LE HAVRE	Le Havre
1	DIEPPE	Dieppe
2	PARIS-ST-LAZARE	Paris-St-Lazare
3	CHERBOURG	Cherbourg
4	CAEN	Caen

Le fichier **stations_villes.csv**, il est créé à partir des données de gares-de-voyageurs.csv et timetables.csv. On commence par extraire les noms des gares dans timetables.csv, puis on les associe à leur ville grâce aux informations du fichier gares-de-voyageurs.csv. Ce fichier contient les noms des gares et leurs coordonnées géographiques.

On utilise ces informations pour créer un DataFrame avec le nom de chaque gare, la ville correspondante et une version simplifiée du nom de la gare (par exemple, "Le Havre" au lieu de "Gare de Le Havre"). Ce fichier fournit ainsi une liste claire des gares et des villes, ce qui peut être très utile pour des analyses géographiques ou pour étudier les données de transport en train.

En résumé, ce fichier permet de relier facilement chaque gare à sa ville, ce qui facilite l'analyse des infrastructures ferroviaires et des trajets qui les connectent.

2.2 Construction du Graphe

2.2.1 Construction du graphe

Dans cette partie, nous allons détailler la manière dont le graphe est construit à partir des données de l'horaire des trains (fichiers CSV) et comment le parcours est effectué à l'aide de l'algorithme de Dijkstra, qui permet de trouver le chemin le plus court entre deux stations.

Chargement des données

Le fichier **timetables_with_cities.csv** contient les informations suivantes :

- **gare_a** et **gare_b** : les deux gares reliées par un trajet,
- **gare_a_city** et **gare_b_city** : les villes correspondantes,
- **duree** : la durée du trajet entre ces deux gares.

Le premier pas consiste à lire ces données et à les organiser de manière à pouvoir construire un graphe. Chaque ligne du fichier représente une connexion entre deux gares. Le processus commence par lire ce fichier CSV, ce qui nous permet d'obtenir un DataFrame **pandas** où chaque ligne représente un trajet entre deux gares, et chaque colonne contient les informations associées.

Le graphe est représenté par un dictionnaire où chaque clé correspond à une gare, et la valeur associée est un autre dictionnaire qui contient les gares connectées à la gare clé ainsi que la durée du trajet entre elles.

Le processus de construction du graphe se déroule comme suit :

1. Pour chaque ligne du DataFrame, on extrait les informations relatives aux deux gares connectées et à la durée du trajet.
2. On ajoute une connexion bidirectionnelle entre ces deux gares dans le graphe. Cela signifie que si une gare A est connectée à une gare B avec une durée d, alors une connexion est ajoutée à la fois de A vers B et de B vers A.

En d'autres termes, chaque connexion entre deux gares est représentée par une arête (A,B) avec un poids , w_{AB} où correspond à la durée du trajet entre A et B. Le graphe peut ainsi être représenté comme : $G = (V, E)$

où V est l'ensemble des sommets (gares) et E est l'ensemble des arêtes (connexions entre les gares). Pour chaque paire de gares A et B, une arête est w_{AB} définie par la durée :

$$E = \{(A, B, w_{AB}) | A, B \in V, w_{AB} \in \mathbb{R}\}$$

Chaque connexion (A,B) a une direction, et son poids est la w_{AB} durée du trajet.

Sauvegarde du graphe

Une fois que le graphe a été construit, il est converti en un DataFrame pour le rendre plus facile à manipuler et à sauvegarder. Ce DataFrame contient trois colonnes : la gare de départ, la gare d'arrivée, et la durée du trajet.

Ensuite, le DataFrame est sauvegardé dans un fichier **Parquet**. Le format Parquet est choisi ici car il est particulièrement adapté pour stocker de grandes quantités de données structurées. Ce format est optimisé pour la lecture et l'écriture efficaces, ce qui permet de gérer des graphes volumineux avec des durées de parcours entre les stations.

Le fichier Parquet permet de garder une version persistante du graphe qui pourra être utilisée par la suite, sans avoir à recomposer le graphe à chaque fois. Cela optimise les performances lorsque l'on travaille avec des graphes complexes et volumineux.

Extrait du fichier parquet :


```
{"station":"Le Havre","connected_station":"Rolleville","duration":26}
{"station":"Le Havre","connected_station":"Fécamp","duration":46}
{"station":"Le Havre","connected_station":"Rouen-Rive-Droite","duration":61}
{"station":"Paris-St-Lazare","connected_station":"Le Havre","duration":138}
{"station":"Paris-St-Lazare","connected_station":"Dieppe","duration":145}
{"station":"Paris-St-Lazare","connected_station":"Rouen-Rive-Droite","duration":97}
```

2.2.2 Algorithmes de Parcours

L'objectif est de trouver le chemin le plus court entre deux stations dans un graphe. Pour ce faire, on utilise l'algorithme de Dijkstra, qui permet de résoudre le problème du plus court chemin dans un graphe pondéré.

2.2.2.1 Principes de l'algorithme de Dijkstra

L'algorithme de Dijkstra fonctionne en partant de la gare de départ et en explorant les voisins de manière itérative. À chaque étape, l'algorithme sélectionne la gare avec la distance minimale calculée jusqu'à présent et explore ses voisins.

L'idée principale derrière l'algorithme est de maintenir une liste des distances minimales calculées depuis la gare de départ vers toutes les autres gares. À chaque étape, l'algorithme met à jour ces distances et ajuste les chemins.

Formellement, pour un graphe G=(V,E), avec un poids associé à chaque arête entre w_{AB} deux sommets A et B, l'algorithme de Dijkstra minimise la fonction de coût d(B), où :

$$d(B) = \min_{C \in V} \{d(C) + w_{CB}\}$$

Cela revient à trouver le chemin le plus court de la gare de départ S à la gare d'arrivée E, en explorant de manière optimale les chemins voisins.

À chaque étape, l'algorithme met à jour la distance minimale à chaque station et enregistre le chemin parcouru. Une fois la gare d'arrivée atteinte, l'algorithme reconstruit le chemin le plus court en remontant les précédents sommets depuis la gare d'arrivée jusqu'à la gare de départ.

2.2.2.2 Représentation du chemin et durée

Une fois le chemin le plus court trouvé, l'algorithme renvoie le chemin sous forme de liste de gares et les durées de parcours entre chaque paire de gares. Ce résultat est ensuite converti en un dictionnaire contenant le chemin, la gare de départ, la gare d'arrivée, et la durée totale du trajet.

Le chemin peut être représenté comme une liste de sommets P=[S,C1,C2,...,E], où S est la gare de départ et E la gare d'arrivée. La durée totale D est la somme des durées des arêtes sur le chemin :

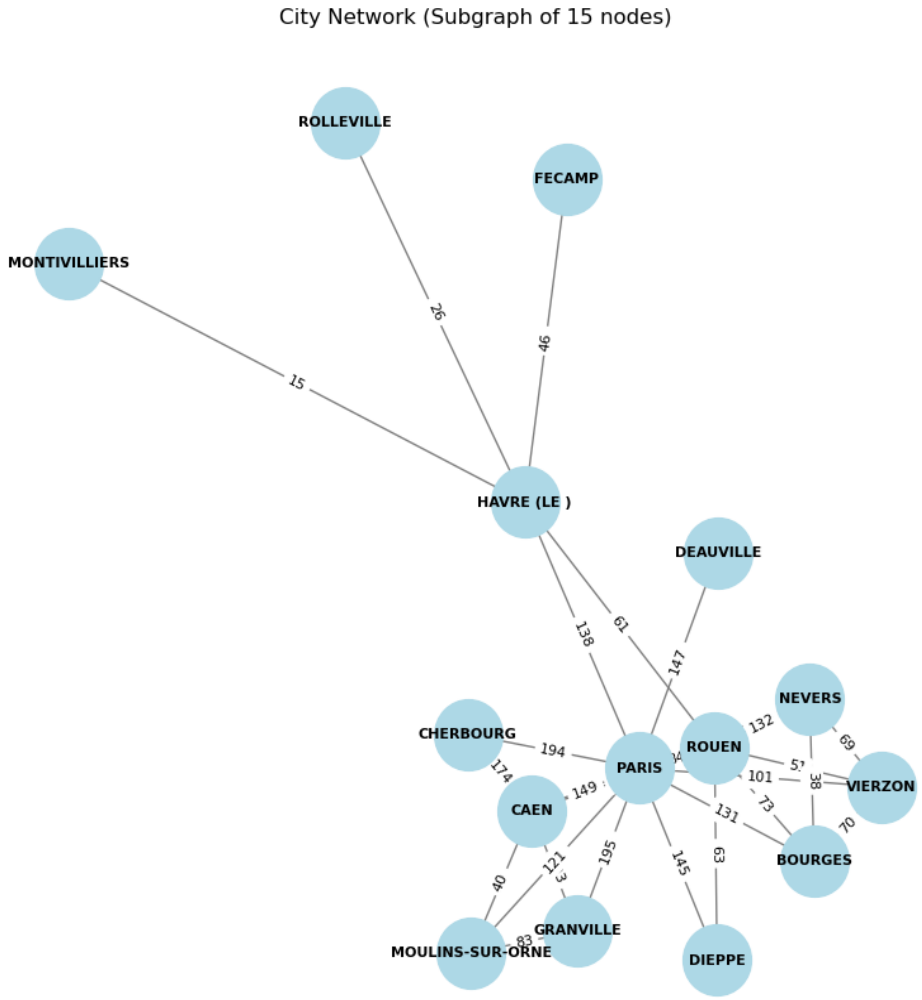
$$D = \sum_{i=1}^{n} w_{P_i, P_{i+1}}$$

P_i et P_{i+1} où sont des gares successives sur le chemin.

La construction du graphe et l'algorithme de Dijkstra permettent de modéliser efficacement les trajets entre les gares et de trouver rapidement le chemin le plus court. Le graphe est construit à partir des données de connexion des gares, et son stockage dans un fichier Parquet optimise les

performances pour des graphes volumineux. L'algorithme de Dijkstra permet ensuite de calculer les trajets les plus rapides entre les gares, en tenant compte des durées de trajet.

2.2.2.3 Exemple de graphe



Le graphe visuel construit ainsi que le graphe utilisé pour parcourir les chemins dépendent entièrement des données contenues dans le fichier Parquet fourni. Ce fichier sert de source principale pour extraire les informations sur les connexions entre les villes, leurs distances respectives et leurs relations.

Les algorithmes implémentés s'appuient sur ces données pour générer un graphe cohérent et précis, en assurant que les relations entre les nœuds reflètent fidèlement les données d'entrée. Toute modification du fichier Parquet se traduira directement par une mise à jour du graphe, ce qui permet une grande flexibilité dans l'analyse et la visualisation des itinéraires.

Ainsi, il est essentiel de garantir que le fichier Parquet contient des données complètes et correctement formatées afin d'obtenir des résultats précis et exploitables.

3 . Reconnaissance Vocale et Transformation en Texte

3.1 Introduction

La reconnaissance vocale est une technologie permettant de convertir la parole humaine en texte exploitable par un système informatique. Dans notre projet, elle constitue une étape clé pour permettre à l'utilisateur d'interagir de manière naturelle avec l'application, en formulant des demandes vocales pour la planification d'itinéraires.

Cette capacité repose sur des techniques avancées de traitement du signal et de Machine Learning pour analyser l'audio et identifier les mots correspondants. Une fois le texte généré, il est utilisé comme entrée pour les modules de détection de langue et d'intention.

Dans notre système, nous avons opté pour une architecture robuste en utilisant des outils éprouvés comme la bibliothèque Python **speech_recognition**. Cela permet non seulement de capturer et traiter l'audio de l'utilisateur, mais aussi de tirer parti de moteurs externes, comme l'API **Google Speech Recognition**, pour une transcription rapide et précise.

3.2 Technologies et Outils

Bibliothèque utilisée : speech_recognition

La bibliothèque Python **speech_recognition** simplifie l'intégration de la reconnaissance vocale dans les applications. Elle offre une interface pour interagir avec plusieurs moteurs de reconnaissance vocale, notamment Google Speech Recognition, Sphinx et d'autres API tierces.

Les fonctionnalités principales de cette bibliothèque incluent :

- **Capture audio via microphone** : Utilisation de l'entrée audio de l'utilisateur.
- **Transcription du discours en texte** : Conversion de l'audio en texte en utilisant un moteur de reconnaissance.
- **Gestion des erreurs** : Détection et traitement des problèmes liés à l'interprétation ou au réseau.

Moteur utilisé : Google Speech Recognition API

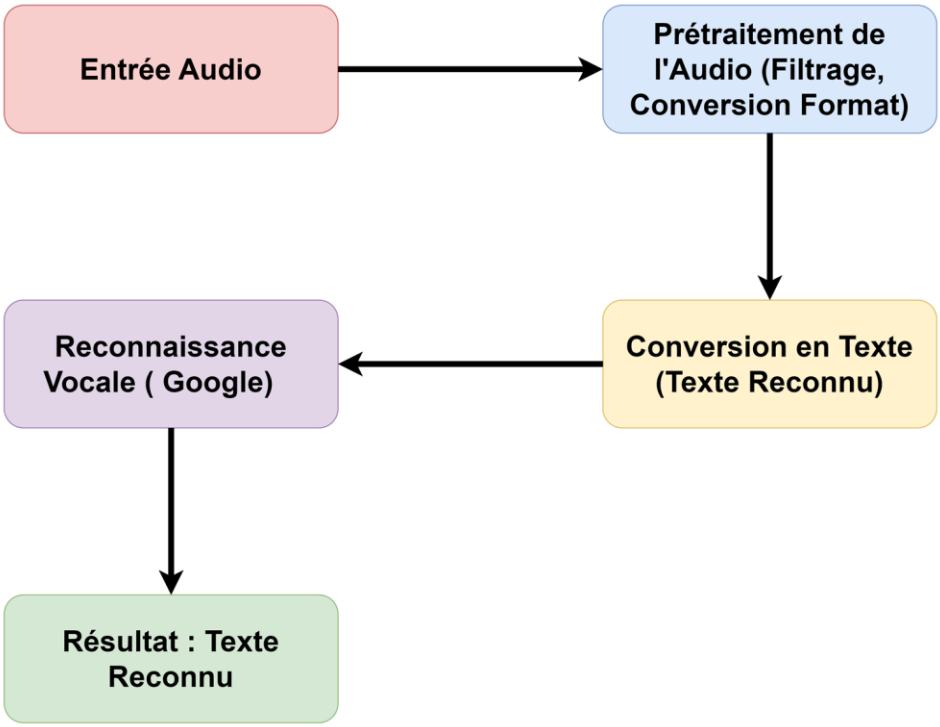
Pour la transcription, le système repose sur le moteur **Google Speech Recognition**, qui utilise des modèles avancés de traitement du langage naturel et d'apprentissage profond. Ce moteur offre les avantages suivants :

- Précision élevée, en particulier pour des langues comme le français.
- Prise en charge de nombreux accents et variations linguistiques.
- Disponibilité en ligne, permettant une mise à jour continue des modèles de reconnaissance.

Étapes du processus

1. **Capture audio** : L'utilisateur s'exprime via le microphone. Le signal audio est capturé en temps réel.
2. **Prétraitement de l'audio** : Le signal est préparé pour être analysé (normalisation, réduction du bruit).
3. **Envoi au moteur** : L'audio est transmis au moteur Google Speech Recognition via une API.
4. **Transcription** : Le moteur convertit l'audio en texte en fonction du modèle linguistique.
5. **Retour au système** : Le texte est renvoyé au programme pour des traitements supplémentaires (par exemple, détection de l'intention).

Schéma illustrant le le fonctionnement de la reconnaissance vocale :

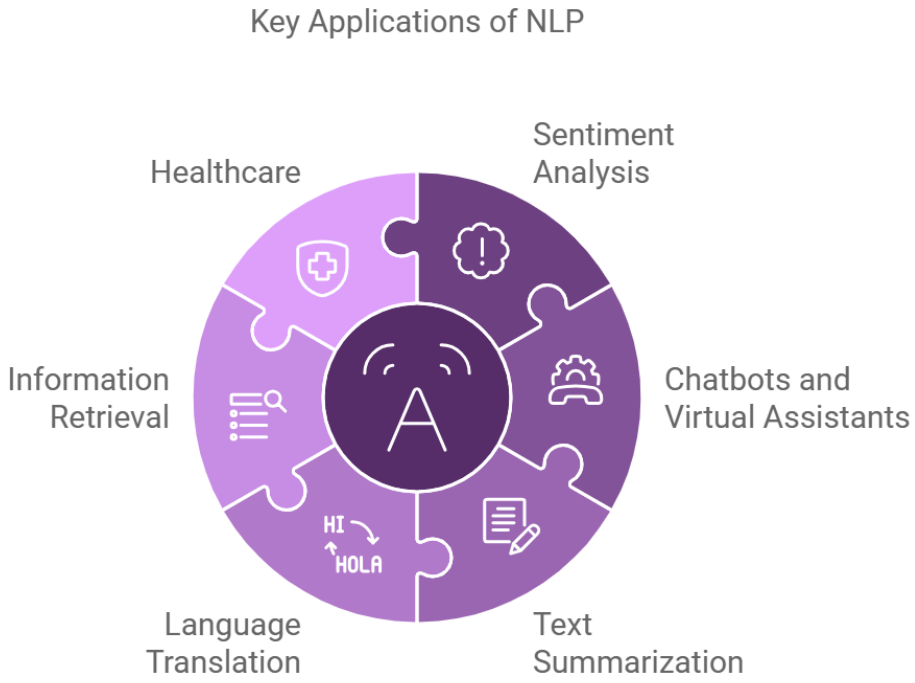


4 . Traitement du Langage Naturel (NLP)

4.1 Définitions et Objectifs du NLP

Le Traitement du Langage Naturel (NLP pour Natural Language Processing) est une sous-discipline de l'intelligence artificielle qui vise à permettre aux ordinateurs de comprendre, analyser, générer et interagir avec le langage humain de manière naturelle. L'objectif principal du NLP est de permettre aux machines de "comprendre" les textes et les paroles humaines, mais aussi de les manipuler de façon pertinente pour accomplir diverses tâches automatisées.

Le NLP repose sur plusieurs techniques et modèles d'apprentissage automatique qui cherchent à traiter le langage humain dans sa forme la plus brute, qu'il soit écrit ou parlé.



Les sous-domaines du NLP

Le NLP comprend plusieurs domaines complémentaires qui se concentrent sur des aspects différents du traitement du langage :

- **Compréhension du Langage Naturel (NLU - Natural Language Understanding)** Le NLU permet à une machine de comprendre le sens d'un texte ou d'une phrase. Cela inclut la compréhension de la grammaire, du vocabulaire, des relations entre les mots, ainsi que de la signification contextuelle. Par exemple, dans une phrase comme "Je veux aller à Paris", le NLU aide à identifier qu'il s'agit d'une demande de voyage et que "Paris" est la destination.
- **Génération du Langage Naturel (NLG - Natural Language Generation)** La NLG permet à une machine de produire du texte compréhensible et pertinent à partir de données brutes. Par exemple, un chatbot peut utiliser la NLG pour répondre de manière

fluide à une question d'un utilisateur : "Quel temps fait-il à Paris ?". Le système pourrait répondre avec un texte généré comme "Il fait 12°C à Paris aujourd'hui."

Les objectifs principaux du NLP sont de rendre les interactions entre humains et machines plus naturelles, en permettant aux ordinateurs de comprendre et générer du langage humain. Parmi les objectifs clés du NLP :

- **Automatiser la compréhension et la génération de textes** : Traitement automatique de documents, conversations et messages, incluant l'extraction d'informations et la génération de résumés.
- **Faciliter les interactions humaines-ordinateurs** : Permettre aux assistants vocaux (comme Siri, Alexa, Google Assistant) de comprendre et répondre à des instructions en langage naturel.
- **Améliorer la traduction automatique** : Utiliser le NLP pour traduire des textes d'une langue à une autre tout en conservant leur sens.

Exemples d'applications concrètes du NLP

Application du NLP	**Description**
Assistants vocaux intelligents	Exemples : Siri, Google Assistant. Ils utilisent le NLP pour comprendre des commandes vocales et effectuer des actions (ex. « Quel temps fait-il ? »).
Analyse des sentiments	Utilisation du NLP pour analyser les opinions des clients sur les réseaux sociaux ou dans les avis en ligne, influençant ainsi les stratégies d'entreprise.
Chatbots et Service Client	Les chatbots utilisent le NLP pour interagir avec les utilisateurs, répondre à des questions et orienter les demandes vers les services appropriés.
Extraction d'information et résumé	Le NLP permet d'extraire des informations clés de documents et de générer des résumés automatiques, utile pour gérer de grands volumes de texte (ex. juridique, médical).

Application au Projet : Extraction des Villes

Dans notre projet, le NLP joue un rôle central pour **extraire les villes** dans les ordres de voyage. Lorsqu'un utilisateur exprime une demande de trajet (par exemple, "Je veux aller de Paris à Lyon"), le système doit identifier correctement les noms des villes, déterminer leur ordre (départ et destination), et gérer les ambiguïtés, comme des villes avec des noms communs (ex. **"Paris"** ou **"Lyon"**). Ce processus fait appel à plusieurs techniques NLP, telles que la **reconnaissance d'entités nommées** (NER), la **désambiguïsation de noms** et l'**analyse syntaxique** des phrases.

4.2 Classifications

4.2.1 Classifications par Tokens

La classification par tokens est une approche utilisée dans le NLP qui consiste à classifier des éléments individuels du texte, appelés **tokens**.

Un token peut être un mot, une ponctuation ou une autre unité sémantique issue d'une phrase. Cette méthode est souvent utilisée dans des tâches comme la **reconnaissance d'entités nommées** (**NER**), où l'objectif est de détecter des **entités spécifiques** comme des lieux, des noms de personnes ou des dates dans le texte.

Dans cette approche, chaque token (mot ou élément) est analysé indépendamment pour déterminer son rôle ou son étiquette dans le texte.

Par exemple, dans une phrase comme "*Je pars de Paris et j'arrive à Lyon*", le modèle peut classifier "*Paris*" et "*Lyon*" comme des entités de type "lieu". Chaque mot ou séquence est donc examiné pour identifier des catégories spécifiques (par exemple, départ et arrivée) en fonction du contexte.

L'un des avantages de la classification par tokens est sa capacité à traiter des textes de manière granulaire, permettant ainsi d'identifier des entités spécifiques et d'analyser les relations entre elles dans un texte donné. Toutefois, cela peut être plus complexe, car il faut s'assurer que les entités sont bien extraites, bien alignées et correctement classifiées.

4.2.2 Classifications par Textes

La classification par texte, quant à elle, traite le texte dans son **ensemble**, plutôt que de manière fragmentée en tokens individuels.

Dans ce cas, l'objectif est d'assigner une ou plusieurs catégories à un document entier, en analysant le **contenu global** de celui-ci. Cette méthode est couramment utilisée pour des tâches comme la **détection de langue, la catégorisation de contenu** (ex : spam vs non-spam) ou encore la **classification d'avis clients en fonction de leur sentiment** (positif ou négatif).

L'une des approches courantes dans la classification par texte est d'utiliser un modèle d'apprentissage automatique pour apprendre des motifs dans le texte, comme des séquences de mots ou des relations sémantiques, afin de prédire la catégorie du texte. Par exemple, un modèle peut apprendre à identifier si une commande de trajet est correctement formulée ou si elle est écrite dans une langue différente, en analysant le texte dans son intégralité.

Cette méthode est souvent plus simple à mettre en œuvre que la classification par tokens, car elle ne nécessite **pas de segmentation du texte en parties plus petites**, et le modèle prend en compte l'ensemble du contenu pour la prédiction. Cependant, elle peut être **moins précise** lorsqu'il s'agit de tâches nécessitant une analyse très fine des unités individuelles du texte, comme la reconnaissance de noms propres ou d'entités spécifiques.

4.3 Named Entity Recognition (NER)

La Reconnaissance d'Entités Nommées (**NER**) est une tâche essentielle du Traitement Automatique des Langues (TAL), qui consiste à **identifier et classer les entités** spécifiques présentes dans un texte, comme les noms de lieux, de personnes, les dates, etc.

Dans le contexte de notre projet, le NER est utilisé pour extraire les informations clés des commandes de trajet, telles que les **villes de départ** (par exemple, "Paris") et **d'arrivée** (par exemple, "Marseille").

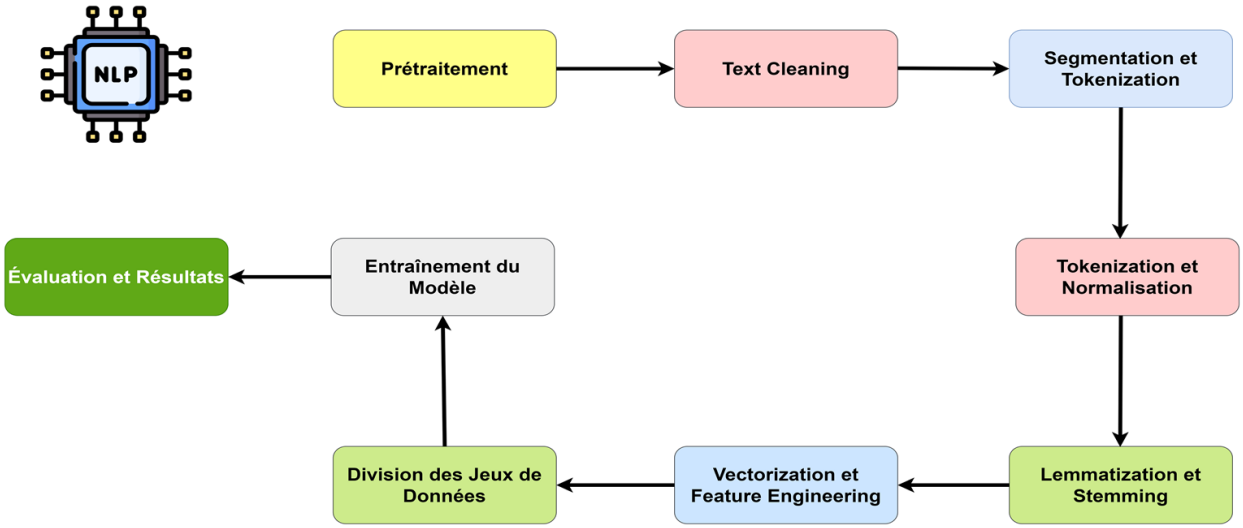
Cette tâche permet ainsi de structurer et d'interpréter les informations présentes dans des phrases textuelles de manière à faciliter leur traitement et leur utilisation dans des systèmes automatisés.

L'utilisation de **modèles préentraînés** comme ceux de **spaCy** nous permet d'effectuer cette tâche de manière rapide et efficace. Le modèle peut ainsi apprendre à reconnaître ces entités spécifiques dans un large éventail de textes, en prenant en compte la variabilité des expressions et des formulations utilisées. Cela permet au système de mieux comprendre les demandes des utilisateurs et d'orienter les actions en conséquence.

4.4 Pipeline NLP

Dans ce projet, nous avons suivi un pipeline **NLP** complète pour entraîner notre modèle de reconnaissance d'entités nommées (NER). Chaque étape du processus a été soigneusement appliquée pour garantir une performance optimale du modèle. Voici une explication détaillée de chaque étape du pipeline, en soulignant l'importance de chaque phase et son rôle dans l'amélioration des résultats.

Voici le schéma ci-dessous qui illustre les différentes étapes de la pipeline NLP suivie dans ce projet, permettant de visualiser clairement le flux de travail appliqué :



4.4.1 Prétraitement

Le prétraitement est une phase essentielle qui consiste à préparer les données brutes pour qu'elles soient utilisables par le modèle. Cette étape inclut plusieurs sous-processus, comme la gestion des données manquantes ou corrompues, et l'adaptation des annotations aux exigences du modèle. Nous avons effectué un nettoyage de ces données pour garantir qu'elles étaient dans un format cohérent et approprié.

Un prétraitement de qualité assure que seules des données valides et bien structurées seront utilisées par le modèle, ce qui réduit le risque d'erreurs pendant l'apprentissage et améliore la qualité des prédictions.

4.4.2 Nettoyage du texte

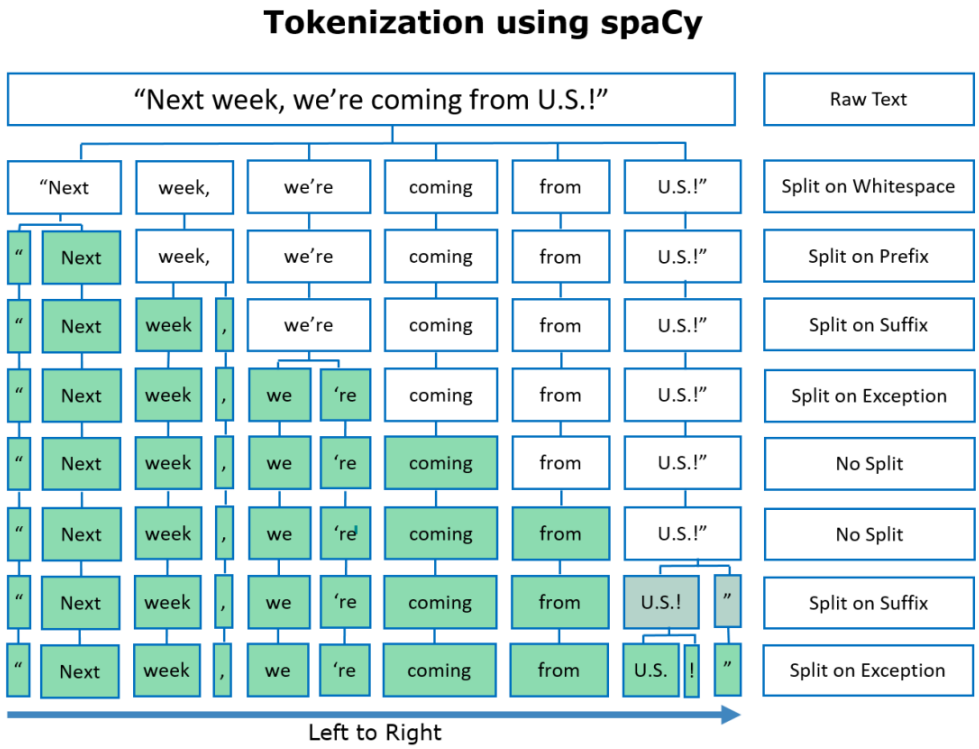
Une fois les données prétraitées, nous avons nettoyé les textes en éliminant toute information superflue ou erronée. Cette étape inclut la correction des erreurs de formatage ou la suppression de tokens mal alignés. Par exemple, des annotations mal positionnées ont été ajustées pour correspondre exactement au texte.

Le nettoyage du texte permet d'éviter que des données mal formatées n'introduisent du bruit dans l'entraînement, ce qui pourrait nuire à la précision du modèle. Cela garantit également que les entités sont bien détectées et alignées.

4.4.3 Segmentation et Tokenisation

La segmentation est le processus de découpage du texte en unités significatives, appelées "tokens" (mots ou groupes de mots). Cette étape est cruciale car elle permet de décomposer le texte brut en éléments individuels qui peuvent être analysés par le modèle. Une fois le texte segmenté, chaque tokens est attribué à une classe ou étiquette spécifique.

La tokenisation est la première étape de traitement d'un texte, essentielle pour la suite du processus. Sans cette décomposition en tokens, il serait difficile pour le modèle de comprendre les relations entre les différentes parties du texte.



4.4.4 Tokenisation et Normalisation

Une fois les phrases découpées en tokens, il est important de normaliser ces tokens, c'est-à-dire de les mettre sous une forme standardisée (par exemple, conversion en minuscules). La normalisation permet d'harmoniser les différentes variations d'un même mot (comme les majuscules, les accents ou les formes différentes).

Cette normalisation permet d'uniformiser les données textuelles, ce qui facilite l'analyse et l'apprentissage du modèle. Par exemple, "Paris" et "paris" seront considérés comme le même mot, ce qui améliore la cohérence du modèle.

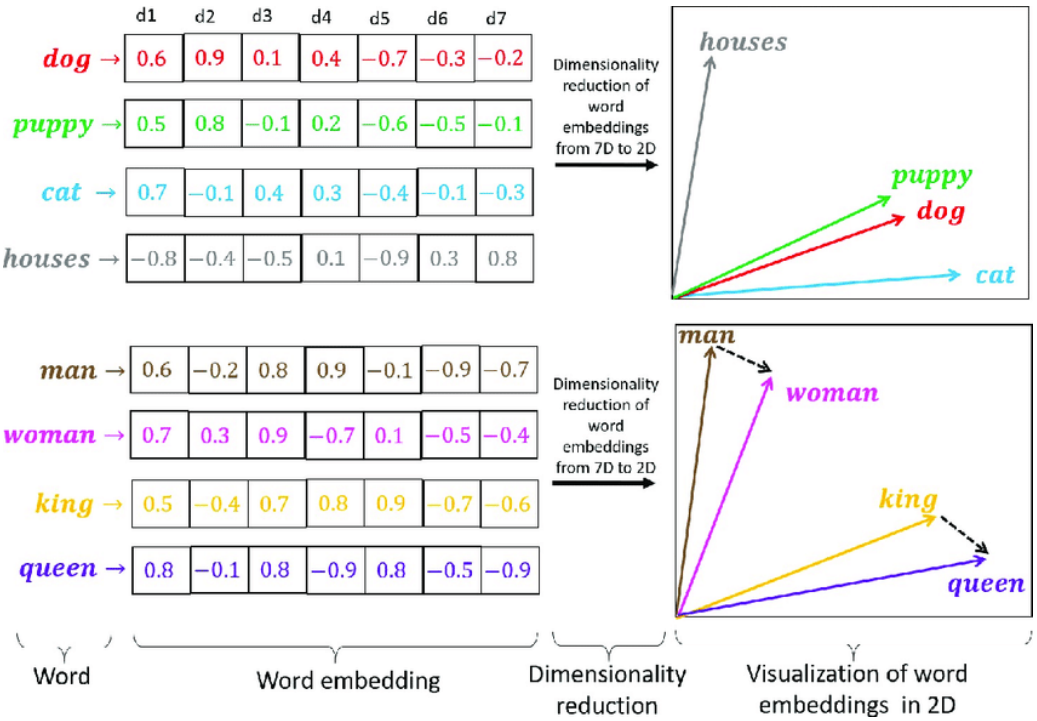
4.4.5 Lemmatisation et Stemming

La lemmatisation et le stemming réduisent les mots à leur racine ou forme de base, comme "marcher", "marché" et "marchait" ramenés à "marcher". La lemmatisation utilise le contexte pour trouver le lemme (forme canonique). Ces techniques regroupent les variantes d'un mot sous une même représentation, facilitant la compréhension des relations entre mots. Cela améliore ainsi la détection des entités et les analyses sémantiques.

4.4.6 Vectorisation et Extraction des caractéristiques

La vectorisation consiste à convertir les mots et phrases en représentations numériques appelées "embeddings" ou vecteurs. Chaque mot est transformé en un vecteur qui capture des informations sur sa signification et ses relations avec d'autres mots. Cette transformation est cruciale pour que le modèle puisse comprendre et traiter les textes de manière numérique.

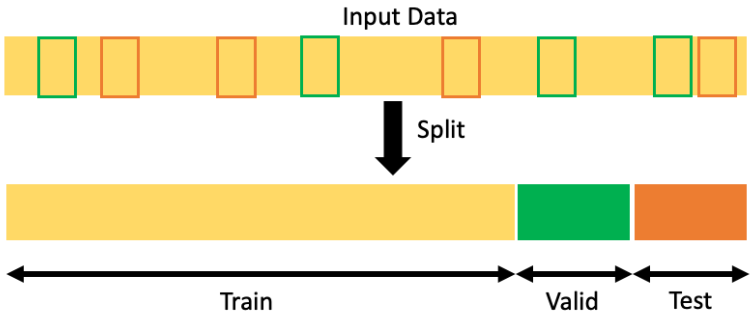
Les modèles de NLP nécessitent des données numériques pour pouvoir effectuer des calculs. La vectorisation permet de convertir le texte en un format numérique tout en préservant le sens sémantique des mots. Cela permet au modèle d'analyser efficacement les relations entre les entités et les mots.



4.4.7 Division des jeux de données

Une fois les données prêtes, il est important de les diviser en trois ensembles distincts : l'entraînement, la validation et le test. L'ensemble d'entraînement est utilisé pour entraîner le modèle, l'ensemble de validation permet d'ajuster les hyperparamètres et d'évaluer les performances pendant l'entraînement, et l'ensemble de test sert à mesurer la performance finale du modèle.

Cette division permet d'éviter le sur-apprentissage (overfitting) et de tester le modèle sur des données qu'il n'a pas vues auparavant. Cela permet de garantir que le modèle est capable de généraliser ses connaissances sur de nouvelles données.



4.4.8 Entraînement modèle

L'entraînement du modèle consiste à utiliser les données d'entraînement pour ajuster les paramètres du modèle et lui permettre de prédire correctement les entités. Pendant l'entraînement, le modèle apprend à détecter des motifs et des structures dans le texte, en utilisant les annotations des entités comme guide.

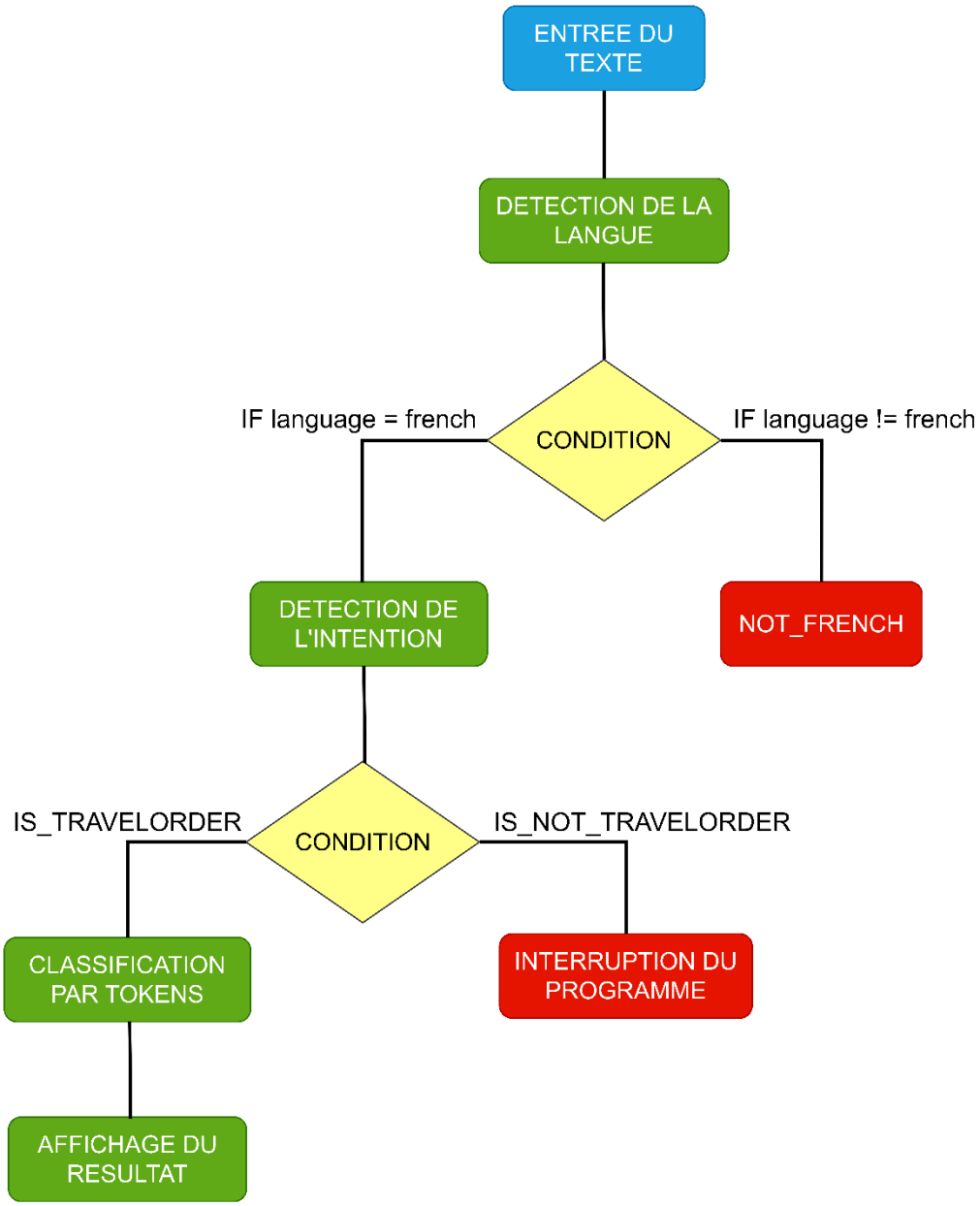
L'entraînement est essentiel pour que le modèle apprenne à effectuer les tâches spécifiques pour lesquelles il a été conçu, ici la détection d'entités comme "DEP" et "ARR". Un bon entraînement optimise les performances du modèle sur de nouvelles données.

4.4.9 Évaluations et résultats

Après l'entraînement, le modèle est évalué sur les jeux de données de validation et de test pour mesurer ses performances. Des métriques telles que la précision, le rappel et le score F1 sont utilisées pour évaluer la qualité des prédictions. Ces scores nous permettent de savoir si le modèle détecte correctement les entités.

L'évaluation permet de mesurer l'efficacité du modèle et d'identifier des domaines d'amélioration. Sans évaluation, il serait impossible de savoir si le modèle est prêt à être déployé sur de nouvelles données.

4.5 Ordre d'exécution et flux de traitement



Le traitement d'une demande de voyage dans notre système suit un **processus** bien défini pour s'assurer que la demande est bien comprise et qu'une réponse adéquate est donnée. Tout d'abord, le système vérifie la **langue** du texte. Si ce n'est pas du français, il arrête immédiatement le traitement et rejette la demande. Si le texte est bien en français, il continue l'analyse.

Ensuite, le système cherche à comprendre **l'intention de la demande**. Si l'intention n'est pas liée à une commande de voyage (par exemple, si la personne pose une question générale ou fait une autre demande), le processus s'arrête également. Cela permet d'éviter de traiter des demandes inutiles et de rendre le système plus efficace.

Si l'intention est bien une commande de voyage, le programme analyse chaque mot du texte pour repérer les informations importantes, comme les villes de départ et d'arrivée. Après cette analyse, le système affiche à l'utilisateur le résultat, par exemple l'itinéraire demandé, ou une autre réponse appropriée.

Ce processus garantit que la demande est traitée rapidement et de manière ciblée, en excluant dès le début les requêtes qui ne concernent pas un voyage.

4.6 Jeux de Données pour le NLP

4.6.1 Création du jeu de données pour l'entraînement du modèle NLP : Méthodologie et annotations

Objectif du Jeu de Données

Le principal objectif de ce jeu de données est de fournir une base d'exemples variés et bien structurés pour entraîner un modèle de traitement du langage naturel (NLP) dédié à deux tâches principales : la **classification de textes** et la **reconnaissance d'entités nommées (NER)**. La classification de textes permet de déterminer l'intention ou le type d'une phrase, tandis que la reconnaissance d'entités nommées permet d'extraire des informations spécifiques, comme les noms de lieux ou de personnes, dans une phrase. Ce jeu de données est nécessaire pour enseigner au modèle comment identifier ces entités et classer les phrases de manière appropriée.

Types de Phrases à Inclure

Le jeu de données inclut plusieurs types de phrases, organisées en catégories distinctes :

1. **Phrases valides** : Ce sont des phrases correctes en français, exprimant des demandes ou des informations pertinentes, comme des trajets ou des itinéraires entre des villes. Ces phrases servent à entraîner le modèle sur des cas d'utilisation réels.
2. **Phrases en langue étrangère** : Certaines phrases sont générées dans des langues autres que le français, pour entraîner le modèle à reconnaître ces cas et identifier que la phrase ne correspond pas au contexte attendu.
3. **Phrases hors contexte** : Ces phrases ne sont pas liées à des trajets ou à des demandes d'itinéraire, mais sont incluses pour aider le modèle à éviter les classifications erronées.

4. **Phrases incompréhensibles ou erronées** : Il s'agit de chaînes de caractères ou de phrases déformées, générées pour tester la capacité du modèle à détecter des erreurs et des incohérences dans le texte.

Méthode de Génération des Phrases Valides

Les **phrases valides** sont construites à l'aide de modèles ou de **templates**. Un template est une structure de phrase avec des parties variables, comme les noms de villes, qui sont ensuite remplacées par des données réelles pour créer des phrases uniques. Par exemple :

- Modèle : "Peux-tu m'aider à trouver un trajet de {départ} à {arrivée} ?"
- Phrase générée : "Peux-tu m'aider à trouver un trajet de PARIS à MARSEILLE ?"

Les variables comme **{départ}** et **{arrivée}** sont remplacées par des villes réelles issues d'une liste de **noms de villes** (ex : Paris, Lyon, Toulouse) afin de générer un large éventail de phrases.

Méthode de Génération des Phrases Invalides

Les **phrases invalides** sont générées pour tester la robustesse du modèle. Ces phrases sont créées pour correspondre à des labels spécifiques selon les tâches d'entraînement :

1. **Dataset de langue (is_not_french)** : Ce label est utilisé pour les phrases en langue étrangère. Par exemple, une phrase en anglais pourrait être étiquetée comme **is_not_french** pour signaler que ce n'est pas du français.
    - Exemple : "I want to travel to Paris." → **is_not_french: 1**
2. **Dataset d'intention (is_correct, is_not_trip, is_unknown)** :
    - **is_correct** : Cette étiquette indique que la phrase est correcte et correspond à une demande de trajet ou d'information pertinente.
    - **is_not_trip** : Cette étiquette est utilisée pour les phrases qui ne concernent pas un trajet (par exemple, des phrases non liées aux voyages).
    - **is_unknown** : Cette étiquette est attribuée aux phrases qui sont difficiles à classer ou à interpréter.
    - Exemple : "He lied when he said he didn't like her." → **is_not_trip: 1**

Annotation et Structuration des Données

Chaque phrase dans le jeu de données est soigneusement **étiquetée** avec les **labels** appropriés pour l'entraînement du modèle. Par exemple :

- Pour le **dataset de langue**, une phrase qui ne correspond pas à du français recevra le label **is_not_french**.
- Pour le **dataset d'intention**, une phrase pertinente aura un label comme **is_correct** ou **is_not_trip** en fonction de son contenu.

Les entités nommées (par exemple, des noms de villes ou de lieux) sont extraites à l'aide de la tâche **NER** (Reconnaissance d'Entités Nommées). Ces entités sont annotées en utilisant des **tags** comme **DEP** (départ) et **ARR** (arrivée) pour les villes de départ et d'arrivée dans les phrases. Un exemple d'annotation NER pour une phrase comme "Je veux aller de PARIS à LYON" serait :

- **Tokens** : ['Je', 'veux', 'aller', 'de', 'PARIS', 'à', 'LYON']
- **NER Tags** : [{'start': 17, 'end': 22, 'label': 'DEP'}, {'start': 25, 'end': 29, 'label': 'ARR'}]

Les entités **PARIS** et **LYON** sont marquées respectivement avec les labels **DEP** (départ) et **ARR** (arrivée).

Exemples de Phrases Générées et Annotations Associées

1. **Phrase valide (demande d'itinéraire)** :
    - Phrase : "As-tu un itinéraire précis de REIMS à VANDŒUVRE-LÈS-NANCY ?"
    - Tokens : ['As', '-', 'tu', 'un', 'itinéraire', 'précis', 'de', 'REIMS', 'à', 'VANDŒUVRE-LÈS-NANCY', '?']
    - NER Tags : [{'start': 30, 'end': 35, 'label': 'DEP'}, {'start': 38, 'end': 57, 'label': 'ARR'}]
2. **Phrase invalide (hors contexte)** :
    - Phrase : "He lied when he said he didn't like her."
    - Intention : **is_not_trip: 1**
3. **Phrase en langue étrangère** :
    - Phrase : "I would like to travel from culoz to buswiller eglise."
    - Langue : **is_not_french: 1**
4. **Phrase incorrecte (incompréhensible)** :
    - Phrase : "?N|ajOLIY6;DOM'mKavLZZnkAi"
    - Intentions : **is_unknown: 1**

Cette méthodologie permet de créer un jeu de données riche et diversifié, adapté à l'entraînement de modèles NLP pour la classification de textes et la reconnaissance d'entités nommées. En combinant des phrases valides, des phrases invalides, et des annotations soigneusement réalisées, nous fournissons aux modèles des exemples représentatifs des scénarios réels, ainsi que des cas plus complexes pour tester leur capacité à identifier des erreurs et des incohérences.

4.6.2 Étiquetage des données pour la classification de texte et la reconnaissance d'entités nommées (NER)

Présentation des ensembles de données

Les données générées dans le cadre de ce projet sont divisées en trois ensembles distincts, chacun visant à entraîner des aspects spécifiques du modèle NLP :

- **Ensemble pour la classification de langue** : Cet ensemble permet de déterminer si une phrase est en français ou dans une autre langue. Les phrases sont étiquetées pour distinguer celles qui sont en français de celles qui ne le sont pas.
- **Ensemble pour la classification d'intention** : Cet ensemble classifie les phrases selon leur intention, telles que des demandes d'itinéraires, de tarifs ou d'informations spécifiques. Les labels dans cet ensemble aident à identifier le type d'intention derrière chaque phrase.
- **Ensemble pour la reconnaissance d'entités nommées (NER)** : Cet ensemble permet de détecter et d'annoter des entités spécifiques dans les phrases, comme les lieux (villes, régions, etc.), afin d'extraire des informations pertinentes.

Détails sur l'étiquetage pour la classification de texte

Pour chaque ensemble de données, des étiquettes (labels) sont attribuées en fonction du type de phrase ou du contexte. Ces labels sont essentiels pour la classification des phrases par le modèle. Voici un aperçu des labels utilisés dans chaque ensemble :

Dataset	Label(s) Possible(s)	Description
Langue	is_not_french	Label indiquant si la phrase est en français (0) ou non (1)
Intention	is_correct, is_not_trip, is_unknown	is_correct : phrase valide ; is_not_trip : hors contexte ; is_unknown : intention inconnue
NER (Reconnaissance d'entités nommées)	B-DEP, I-DEP, B-ARR, I-ARR	Labels indiquant les entités "départ" (B-DEP, I-DEP) et "arrivée" (B-ARR, I-ARR)

Détails sur l'annotation des entités nommées (NER)

L'annotation des entités nommées (NER) consiste à identifier et marquer les entités spécifiques, telles que les villes ou les régions, dans les phrases. Ces entités sont importantes pour l'entraînement du modèle afin qu'il puisse reconnaître les informations géographiques dans des contextes variés.

- **Approche BERT, SpaCy et Conditional Random Fields (CRF)**

Approche	Description	Annotations
BERT	Chaque mot est découpé en tokens, et les labels sont prédits pour chaque token.	"REIMS" -> B-DEP, "VANDŒUVRE-LÈS-NANCY" -> B-ARR
SpaCy	Utilise les spans de texte pour annoter les entités avec les indices de début et de fin.	"REIMS" -> DEP, "VANDŒUVRE-LÈS-NANCY" -> ARR
Conditional Random Fields (CRF)	L'annotation est basée sur les features de chaque token, et les relations entre les tokens dans la séquence sont analysées pour prédire les entités.	"REIMS" -> B-DEP, "à" -> O, "VANDŒUVRE-LÈS-NANCY" -> B-ARR

Tokenization et Attribution des Labels

La tokenisation consiste à diviser les phrases en unités plus petites (tokens). Ces tokens sont ensuite associés à des labels NER. Voici un exemple détaillé de la manière dont les labels sont attribués aux entités dans différents ensembles de données.

Exemple de phrase :
"As-tu un itinéraire précis de REIMS à VANDŒUVRE-LÈS-NANCY ?"

- **Tokenisation** :
  Tokens : ['As', '-', 'tu', 'un', 'itinéraire', 'précis', 'de', 'REIMS', 'à', 'VANDŒUVRE-LÈS-NANCY', '?']
- **Attribution des labels** :
  - REIMS : B-DEP (début de l'entité "départ")
  - VANDŒUVRE-LÈS-NANCY : B-ARR (début de l'entité "arrivée")

Exemple NER en SpaCy :
La phrase est analysée, et chaque entité est identifiée par ses indices de début et de fin.

Label	As	-	tu	un	itinéraire	précis	de	Reims	à	Paris	?
Token	O	O	O	O	O	O	O	B-DEP	O	B-ARR	O

Tableaux des ensembles générés

Ensemble Langue :

Sentence	is_not_french
Y a-t-il un moyen d'aller de Montreux-Vieux à la gare de La Roche-Sur-Foron ?	0
I would like to travel from culoz to buswiller eglise.	1
He lied when he said he didn't like her.	1

Ensemble Intention :

Sentence	is_correct	is_not_trip	is_unkown
Y a-t-il un moyen d'aller de Montreux-Vieux à la gare de La Roche-Sur-Foron ?	1	0	0
I would like to travel from culoz to buswiller eglise.	0	0	0
He lied when he said he didn't like her.	0	1	0

Ensemble NER (Reconnaissance d'Entités Nommées) :

Text	Tokens	ner_tags	spacy_ner_tags
As-tu un itinéraire précis de REIMS à VANDŒUVRE-LÈS-NANCY ?	['As', '-', 'tu', 'un', 'itinéraire', 'précis', 'de', 'REIMS', 'à', 'VANDŒUVRE-LÈS-NANCY', '?']	[0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 0]	[{'start': 30, 'end': 35, 'label': 'DEP'}, {'start': 38, 'end': 57, 'label': 'ARR'}]
Comment puis-je aller à PETIT-BOURG en venant de VILLEPINTE ?	['Comment', 'puis', '-je', 'aller', 'à', 'PETIT-BOURG', 'en', 'venant', 'de', 'VILLEPINTE', '?']	[0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0]	[{'start': 24, 'end': 35, 'label': 'ARR'}, {'start': 49, 'end': 59, 'label': 'DEP'}]

L'étiquetage des données dans le cadre de ce projet vise à fournir des ensembles cohérents pour l'entraînement du modèle NLP. Chaque phrase est soigneusement annotée en fonction de son type (langue, intention, ou entité nommée), ce qui permet au modèle d'apprendre les relations contextuelles et géographiques. Les méthodologies de tokenisation et d'annotation des entités utilisées (BERT, SpaCy, CRF) garantissent une extraction précise des informations pertinentes, facilitant ainsi le développement d'un modèle robuste pour la classification de textes et la reconnaissance d'entités nommées.

4.6.3 Problématiques rencontrées

Dans ce projet, plusieurs défis ont été identifiés et traités lors de la préparation des jeux de données destinés à entraîner des modèles de classification de texte et de reconnaissance d'entités nommées (NER). Ci-dessous, nous présentons une analyse détaillée des principales problématiques rencontrées, ainsi que les solutions mises en œuvre pour y remédier, le tout expliqué dans un langage clair et accessible.

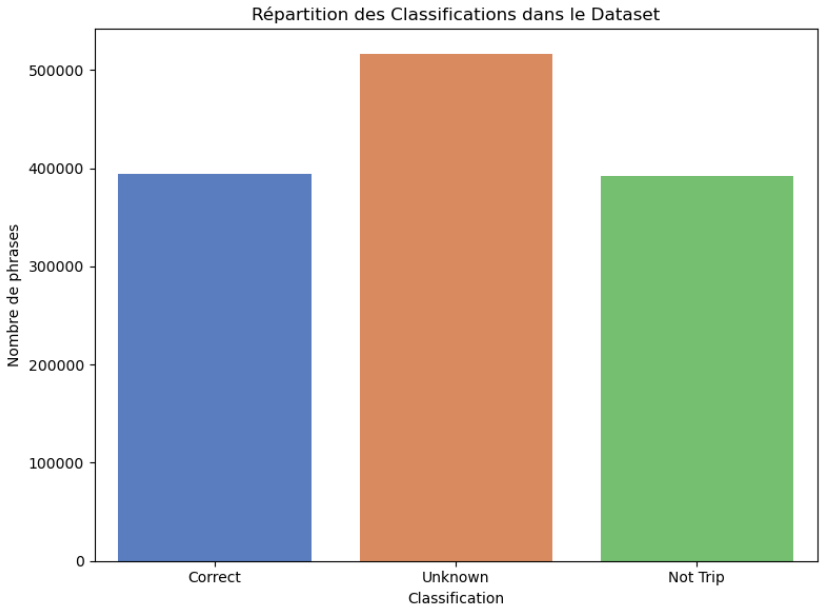
4.6.3.1 Duplications de phrases et déséquilibre des classes

Problématique

Lors de la génération automatique de phrases à partir de templates, des doublons peuvent apparaître. Ces duplications sont dues à des erreurs dans la génération ou à un nombre limité de modèles utilisés pour créer la diversité attendue. Par ailleurs, le nombre de phrases valides, de phrases en langue étrangère, de phrases hors contexte et de phrases erronées n'est pas toujours équilibré, ce qui peut biaiser l'apprentissage du modèle.

Solution

- **Suppression des doublons :** Après la génération, la méthode *drop_duplicates* est utilisée pour éliminer les phrases en double, garantissant ainsi que chaque phrase est unique.
- **Échantillonnage aléatoire :** L'utilisation de *random.sample* et *np.random.choice* permet de sélectionner un nombre défini de phrases pour chaque combinaison, ce qui aide à limiter le déséquilibre.
- **Génération par "chunks" :** En générant les données en petits lots et en fusionnant plusieurs DataFrames, on obtient une diversité suffisante tout en conservant un bon équilibre entre les différentes classes.



4.6.3.2 Problème du point final dans l'étiquetage NER

Problématique

Dans l'annotation NER, les signes de ponctuation, notamment les points finaux, sont souvent collés aux mots. Cela peut entraîner des erreurs lors de la tokenisation, car le mot associé à une ponctuation peut être mal segmenté et ainsi recevoir un mauvais label.

Solution

- **Séparation de la ponctuation :** Une fonction de tokenisation a été implémentée qui vérifie si le dernier caractère d'un token est une ponctuation (comme un point, point d'exclamation ou d'interrogation). Si c'est le cas, le mot est séparé de la ponctuation en deux tokens distincts. Cette approche permet d'appliquer correctement les étiquettes NER aux mots et d'éviter les erreurs liées à la fusion de la ponctuation avec les entités.

4.6.3.3 Gestion des formulations prépositionnelles

Problématique

En français, la préposition "de" se change en "d'" lorsque le mot qui suit commence par une voyelle ou un accent. Si cette règle n'est pas appliquée correctement, cela peut entraîner des incohérences dans les phrases générées, affectant ainsi la qualité des données d'entraînement.

Solution

- **Utilisation d'expressions régulières :** Lors du nettoyage des phrases, une expression régulière est utilisée pour détecter quand un mot débute par une voyelle (ou une voyelle accentuée) et pour remplacer "de" par "d'" en conséquence. Cette correction grammaticale assure que les noms de villes sont présentés de manière cohérente et facilite la reconnaissance des entités par le modèle.

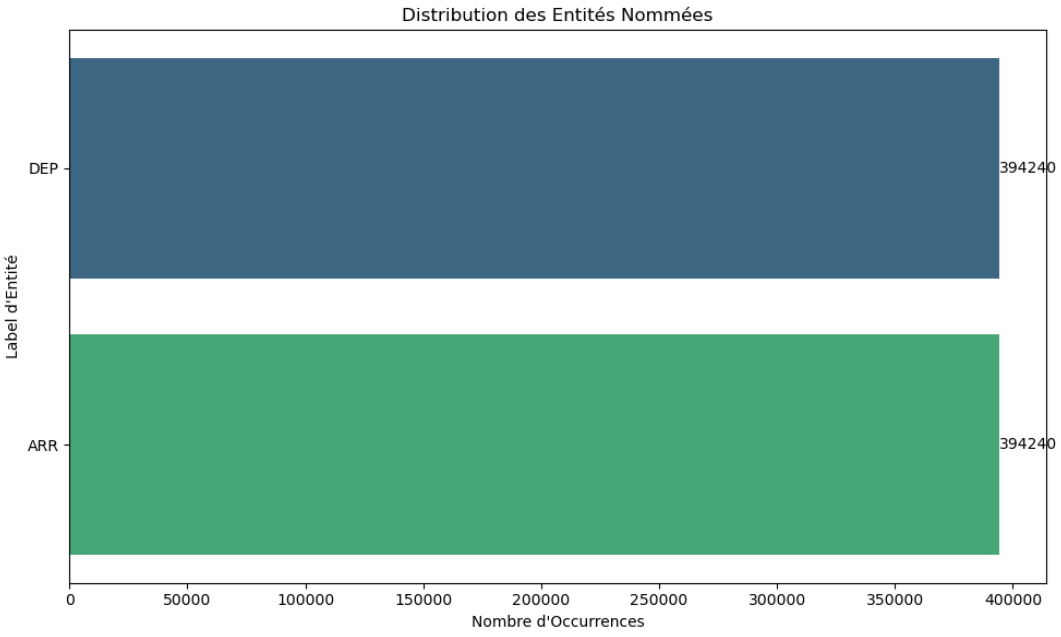
4.6.3.4 Erreurs dans les templates inversant les entités DEP et ARR

Problématique

Certains templates de phrases peuvent être mal conçus et inverser l'ordre des entités, c'est-à-dire que le lieu de départ (marqué par DEP) et le lieu d'arrivée (marqué par ARR) sont interchangés. Cela induit une confusion lors de l'entraînement, car le modèle apprend des associations erronées.

Solution

- **Règles explicites de taggage :** Le code attribue des étiquettes en comparant chaque token avec les mots composant les noms de villes. Si le token correspond à une partie du nom du lieu de départ, il reçoit le label DEP, sinon, s'il correspond au lieu d'arrivée, il reçoit le label ARR.
- **Vérification et validation :** Des mécanismes de validation sont intégrés pour corriger automatiquement certaines inversions, garantissant ainsi une attribution cohérente des étiquettes.



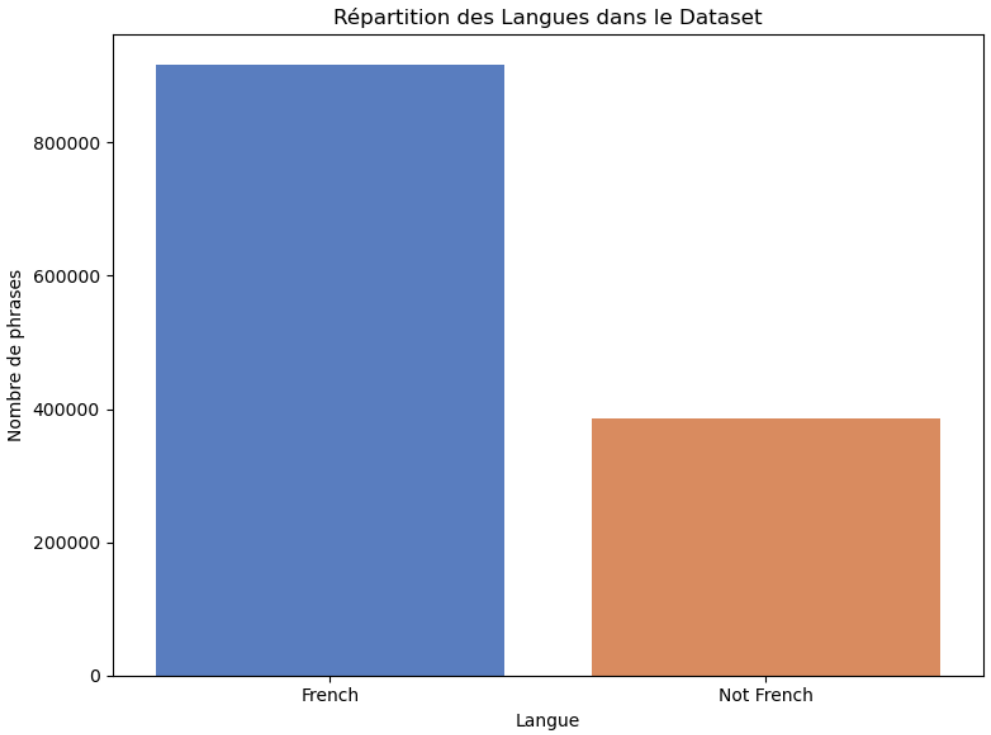
4.6.3.5 Équilibre des jeux de données en classification et reconnaissance d'entités

Problématique

Pour obtenir un modèle performant, il est crucial que chaque classe (phrases valides, phrases en langue étrangère, phrases hors contexte, phrases erronées) soit représentée de manière équilibrée dans le jeu de données. Un déséquilibre peut biaiser l'apprentissage et limiter la capacité du modèle à généraliser.

Solution

- **Analyse statistique et graphiques** : Des analyses statistiques et graphiques ont été réalisées pour visualiser la répartition des classes et identifier d'éventuels déséquilibres.
- **Techniques d'échantillonnage :** Des méthodes de sur-échantillonnage et de sous-échantillonnage ont été appliquées lors de la génération pour assurer une représentation équilibrée de chaque catégorie.
- **Fusion de DataFrames avec suppression des doublons :** En générant plusieurs lots de données puis en les fusionnant après suppression des doublons, un jeu de données global équilibré a été obtenu.



4.6.3.6 Impact du fine-tuning excessif et recommandations

Problématique

Un entraînement trop poussé (fine-tuning excessif) sur un dataset limité ou déséquilibré peut conduire à de l'overfitting. Le modèle apprend alors trop précisément sur les exemples d'entraînement, ce qui réduit sa capacité à généraliser sur des données nouvelles.

Solution et recommandations :

- **Ajustement du nombre d'échantillons et d'époques :** Le code prévoit la génération de données par lots (chunks) et fixe un nombre défini d'échantillons pour chaque classe, ce qui permet d'éviter un biais et de limiter le risque de surapprentissage.
- **Techniques de régularisation :** Suivant les recommandations de la communauté Hugging Face, il est conseillé d'utiliser des techniques telles que le dropout, l'early stopping et la validation croisée pour surveiller la performance du modèle et prévenir l'overfitting.
- **Suivi de la répartition des classes :** L'utilisation de graphiques de distribution permet de vérifier que chaque classe est bien représentée, et d'ajuster le nombre d'époques d'entraînement en fonction de la convergence observée sur un ensemble de validation.

Le traitement des jeux de données pour l'entraînement de modèles NLP comporte plusieurs défis, allant des duplications de phrases et du déséquilibre des classes à la gestion fine de la ponctuation, des prépositions et des erreurs dans les templates d'entités. Grâce à une série de techniques de nettoyage, d'échantillonnage et de vérification, ces problèmes ont été efficacement atténués. Les graphiques de répartition (des entités, des classifications et des langues) ont permis de surveiller l'équilibre des données, tandis que des ajustements dans le fine-tuning (nombre d'échantillons et d'époques) et l'application de techniques de régularisation, conformément aux recommandations de Hugging Face, ont aidé à prévenir l'overfitting. Ces efforts combinés ont permis de construire un dataset robuste et équilibré, améliorant ainsi significativement la performance et la généralisation des modèles NLP.

5 . Expérimentations et Résultats

Dans cette partie, nous explorons les différentes techniques de modélisation utilisées dans le traitement du langage naturel (NLP), en mettant l'accent sur la classification de texte et l'identification d'entités nommées (NER). Ces deux tâches sont essentielles dans notre projet, car elles permettent d'analyser et d'extraire les informations clés des ordres de voyage reçus sous forme textuelle.

Pour **l'identification d'entités nommées**, plusieurs modèles ont été utilisés afin de repérer efficacement les noms de villes, départs et destinations. Camembert et DistilBERT, qui s'appuient sur des architectures de type transformer, offrent une capacité d'apprentissage contextuel approfondie, leur permettant de reconnaître les entités malgré les variations de syntaxe ou les erreurs typographiques. SpaCy, avec son pipeline NLP optimisé, permet un traitement rapide et

efficace, tandis que les Conditional Random Fields (CRF), bien que plus anciens, restent pertinents pour la segmentation et l'étiquetage de séquences textuelles.

En ce qui concerne la **classification de texte**, la détection de langue repose sur des méthodes variées, allant de l'approche statistique avec Naive Bayes aux réseaux neuronaux comme les CNN et LSTM, qui capturent les relations complexes entre les mots. Pour la détection d'intention, différentes techniques ont été exploitées, notamment les modèles Bag of Words (bigram, unigram, SLA), qui analysent la fréquence des termes, ainsi que des réseaux de neurones plus avancés tels que GPT, GRU et SAN. Ces derniers permettent une meilleure compréhension du contexte et des nuances des phrases, améliorant ainsi la précision de l'interprétation des commandes de voyage.

L'évaluation des performances de ces modèles a permis de choisir les plus adaptés aux contraintes de notre projet, en tenant compte de l'équilibre entre précision, temps de traitement et consommation de ressources.

5.1 Présentation générale des tests et de l'approche comparative

L'expérience menée dans ce projet avait pour but de tester différentes méthodes de traitement automatique du langage (NLP) pour comprendre et traiter des demandes de voyage. Nous nous sommes concentrés sur trois points principaux : détecter la langue utilisée, déterminer si la phrase est bien une demande de voyage, et extraire les informations importantes comme les villes de départ et d'arrivée.

Pour comparer les méthodes, nous avons évalué plusieurs modèles de classification et de reconnaissance d'informations. Comme dit précédemment dans le rapport, pour en être sûr que nos tests étaient solides, nous avons créé un ensemble de données varié, incluant des phrases correctes, des fautes de frappe, et des variations dans la façon de s'exprimer. Chaque modèle a été entraîné et testé sur ces données, en utilisant des mesures classiques comme la précision, le rappel et le F1-score.

D'abord, nous avons testé plusieurs algorithmes pour détecter la langue, en utilisant des méthodes simples comme Naïve Bayes et des techniques plus avancées comme les réseaux de neurones (CNN et LSTM). Ensuite, pour déterminer si une phrase était une demande de voyage, nous avons comparé des méthodes traditionnelles (comme Bag of Words et SVM) avec des modèles plus récents comme GPT et SAN.

Enfin, pour repérer les villes de départ et d'arrivée, nous avons essayé des modèles séquentiels comme CRF et des outils modernes de NLP comme SpaCy et CamemBERT. Nous avons particulièrement regardé la rapidité et la précision des modèles pour trouver les solutions les plus adaptées à notre problème.

5.2 Modèles de Classification de Texte

5.2.1 Détection de la Langue

5.2.1.1 Naïves Bayes

Naïve Bayes est un algorithme de classification basé sur le **théorème de Bayes** et utilisé dans de nombreux problèmes de NLP, notamment la détection de langue. Il suppose que les caractéristiques (mots dans un texte) sont indépendantes les unes des autres, ce qui simplifie grandement les calculs tout en offrant de très bonnes performances en pratique.

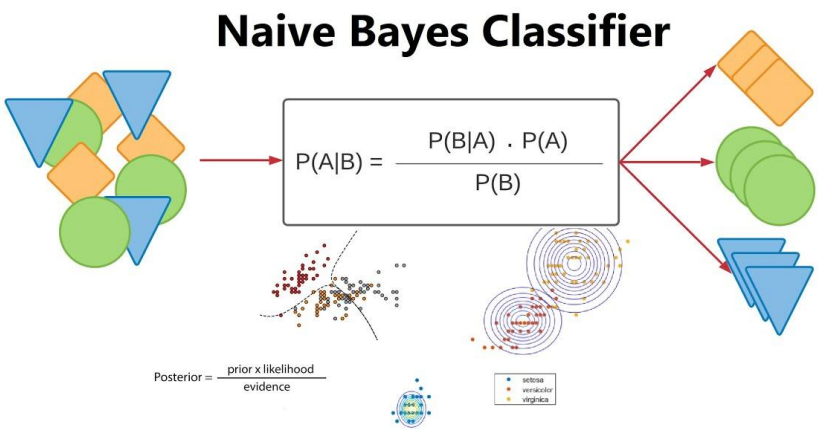
Le **théorème de Bayes** s'écrit :

$$P(C \mid X) = \frac{P(X \mid C)P(C)}{P(X)}$$

Où,

- P (C | X) est la probabilité qu'un texte appartienne à la classe C (ex : "French" ou "Not French") en connaissant X, la phrase donnée.
- P (X | C) est la probabilité d'obtenir X si la classe est C.
- P (C) est la probabilité a priori d'avoir la classe C (ex : fréquence des phrases françaises dans l'ensemble de données).
- P(X) est la probabilité totale d'obtenir X (toutes classes confondues).

En NLP, on utilise une version appelée Naïve Bayes multinomial, où $P\ (\ X \mid C\ )$ P(X|C) est estimée à partir des fréquences des mots dans chaque classe.



Prétraitement et Vectorisation

Avant d'entraîner le modèle, les phrases sont transformées en **matrices numériques** à l'aide de la méthode **TF-IDF (Term Frequency - Inverse Document Frequency)**. Cette technique pondère chaque mot en fonction de son importance dans un document tout en réduisant l'impact des mots trop courants. La formule du TF-IDF est :

$$TFIDF(w) = TF(w) \times IDF(w)$$

Où,

- TF (W) est la fréquence du mot tv dans une phrase.
- IDF (w) = log(N/DF(w)), avec N le nombre total de documents et DF(w) le nombre de documents contenant w.

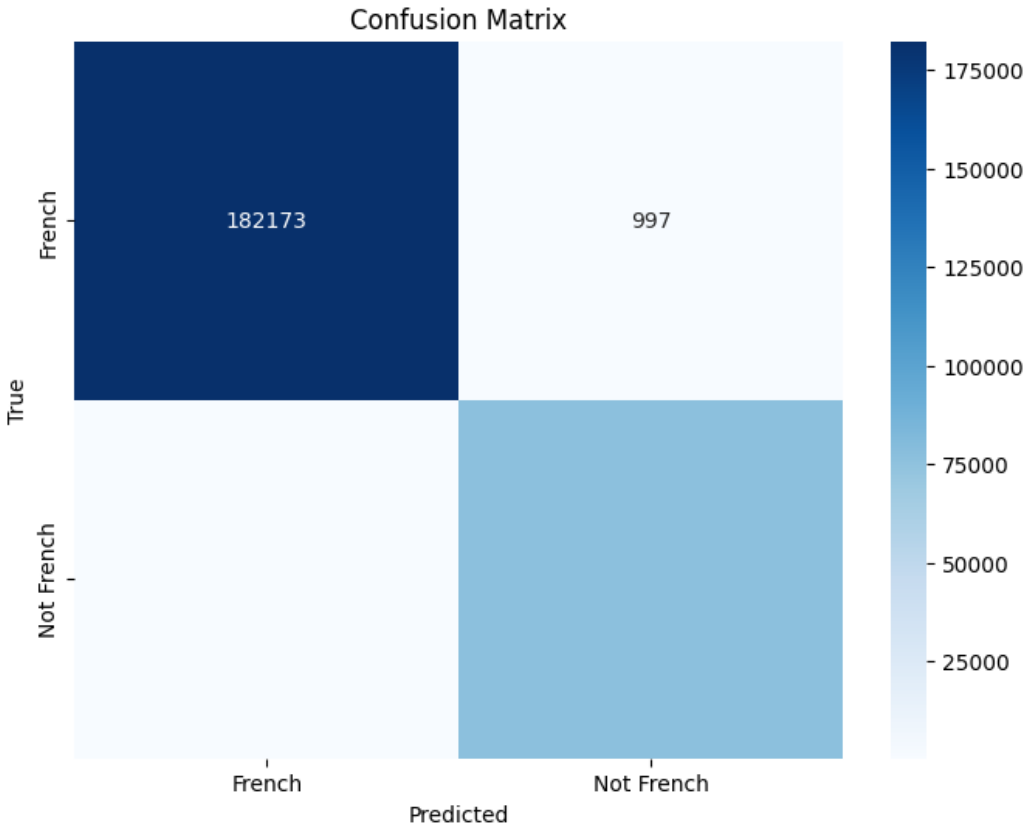
Entraînement et Évaluation

Le modèle a été entraîné sur **1 041 396 phrases** et testé sur un ensemble de **260 349 phrases**. Les résultats obtenus montrent une très bonne performance :

- **Précision globale (accuracy) : 99.45%**
- **F1-score : 0.9907**
- **AUC (Aire sous la courbe ROC) : 0.9944**

	precision	recall	f1-score	support
0	1.00	0.99	1.00	183170
1	0.99	0.99	0.99	77179
accuracy			0.99	260349
macro avg	0.99	0.99	0.99	260349
weighted avg	0.99	0.99	0.99	260349

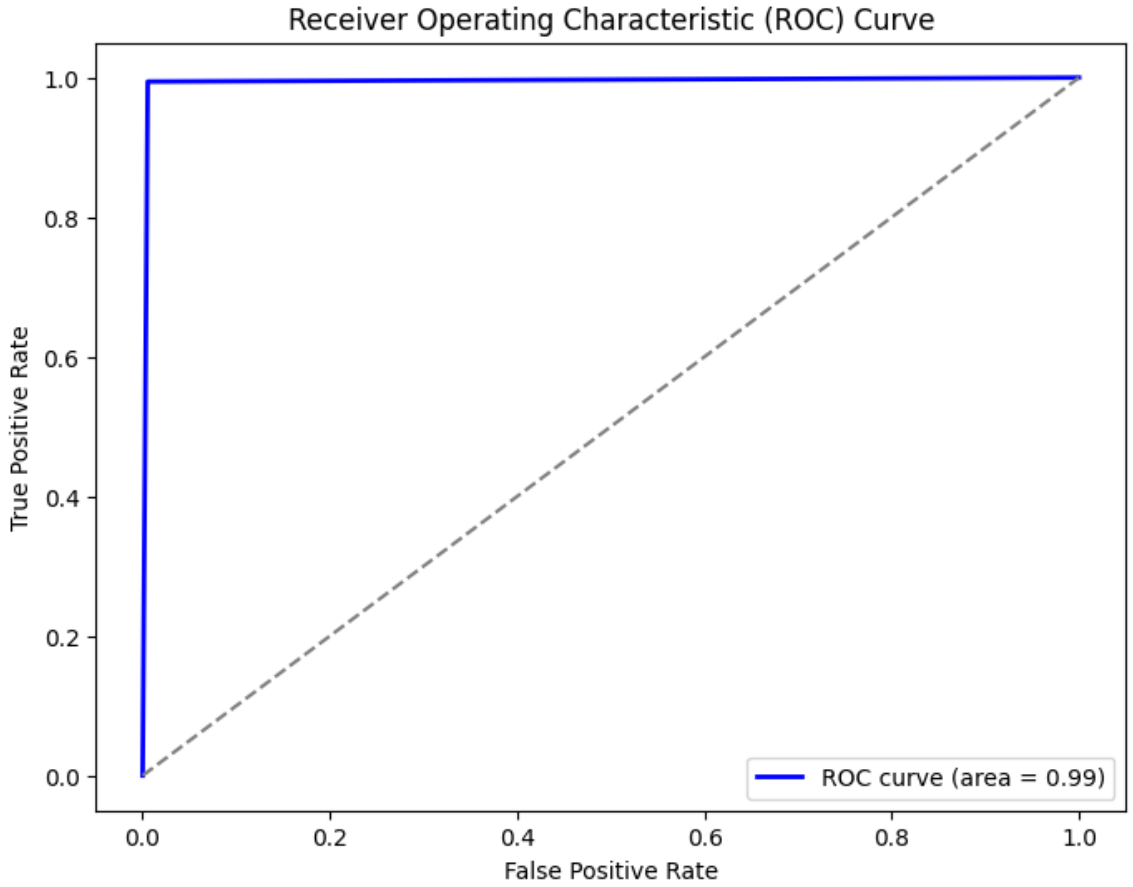
La **matrice de confusion** permet de visualiser les performances du modèle :



	Prédit : French	**Prédit : Not French**
Vrai : French	182 173	997
Vrai : Not French	437	76 742

- Le modèle identifie **correctement 182 173 phrases françaises**, avec seulement **997 erreurs** (faux négatifs).
- Il détecte **76 742 phrases non françaises**, avec **437 erreurs** (faux positifs).
- Le **taux d'erreur est donc très faible**.

La **courbe ROC** (Receiver Operating Characteristic) montre une très bonne séparation entre les classes, avec une **AUC proche de 1** (0.9944), ce qui indique que le modèle discrimine très bien entre les phrases en français et celles dans d'autres langues.



Le modèle Naïve Bayes combiné avec TF-IDF s'avère être une solution efficace pour la détection de la langue dans ce projet. Il offre une grande rapidité d'entraînement et d'exécution, avec une précision élevée même sur de grands ensembles de données, tout en étant facile à interpréter et à optimiser. Toutefois, ce modèle repose sur l'hypothèse d'indépendance des mots, ce qui ne correspond pas totalement à la réalité du langage naturel, et il peut être sensible aux mots rares ou aux fautes d'orthographe. Pour aller plus loin, il serait intéressant de comparer ses performances avec celles de modèles plus avancés, comme les LSTM ou les Transformers.

5.2.1.2 CNN (Convolutional Neural Networks)

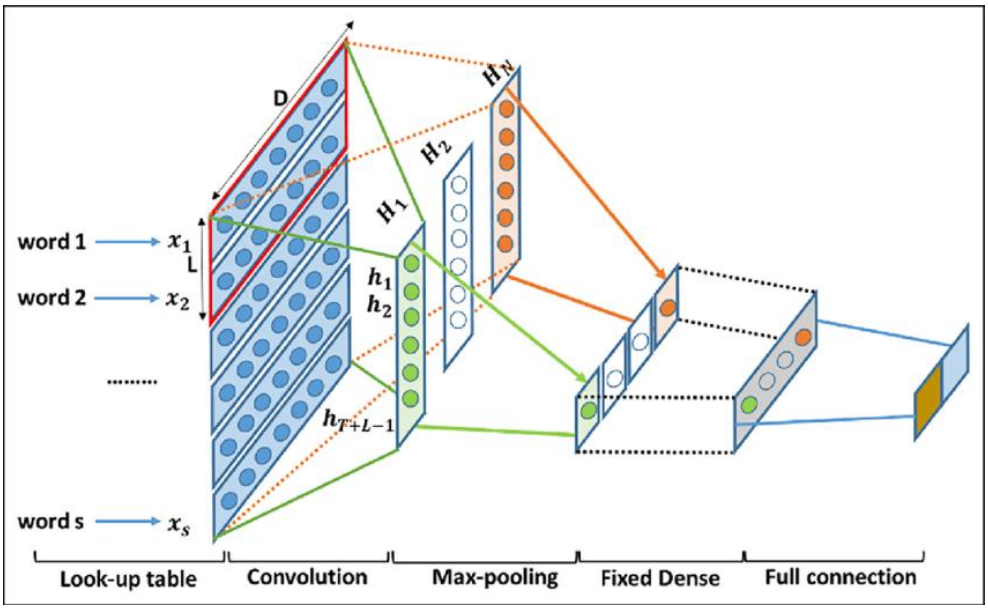
Les **Réseaux de Neurones Convolutionnels** (CNN) sont des modèles d'apprentissage profond largement utilisés pour traiter des données structurées en forme de grille, comme les images. Mais ils peuvent aussi être très efficaces pour traiter du texte. L'idée derrière un CNN est qu'il est capable de **repérer des motifs ou des structures locales** dans les données, ce qui est particulièrement utile pour identifier des caractéristiques spécifiques dans des phrases ou des mots qui peuvent aider à distinguer différentes langues.

Les CNN sont composés de différentes couches qui jouent un rôle spécifique :

1. **Couches Convolutives** : Ces couches utilisent des petits filtres ou noyaux pour parcourir le texte (ou une image) et extraire des motifs locaux. Par exemple, un filtre pourrait détecter une séquence de mots ou un n-gramme spécifique qui est typique dans une langue donnée.
2. **Couches de Pooling** : Une fois qu'un filtre a détecté un motif dans le texte, la couche de **pooling** simplifie et réduit la taille des données extraites, tout en conservant les informations

essentielles. C'est comme si on prenait la moyenne ou le maximum d'une petite fenêtre de données.

3. **Couches Complètement Connectées (Fully Connected)** : Ces couches utilisent les informations extraites par les couches précédentes pour prendre une décision finale, comme déterminer si le texte est en français ou non.



Prétraitement des Données et Vectorisation

Pour entraîner un modèle CNN à comprendre du texte, il faut d'abord convertir les phrases en **données numériques** que le modèle puisse traiter. Voici les étapes :

- **Tokenisation** : Le texte est découpé en mots (ou tokens). Par exemple, la phrase "Je veux partir de Marseille à Paris" devient une liste de mots : ["Je", "veux", "partir", "de", "Marseille", "à", "Paris"].
- **Encodage des mots** : Chaque mot est ensuite converti en un identifiant numérique unique à l'aide d'un **vocabulaire**. Cela permet de transformer les mots en nombres, ce qui est plus facile à manipuler pour un modèle de machine learning.
- **Padding** : Comme toutes les phrases n'ont pas la même longueur, on doit ajouter des **zéros** à la fin des phrases plus courtes pour les amener à une longueur uniforme. Cela permet au modèle de traiter des entrées de taille constante.

Définition du Modèle CNN pour la Détection de Langue

Le modèle CNN que nous avons utilisé pour la détection de la langue est structuré de la manière suivante :

1. **Embedding Layer (Couche d'Embedding)** : Cette couche transforme les identifiants numériques des mots en **vecteurs d'embedding**, c'est-à-dire en représentations denses des mots dans un espace de caractéristiques. Chaque mot est ainsi représenté par un vecteur de taille 50 (par exemple), permettant au modèle de capturer les relations sémantiques entre les mots.

2. **Couches Convolutives (Conv1D)** : Les deux premières couches convolutives utilisent des **filtres de taille 5** pour extraire des motifs locaux dans le texte. Ces couches apprennent à détecter des n-grammes (groupes de mots) spécifiques dans le texte, comme des séquences communes en français.
3. **Couches de Pooling (MaxPool1D)** : Après chaque couche convolutive, la couche de **pooling** permet de **réduire la taille** des sorties des filtres. Concrètement, au lieu de garder toutes les informations de la fenêtre, on garde seulement la valeur maximale. Cela permet de simplifier le modèle tout en conservant les informations importantes. Par exemple, après chaque fenêtre de 2 mots, on garde seulement le plus grand de ces mots.
4. **Couches Complètement Connectées (Fully Connected Layers)** : Après avoir extrait des caractéristiques locales avec les couches convolutives et de pooling, on a un certain nombre de données que l'on **aplatit** (flatten) pour les envoyer dans des couches entièrement connectées. Ces couches prennent les caractéristiques extraites et les utilisent pour **classer** la phrase. La dernière couche utilise une **activation sigmoïde** pour donner une probabilité que la phrase soit en français ou non (not french).

Entraînement du Modèle

Le modèle est entraîné sur **10 époques** avec une fonction de perte **BCELoss** (Binary Cross-Entropy Loss), adaptée aux problèmes de classification binaire (ici, "Français" vs "Non-Français"). L'optimiseur **Adam** ajuste les poids du modèle à chaque époque pour minimiser la perte.

Voici un exemple de l'évolution de la perte au fil des époques :

- **Epoch 1** : Perte = 0.0106
- **Epoch 2** : Perte = 0.0028
- **Epoch 10** : Perte = 0.0045

Cette diminution de la perte indique que le modèle apprend progressivement à mieux prédire si une phrase est en français ou non.

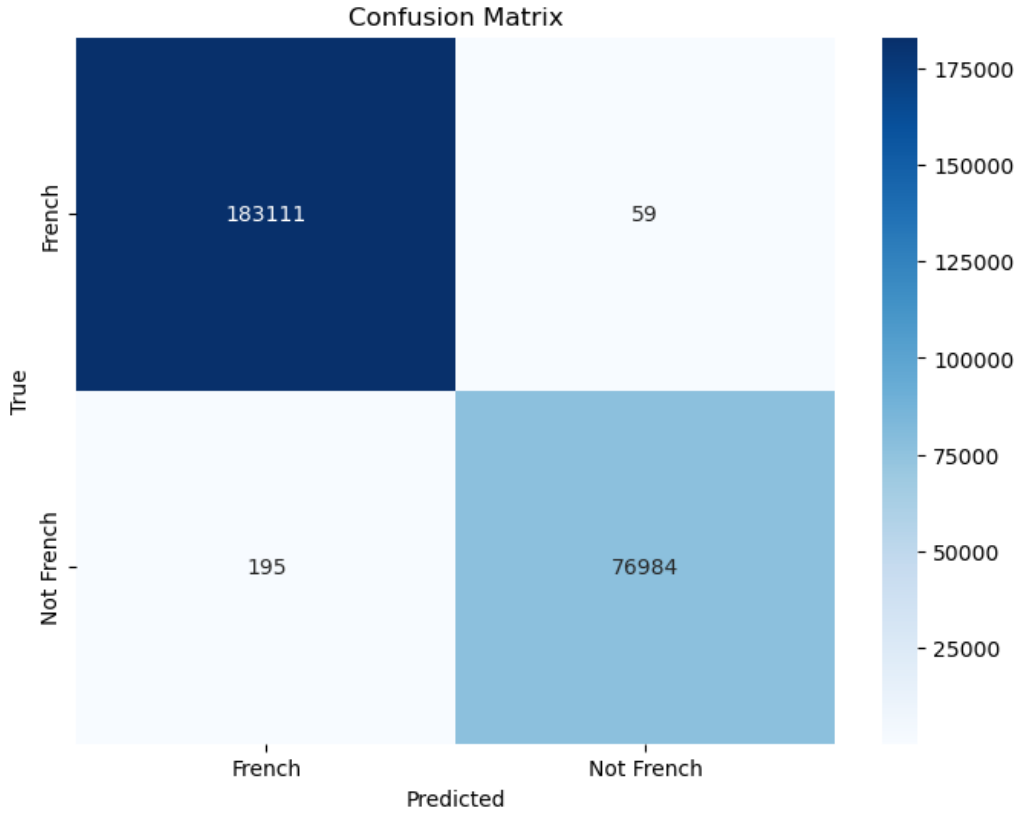
Évaluation du Modèle

Une fois l'entraînement terminé, nous avons évalué le modèle sur un ensemble de test comprenant **260 349 phrases**. Les résultats sont impressionnants, avec une **précision de 99.90%**. Voici quelques points clés du **rapport de classification** :

- **Précision** et **rappel** pour les deux classes (français et non-français) sont égaux à **1.00**, ce qui signifie que le modèle identifie parfaitement les phrases françaises et non françaises.

```
                precision    recall  f1-score   support

           0.0       1.00      1.00      1.00    183170
           1.0       1.00      1.00      1.00     77179

      accuracy                           1.00    260349
     macro avg       1.00      1.00      1.00    260349
  weighted avg       1.00      1.00      1.00    260349

### Accuracy ###
Accuracy: 0.9990
```

La **matrice de confusion** montre que le modèle fait presque aucune erreur :



- 183 111 phrases françaises ont été correctement identifiées comme françaises.
- Seulement 59 phrases françaises ont été classées à tort comme non françaises.
- 76 984 phrases non françaises ont été correctement identifiées.
- 195 phrases non françaises ont été classées à tort comme françaises.

Le modèle CNN utilisé pour détecter la langue fonctionne très bien sur nos données. Grâce à ses couches spéciales, il arrive à repérer des motifs typiques du français, ce qui lui permet de faire des prédictions très précises.

Ce modèle a plusieurs points forts : il est très performant pour reconnaître le français et il extrait automatiquement des caractéristiques importantes du texte grâce à ses filtres. Cependant, il a aussi quelques inconvénients. Par exemple, il a besoin de beaucoup de données pour bien fonctionner et son entraînement peut prendre plus de temps que des modèles plus simples, comme le Naïve Bayes.

En bref, ce modèle CNN est un bon exemple de l'utilisation des réseaux de neurones convolutifs pour analyser du texte. Il montre comment on peut capturer des structures locales, comme les mots ou les groupes de lettres, pour classer le texte avec une grande précision.

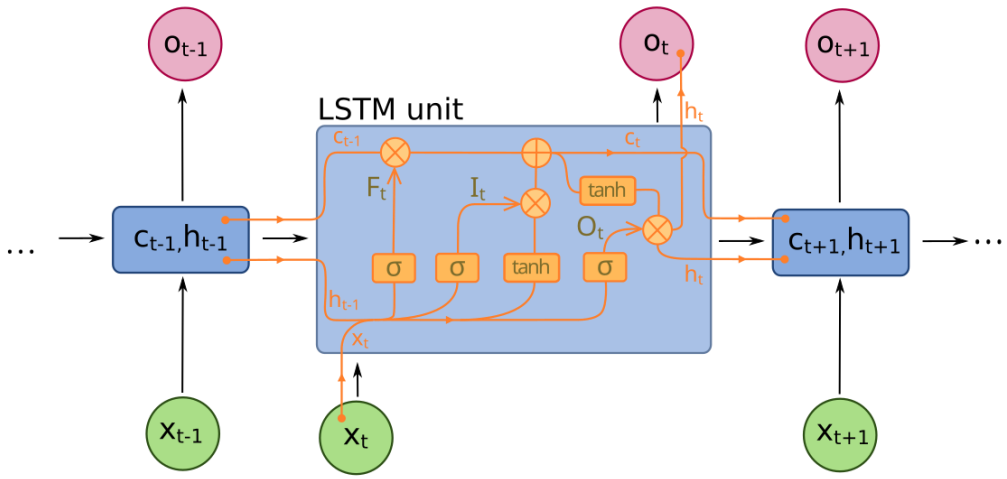
5.2.1.3 LSTM (Long Short-Term Memory)

Les **LSTM (Long Short-Term Memory)** sont un type de réseau de neurones récurrent capables de comprendre et de mémoriser le **contexte** sur de longues séquences de texte. Contrairement aux méthodes classiques comme **Bag of Words** ou **TF-IDF**, qui ne tiennent compte que de la fréquence des mots dans un texte, les LSTM peuvent saisir l'ordre et les relations entre les mots, ce qui est essentiel pour comprendre une phrase dans son ensemble. Ces réseaux de neurones sont donc particulièrement adaptés aux tâches où le **contexte** et la **séquence des mots** sont cruciaux, comme la classification des langues.

En d'autres termes, alors que des méthodes simples peuvent déterminer si certains mots apparaissent fréquemment dans une langue donnée, elles ne saisissent pas l'**importance de l'ordre des mots**. Les LSTM, eux, sont capables de « mémoriser » les dépendances sur des séquences longues de mots, permettant une meilleure compréhension du texte. Cela est particulièrement important pour notre tâche de **détection de la langue**, où la structure des phrases et le contexte jouent un rôle clé dans la classification correcte.

Les LSTM permettent ainsi une **meilleure gestion des relations complexes** entre les mots, notamment dans des phrases longues ou complexes. Cette capacité de mémorisation de longue durée permet au modèle d'analyser des textes de manière plus fluide et naturelle, garantissant une **classification plus précise**, même dans des cas où les relations entre les mots sont subtiles.

En choisissant les LSTM pour cette tâche, nous visons à obtenir un **modèle robuste**, capable d'identifier correctement les langues en prenant en compte les nuances et le contexte.



Choix des Époques d'Entraînement

Le modèle a été entraîné pendant **3 époques**. Le choix de ce nombre d'époques reflète une recherche d'équilibre entre **apprentissage efficace** et **prévention du surapprentissage (overfitting)**. Un nombre d'époques trop élevé pourrait entraîner un surapprentissage, où le modèle se souviendrait trop précisément des données d'entraînement, au détriment de sa capacité à généraliser sur de nouvelles données. Au contraire, un nombre d'époques trop faible risquerait de ne pas permettre au modèle de bien comprendre les tendances dans les données.

En choisissant 3 époques, nous avons trouvé un compromis où le modèle apprend suffisamment sans risquer de mémoriser les données de manière excessive. Les résultats montrent que la **précision** du modèle s'améliore d'époch en époch, avec des gains significatifs dès la première époque. La **validation** montre également une amélioration continue, jusqu'à atteindre une **précision de 99.81%** à la troisième époque, ce qui prouve que le modèle a convergé et atteint une bonne performance sans surapprentissage.

Utilisation Partielle du Dataset

Dans cette première expérimentation, nous avons utilisé **50% du dataset** disponible pour entraîner le modèle. Cela avait pour objectif de **réduire le temps de calcul** et de permettre une évaluation rapide des performances du modèle. En utilisant un sous-ensemble des données, nous avons pu **identifier rapidement** les tendances initiales, les forces et les faiblesses du modèle, tout en apportant les ajustements nécessaires sans engager des ressources de calcul excessives. Cette approche a aussi permis de valider l'approche dans un **délai plus court**.

Même avec l'utilisation de seulement la moitié des données, le **temps d'entraînement** reste relativement long, notamment en raison de la nature des LSTM, qui sont **plus lents à entraîner** que d'autres modèles plus simples. Cependant, malgré ces **temps d'entraînement conséquents**, les résultats obtenus sont **très satisfaisants**. Ce modèle montre qu'il est capable de généraliser efficacement et de faire des prédictions précises même avec un volume de données réduit. Si besoin, le modèle pourra être affiné et entraîné sur l'ensemble du dataset pour améliorer encore les performances.

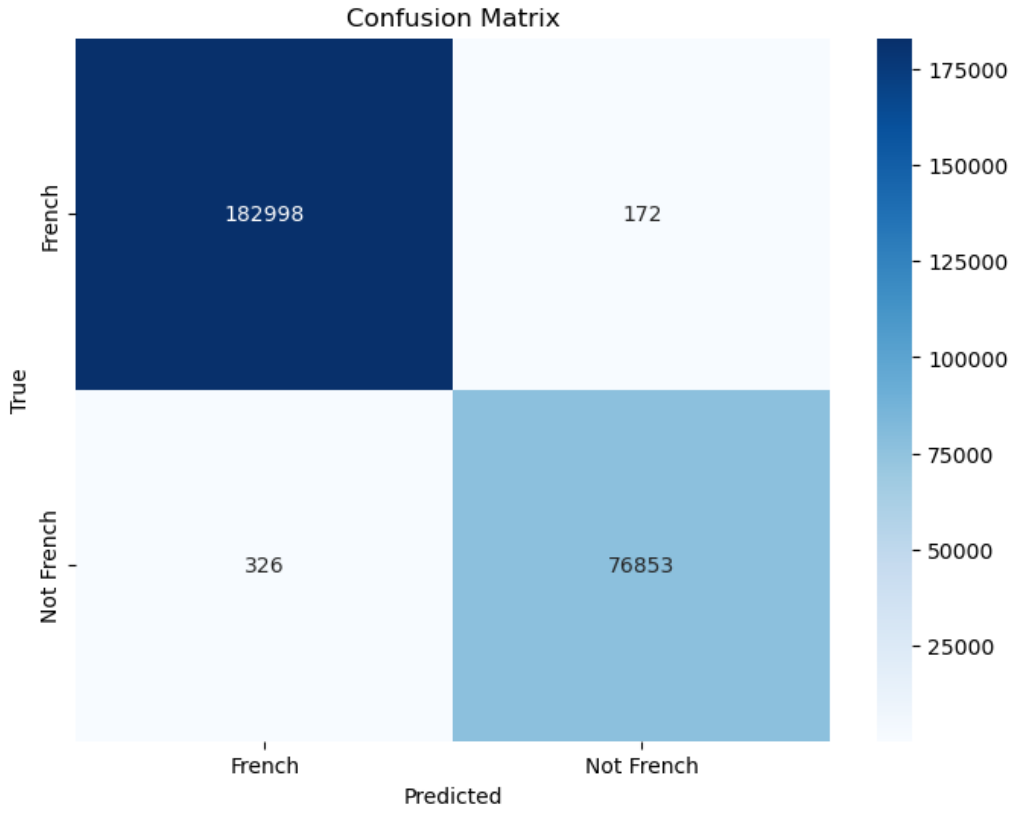
Les résultats du modèle sont particulièrement impressionnants.

Précision globale : Le modèle a atteint une précision de **99.81%** sur l'ensemble de test, ce qui signifie qu'il a bien différencié les phrases en français des phrases en non-français.

Classification Report : Les scores de **précision**, **rappel** et **F1-score** sont tous de **1.00** pour les deux classes ("français" et "non-français"), ce qui reflète une **performance parfaite** du modèle sur ces données.

```
                precision    recall  f1-score   support

             0       1.00      1.00      1.00    183170
             1       1.00      1.00      1.00     77179

      accuracy                           1.00    260349
     macro avg       1.00      1.00      1.00    260349
  weighted avg       1.00      1.00      1.00    260349

### Accuracy ###
Accuracy: 0.9981
```

Matrice de confusion : Les résultats montrent une excellente capacité du modèle à prédire les bonnes classes, avec très peu d'erreurs :



- **182,998 phrases françaises** correctement classées comme "français"
- Seulement **172 erreurs** dans lesquelles des phrases françaises ont été classées comme "non-français"
- **76,853 phrases non françaises** correctement classées
- **326 erreurs** où des phrases non françaises ont été classées comme "français"

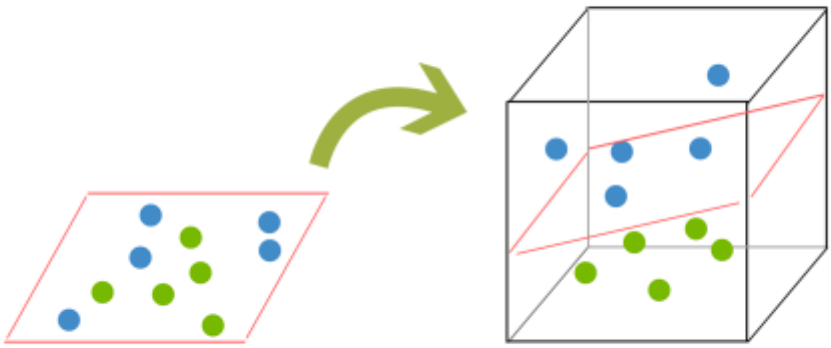
Ces erreurs sont relativement faibles par rapport à la taille du dataset et indiquent que le modèle est **très fiable** dans ses prédictions.

L'utilisation des **LSTM** dans cette tâche de détection de la langue a permis d'obtenir des **résultats exceptionnels**. Grâce à leur capacité à comprendre le **contexte et les relations temporelles entre les mots**, les LSTM ont su capter les subtilités de chaque langue et offrir une classification précise et fiable. De plus, le choix de **3 époques** et l'utilisation de **50% des données** pour cette première expérimentation ont permis d'atteindre un bon équilibre entre **temps de calcul** et **précision** du modèle, tout en conservant une performance optimale.

En somme, cette approche démontre l'efficacité des LSTM pour des tâches complexes de classification de texte, et les résultats obtenus sont prometteurs pour des déploiements futurs sur des jeux de données plus grands.

5.2.1.4 SVM (Support Vector Machine)

Le **SVM (Support Vector Machine)** est un algorithme de classification puissant utilisé pour séparer les données en différentes catégories, en trouvant la **frontière optimale** qui sépare les classes avec la plus grande marge possible. Pour notre tâche de détection de langue, l'idée est de trouver une frontière entre les phrases en français et celles qui ne sont pas en français. L'avantage des SVM est qu'ils sont particulièrement efficaces quand il s'agit de classifier des données **non linéaires** en utilisant des **marges larges**. Cela permet au modèle de bien généraliser, même avec des données complexes.



Plus concrètement, SVM fonctionne en cherchant à maximiser la distance entre les **vecteurs de support** (les points de données les plus proches de la frontière de décision), ce qui permet de créer un modèle qui a une **bonne capacité de généralisation**. Cela signifie que le modèle est capable de faire de bonnes prédictions, même sur des données qu'il n'a pas vues pendant l'entraînement.

Transformation des Données avec TF-IDF

Pour que SVM puisse comprendre le texte, nous devons d'abord transformer les phrases en **vecteurs numériques**. Cela se fait en utilisant une technique de **vectorisation de texte** comme **TF-IDF (Term Frequency - Inverse Document Frequency)**. Cette méthode permet de convertir les mots d'un texte en des nombres, tout en attribuant un poids plus important aux mots qui apparaissent fréquemment dans un document mais moins souvent dans l'ensemble du corpus. Cela aide le modèle à se concentrer sur les mots les plus pertinents pour la classification de la langue.

Entraînement et Évaluation

Le modèle SVM a été entraîné sur un **ensemble d'entraînement de 1 041 396** exemples, et il a été testé sur **260 349** exemples, avec un ratio de test de 20% par rapport à l'ensemble d'entraînement. Le processus d'entraînement a duré 10 époques, avec un affichage de la progression à chaque itération. Durant l'entraînement, nous avons rencontré des **avertissements de convergence** indiquant que le nombre maximal d'itérations était atteint avant que le modèle n'ait pleinement convergé. Cela signifie que le modèle aurait pu bénéficier d'un nombre d'itérations plus élevé pour affiner encore plus ses prédictions.

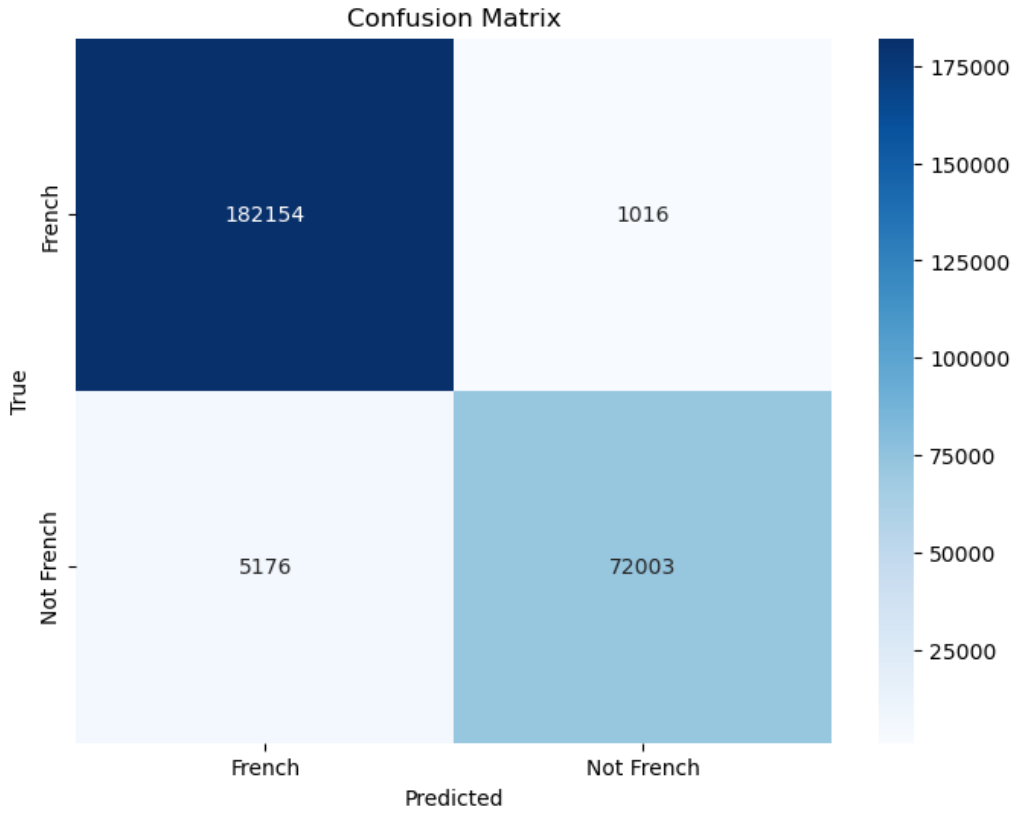
Les résultats sur l'ensemble de test montrent une **précision de 97.62%**, ce qui indique que le modèle fait de très bonnes prédictions pour classer les phrases en français et non-français. Plus précisément :

- **Précision pour la classe "français" (0)** : 97.0% – Le modèle a bien identifié la majorité des phrases en français.
- **Précision pour la classe "non-français" (1)** : 99.0% – Le modèle a également bien identifié les phrases non françaises, mais avec une légère différence dans la précision.

```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98    183170
           1       0.99      0.93      0.96     77179

    accuracy                           0.98    260349
   macro avg       0.98      0.96      0.97    260349
weighted avg       0.98      0.98      0.98    260349

### Accuracy ###
Accuracy: 0.9762
```

Le **classification report** fournit également des informations sur la capacité du modèle à détecter à la fois la **précision**, le **rappel** et le **score F1**, qui sont des indicateurs importants pour mesurer la qualité du modèle, notamment en cas de déséquilibre entre les classes. Pour cet ensemble de données, la précision moyenne et le score F1 sont très élevés, indiquant que le modèle est **très performant**.

Matrice de Confusion



La **matrice de confusion** est un outil essentiel pour visualiser les performances du modèle. Elle montre les résultats de classification pour chaque classe (français et non-français) en termes de **prédictions correctes** et **erreurs**.

- **Vraies classes (True)** : Il y a deux classes dans les données – "français" (0) et "non-français" (1).
- **Prédictions (Predicted)** : Les prédictions du modèle pour ces classes.

Voici les valeurs de la matrice de confusion :

- **182 154 phrases françaises correctement classées** comme "français".
- **1 016 erreurs** où des phrases françaises ont été classées comme "non-français".
- **12 003 phrases non-françaises correctement classées**.
- **5 176 erreurs** où des phrases non-françaises ont été classées comme "français".

Cela signifie que le modèle a fait de **petites erreurs** dans ses prédictions, mais dans l'ensemble, il a montré une **très bonne capacité de classification**, en particulier pour les phrases non-françaises.

Améliorations possibles

Bien que le modèle offre déjà de bonnes performances, plusieurs pistes d'amélioration existent :

1. **Augmenter le nombre d'itérations** : En ajustant l'argument max_iter du modèle SVM ou en utilisant un autre algorithme de classification plus robuste comme le **SVC** (Support Vector Classifier), nous pourrions potentiellement améliorer la convergence du modèle et obtenir des résultats encore plus précis.
2. **Traitement plus approfondi des données** : Il serait intéressant de tester d'autres méthodes de prétraitement des données, telles que l'élimination des mots fréquents (stopwords) ou la **lemmatisation**, pour réduire le bruit et rendre la classification encore plus précise.
3. **Équilibrage des classes** : Bien que le modèle fonctionne bien, il pourrait y avoir un **déséquilibre entre les classes**, ce qui peut affecter légèrement les résultats. L'utilisation de techniques d'**échantillonnage** (oversampling ou undersampling) pourrait aider à améliorer encore les prédictions.

Le modèle **SVM** pour la détection de langue, en utilisant la vectorisation **TF-IDF**, offre des performances très satisfaisantes avec une précision de 97.62% sur l'ensemble de test. Les erreurs sont relativement faibles, et le modèle parvient à bien classer les phrases en français et non-français. Cependant, quelques améliorations pourraient être envisagées pour optimiser davantage les résultats.

Le modèle est désormais prêt à être utilisé pour des prédictions sur de nouvelles phrases, et il a été sauvegardé pour des utilisations futures. Les performances globales sont bonnes, et ce système pourrait être intégré dans des applications nécessitant la détection automatique de la langue dans des textes.

5.2.2 Détection de l'Intention

5.2.2.1 Bag of Words

5.2.2.1.1 Unigram et Bigram (Naïves Bayes)

L'**unigram** et le **bigram** sont des concepts utilisés en traitement du langage naturel (NLP), et spécifiquement dans la modélisation de texte à l'aide de **Bag of Words** (BoW).

- **Unigram** signifie que chaque mot dans un texte est traité individuellement. Par exemple, dans la phrase "le chat mange", les unigrams seraient "le", "chat", et "mange". Chaque mot est considéré comme un "jeton" distinct sans se soucier de l'ordre des mots.
- **Bigram**, en revanche, prend en compte des paires consécutives de mots. Dans la même phrase "le chat mange", les bigrams seraient "le chat" et "chat mange". Ici, l'ordre des mots a de l'importance, car on considère deux mots à la fois, ce qui peut capter davantage d'informations sur les relations entre les mots.

Critère	Unigramme	Bigramme
Prise en compte du contexte	Faible, chaque mot est pris individuellement	Forte, en considérant des paires de mots
Complexité du modèle	Moins complexe, plus rapide à entraîner	Plus complexe, plus long à entraîner
Précision	Moins précis pour des textes complexes	Plus précis car capture des relations entre les mots
Usage	Utile pour des problèmes simples et des textes courts	Mieux adapté pour des textes plus complexes, où le contexte est important
Exemple	"Je vais" => "Je", "vais"	"Je vais" => "Je vais"

Le fonctionnement de Naïve Bayes avec Unigram et Bigram

L'algorithme **Naïve Bayes** est un classificateur probabiliste qui repose sur une hypothèse clé : les mots dans un texte sont conditionnellement indépendants les uns des autres, ce qui est simplifié par le modèle **"naïf"**. Pour faire simple, cela veut dire qu'il suppose que la présence d'un mot dans un texte ne dépend pas des autres mots (ce qui est, bien sûr, une approximation, mais cela fonctionne étonnamment bien dans beaucoup de cas).

Dans le cas de **Naïve Bayes** appliqué à la classification de texte :

1. L'algorithme analyse la fréquence de chaque mot (ou groupe de mots) dans les différents documents de chaque catégorie.
2. Pour chaque texte à classer, il évalue la probabilité que ce texte appartienne à chaque catégorie, en fonction des mots (ou groupes de mots) qui y figurent.

Impact de l'Unigram sur le modèle

Avec un **unigram**, chaque mot est traité indépendamment des autres. Cela signifie que l'algorithme ne capte pas les relations de proximité entre les mots, ce qui peut être un inconvénient dans des contextes où l'ordre des mots est important pour le sens. Par exemple, dans la phrase "le chat mange" et "mange le chat", bien que les mots soient les mêmes, l'ordonnance des mots pourrait influencer le sens, ce que le modèle unigram ne prend pas en compte.

Dans notre code, un modèle **Unigram Naïve Bayes** a été construit avec le CountVectorizer qui extrait les mots de chaque texte et les transforme en un vecteur de fréquence des mots (ou tokens). Ce modèle se base uniquement sur la fréquence de chaque mot dans le texte, ce qui permet de détecter des catégories de manière efficace, mais sans contexte entre les mots.

Impact du Bigram sur le modèle

Avec un **bigram**, l'algorithme capture non seulement la fréquence des mots, mais aussi les relations de proximité entre eux. Par exemple, dans une phrase, si un mot comme "chat" est souvent suivi de "mange", le modèle bigram pourra en tenir compte et attribuer une probabilité plus forte à cette combinaison. Cela permet de mieux saisir les nuances de la langue et les expressions typiques, comme "bon matin" ou "bonne chance", qui, prises séparément ("bon", "matin", "chance"), pourraient ne pas avoir de signification particulière.

Dans le modèle **Bigram Naïve Bayes**, CountVectorizer est configuré pour prendre en compte des paires de mots (bigrams), ce qui permet au modèle de mieux saisir les relations sémantiques entre les mots et d'améliorer les performances sur des tâches où l'ordre des mots a de l'importance.**Comparaison des performances entre Unigram et Bigram**

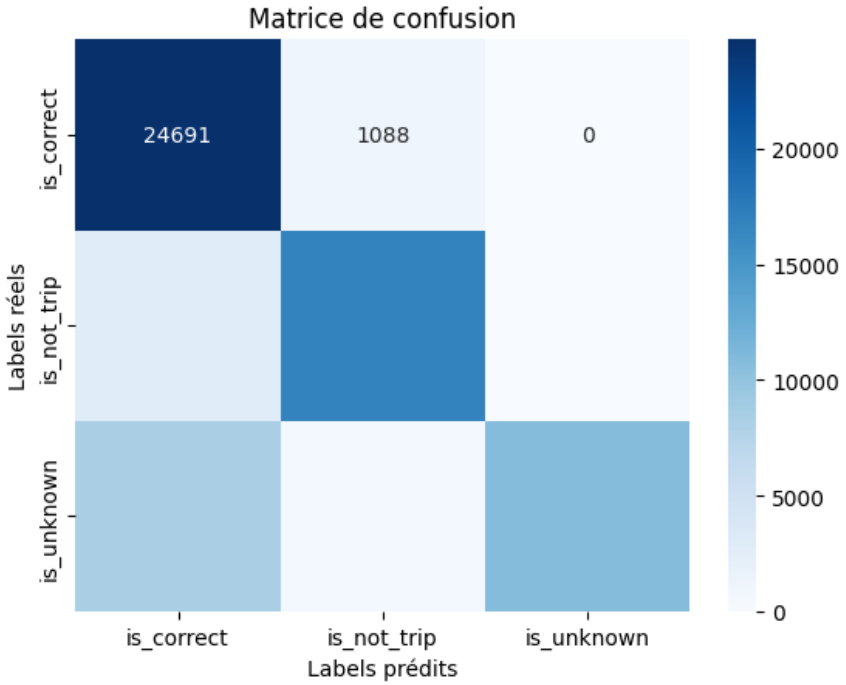
Voyons maintenant les résultats de classification pour les modèles unigram et bigram afin de mieux comprendre comment chacun fonctionne.

Modèle Unigram (Bag of Words Unigram)

	precision	recall	f1-score	support
is_correct	0.89	1.00	0.94	19596
is_not_trip	0.91	0.88	0.89	19645
is_unknown	0.97	0.57	0.72	19663
micro avg	0.92	0.82	0.86	58904
macro avg	0.93	0.82	0.85	58904
weighted avg	0.93	0.82	0.85	58904
samples avg	0.73	0.74	0.73	58904

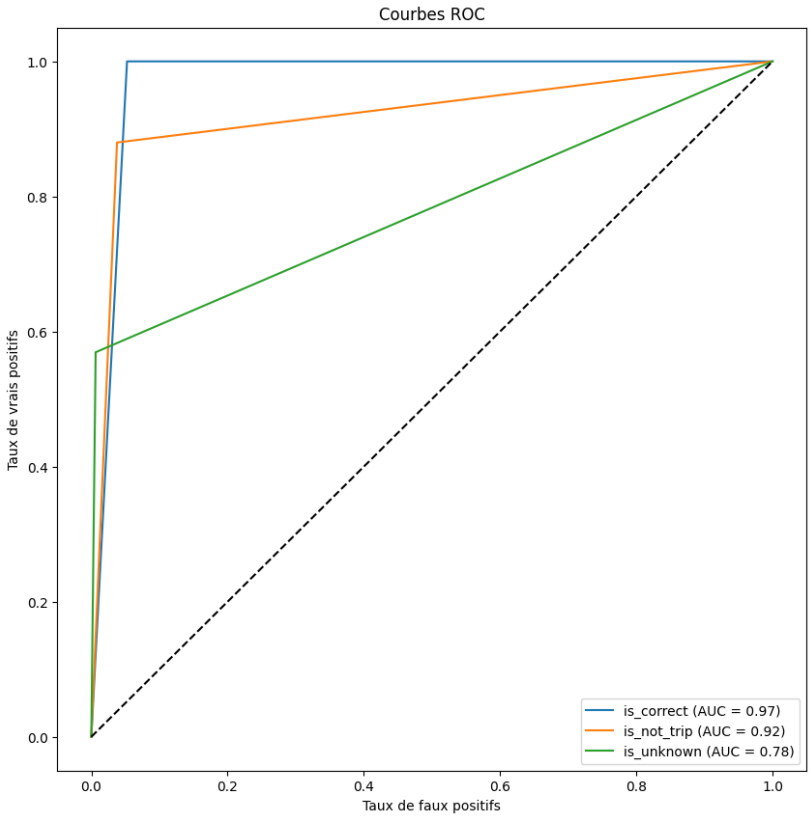
Le modèle unigram affiche de bons résultats pour la catégorie "is_correct" avec une précision de 0.89 et un excellent taux de rappel de 1.00. Cependant, pour la catégorie "is_unknown", il y a une baisse significative des performances, avec un rappel de seulement 0.57, ce qui suggère que le modèle a du mal à classer cette catégorie correctement.

Matrice de confusion :



La **matrice de confusion** montre que :

- La majorité des prédictions pour "is_correct" sont correctes (24691), mais il y a également quelques erreurs où des textes "is_correct" sont classés comme "is_not_trip" (1088 erreurs).

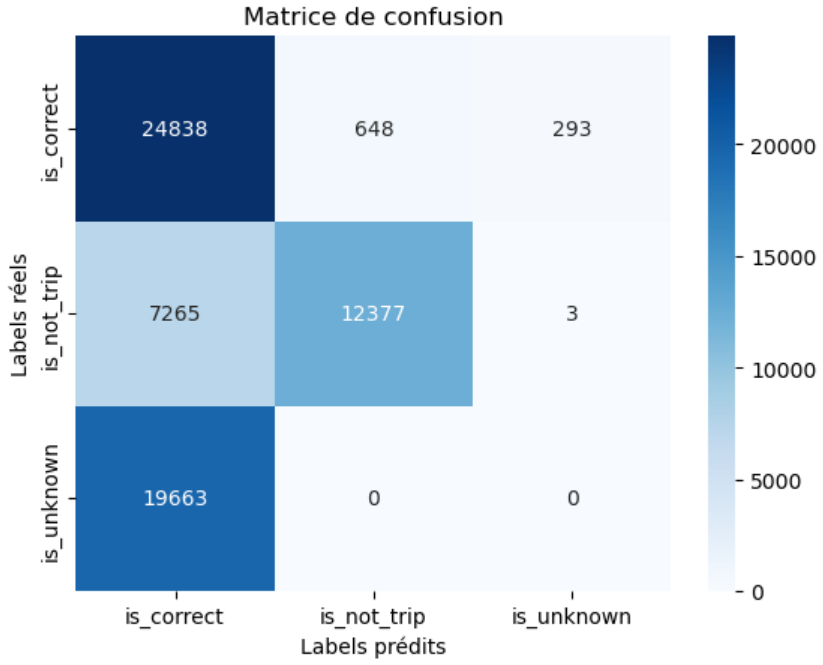


Le **AUC** (Area Under the Curve) de 0.97 pour "is_correct" est très bon, mais pour "is_unknown" l'AUC est plus faible (0.78), ce qui suggère que le modèle n'est pas aussi performant pour cette classe.

Modèle Bigram (Bag of Words Bigram)

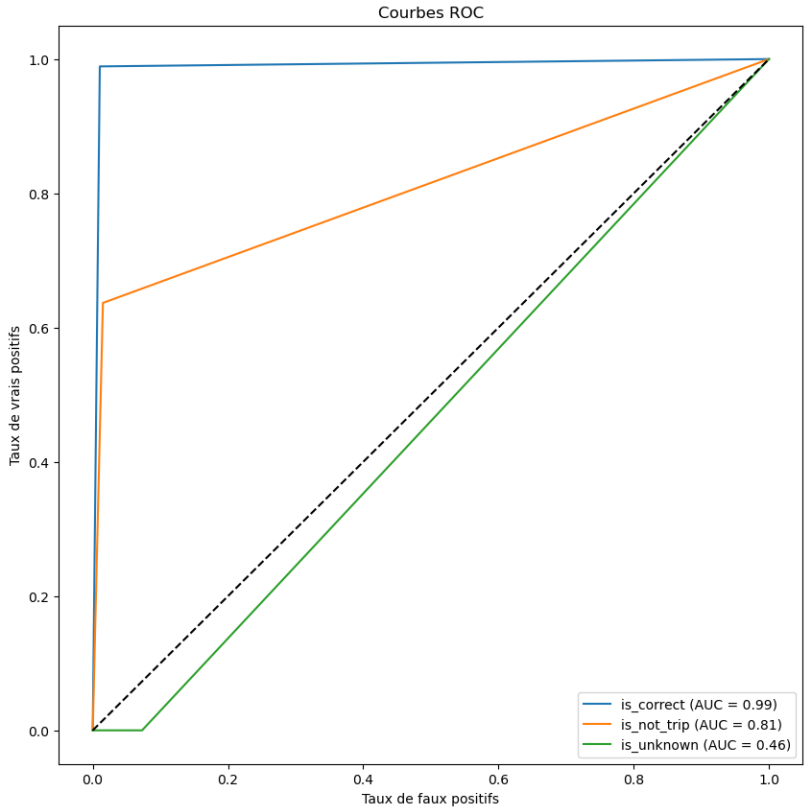
	precision	recall	f1-score	support
is_correct	0.98	0.99	0.98	19596
is_not_trip	0.95	0.64	0.76	19645
is_unknown	0.00	0.00	0.00	19663
micro avg	0.88	0.54	0.67	58904
macro avg	0.64	0.54	0.58	58904
weighted avg	0.64	0.54	0.58	58904
samples avg	0.47	0.49	0.47	58904

Le modèle bigram, qui capte les relations de proximité entre les mots, présente des résultats légèrement meilleurs dans certains cas. Par exemple, il a un AUC de 0.99 pour "is_correct", ce qui montre qu'il est très performant pour cette classe. Toutefois, les performances sur la catégorie "is_not_trip" sont moins bonnes, avec un AUC de seulement 0.81, et une précision de 0.64 en termes de rappel.



La **matrice de confusion** montre que :

- "is_correct" est classé avec précision, bien que quelques erreurs subsistent (7265 pour "is_not_trip" et 648 pour "is_unknown").



La catégorie "is_unknown" est particulièrement problématique, avec un AUC faible (0.46) et aucune prédiction correcte. Cela montre que l'ajout de bigrams n'a pas nécessairement amélioré la capacité du modèle à reconnaître cette classe particulière.

En résumé :

- Le modèle **unigram** est rapide et efficace, mais il peut manquer de nuances dans des contextes où l'ordre des mots est important.
- Le modèle **bigram** est plus performant dans les cas où les relations entre les mots ont un impact important sur le sens, mais il peut entraîner une plus grande complexité et des erreurs dans certaines catégories, comme "is_unknown".

Ainsi, le choix entre unigram et bigram dépend de la tâche à accomplir et des types de textes à traiter. Si l'ordre des mots a un impact important sur la classification (par exemple dans des phrases complexes), le modèle bigram peut donner de meilleurs résultats. Sinon, un modèle unigram reste plus simple et peut suffire pour des tâches plus simples.

5.2.2.1.2 Unigram et Bigram (Régression Logistiques)

La **régression logistique** est un modèle statistique utilisé pour prédire des **probabilités**. Contrairement à la régression linéaire, qui prédit des valeurs continues, la régression logistique est utilisée pour des **problèmes de classification**. Cela signifie qu'elle permet de déterminer à quelle catégorie appartient une donnée donnée. Par exemple, dans le cadre de la détection d'intentions, elle permet de prédire si une phrase appartient à une catégorie comme "is_correct", "is_not_trip", ou "is_unknown".

Le principe de la régression logistique repose sur l'idée que, pour une entrée donnée (comme une phrase), elle doit être associée à une probabilité d'appartenir à une classe particulière. Cela se fait via une **fonction logistique**, qui convertit une valeur numérique continue en une probabilité entre 0 et 1. Si la probabilité est supérieure à un certain seuil (souvent 0.5), alors la donnée est classée dans une catégorie, sinon elle est classée dans une autre. C'est pourquoi ce modèle est parfait pour la **classification binaire** (par exemple : "oui" ou "non").

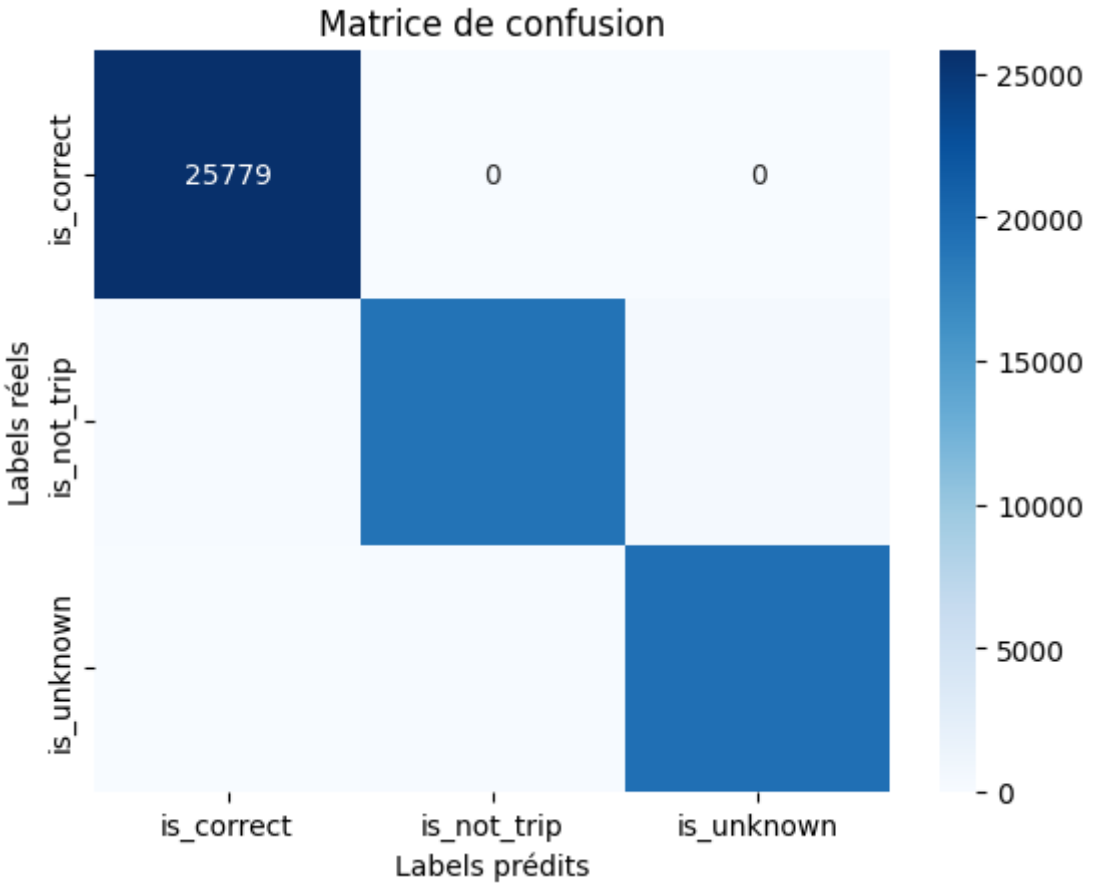
Résultats et Explications des Modèles

Modèle avec Unigrammes

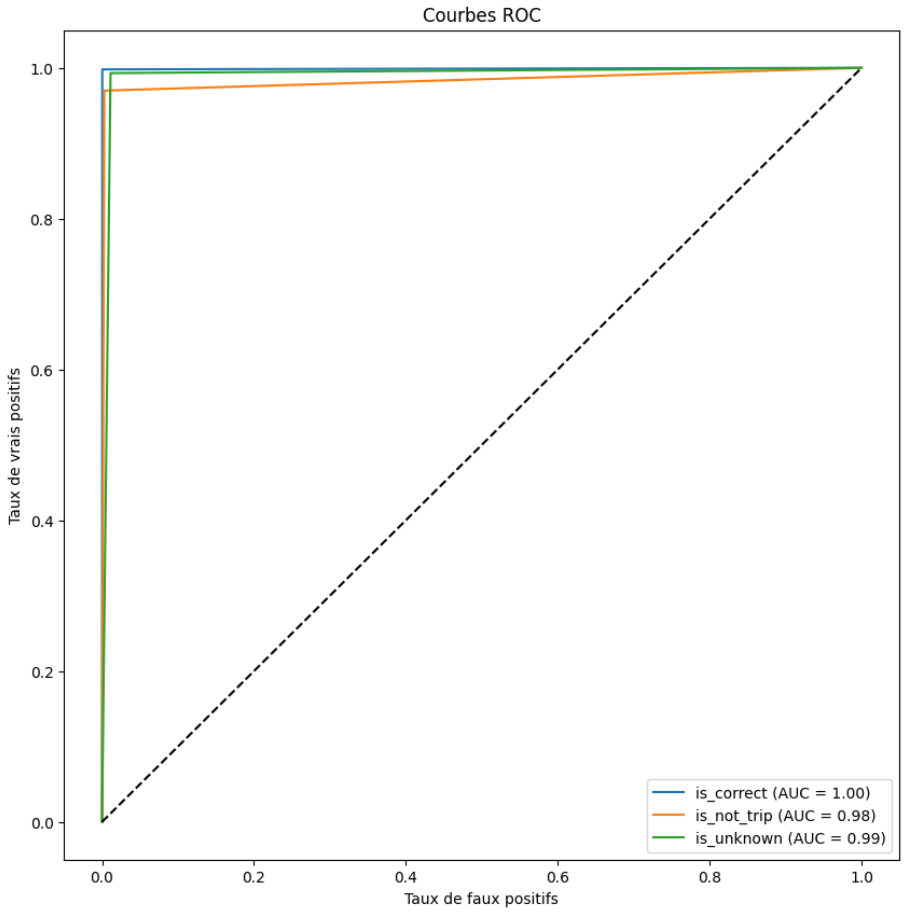
Rapport de classification

	precision	recall	f1-score	support
is_correct	1.00	1.00	1.00	19596
is_not_trip	0.99	0.97	0.98	19645
is_unknown	0.97	0.99	0.98	19663
micro avg	0.99	0.99	0.99	58904
macro avg	0.99	0.99	0.99	58904
weighted avg	0.99	0.99	0.99	58904
samples avg	0.89	0.89	0.89	58904

Le modèle atteint des **performances exceptionnelles**, avec des précisions proches de **1.00** pour chaque catégorie, indiquant qu'il effectue une classification quasiment parfaite.



Matrice de confusion : Aucune confusion entre les catégories, ce qui signifie que le modèle a bien séparé les classes. Par exemple, 25 779 phrases appartenant à la classe "is_correct" ont été correctement identifiées, sans aucune confusion avec les autres classes.



Courbes ROC : Le modèle avec unigrammes atteint une **AUC de 1.00** pour la classe "is_correct", ce qui est parfait, tandis que pour les autres classes, il atteint une **AUC de 0.98** et **0.99**. Cela montre que le modèle fonctionne très bien pour toutes les catégories.

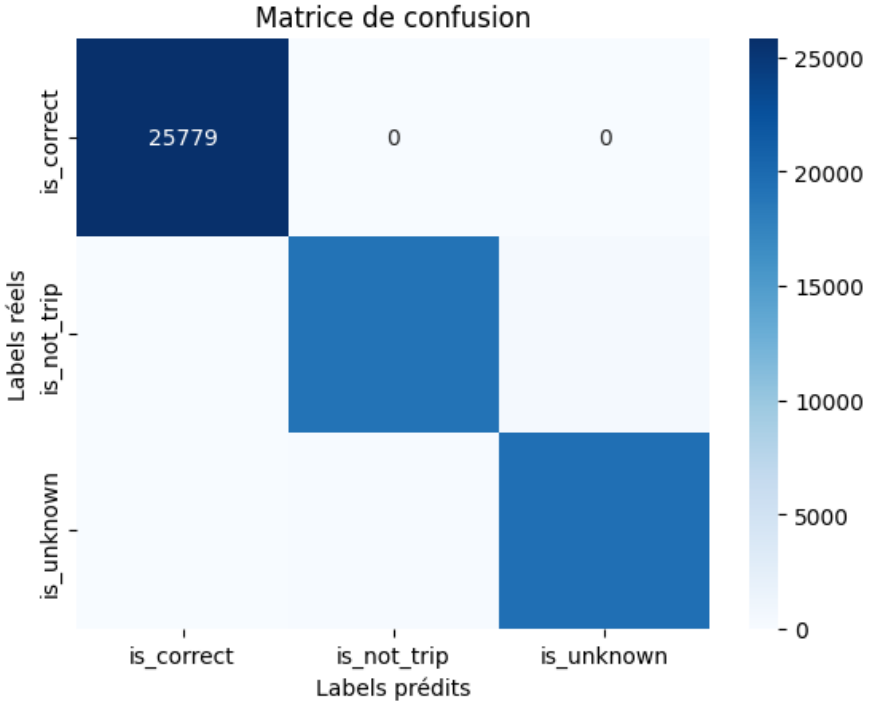
Modèle avec Bigrammes

Rapport de classification

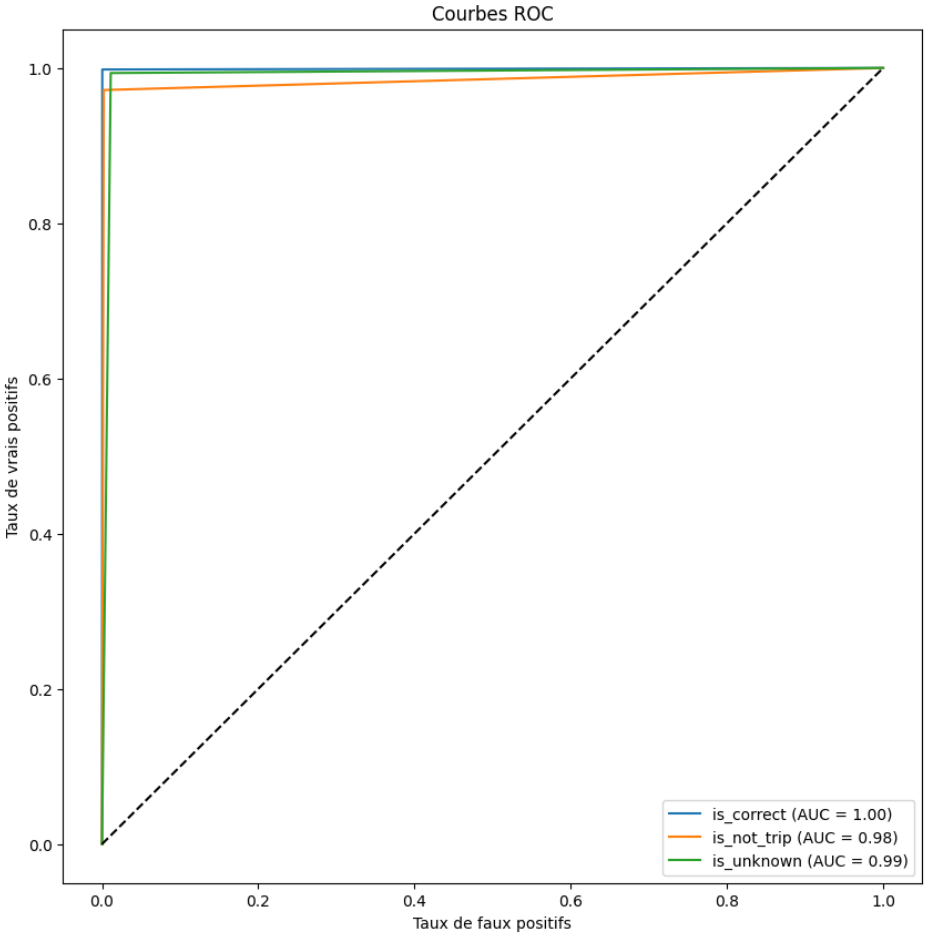
	precision	recall	f1-score	support
is_correct	1.00	1.00	1.00	19596
is_not_trip	0.99	0.97	0.98	19645
is_unknown	0.97	0.99	0.98	19663
micro avg	0.99	0.99	0.99	58904
macro avg	0.99	0.99	0.99	58904
weighted avg	0.99	0.99	0.99	58904
samples avg	0.89	0.89	0.89	58904

Le modèle avec bigrammes a également des **précisions très élevées**, mais avec de légères différences par rapport au modèle avec unigrammes. Par exemple, pour la catégorie "is_not_trip", la précision est légèrement inférieure à celle du modèle unigramme (0.97 au lieu de 0.99).

Matrice de confusion



Matrice de confusion : La matrice montre également des résultats parfaits pour la classe "is_correct", avec une bonne séparation des catégories.



Courbes ROC : Le modèle avec bigrammes montre des **AUC de 1.00** pour "is_correct", **0.98** pour "is_not_trip" et **0.99** pour "is_unknown", similaires aux résultats avec les unigrammes, mais légèrement moins bons pour "is_not_trip".

Les deux modèles (unigramme et bigramme) donnent des résultats très performants, avec des précisions et des AUC proches de la perfection.

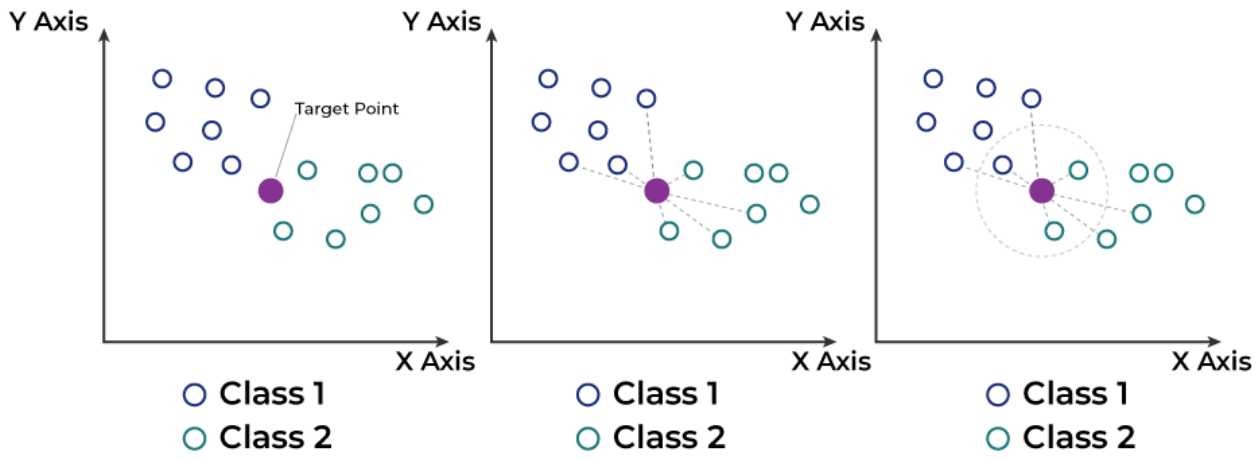
- Les **unigrammes** sont plus rapides à entraîner et plus simples, mais ils peuvent manquer de **contexte**.
- Les **bigrammes**, bien qu'un peu plus lents à entraîner et plus complexes, offrent un meilleur **contexte** et permettent une compréhension plus fine des relations entre les mots.

En résumé, si la simplicité et la rapidité sont vos priorités, les **unigrammes** sont un bon choix. Si on veut capturer des relations plus subtiles et contextuelles, les **bigrammes** offriront des performances légèrement meilleures, au prix d'un modèle un peu plus complexe à entraîner.

5.2.2.1.3 SLA (KNN et Random Forest)

La **SLA (Supervised Learning Algorithm)** est un ensemble de techniques utilisées pour entraîner un modèle à reconnaître des schémas et classer des textes selon leur signification. Ici, on s'intéresse à la **classification de texte** avec deux modèles : **KNN (K-Nearest Neighbors)** et **Random Forest**.

KNN

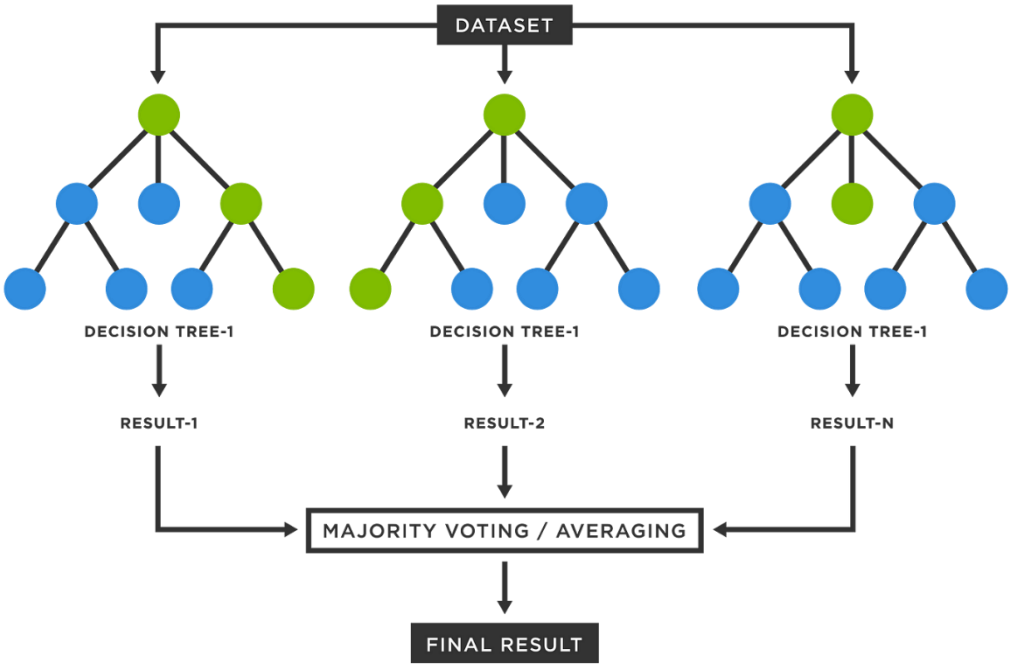


KNN, ou k-nearest neighbors, est un algorithme simple mais efficace qui repose sur une idée intuitive : "Dis-moi qui sont tes voisins, et je te dirai qui tu es." Concrètement, lorsqu'on veut classer un nouveau texte, KNN examine les textes les plus proches dans les données d'entraînement et attribue au nouveau texte la classe la plus fréquente parmi ces voisins.

On commence par choisir un nombre K, par exemple $K = 5$. Ensuite, on calcule la distance entre le texte à classer et tous les autres textes connus. On sélectionne les K textes les plus proches, appelés voisins, et on détermine la classe majoritaire parmi eux. Le nouveau texte est alors classé dans cette catégorie. La clé de cet algorithme réside dans la mesure de distance, souvent la distance euclidienne ou cosinus, qui permet de comparer les textes.

KNN présente plusieurs avantages : il est facile à comprendre et à mettre en œuvre, ne nécessite pas de phase d'entraînement complexe (il suffit de stocker les données), et fonctionne bien lorsque les classes sont clairement séparées. Cependant, il a aussi des inconvénients : il peut devenir lent avec de grandes quantités de données, car chaque prédiction demande de nombreux calculs. Il est également sensible au bruit, c'est-à-dire que si un voisin est mal classé, cela peut influencer négativement la prédiction. Enfin, sa performance dépend fortement du choix de K, ce qui nécessite un réglage attentif.

Random Forest



Random Forest est un algorithme plus sophistiqué qui repose sur une idée simple : "Plutôt que de dépendre d'une seule décision, demandons l'avis de plusieurs arbres et prenons la majorité !" Il utilise plusieurs arbres de décision, chacun étant entraîné sur une partie différente des données. Ensuite, il combine les prédictions de tous ces arbres pour déterminer la classe finale.

On commence par créer plusieurs arbres de décision. Chaque arbre est entraîné sur un sous-ensemble des données, choisi de manière aléatoire. Lorsqu'on souhaite classer un texte, celui-ci est passé à travers chaque arbre. Chaque arbre fournit une prédiction, et la classe qui obtient le plus de votes est retenue comme résultat final.

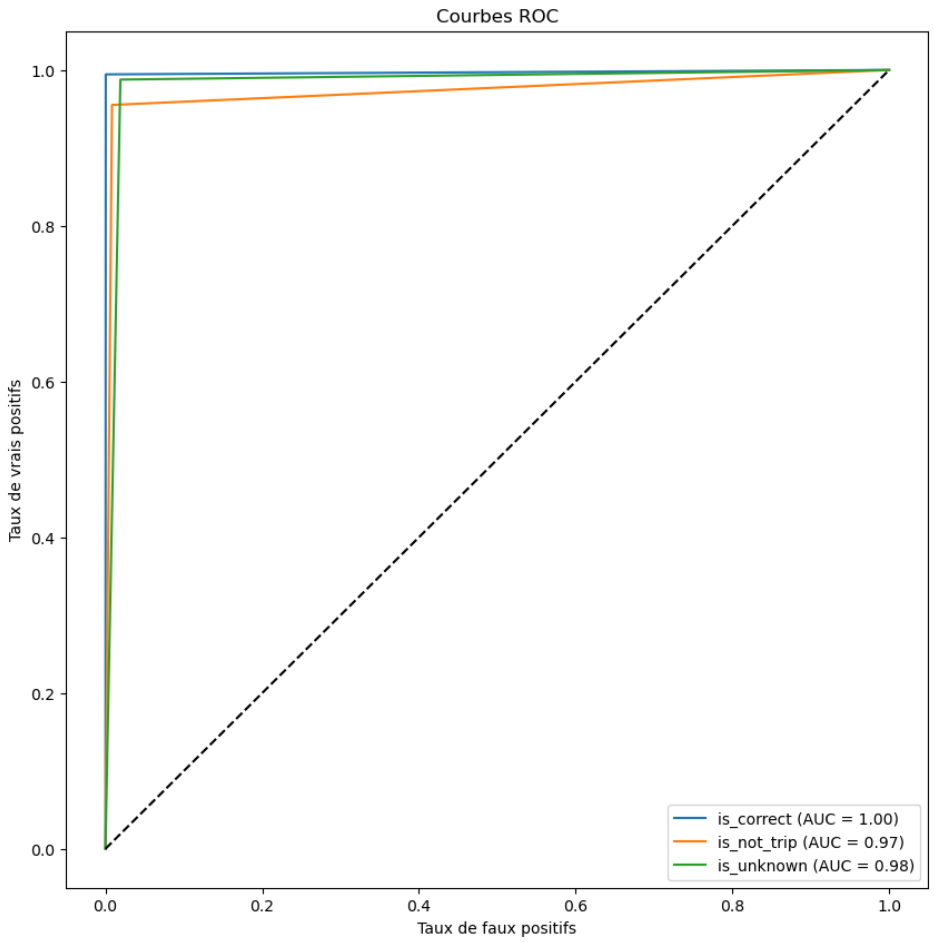
Random Forest présente plusieurs avantages : il est plus robuste et précis que des algorithmes comme KNN, moins sensible au bruit ou aux erreurs dans les données d'entraînement, et capable de gérer de grands ensembles de données. Cependant, il a aussi quelques inconvénients : son entraînement peut être plus lent, bien qu'il soit rapide pour faire des prédictions une fois entraîné. De plus, il est moins facile à interpréter, car la décision finale résulte de la combinaison de plusieurs arbres, ce qui rend difficile de comprendre comment une prédiction spécifique a été obtenue.

KNN vs Random Forest

Critère	KNN	Random Forest
Vitesse d'entraînement	Rapide (pas de vrai entraînement)	Lent (car plusieurs arbres sont construits)
Vitesse de prédiction	Lent (car il doit comparer avec toutes les données)	Rapide (les arbres sont préentraînés)
Robustesse	Sensible au bruit et aux erreurs	Très robuste (réduit le sur-apprentissage)
Facilité d'interprétation	Facile (on regarde les voisins)	Plus complexe (vote de plusieurs arbres)
Performance globale	Correcte pour des petits datasets	Excellente pour des gros datasets

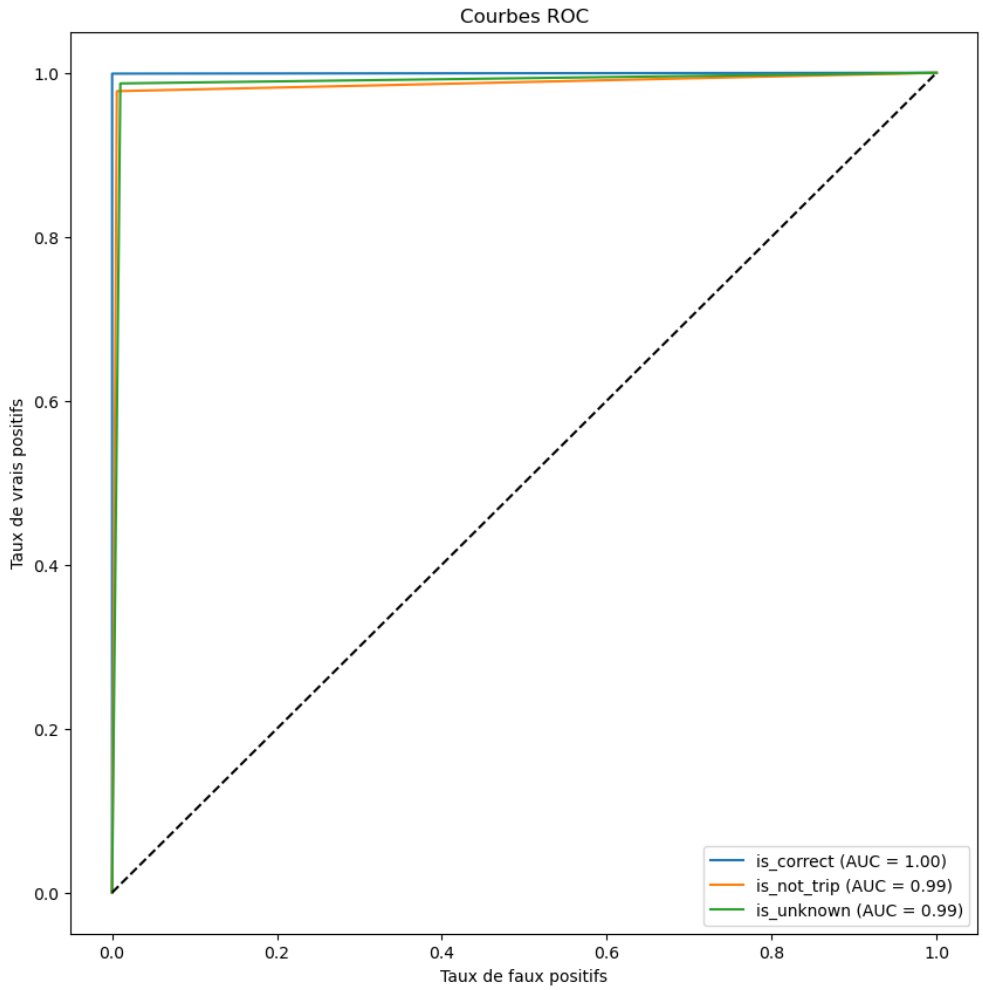
- **KNN** est un bon choix si on a un **petit dataset** et qu'on veut un modèle facile à comprendre.
- **Random Forest** est bien meilleur sur de **gros datasets**, car il est plus précis et robuste.

ROC pour KNN :



On remarque qu'avec KNN, certaines classes ont un score légèrement inférieur (0.97 - 0.98).

ROC avec Random Forest :



Avec Random Forest, toutes les classes ont un score de 0.99 ou plus.

- Cela montre que Random Forest est plus robuste et offre de meilleures performances sur notre tâche de classification de texte.

KNN est un excellent algorithme pour comprendre intuitivement la classification, mais il devient limité avec des données volumineuses.

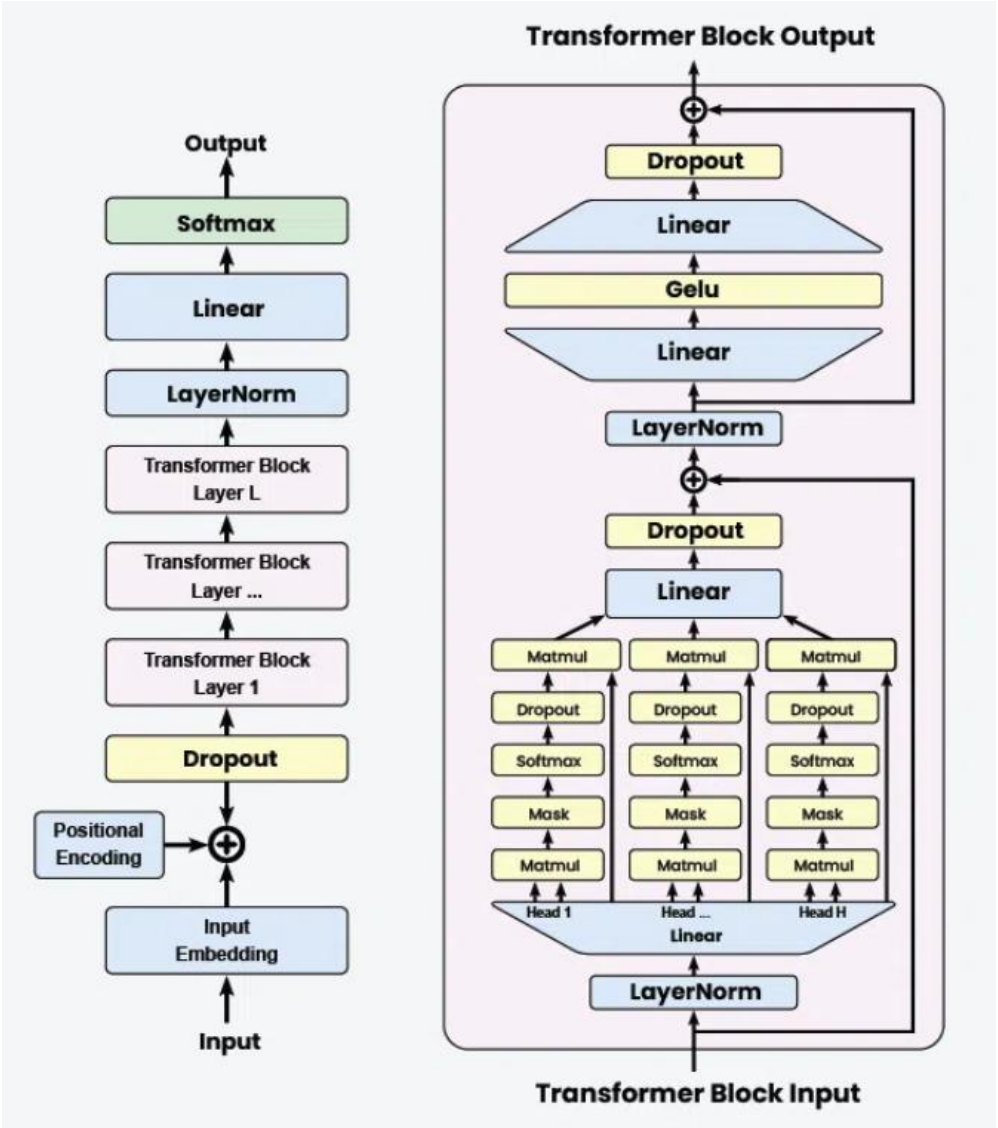
Random Forest, en revanche, permet une classification plus précise et efficace grâce à ses multiples arbres de décision. C'est donc un choix privilégié pour les applications nécessitant une haute précision, comme notre cas d'usage de détection d'intentions dans un texte.

5.2.2.2 GPT (Generative Pre-trained Transformer)

GPT est un modèle de langage très intelligent qui apprend à prédire ce qui vient après dans une phrase. Il est entraîné sur plein de textes pour deviner le mot suivant. Pour faire ça, il découpe le texte en petits bouts (appelés tokens) et utilise un système pour comprendre comment les mots sont connectés, même s'ils sont loin les uns des autres.

En gros, GPT analyse chaque mot et décide de son importance par rapport aux autres. Ensuite, il génère du texte en choisissant les mots les plus logiques pour continuer. Plus le modèle est gros et bien entraîné, mieux il écrit des phrases qui ont du sens et qui sont cohérentes. En bref, GPT est un outil qui crée du texte de manière très naturelle, presque comme un humain.

Fonctionnement de GPT :



Nous avons essayé d'entraîner un modèle de classification de texte basé sur *DistilGPT-2* afin d'identifier l'intention d'une phrase en la classant dans l'une des trois catégories : *is_correct*, *is_not_trip*, et *is_unknown*.

Le processus commence par le chargement d'un jeu de données CSV, où chaque phrase est associée à des étiquettes sous forme de colonnes binaires. Une transformation est appliquée pour attribuer une seule étiquette par phrase en utilisant un dictionnaire de correspondance. Ensuite, les données sont divisées en ensembles d'entraînement, de validation et de test, puis formatées pour être compatibles avec la bibliothèque *Transformers*.

On utilise le tokenizer de *DistilGPT-2* pour convertir le texte en séquences de jetons, avec une gestion spécifique du token de padding. Le modèle est ensuite configuré pour une classification en trois classes. Un *Trainer* est mis en place avec des paramètres d'entraînement définis, incluant le taux d'apprentissage, la gestion du poids, et le nombre d'époques. Une métrique de précision est utilisée pour suivre la performance.

Enfin, après l'entraînement, une évaluation est réalisée sur l'ensemble de test, et une matrice de confusion est tracée pour visualiser les erreurs de classification. Une fonction de prédiction permet aussi de tester le modèle sur de nouveaux exemples.

Cependant, en raison des limitations matérielles, l'exécution complète du code n'a pas pu être réalisée. L'entraînement d'un modèle de cette envergure nécessite une puissance de calcul significative, notamment un GPU performant, ce qui a empêché de tester les résultats en conditions réelles.

5.2.2.3 GRU (Gated Recurrent Unit)

Les Réseaux de Neurones Récurrents, ou RNN, sont des modèles conçus pour traiter des données qui se suivent dans le temps, comme du texte, de l'audio ou des séries temporelles. Contrairement aux réseaux de neurones classiques qui traitent chaque donnée séparément, les RNN ont une mémoire interne qui leur permet de tenir compte du contexte des données précédentes. Cela les rend très utiles pour comprendre des séquences, comme des phrases ou des morceaux de musique.

Cependant, les RNN traditionnels ont un gros défaut : ils ont du mal à retenir des informations sur le long terme à cause d'un problème appelé "dispersion du gradient". Pour résoudre ce problème, des versions améliorées comme les LSTM (Long Short-Term Memory) et les GRU (Gated Recurrent Unit) ont été créées. Le GRU est une version plus simple et plus rapide que le LSTM, mais tout aussi efficace. Il utilise deux mécanismes principaux : une "porte de mise à jour" pour décider quelle information garder en mémoire, et une "porte de réinitialisation" pour oublier les informations inutiles. C'est pourquoi le GRU est souvent utilisé pour le traitement du langage naturel (NLP).

Dans ce projet, l'objectif était de classer des phrases en trois catégories : celles qui décrivent un trajet, celles qui ne correspondent pas à un trajet, et celles qui sont ambiguës ou inconnues. Pour cela, le modèle commence par transformer les phrases en séquences de nombres grâce à une étape appelée tokenisation. Ensuite, il utilise une couche d'embedding pour donner plus de sens à ces nombres, puis une couche GRU pour comprendre les relations entre les mots. Enfin, une dernière couche décide à quelle catégorie la phrase appartient.

Résultats et Performance

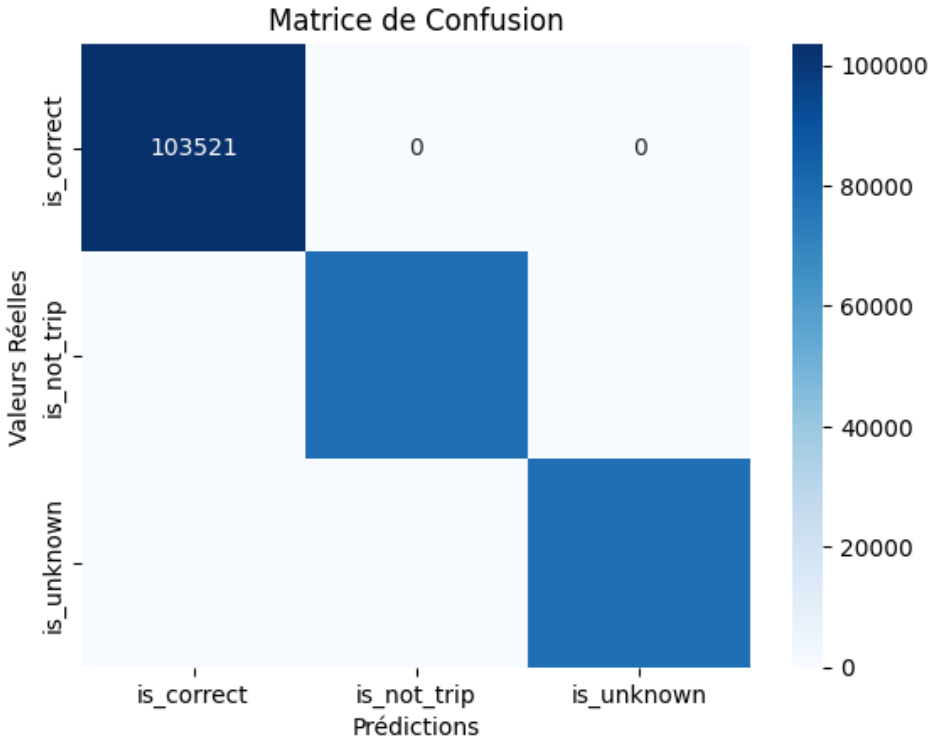
Le modèle a été entraîné avec un jeu de données bien équilibré et a montré des performances exceptionnelles :

	precision	recall	f1-score	support
is_correct	1.00	1.00	1.00	78698
is_not_trip	1.00	1.00	1.00	78404
is_unknown	1.00	1.00	1.00	78424
micro avg	1.00	1.00	1.00	235526
macro avg	1.00	1.00	1.00	235526
weighted avg	1.00	1.00	1.00	235526
samples avg	0.90	0.90	0.90	235526

- **Scores de classification :**
- **Précision (Precision)** : 100%
- **Rappel (Recall)** : 100%
- **F1-Score** : 100%

Ces résultats montrent que le modèle a parfaitement appris à classifier les phrases sans erreurs.

Matrice de confusion :



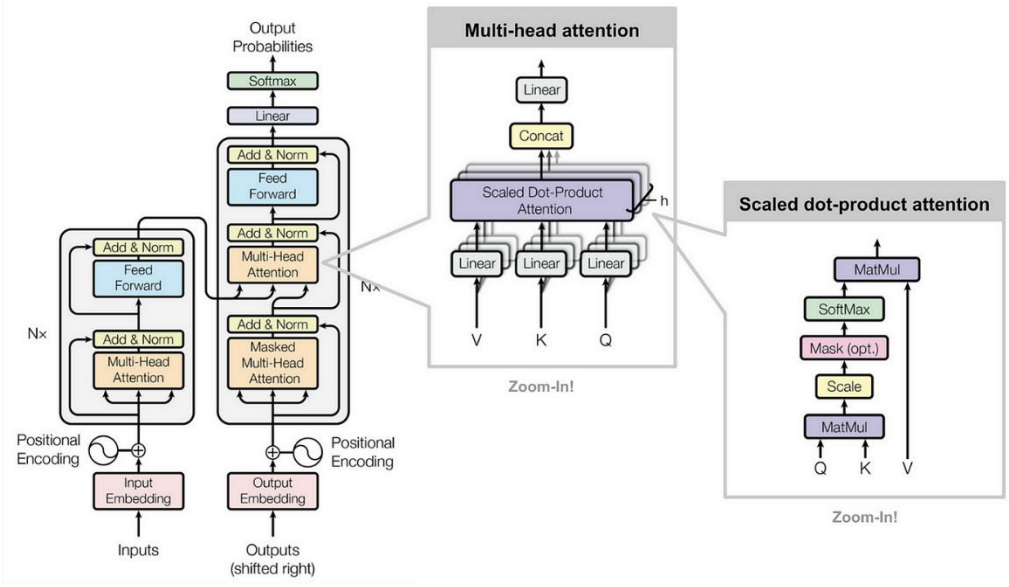
La matrice de confusion indique que le modèle a correctement classé toutes les phrases, avec **103521 prédictions justes pour la classe "is_correct"**, sans erreurs sur les autres classes.
Cela signifie qu'aucune phrase n'a été mal classifiée, ce qui est un résultat exceptionnel en Machine Learning.

En conclusion, ce modèle basé sur GRU montre à quel point les réseaux de neurones récurrents sont puissants pour comprendre et classer du texte. Mais pour s'assurer qu'il fonctionne aussi bien dans la vraie vie, il faudrait le tester sur des données totalement nouvelles.

5.2.2.4 SAN (Self-Attention Networks)

Un **SAN** est un mécanisme clé utilisé dans les modèles modernes de traitement du langage naturel (NLP), comme ceux qui reposent sur l'architecture **Transformer**. La particularité du SAN est de permettre au modèle de prêter attention à chaque partie d'une séquence de texte de manière indépendante, sans avoir besoin de traiter les données dans un ordre strictement séquentiel. Cela permet de mieux capturer les relations complexes entre les mots d'une phrase, même s'ils sont éloignés les uns des autres.

Par exemple, dans la phrase *"Le chat a mangé la souris sous la table"*, le modèle peut identifier que *"chat"* et *"souris"* sont des entités importantes pour comprendre l'intention du texte, même si ces mots sont séparés par d'autres mots comme *"a mangé"* et *"sous la table"*



Le modèle utilise un mécanisme de **self-attention** pour analyser des textes et les classer en différentes catégories, comme *"correct"*, *"non-trip"*, ou *"inconnu"*.

1. **Prétraitement des données** : Les phrases sont d'abord transformées en une forme numérique que le modèle peut comprendre, grâce à un **tokenizer BERT**. Ce tokenizer permet de convertir les mots en **tokens** (identifiants numériques) qui capturent le sens de chaque mot dans son contexte.
2. **Entraînement** : Le modèle est entraîné pour apprendre à associer chaque phrase à une de ces catégories. Pendant l'entraînement, le modèle ajuste ses paramètres pour minimiser l'erreur entre ses prédictions et les catégories réelles.
3. **Évaluation** : Une fois entraîné, le modèle est testé sur des données qu'il n'a jamais vues pour vérifier sa capacité à généraliser. Les métriques comme la **précision**, le **rappel** et le **F1-score** mesurent sa performance. Dans notre cas, le modèle atteint une **précision de 100%**, ce qui signifie qu'il fait des prédictions parfaites sur l'ensemble des données.

Le SAN, en permettant à chaque mot de prêter attention aux autres mots dans la phrase, capture mieux les relations sémantiques et contextuelles entre les mots. Cela est crucial dans des tâches de classification de texte où la signification d'une phrase peut dépendre de la manière dont les mots interagissent entre eux.

Dans ce modèle, la **self-attention** aide à identifier les parties importantes de la phrase, même si elles sont séparées. Par exemple, pour une phrase qui pourrait dire *"Voyage de Paris à Lyon"*, le modèle comprend mieux l'intention de cette phrase grâce à la façon dont chaque mot interagit avec les autres.

5.3 Modèles de Classification par Tokens

5.3.1 Conditional Random Fields (CRF)

Dans notre projet, nous utilisons testé un modèle **CRF (Conditional Random Fields)** pour la classification des tokens dans les phrases. Ce modèle est particulièrement efficace pour les tâches de **Reconnaissance d'Entités Nommées (NER)**, où chaque mot ou groupe de mots doit être classé dans une catégorie spécifique, comme **lieu de départ (DEP)**, **lieu d'arrivée (ARR)**, ou **autres mots (O)**.

Le modèle **CRF** est un modèle probabiliste utilisé pour l'étiquetage de séquences. Contrairement aux modèles classiques qui analysent chaque mot de manière indépendante, CRF prend en compte le contexte en considérant la relation entre les mots d'une même phrase. Cette approche est essentielle pour des tâches comme la nôtre, où la signification d'un mot peut dépendre de sa position et des mots qui l'entourent. Par exemple, dans la phrase *"Voyage de Paris à Lyon"*, le mot "Paris" doit être reconnu comme un point de départ, tandis que "Lyon" est une destination. Un modèle qui analyserait ces mots sans contexte pourrait les identifier comme des lieux sans préciser leur rôle exact. Grâce à CRF, le modèle est capable d'exploiter la structure de la phrase pour améliorer la précision de ses prédictions.

Préparation des données

Avant d'entraîner le modèle, il est nécessaire de transformer les **phrases en séquences exploitables**. Chaque mot est associé à une **série de caractéristiques (features)** qui permettent au modèle d'apprendre. Ces caractéristiques incluent :

- Le mot lui-même et sa version en **minuscules**
- La **présence de majuscules**
- La position du mot dans la phrase (**début ou fin de phrase**)
- La nature des mots **avant et après**

En intégrant ces informations, CRF peut mieux comprendre les relations entre les mots et améliorer la classification.

Entraînement du modèle

L'entraînement du modèle se fait sur un jeu de données divisé en trois parties :

1. Ensemble d'apprentissage (training set)
2. Ensemble de validation (validation set)
3. Ensemble de test (test set)

Nous avons entraîné notre modèle CRF sur 20 époques. À chaque étape, il ajuste ses poids pour améliorer ses prédictions. Les résultats montrent une grande stabilité, avec un F1-score de 98,46% dès la première époque, conservé jusqu'à la fin de l'entraînement.

Evaluation et Performance

	precision	recall	f1-score	support
ARR	0.97	0.96	0.96	17933
DEP	0.98	0.98	0.98	17953
O	0.99	0.99	0.99	98169
accuracy			0.98	134055
macro avg	0.98	0.98	0.98	134055
weighted avg	0.98	0.98	0.98	134055

Sur le jeu de test, le modèle atteint une précision remarquable :

- 98% de F1-score pour les lieux de départ (DEP)
- 96% de F1-score pour les lieux d'arrivée (ARR)
- 99% de F1-score pour les autres mots (O)

Au total, notre modèle affiche une précision de 98% sur 134 055 tokens, démontrant son efficacité et sa fiabilité pour la classification des tokens dans les phrases.

L'approche basée sur **CRF** est particulièrement bien adaptée à notre projet. Elle permet de prendre en compte **les dépendances entre les mots** tout en exploitant **des informations contextuelles** pour affiner les prédictions. Contrairement aux modèles plus simples, qui peuvent commettre des erreurs en classant un mot uniquement en fonction de sa forme, CRF analyse **l'ensemble de la phrase** et les relations entre les tokens pour offrir une **compréhension plus fine et plus précise** du texte.

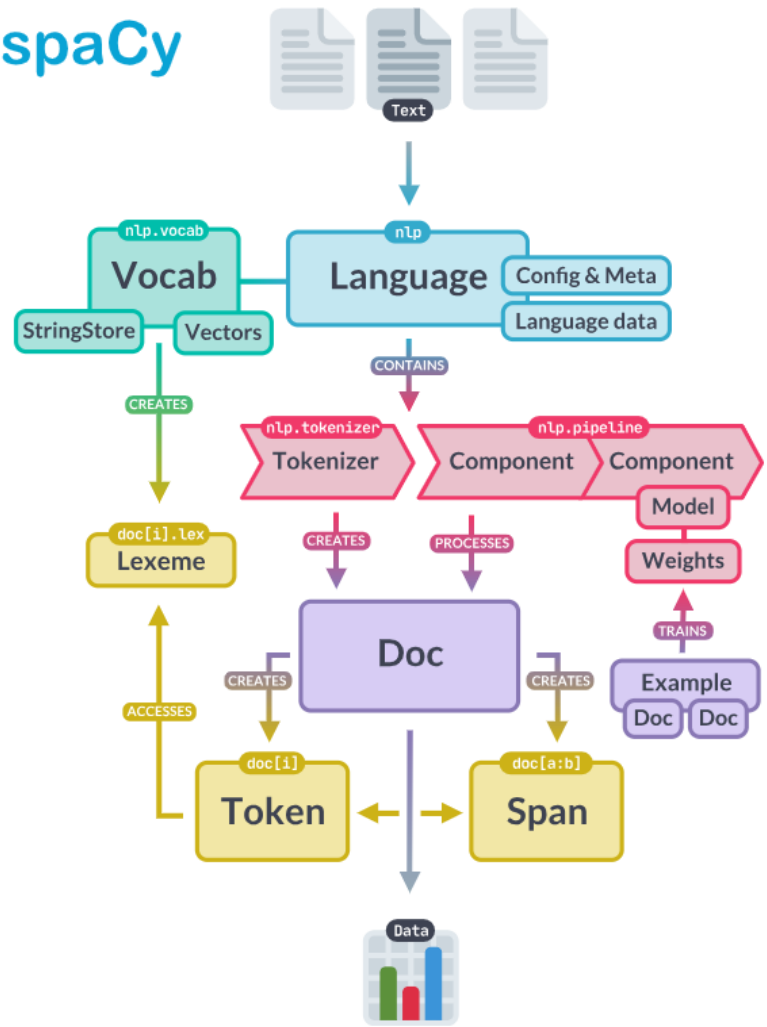
Grâce à cette méthode, nous pouvons **automatiser l'extraction d'informations clés** dans des descriptions d'itinéraires ou de voyages avec **une très haute précision**, rendant notre modèle particulièrement robuste et efficace pour ce type de tâche.

5.3.2 Modèle SpaCy

spaCy est une bibliothèque de **traitement automatique du langage naturel (NLP)** conçue pour offrir des performances rapides et efficaces. Contrairement à d'autres bibliothèques comme **NLTK**, qui sont davantage axées sur l'expérimentation académique, spaCy est optimisé pour des applications industrielles et propose des **modèles pré-entraînés** qui permettent d'effectuer diverses tâches de NLP :

- **Segmentation des phrases** : séparation d'un texte en phrases distinctes.
- **Tokenization** : découpage d'un texte en mots ou en sous-mots appelés **tokens**.

- **Étiquetage grammatical (POS tagging)** : identification des catégories grammaticales de chaque mot (nom, verbe, adjectif, etc.).
- **Reconnaissance des entités nommées (NER - Named Entity Recognition)** : détection et classification des entités comme les noms de lieux, de personnes, ou d'organisations.
- **Analyse de dépendances** : détermination des relations syntaxiques entre les mots d'une phrase.
- L'un des grands avantages de spaCy est son **efficacité computationnelle** : il est conçu pour être rapide et optimisé, ce qui en fait un excellent choix pour des applications en production.



Dans notre projet, nous utilisons spaCy pour effectuer une tâche de **classification des tokens** dans des phrases décrivant des trajets. L'objectif est d'identifier **les points de départ (DEP) et les points d'arrivée (ARR)** en fonction du contexte.

Par exemple, considérons la phrase :

"Voyage de Paris à Lyon"

Dans ce cas, nous voulons que le modèle reconnaisse **"Paris" comme un point de départ (DEP)** et **"Lyon" comme une destination (ARR)**.

Le problème est que ces mots, pris individuellement, sont simplement des noms de villes. Un modèle naïf qui analyse chaque mot séparément ne pourrait pas déterminer leur rôle précis dans la phrase. C'est là que **spaCy et l'apprentissage supervisé entrent en jeu**.

Nous suivons plusieurs étapes pour entraîner notre modèle :

1. **Chargement du modèle spaCy** : Nous utilisons un modèle pré-entraîné en français (**fr_core_news_sm, md ou lg**), qui comprend déjà une compréhension de la langue.
2. **Ajout de nouvelles étiquettes** : Nous modifions le pipeline du modèle en ajoutant deux nouvelles catégories d'entités : **DEP (Départ)** et **ARR (Arrivée)**.
3. **Préparation des données d'entraînement** : Nous utilisons un fichier CSV contenant des phrases annotées avec des tokens et des étiquettes NER.
4. **Alignement et validation des entités** : Nous nous assurons que les entités identifiées correspondent bien aux mots de la phrase et sont correctement positionnées.
5. **Entraînement du modèle** : Le modèle est ajusté sur notre ensemble de données pour apprendre à identifier correctement les entités DEP et ARR.
6. **Évaluation des performances** : Après l'entraînement, nous évaluons le modèle sur un jeu de test indépendant et analysons des métriques comme le **F1-score, la précision et le rappel**.

5.3.2.1 Comparaison des modèles spaCy : sm, md et lg

spaCy propose trois tailles de modèles pour le français :

1. **fr_core_news_sm** (small) : modèle léger et rapide, mais moins précis.
2. **fr_core_news_md** (medium) : modèle intermédiaire, qui offre un bon compromis entre rapidité et précision.
3. **fr_core_news_lg** (large) : modèle le plus avancé, avec un vocabulaire plus riche et des embeddings de mots plus détaillés.

Nous avons testé ces trois modèles sur notre tâche et comparé leurs performances.

5.3.2.1.1 Performance des modèles

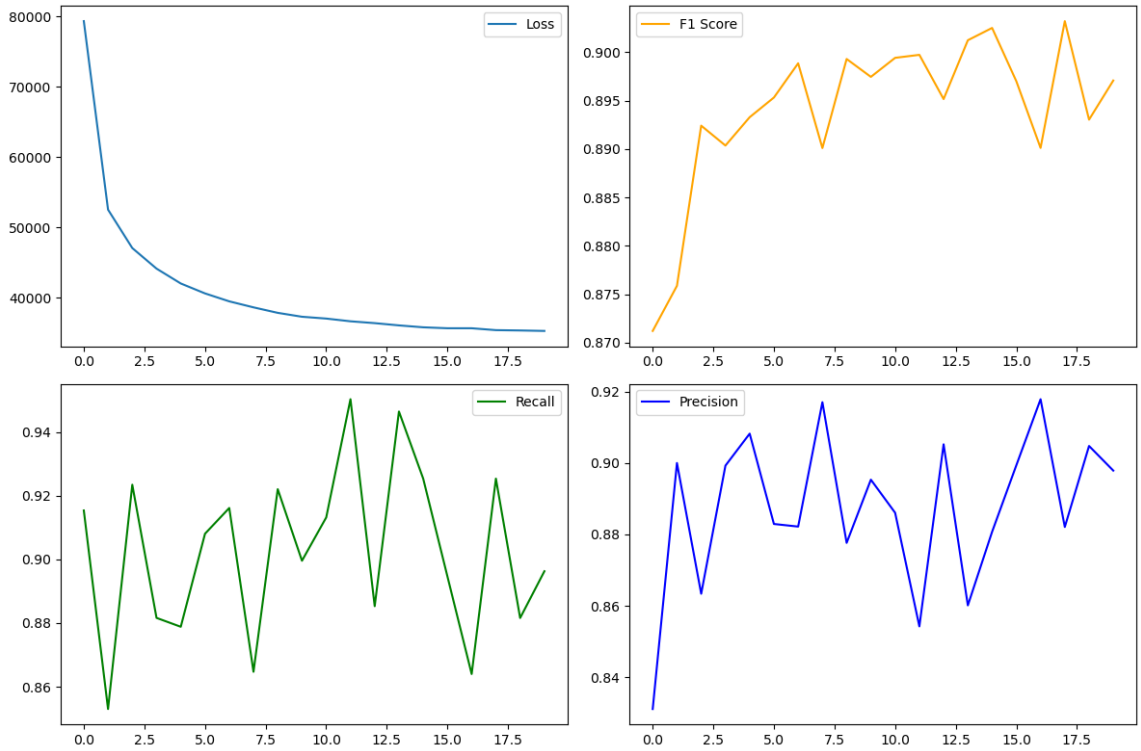
F1-score, précision et rappel

Modèle	Validation F1 Score	Précision	Rappel	Test F1 Score	Précision	Rappel
Sm	0.8645	0.8756	0.8537	0.8650	0.8760	0.8543
Md	0.8753	0.8740	0.8766	0.8755	0.8742	0.8768
Lg	0.8655	0.8757	0.8555	0.8661	0.8770	0.8556

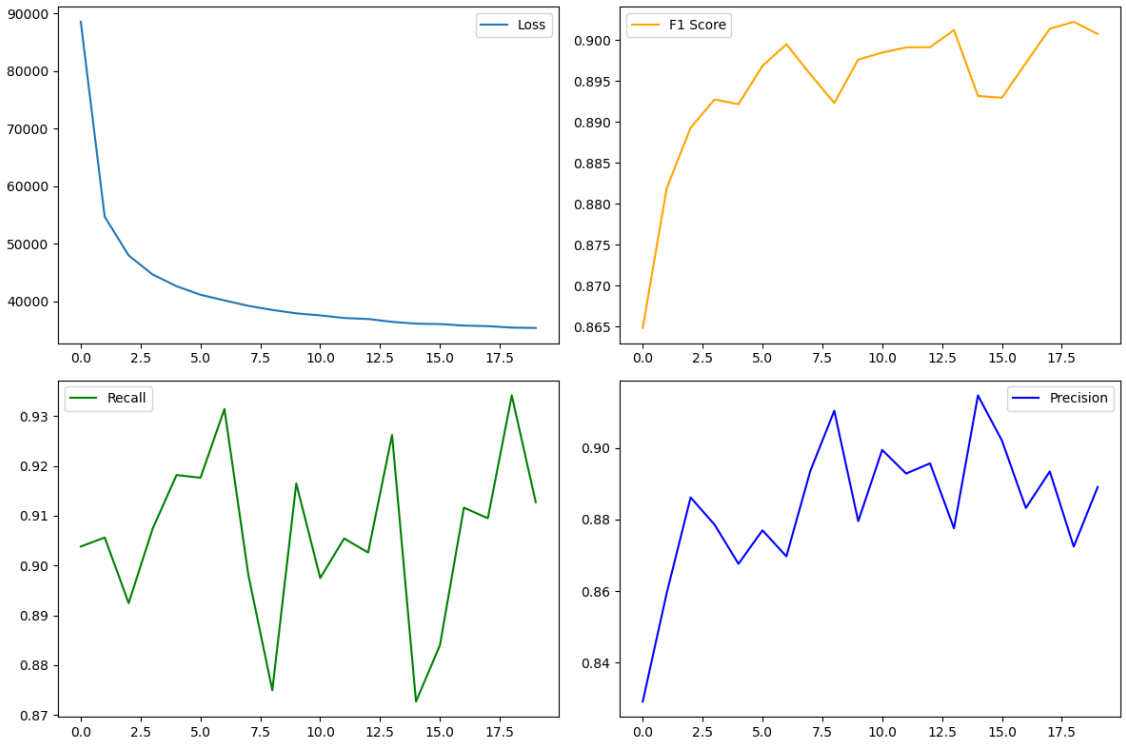
- **fr_core_news_md a les meilleures performances globales**, avec un **F1-score légèrement plus élevé**.
- **fr_core_news_sm est un peu en retrait**, mais reste performant malgré sa petite taille.
- **fr_core_news_lg, bien qu'étant le plus grand modèle, ne surpasse pas md de manière significative**, ce qui remet en question son utilité pour cette tâche.

5.3.2.1.2 Analyse des courbes d'apprentissage

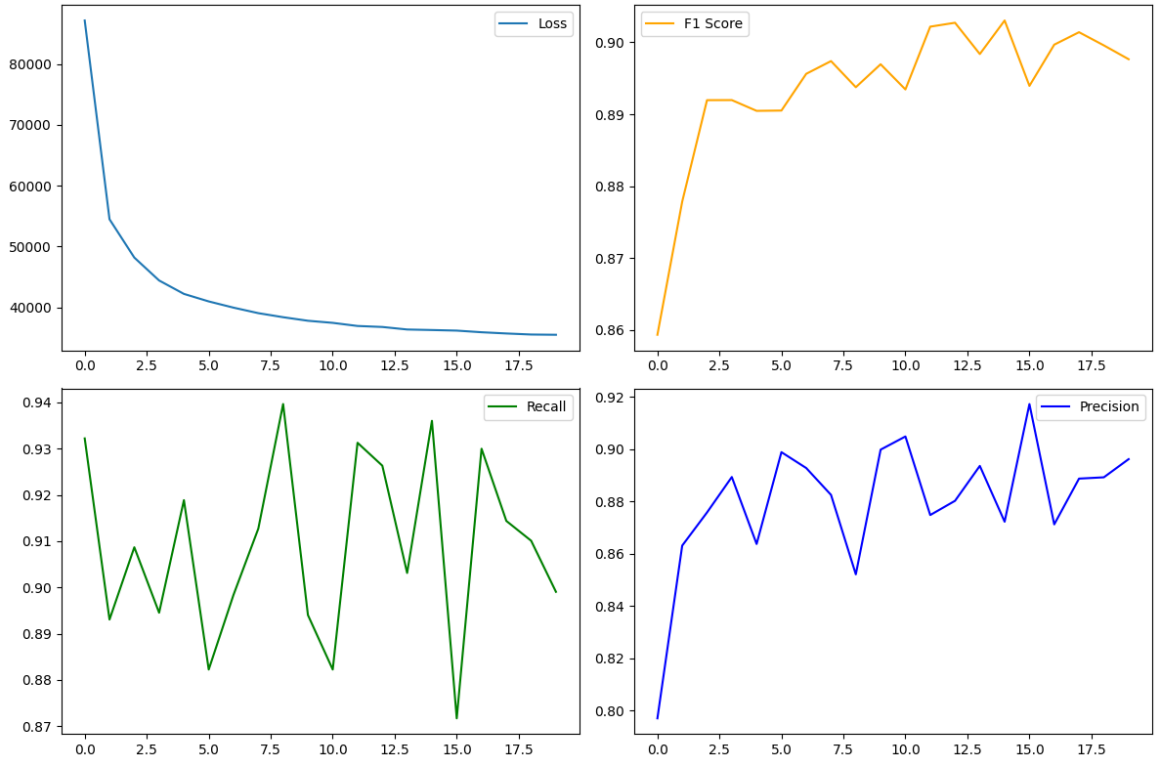
Pour SM :



Pour MD :



Pour LG :



Courbe de loss (perte d'apprentissage)

- La loss diminue progressivement, ce qui est un signe que le modèle apprend correctement.
- À la fin de l'entraînement, **une forte diminution se produit**, ce qui pourrait indiquer un surajustement aux données d'entraînement.

Courbes de précision, rappel et F1-score

- **F1-score** :
  - Se stabilise entre **0.88 et 0.90** pour les modèles **sm et md**.
  - Pour **md et lg**, il reste légèrement supérieur à **0.895** en fin d'entraînement.
- **Précision** :
  - Atteint **0.90 au début**, puis oscille entre **0.88 et 0.90** pour tous les modèles.
  - Le modèle md semble être **plus stable**, avec moins de fluctuations.
- **Rappel** :
  - Pour le modèle md, il est **plus élevé (0.91-0.93)**, ce qui signifie qu'il réussit mieux à détecter les entités sans en oublier trop.
  - sm et lg sont légèrement inférieurs, bien que lg soit proche de md.

Nous concluons donc que :

- **Le modèle md est le plus adapté**, car il offre un excellent compromis entre **précision, rappel et stabilité**. Il est suffisamment puissant pour capturer le contexte des phrases, sans être aussi lourd que lg.
- **Le modèle sm reste une bonne option** si l'on privilégie la rapidité et que l'on accepte une légère baisse de précision.
- **Le modèle lg n'apporte pas une amélioration significative** par rapport à md, bien qu'il soit plus coûteux en ressources.

Pourquoi md est-il le meilleur choix ?

1. **Bon équilibre entre performance et rapidité** :
   - Il offre une **meilleure précision et un meilleur rappel que sm**, sans être aussi lourd que lg.
2. **Plus stable sur la durée** :
   - Ses courbes d'apprentissage montrent une **bonne convergence**, avec peu de fluctuations.
3. **Efficacité sur notre tâche spécifique** :
   - Il capture mieux les relations entre les mots et attribue correctement les étiquettes DEP et ARR.

Alors en ce qui concerne la **classification des tokens** avec spaCy dans des descriptions de trajets, nous utilisions **fr_core_news_md** comme modèle optimal. Il offre **un excellent compromis entre précision et performance**, garantissant un bon niveau d'exactitude tout en restant suffisamment rapide pour une utilisation pratique.

5.3.3 Modèle BERT

BERT (Bidirectional Encoder Representations from Transformers) est un modèle de traitement du langage naturel (NLP) développé par Google en 2018. Il repose sur l'architecture Transformer et se distingue par sa capacité à analyser les mots dans leur contexte bidirectionnel, c'est-à-dire en tenant compte à la fois des mots précédents et suivants dans une phrase. Cette approche permet à BERT de mieux comprendre le sens des mots et d'améliorer les performances sur diverses tâches NLP comme l'analyse des sentiments, la traduction, ou encore la reconnaissance d'entités nommées (NER).

5.3.3.1 Modèle DistilBERT

DistilBERT est une version allégée de BERT, développée pour être plus rapide et moins gourmande en ressources tout en conservant la plupart des performances du modèle original. Il est entraîné avec une méthode appelée "knowledge distillation", qui consiste à apprendre à partir d'un modèle plus grand (BERT) pour réduire sa taille et ses besoins en calculs tout en maintenant une bonne précision. DistilBERT est particulièrement adapté aux applications où la vitesse et l'efficacité sont cruciales.

Dans notre projet, nous utilisons DistilBERT pour une tâche de reconnaissance d'entités nommées (NER) spécifique au domaine des itinéraires de voyage. L'objectif est d'identifier les villes de départ (B-DEP, I-DEP) et d'arrivée (B-ARR, I-ARR) dans les phrases des utilisateurs.

1. Préparation des données

Les données utilisées sont stockées dans un fichier CSV, où chaque phrase est annotée avec des tokens (mots ou groupes de mots) et des étiquettes correspondantes (O pour les mots neutres, B-DEP/I-DEP pour les lieux de départ et B-ARR/I-ARR pour les lieux d'arrivée). Avant d'entraîner notre modèle, nous devons :

- Vérifier et filtrer les données incorrectes.
- Convertir les annotations sous un format compréhensible par le modèle.
- Séparer les données en ensembles d'entraînement, de validation et de test.

2. Tokenisation et alignement des labels

Nous utilisons le tokenizer de DistilBERT pour transformer les phrases en séquences de tokens adaptées au modèle. Un défi ici est que certains mots peuvent être divisés en sous-mots (ex: "Strasbourg" peut être séparé en "Stra" et "sbourg"). Nous devons aligner correctement les étiquettes (ner_tags) avec ces tokens pour éviter des erreurs d'apprentissage.

3. Entraînement du modèle

Nous utilisons un ensemble de paramètres spécifiques pour optimiser l'apprentissage du modèle :

- **Batch size** : 4 (nombre d'exemples traités à la fois)
- **Nombre d'époques** : 3 (nombre de passages sur l'ensemble des données)
- **Taux d'apprentissage** : 2e-5 (contrôle la vitesse d'ajustement des poids du modèle)
- **Utilisation du GPU** : Activation de cuda si disponible pour accélérer l'entraînement

Nous utilisons l'algorithme d'optimisation Adam et une pondération des gradients pour améliorer la stabilité de l'entraînement.

4. Évaluation des performances

Après l'entraînement, nous évaluons le modèle en utilisant plusieurs métriques :

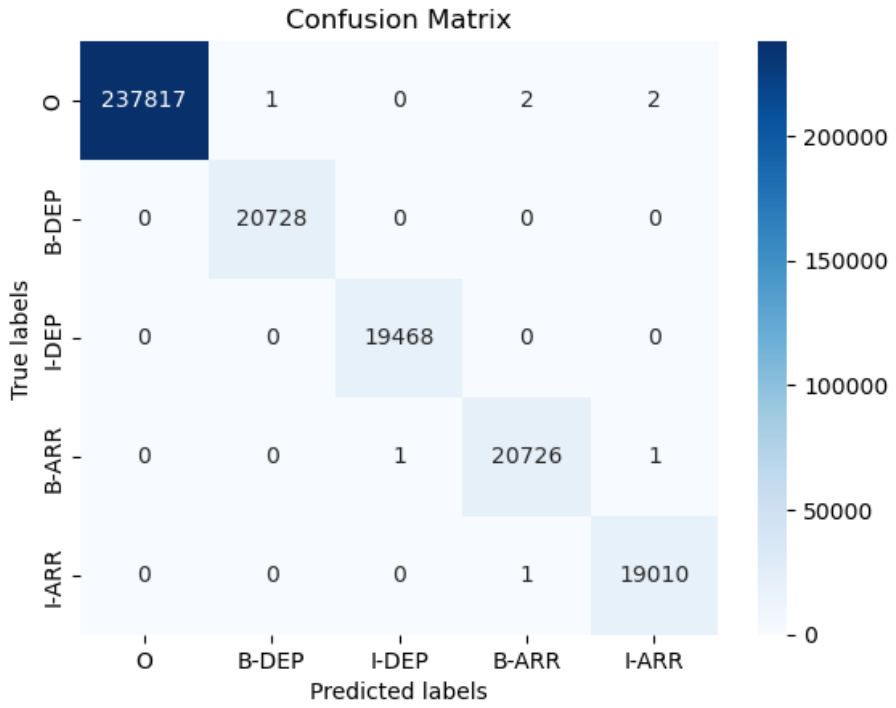
- **Précision** (Precision) : Proportion des prédictions correctes parmi les entités détectées.
- **Rappel** (Recall) : Proportion des entités réelles correctement identifiées par le modèle.
- **F1-score** : Moyenne harmonique de la précision et du rappel.
- **Exactitude globale** (Accuracy) : Pourcentage des tokens correctement classés.

Les résultats obtenus montrent des performances très élevées du modèle sur la tâche de reconnaissance d'entités nommées (NER). Voici une interprétation détaillée des métriques :

1. **Précision (Precision) : 0.9988 (99.88%)**
   La précision mesure la proportion de prédictions correctes parmi toutes les prédictions positives faites par le modèle. Une précision de 99.88% signifie que presque toutes les entités prédites par le modèle sont correctes.
2. **Rappel (Recall) : 0.9989 (99.89%)**
   Le rappel mesure la proportion d'entités réelles correctement identifiées par le modèle. Un rappel de 99.89% indique que le modèle a réussi à détecter presque toutes les entités présentes dans les données.
3. **F1-score : 0.9989 (99.89%)**
   Le F1-score est une moyenne harmonique de la précision et du rappel. Un score de 99.89% montre que le modèle atteint un équilibre excellent entre la précision et le rappel.
4. **Exactitude (Accuracy) : 0.9998 (99.98%)**
   L'exactitude mesure la proportion totale de prédictions correctes (entités et non-entités) par rapport à l'ensemble des prédictions. Une exactitude de 99.98% signifie que le modèle fait très peu d'erreurs dans l'ensemble de la tâche.

5. Analyse avec Matrice de confusion

Nous utilisons une matrice de confusion pour visualiser les erreurs du modèle. Chaque ligne représente une étiquette réelle et chaque colonne une étiquette prédite.



Les valeurs élevées sur la diagonale montrent que le modèle fait très peu d'erreurs, mais il y a quelques confusions entre B-ARR et O, ainsi qu'entre B-ARR et I-ARR.

Grâce à DistilBERT, nous avons pu entraîner un modèle précis et efficace pour la reconnaissance d'entités nommées dans le contexte des itinéraires de voyage. Ses performances sont très élevées avec un F1-score de près de 99.98%. L'utilisation de la matrice de confusion et des métriques nous a permis d'affiner et de valider le modèle pour une meilleure compréhension et extraction des entités clés dans les phrases des utilisateurs.

5.3.3.2 Modèle CamemBERT

CamemBERT est un modèle de langage basé sur BERT, spécialement conçu pour le français. Il a été entraîné sur un large corpus de textes en français, ce qui lui permet de comprendre et de traiter efficacement la langue française dans différentes tâches de traitement du langage naturel (NLP), comme la classification de texte, l'analyse des sentiments, et la reconnaissance d'entités nommées (NER).

CamemBERT fonctionne en utilisant la méthode de l'apprentissage auto-supervisé : il est pré-entraîné sur une grande quantité de texte en masquant certains mots et en apprenant à les prédire. Cela lui permet de comprendre le contexte et la signification des mots dans une phrase.

Dans notre projet, nous utilisons CamemBERT pour la reconnaissance d'entités nommées (NER), une tâche qui consiste à identifier des éléments spécifiques dans un texte, comme des lieux de départ et d'arrivée dans des ordres de voyage.

1. Chargement et préparation des données

Nous avons un fichier CSV contenant des tokens (mots ou groupes de mots) et leurs étiquettes associées. Les étiquettes suivent le format standard BIO :

- **O** : Mot n'appartenant à aucune entité
- **B-DEP** : Début d'une entité de type départ
- **I-DEP** : Suite d'une entité de type départ
- **B-ARR** : Début d'une entité de type arrivée
- **I-ARR** : Suite d'une entité de type arrivée

Nous chargeons ces données et supprimons les lignes invalides (qui ne contiennent pas de tokens ou d'étiquettes valides). Ensuite, nous divisons notre jeu de données en trois parties :

- **Train** (60%) : Utilisé pour entraîner le modèle
- **Validation** (15%) : Utilisé pour ajuster les paramètres
- **Test** (25%) : Utilisé pour évaluer les performances finales

2. Tokenisation et alignement des étiquettes

CamemBERT nécessite que le texte soit transformé en tokens. Nous utilisons son tokenizer pour diviser les phrases en sous-mots tout en conservant les correspondances avec les étiquettes initiales. Les tokens spéciaux comme [CLS] et [SEP] sont ignorés dans le calcul des erreurs.

3. Entraînement du modèle

Nous utilisons un modèle pré-entraîné CamemBERT que nous affinons sur notre dataset. Le modèle apprend à prédire les étiquettes des tokens en minimisant l'erreur de classification. Les hyperparamètres principaux sont :

- **Taux d'apprentissage** : 2e-5
- **Taille du batch** : 16
- **Nombre d'époques** : 3
- **Pondération des poids** : 0.01

4. Évaluation du modèle

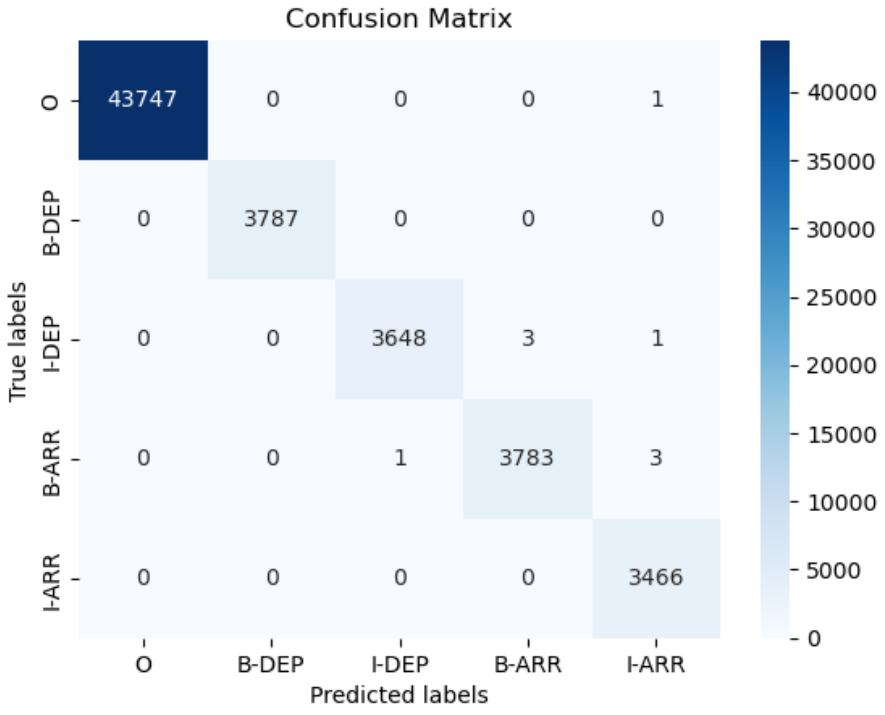
Après l'entraînement, nous évaluons les performances du modèle en mesurant plusieurs métriques :

- **Précision** : 99.88%
- **Rappel** : 99.89%
- **F1-score** : 99.88%
- **Exactitude** : 99.98%

Ces résultats montrent que le modèle est extrêmement performant et fait très peu d'erreurs.

5. Matrice de confusion

Nous visualisons les performances avec une matrice de confusion qui compare les prédictions du modèle aux valeurs réelles.



On observe que la grande majorité des prédictions sont correctes. Les erreurs sont rares et concernent principalement la confusion entre **I-DEP** et **B-ARR**, mais elles sont négligeables.

CamemBERT est un modèle puissant pour le NLP en français. Son adaptation à la tâche de reconnaissance d'entités nommées dans notre projet a donné des résultats très précis. Grâce à un bon pré-traitement des données et une tokenisation bien adaptée, notre modèle atteint une performance exceptionnelle, avec une exactitude de 99.98%.

Les prochaines étapes pourraient inclure l'expérimentation avec d'autres architectures comme CamemBERT large ou la combinaison avec des modèles de post-traitement pour affiner encore davantage les prédictions.

5.4 Analyse des performances matérielles et Impact énergétique

5.4.1 Présentation des ressources matérielles

Pour les expérimentations menées dans ce projet, l'entraînement des modèles de traitement du langage naturel (NLP) a été réalisé sur une machine personnelle avec les spécifications suivantes :

- **Processeur** : Intel® Core™ i7-8565U (4 cœurs, 8 threads, fréquence max. de 1,8 GHz).
- **Mémoire RAM** : 8 Go (2 modules de 4 Go, fréquence 2400 MHz, fabricant : Samsung).
- **Système d'exploitation** : Windows 11 Famille (Version 10.0.26100, build 26100).

Les performances de la machine ont été particulièrement sollicitées lors de l'entraînement des modèles de machine learning, notamment pour les architectures plus lourdes comme **Camembert** et **GPT-2**, qui nécessitent une consommation élevée en CPU et en RAM. Les ressources matérielles de l'ordinateur se sont avérées être un facteur limitant pour l'entraînement de certains modèles plus complexes, en raison de la charge élevée qu'ils engendrent sur le système.

5.4.2 Analyse de la consommation CPU/RAM

Classification par Tokens

Modèle	Méthode / Architecture	Utilisation CPU (%)	Utilisation RAM (%)	Temps d'entraînement estimé	Remarques sur l'empreinte
spaCy	Pipeline NLP classique	~85%	~65%	~8 heures	Bon compromis, mais le long run en 8h augmente la consommation.
Camembert	Transformer (modèle de type BERT)	~95%	~85%	~13 heures	Très gourmand : CPU et RAM quasiment à leur max, coût élevé.
DistilBERT	Version allégée de BERT	~90%	~80%	~10 heures	Moins lourd que Camembert, mais toujours une forte charge.
Conditional Random Fields (CRF)	Modèle statistique pour séquence	~60%	~40%	~6 heures	Relativement léger, impact faible sur le coût énergétique.

Classification par Textes (détection de langue)

Modèle	Méthode / Architecture	Utilisation CPU (%)	Utilisation RAM (%)	Temps d'entraînement estimé	Remarques sur l'empreinte
Naive Bayes	Méthode statistique classique	~40%	~30%	~1 heure	Très léger, faible impact énergétique.
CNN	Réseau de neurones convolutionnel	~95%	~80%	~8 heures	Consommation importante sur CPU et RAM.
LSTM	Réseau de neurones récurrent (LSTM)	~95%	~80%	~10 heures	Long entraînement, similaire aux CNN en coût.
SVM	Machine à vecteurs de support	~85%	~70%	~6 heures	Modéré, moins gourmand que les NN profonds.

Classification par Textes (détection d'intention)

Modèle	Méthode / Architecture	Utilisation CPU (%)	Utilisation RAM (%)	Temps d'entraînement estimé	Remarques sur l'empreinte
Bag of Words	Modèle basé sur la fréquence des mots	~50%	~40%	~1 heure	Très léger et rapide à entraîner.
GPT-2	Transformer de génération (adapté ici)	~100%	~95%	……..	Très gourmand en ressources, empreinte carbone élevée.
GRU	Réseau récurrent avec GRU	~95%	~85%	~15 heures	Charge similaire à d'autres NN, assez intensif.
SAN	Réseau à auto-attention (Self-Attention Network)	~95%	~90%	~16 heures	Comparable aux GRU en termes de charge.

5.4.3 Comparaison des temps d'entraînement

Les temps d'entraînement des modèles varient considérablement selon leur complexité et les ressources nécessaires.

- **Modèles de classification par tokens (spaCy, Camembert, DistilBERT, CRF)** : Les modèles basés sur des architectures complexes comme **Camembert** et **DistilBERT** peuvent prendre de 8 à 13 heures d'entraînement, avec des consommations de ressources CPU et RAM particulièrement élevées. En revanche, **spaCy** est un compromis plus léger, mais l'entraînement reste long, à environ 8 heures.
- **Modèles de classification par textes (Naive Bayes, CNN, LSTM, SVM)** : Les modèles comme **Naive Bayes** sont très rapides et peu gourmands (environ 1 heure), tandis que les **CNN** et **LSTM** nécessitent entre 6 et 10 heures, avec une consommation élevée des ressources. **SVM** est modéré en termes de ressources et de temps d'entraînement.
- **Modèles de détection d'intention (Bag of Words, GPT-2, GRU, SAN)** : **Bag of Words** est extrêmement rapide, ne nécessitant qu'une heure. En revanche, des modèles comme **GPT-2**, **GRU**, et **SAN** sont très gourmands en ressources, avec des temps d'entraînement de 15 à 16 heures.

L'entraînement des modèles sur un PC portable avec des ressources limitées montre que certains modèles, notamment les plus complexes comme **GPT-2** et **Camembert**, ont des empreintes matérielles et énergétiques considérables, ce qui impacte significativement les temps d'entraînement. Les modèles plus légers comme **Naive Bayes**, **Bag of Words**, et **Conditional Random Fields (CRF)** sont bien adaptés aux ressources matérielles disponibles, offrant un compromis

acceptable entre performance et temps d'exécution. Les modèles plus lourds, bien qu'efficaces, nécessitent des machines plus puissantes ou des optimisations supplémentaires pour réduire l'empreinte énergétique et améliorer l'efficacité des calculs.

6 . Choix Finaux et Modèle Retenu

6.1 Choix du Modèle Final

Après avoir exploré plusieurs approches et évalué différentes techniques en termes de précision, de performance et de complexité, nous avons pris les décisions suivantes pour chaque composant du système de traitement automatique du langage naturel (NLP) :

- **Détection de la langue :** Nous avons choisi d'utiliser **Naïve Bayes**, qui s'est révélé particulièrement efficace pour la tâche de classification des langues. Ce modèle a non seulement fait preuve d'une rapidité d'exécution impressionnante, mais aussi d'une précision satisfaisante pour distinguer le français des autres langues. Son approche probabiliste et sa simplicité en font un excellent choix pour ce type de problème où les critères sont principalement la rapidité et l'efficacité.
- **Détection de l'intention (ordre de voyage ou non) :** Pour cette tâche, nous avons opté pour **Bag of Words (Bigram) + Naïve Bayes**. Cette approche, qui utilise des modèles statistiques de bigrammes (paires de mots), a montré une grande efficacité dans la classification des phrases en tant qu'ordres de voyage ou non. En combinant la représentation simple des mots avec les bigrammes pour capturer des relations contextuelles locales, et en appliquant Naïve Bayes pour la classification, nous avons trouvé un bon compromis entre précision et performance. Ce modèle est particulièrement adapté pour ce genre de tâche, où la capture des relations entre les mots voisins et l'analyse probabiliste des phrases est essentielle, tout en restant relativement léger et rapide à l'exécution.
- **Reconnaissance des entités nommées (NER) :** Pour la tâche d'extraction d'entités telles que les villes de départ et d'arrivée, nous avons opté pour **SpaCy md**. Ce modèle présente un compromis optimal entre la performance et la précision. SpaCy est réputé pour sa capacité à traiter de grandes quantités de texte tout en restant rapide et efficace. La version "medium" de SpaCy a montré une meilleure capacité à identifier les entités pertinentes, telles que les noms de lieux, tout en maintenant des performances acceptables dans un cadre de projet comme le nôtre.

Ce choix final est le fruit d'une analyse approfondie des résultats obtenus lors de nos évaluations. En combinant des modèles simples mais efficaces pour les tâches de détection de langue et d'intention avec un modèle plus complexe mais optimisé pour la reconnaissance d'entités, nous avons mis en place un système cohérent, performant et évolutif.

Ce choix final est le fruit d'une analyse approfondie des résultats obtenus lors de nos évaluations. En combinant des modèles simples mais efficaces pour les tâches de détection de langue et

d’intention avec un modèle plus complexe mais optimisé pour la reconnaissance d’entités, nous avons mis en place un système cohérent, performant et évolutif.

6.2 Conclusion

Ce projet a permis de concevoir un système de traitement du langage naturel performant capable de comprendre et d’interpréter des ordres de voyage formulés en français. À travers ce processus, nous avons exploré des concepts clés du NLP, y compris la détection de la langue, l’identification de l’intention derrière les phrases et l’extraction d’informations cruciales comme les villes de départ et d’arrivée.

Les tests réalisés sur les différents modèles ont validé l’approche adoptée, notamment en termes de précision et de rapidité, avec des résultats prometteurs dans la classification des intentions et la reconnaissance des entités. Le système a ainsi montré sa capacité à discriminer efficacement les demandes pertinentes d’un ordre de voyage tout en offrant une réponse structurée qui pourra, à terme, alimenter un algorithme d'optimisation de chemin pour un parcours optimal.

En termes d'intégration, la combinaison de ces modules avec un algorithme de recherche de chemin a permis de proposer une solution fonctionnelle pour la résolution des ordres de voyage, capable d’afficher des itinéraires adaptés aux besoins des utilisateurs. Le système a été testé et fonctionne de manière fiable, avec une prise en charge complète des scénarios les plus courants.

Bien que nous ayons obtenu de bons résultats, plusieurs pistes d'amélioration subsistent. Premièrement, le modèle de classification de l’intention pourrait être affiné afin d’augmenter sa précision dans la détection des ordres de voyage. De plus, l'ajout d’un module de correction orthographique et de gestion des fautes de frappe permettrait de rendre le système encore plus robuste face aux entrées réelles et variées des utilisateurs. Enfin, une extension du système pour gérer d'autres modes de transport, en complément des trajets en train, pourrait être une évolution intéressante, bien que cela ajoute une complexité considérable au problème.

En somme, ce projet a été une expérience enrichissante, tant sur le plan technique que pratique, et nous a permis d’approfondir nos connaissances en NLP, en modélisation de graphes et en intégration de systèmes complexes. Nous avons développé une solution applicative qui répond à un besoin réel et qui pourrait être facilement améliorée et adaptée à des cas d'usage plus complexes à l'avenir.