Documentation Technique Détaillée

# Notebook: detection\_intention.ipynb

Classification d'Intentions et Named Entity Recognition  
Utilisant CamemBERT et Transformers  
  
*Avec explications ligne par ligne et justifications des choix techniques*

# **Table des Matières**

* 1. Introduction et Vue d'Ensemble
* 2. Cellule 1 : En-tête et Présentation
* 3. Cellule 2 : Vérification GPU et Installation
* 4. Cellule 3 : Montage Google Drive
* 5. Cellule 4 : Chargement des Datasets
* 6. Cellule 5 : Preprocessing et Parsing des Entités
* 7. Cellule 6 : Préparation HuggingFace Datasets
* 8. Cellule 7 : Fine-Tuning Intent Classification
* 9. Cellule 8 : Évaluation et Post-Processing
* 10. Cellule 9 : Préparation Dataset NER
* 11. Cellule 10 : Fine-Tuning NER
* 12. Cellule 11 : Évaluation NER
* 13. Cellule 12 : Pipeline d'Inférence Complet
* 14. Annexe : Glossaire et Références

# **1. Introduction et Vue d'Ensemble**

Ce notebook implémente un système complet de traitement du langage naturel (NLP) pour la compréhension et l'extraction d'informations à partir de requêtes de voyage en français. Il combine deux tâches principales :

* Classification d'Intention : Détermine si une phrase concerne un voyage (TRIP), ne concerne pas un voyage (NOT\_TRIP), est incompréhensible (UNKNOWN), ou est dans une langue étrangère (NOT\_FRENCH)
* Named Entity Recognition (NER) : Extrait les villes de départ (Departure) et de destination (Destination) des requêtes de voyage

## **Pourquoi CamemBERT ?**

CamemBERT est un modèle de langage pré-entraîné spécifiquement pour le français, basé sur l'architecture RoBERTa. Voici les raisons de ce choix :

* Entraîné sur 138 Go de texte français (OSCAR corpus)
* Comprend les nuances, la sémantique et la syntaxe du français
* Performances supérieures à mBERT sur les tâches françaises
* Compatible avec l'écosystème HuggingFace Transformers

## **Architecture Globale**

Le système suit un pipeline en 3 étapes :

* Étape 1 : Fine-tuning d'un modèle de classification d'intention (4 classes)
* Étape 2 : Fine-tuning d'un modèle NER pour extraction d'entités
* Étape 3 : Combinaison des deux modèles pour inférence complète

# **2. Cellule 1 : En-tête et Présentation (Markdown)**

# Detection d'Intention avec CamemBERT (Transformers)  
  
\*\*Notebook optimise pour Google Colab GPU avec Transformers\*\*  
  
Ce notebook utilise CamemBERT pour :  
1. Classification d'intent (TRIP, NOT\_TRIP, UNKNOWN, NOT\_FRENCH)  
2. Named Entity Recognition (Departure, Destination)  
  
\*\*Ameliorations par rapport au baseline TF-IDF\*\* :  
- Comprehension semantique profonde du texte  
- Detection precise des langues etrangeres  
- NER contextuel (pas juste un gazetteer)  
- Fine-tuning sur donnees francaises

Cette cellule Markdown sert de documentation en en-tête du notebook.

### **Explication détaillée :**

* Titre principal : Indique l'utilisation de CamemBERT et Transformers
* Optimisation Colab : Précise que le notebook est conçu pour Google Colab avec GPU
* Les 4 classes d'intention : TRIP (voyage), NOT\_TRIP (pas de voyage), UNKNOWN (incompréhensible), NOT\_FRENCH (langue étrangère)
* NER : Extraction de 2 types d'entités (Departure et Destination)

**💡 Pourquoi mentionner TF-IDF:** TF-IDF (Term Frequency-Inverse Document Frequency) est une approche baseline classique qui ne comprend pas la sémantique. CamemBERT offre une compréhension contextuelle profonde.

# **3. Cellule 2 : Vérification GPU et Installation des Dépendances**

import torch  
  
print('='\*70)  
print('DETECTION INTENTION - Transformers + CamemBERT')  
print('='\*70)  
  
print(f'\nPyTorch version: {torch.\_\_version\_\_}')  
print(f'CUDA disponible: {torch.cuda.is\_available()}')

### **Ligne par ligne :**

Ligne 1 : import torch

* Importe PyTorch, le framework de deep learning utilisé par Transformers
* PyTorch gère les tenseurs, les calculs GPU et l'entraînement des réseaux de neurones

Lignes 3-5 : Bannière d'affichage

* Affiche un séparateur visuel pour mieux identifier le début du notebook

Ligne 7 : torch.\_\_version\_\_

* Affiche la version de PyTorch installée (important pour la compatibilité)

Ligne 8 : torch.cuda.is\_available()

* Vérifie si CUDA (API pour GPU NVIDIA) est disponible
* Retourne True si un GPU est détecté, False sinon

**💡 Importance du GPU:** L'entraînement de modèles Transformers est 10-50x plus rapide sur GPU. Sans GPU, CamemBERT-base peut prendre plusieurs heures à entraîner.

if torch.cuda.is\_available():  
 print(f'GPU: {torch.cuda.get\_device\_name(0)}')  
 print(f'Memoire GPU: {torch.cuda.get\_device\_properties(0).total\_memory / 1e9:.2f} GB')  
else:  
 print('ATTENTION: Pas de GPU detecte!')  
 print('Allez dans Runtime > Change runtime type > GPU')

### **Ligne par ligne :**

Ligne 1 : if torch.cuda.is\_available()

* Condition : Si GPU disponible, afficher les détails

Ligne 2 : torch.cuda.get\_device\_name(0)

* Récupère le nom du GPU (ex: 'Tesla T4', 'A100')
* L'argument 0 signifie le premier GPU (GPU index 0)

Ligne 3 : torch.cuda.get\_device\_properties(0).total\_memory

* Récupère la mémoire totale du GPU en octets
* Division par 1e9 (1 milliard) convertit en gigaoctets (Go)
* :.2f formate à 2 décimales (ex: 15.36 GB)

**⚠️ Pourquoi vérifier la mémoire GPU:** CamemBERT-base nécessite environ 2-4 GB de VRAM. Si la mémoire est insuffisante, réduire le batch size ou utiliser la précision mixte FP16.

!pip install -q transformers datasets evaluate seqeval accelerate scikit-learn langdetect  
print('\nInstallation terminee!')

### **Ligne par ligne :**

Ligne 1 : !pip install...

* Le '!' exécute une commande shell dans Colab/Jupyter
* L'option '-q' (quiet) réduit la verbosité de l'installation

Packages installés et leurs rôles :

* transformers : Bibliothèque HuggingFace pour charger et entraîner CamemBERT
* datasets : Gestion efficace des datasets (tokenization batched, caching)
* evaluate : Calcul des métriques (accuracy, F1, precision, recall)
* seqeval : Métriques spécialisées pour le NER (format BIO)
* accelerate : Optimisation GPU/TPU et entraînement distribué
* scikit-learn : Calcul des class weights et métriques
* langdetect : Détection automatique de la langue (post-processing)

**💡 Pourquoi ces bibliothèques spécifiques:** Transformers + Datasets forment l'écosystème standard pour le NLP moderne. Seqeval est conçu pour évaluer correctement les entités (pas juste token par token).

# **4. Cellule 3 : Montage Google Drive et Configuration**

from google.colab import drive  
import os  
  
# Monter Google Drive  
drive.mount('/content/drive')

### **Ligne par ligne :**

Ligne 1 : from google.colab import drive

* Importe le module pour monter Google Drive dans Colab
* Fonctionne uniquement dans Google Colab, pas en local

Ligne 2 : import os

* Module Python pour interactions système (chemins, variables d'environnement)

Ligne 5 : drive.mount('/content/drive')

* Monte Google Drive dans le système de fichiers Colab
* Après montage, les fichiers Drive sont accessibles via /content/drive/MyDrive/
* Demande une autorisation OAuth la première fois

**💡 Pourquoi utiliser Google Drive:** Les sessions Colab sont éphémères (durée maximale 12h). Google Drive permet de :  
• Stocker les datasets de manière persistante  
• Sauvegarder les modèles entraînés  
• Reprendre l'entraînement après déconnexion

# Desactiver WandB  
os.environ['WANDB\_DISABLED'] = 'true'  
  
# Chemins  
workdir = '/content/drive/MyDrive/dataset'  
os.makedirs(workdir, exist\_ok=True)  
  
print('Working directory:', workdir)  
print('WandB: DESACTIVE')

### **Ligne par ligne :**

Ligne 2 : os.environ['WANDB\_DISABLED'] = 'true'

* Définit une variable d'environnement pour désactiver Weights & Biases
* WandB est un outil de tracking d'expériences (perte, métriques, visualisations)
* Raison de la désactivation : Évite la demande de clé API et accélère l'exécution

Ligne 5 : workdir = '/content/drive/MyDrive/dataset'

* Définit le répertoire de travail où sont stockés les datasets et modèles
* /content/drive/MyDrive/ correspond à la racine de votre Google Drive

Ligne 6 : os.makedirs(workdir, exist\_ok=True)

* Crée le répertoire s'il n'existe pas
* exist\_ok=True : Ne lève pas d'erreur si le répertoire existe déjà
* Important pour éviter les erreurs lors de ré-exécutions

**⚠️ Structure attendue dans Google Drive:** Le notebook s'attend à trouver :  
• /content/drive/MyDrive/dataset/train\_set.csv  
• /content/drive/MyDrive/dataset/test\_set.csv  
Assurez-vous d'avoir téléchargé ces fichiers dans Drive avant l'exécution.

# **5. Cellule 4 : Chargement et Analyse des Datasets**

import pandas as pd  
  
train\_path = os.path.join(workdir, 'train\_set.csv')  
test\_path = os.path.join(workdir, 'test\_set.csv')  
  
# Verification existence  
for p in [train\_path, test\_path]:  
 if not os.path.exists(p):  
 print(f'ERREUR: Fichier non trouve: {p}')  
 raise FileNotFoundError(p)

### **Ligne par ligne :**

Ligne 1 : import pandas as pd

* Pandas : Bibliothèque pour manipulation de données tabulaires (DataFrames)
* Utilisée pour charger, nettoyer et analyser les CSV

Lignes 3-4 : Construction des chemins

* os.path.join() : Construit un chemin de manière portable (Windows/Linux/Mac)
* Combine workdir avec le nom du fichier CSV

Lignes 7-10 : Vérification d'existence

* Boucle for : Itère sur les deux chemins (train et test)
* os.path.exists(p) : Vérifie si le fichier existe
* raise FileNotFoundError(p) : Stoppe l'exécution avec un message clair
* Raison : Évite des erreurs cryptiques plus tard dans le code

**💡 Bonne pratique : Fail Fast:** Vérifier l'existence des fichiers dès le début permet de détecter les problèmes avant de lancer un long entraînement. C'est le principe 'fail fast'.

# Chargement  
train\_df = pd.read\_csv(train\_path, encoding='utf-8')  
test\_df = pd.read\_csv(test\_path, encoding='utf-8')  
  
print(f'Train shape: {train\_df.shape}')  
print(f'Test shape: {test\_df.shape}')

### **Ligne par ligne :**

Lignes 2-3 : pd.read\_csv()

* Charge les fichiers CSV en DataFrames Pandas
* encoding='utf-8' : Important pour les accents français (é, è, à, etc.)
* train\_df et test\_df sont maintenant des tableaux 2D (lignes × colonnes)

Lignes 5-6 : Affichage des dimensions

* .shape : Retourne un tuple (nb\_lignes, nb\_colonnes)
* Ex: (8000, 3) signifie 8000 exemples avec 3 colonnes

print('\nDistribution des classes (Train):')  
print(train\_df['intent'].value\_counts())  
print(f'\nPourcentages:')  
print(train\_df['intent'].value\_counts(normalize=True).apply(lambda x: f'{x:.2%}'))  
  
display(train\_df.head(3))

### **Ligne par ligne :**

Ligne 2 : train\_df['intent'].value\_counts()

* Compte le nombre d'occurrences de chaque classe d'intention
* Ex: TRIP: 5200, NOT\_TRIP: 1400, UNKNOWN: 800, NOT\_FRENCH: 600
* Crucial pour détecter les déséquilibres de classes

Ligne 4 : .value\_counts(normalize=True)

* normalize=True : Convertit les comptes en proportions (somme = 1.0)
* .apply(lambda x: f'{x:.2%}') : Formate en pourcentages (ex: 0.65 → 65.00%)

Ligne 6 : display(train\_df.head(3))

* .head(3) : Sélectionne les 3 premières lignes
* display() : Affiche joliment le DataFrame dans Jupyter/Colab
* Permet de vérifier visuellement la structure des données

**⚠️ Déséquilibre de classes détecté:** Le dataset contient ~65% de TRIP, ~14% de NOT\_TRIP, ~11% de NOT\_FRENCH, ~10% de UNKNOWN. Ce déséquilibre nécessite l'utilisation de class weights lors de l'entraînement (voir Cellule 7).

# **6. Cellule 5 : Preprocessing et Parsing des Entités JSON**

import json  
  
# Nettoyer les colonnes necessaires  
train\_df = train\_df[['text', 'intent', 'entities']].dropna(subset=['text', 'intent']).reset\_index(drop=True)  
test\_df = test\_df[['text', 'intent', 'entities']].dropna(subset=['text', 'intent']).reset\_index(drop=True)

### **Ligne par ligne :**

Ligne 1 : import json

* Module Python pour parser/générer du JSON
* Nécessaire car la colonne 'entities' contient du JSON

Ligne 4 : train\_df[['text', 'intent', 'entities']]

* Sélectionne uniquement les 3 colonnes nécessaires
* Supprime les colonnes inutiles (optimisation mémoire)

.dropna(subset=['text', 'intent'])

* Supprime les lignes où 'text' ou 'intent' est NaN (valeur manquante)
* subset=['text', 'intent'] : Ne vérifie que ces 2 colonnes
* Note : 'entities' peut être NaN (phrases NOT\_TRIP n'ont pas d'entités)

.reset\_index(drop=True)

* Réinitialise l'index après suppression de lignes
* drop=True : Ne garde pas l'ancien index comme colonne
* Résultat : Index propre de 0 à N-1

def parse\_entities\_field(row):  
 """Parse la colonne entities (JSON) et valide."""  
 try:  
 ents = json.loads(row['entities']) if pd.notna(row['entities']) else []  
 except:  
 ents = []  
   
 valid = []  
 txt = row.get('text', '')  
 for ent in ents:  
 if isinstance(ent, dict) and 'start' in ent and 'end' in ent and 'label' in ent:  
 if 0 <= ent['start'] < ent['end'] <= len(txt):  
 valid.append(ent)  
 return valid

### **Ligne par ligne :**

Ligne 1 : def parse\_entities\_field(row)

* Définit une fonction qui prend une ligne (row) du DataFrame
* Sera appliquée à chaque ligne avec .apply()

Lignes 3-6 : Tentative de parsing JSON

* pd.notna(row['entities']) : Vérifie si la cellule n'est pas NaN
* json.loads() : Convertit la chaîne JSON en liste Python
* Format attendu : '[{"start":0, "end":5, "label":"Departure"}, ...]'
* try...except : Gère les JSON mal formés (retourne [] vide)

Lignes 8-13 : Validation des entités

* valid = [] : Liste pour stocker les entités valides
* row.get('text', '') : Récupère le texte, '' si absent

Ligne 11 : Vérifications de sécurité

* isinstance(ent, dict) : Vérifie que c'est un dictionnaire
* 'start' in ent : Vérifie la présence des clés obligatoires

Ligne 12 : Validation des indices

* 0 <= ent['start'] : Start ne peut pas être négatif
* ent['start'] < ent['end'] : End doit être après Start
* ent['end'] <= len(txt) : End ne peut pas dépasser la longueur du texte
* Raison : Évite les erreurs d'indexation lors du NER

**💡 Pourquoi valider rigoureusement:** Des données mal formatées peuvent causer des erreurs silencieuses lors de l'entraînement NER, rendant le débogage très difficile. Cette validation garantit la cohérence.

# Parser les entities  
train\_df['parsed\_entities'] = train\_df.apply(parse\_entities\_field, axis=1)  
test\_df['parsed\_entities'] = test\_df.apply(parse\_entities\_field, axis=1)  
  
print('Entities parsees avec succes!')  
print(f'\nExemple TRIP avec entities:')  
trip\_ex = train\_df[train\_df['intent'] == 'TRIP'].iloc[0]  
print(f' Texte: {trip\_ex["text"]}')  
print(f' Entities: {trip\_ex["parsed\_entities"]}')

### **Ligne par ligne :**

Lignes 2-3 : .apply(parse\_entities\_field, axis=1)

* .apply() : Applique une fonction à chaque ligne
* axis=1 : Applique par ligne (axis=0 serait par colonne)
* Crée une nouvelle colonne 'parsed\_entities' avec les listes validées

Ligne 7 : train\_df[train\_df['intent'] == 'TRIP'].iloc[0]

* train\_df['intent'] == 'TRIP' : Filtre booléen, retourne True/False
* train\_df[...] : Sélectionne les lignes où la condition est True
* .iloc[0] : Prend la première ligne du résultat filtré

Lignes 8-9 : Affichage d'exemple

* Vérifie visuellement que le parsing a fonctionné
* Exemple de sortie :
* Texte: Aller de Paris à Lyon
* Entities: [{'start': 9, 'end': 14, 'label': 'Departure'}, ...]

# **7. Cellule 6 : Préparation des Datasets HuggingFace**

from datasets import Dataset  
from sklearn.preprocessing import LabelEncoder  
  
# Encoder les labels d'intent  
label\_encoder = LabelEncoder()  
all\_intents = list(set(list(train\_df['intent'].unique()) + list(test\_df['intent'].unique())))  
label\_encoder.fit(all\_intents)

### **Ligne par ligne :**

Ligne 1 : from datasets import Dataset

* HuggingFace Datasets : Format optimisé pour les pipelines Transformers
* Avantages : Mémoire efficace, tokenization batched, caching automatique

Ligne 2 : from sklearn.preprocessing import LabelEncoder

* Convertit les labels textuels (TRIP, NOT\_TRIP...) en entiers (0, 1, 2, 3)
* Nécessaire car les modèles PyTorch travaillent avec des indices numériques

Ligne 6 : Construction de la liste complète des intents

* train\_df['intent'].unique() : Intents uniques dans train
* test\_df['intent'].unique() : Intents uniques dans test
* list(set(...)) : Union des deux ensembles (sans doublons)
* Raison : S'assurer que toutes les classes sont présentes, même si rare

Ligne 7 : label\_encoder.fit(all\_intents)

* Apprend le mapping : {'NOT\_FRENCH': 0, 'NOT\_TRIP': 1, 'TRIP': 2, 'UNKNOWN': 3}
* L'ordre est alphabétique par défaut

print('Classes d\'intent:')  
for i, label in enumerate(label\_encoder.classes\_):  
 print(f' {i}: {label}')  
  
# Creer les datasets HuggingFace  
train\_df['label'] = label\_encoder.transform(train\_df['intent'])  
test\_df['label'] = label\_encoder.transform(test\_df['intent'])  
  
intent\_train\_dataset = Dataset.from\_pandas(train\_df[['text', 'label']])  
intent\_test\_dataset = Dataset.from\_pandas(test\_df[['text', 'label']])

### **Ligne par ligne :**

Lignes 1-3 : Affichage du mapping

* enumerate() : Retourne (index, valeur) pour chaque élément
* label\_encoder.classes\_ : Tableau des labels dans l'ordre interne

Ligne 6 : label\_encoder.transform()

* Convertit les labels textuels en entiers
* Ex: ['TRIP', 'NOT\_TRIP', 'TRIP'] → [2, 1, 2]
* Crée une nouvelle colonne 'label' dans le DataFrame

Ligne 9 : Dataset.from\_pandas()

* Convertit DataFrame Pandas → HuggingFace Dataset
* Sélectionne uniquement ['text', 'label'] (colonnes nécessaires)
* Format optimisé pour .map(), tokenization, etc.

**💡 Pourquoi HuggingFace Datasets:** Dataset HF utilise Apache Arrow pour un accès mémoire ultra-rapide. La tokenization est mise en cache, évitant les recalculs lors de ré-exécutions.