EPITECH. TECHNOLOGY

Rapport projet IA

T-AIA-901

2024

Thibault Conte
Alexandre Dupont
Anass Elfatni
Nicolas Laurens
Alban Teissier

TABLE DES MATIÈRES

Travel Order Resolver

Objectif

Le projet vise à mettre en place un workflow pour proposer à un utilisateur, le trajet le plus court entre deux destination qu'il aura choisis.
Le processus débutera par l'expression de la phrase, qu'elle soit sous forme de texte ou de commande vocale.
Ensuite, un modèle d'intelligence artificielle dédié au traitement du langage sera utilisé pour extraire les entités nommées, en l'occurrence les villes de départ ainsi que les villes d'arrivées.
Par la suite, un algorithme sera utilisé pour déterminer le meilleur chemin entre les deux villes identifiées.
Il est important de noter que le "meilleur chemin" sera défini en termes de liaisons ferroviaires, et les poids associés à ces chemins seront basés sur les temps de trajet entre les gares correspondantes.

Le projet se découpe en trois points :

Speech Recognition

Algorithme permettant la conversion de la parole humaine en texte écrit.

Natural Language Processing

Ensemble de modèles d'intelligence artificielle dédiés au traitement du langage humain.

Path Finding

Un algorithme qui, en se basant sur des poids attribués à différents points, identifie le chemin optimal entre eux en termes de coût, déterminant ainsi la trajectoire la plus avantageuse.

Speech Recognition

Introduction

Pour ce projet, cette composante n'est pas au cœur des préoccupations. L'objectif était plutôt de trouver une solution existante pour convertir la parole humaine en texte. Étant donné que cette technologie est déjà disponible, nous avons opté pour l'utilisation d'une bibliothèque open source offrant cette fonctionnalité.

Speech Recognition Library

Cette bibliothèque Python est un outil fournissant des fonctionnalités de reconnaissance vocale. Elle permet la conversion d'enregistrements audio contenant des paroles humaines en texte écrit. En plus de cela, la bibliothèque offre une interface permettant de spécifier la langue à écouter, ce qui revêt une importance particulière dans notre projet.

License : 3-clause BSD

Implémentation

Installation : *pip install SpeechRecognition*

```python
import speech_recognition as sr
import keyboard


class cSpeechToText:
    def __init__(self):
        self.recognizer = sr.Recognizer()
        self.sentence = ""

    def start(self):
        with sr.Microphone() as source:
            listening = False
            print("Please press 'ctrl' before talk")
            while True:
                try:
                    if keyboard.is_pressed('ctrl'):
                        if not listening:
                            print("Start listening ...")
                            listening = True
                            self.recognizer.adjust_for_ambient_noise(source=source)
                            audio_data = self.recognizer.listen(source)
                            self.sentence = self.recognizer.recognize_google(audio_data, language="fr-FR")
                    elif listening:
                        print("Stop listening ")
                        listening = False
                        return self.sentence

                except sr.UnknownValueError:
                    print("Error: google don't reconize the audio")

                except sr.RequestError as e:
                    print(f"Error google request: {str(e)}")

                except Exception as e:
                    print(f"Error speech to text: {str(e)}")
```


Documentation de Speech Recognition : https://pypi.org/project/SpeechRecognition
Articles et tutoriels pertinents : https://github.com/Uberi/speech_recognition#readme

Natural Language Processing

Introduction

Pour détecter les villes d'origine et d'arrivée dans une phrase, nous avons utilisé un modèle d'intelligence artificielle spécialisé dans la reconnaissance d'entités nommées (NER). Pour ce faire, nous avons choisi d'ajuster des modèles préexistants, tirant parti de leur entraînement sur des ensembles de données volumineux. Afin de personnaliser le modèle en fonction de notre cas spécifique, la création d'un dataset personnalisé s'est avérée nécessaire. Dans cette section, nous aborderons d'abord la conception de ce dataset, puis nous explorerons la mise en œuvre du NER dans notre contexte.

Dataset generator

Pour élaborer notre dataset, nous avons initié le processus en répertoriant les différentes formulations de phrases pouvant être utilisées pour décrire un itinéraire, tant au niveau de la syntaxe que du lexique employé. Afin d'accroître la variété des phrases générées, nous avons mis en place un algorithme. À partir de patrons préalablement définis dans des fichiers texte, cet algorithme nous a permis de générer plusieurs centaines de phrases distinctes.

Exemple d'un fichier text :

```
[Génération 1 - Valide]

[construction]
Verbe -- intention valide -- mot start -- villeOrigine -- mot end -- villeArrive
Verbe -- intention valide -- mot end -- villeOrigine -- mot start -- villeArrive
mot start -- villeOrigine -- Verbe -- intention valide-- mot end -- villeArrive

[Exemple]
j'envisage de voyager de Toulouse à Paris
nous projetons de partir à Paris depuis Toulouse
de Toulouse j'envisage de voyager à Paris

[Verbe]
j'envisage de
je projette de
je pense
je compte
j'anticipe de
je programme de
je souhaite
je souhaiterais
j'ai l'intention de
je songe
je prévois de
j'aimerais
nous envisageons de
nous projetons de
nous pensons
```

Pour accroître la variété de nos phrases générées, nous avons décidé que, pour chaque structure de patrons de phrases considérée comme "valide" (phrases contenant un itinéraire), nous avons conçus un patron qui utilise le même lexique et la même syntaxe pour créer des phrases considérées comme "non valides" (ne contenant pas d'itinéraire).

Exemple d'un fichier text non valide :

```
[Génération 1 - Invalide]

[construction]
Verbe -- intention non valide --*

[Exemple]
j'envisage de pratiquer la photographie

[Non Classer]
de créer
de consacrer plus de temps
de poursuivre

[Verbe]
j'envisage de
je projette de
je pense
je compte
j'anticipe de
je programme de
je souhaite
```

C'est ainsi que nous aboutissons à un total de 9 fichiers texte, chacun renfermant des patrons de structures et de lexiques de phrases distincts, combinant des formulations valides avec des formulations invalides.

L'algorithme créé prend en charge la lecture de tous les fichiers mentionnés précédemment, ainsi que de fichiers contenant les noms des villes de France et des gares. En fonction d'une quantité prédéfinie, il génère le nombre souhaité de phrases de manière équitable entre les différents patrons présents dans les fichiers. Une fois les phrases générées, l'algorithme les enregistre dans un fichier CSV en précisant, pour chaque phrase, des informations telles que sa validité, ainsi que les villes d'origine et d'arrivée, le cas échéant.

```python
class cGenerateSentence:
    def __init__(self):
        self.c_stops = cStops()
        self.c_time_tables = cTimeTables()
        self.c_list_des_gares = cListDesGares()


        self.c_generation_1_valid = cGeneration_1_Valid()
        self.c_generation_1_invalid = cGeneration_1_Invalid()

        self.c_generation_2_valid = cGeneration_2_Valid()
        self.c_generation_2_invalid = cGeneration_2_Invalid()

        self.c_generation_3_valid = cGeneration_3_Valid()

        self.c_generation_4_valid = cGeneration_4_Valid()
        self.c_generation_4_invalid =cGeneration_4_Invalid()

        self.c_generation_invalid_depuis = cGeneration_Invalide_Depuis()

        self.list_of_city = []

        self.list_of_sentences = []
        self.list_csv_row = []
        self.list_csv_column = []

        self.path_csv = "generate_sentence.csv"
        self.path_column_csv = "column_test_model.csv"


        self.__set_up()
        self.__filter_name_city()
        self.__set_list_name_city()
```

```python
def generate_by_number(self, number):
    self.__generate_sentences(number)
    self.__get_list_of_sentences()
    self.__get_list_of_csv_row()
    self.__get_list_of_csv_column()
    self.__write_csv()
    self.__write_column_csv()
```

```python
def __write_csv(self):
    if not os.path.isfile(self.path_csv):
        with open(self.path_csv, 'w', newline='',encoding='utf-8') as csvfile:
            csv_writer = csv.writer(csvfile,delimiter=';')
            csv_writer.writerow(['ID','SENTENCE','VALID','ORIGIN','ARRIVAL','STEP_OVER','TYPE'])

    with open(self.path_csv, 'a', newline='',encoding='utf-8') as csvfile:
        csv_writer = csv.writer(csvfile,delimiter=';')

        for i in range(len(self.list_csv_row)):
                row = self.list_csv_row[i]
                row[0] = i
                csv_writer.writerow(row)
```

Dataset généré :

```
ID;SENTENCE;VALID;ORIGIN;ARRIVAL;STEP_OVER;TYPE
0;j'aimerais aller depuis AIGUEBELETTE-LE-LAC vers CHILLY-MAZARIN;1;AIGUEBELETTE-LE-LAC;CHILLY-MAZARIN;;
1;je compte voyager de CHEVILLON vers SOULTZ-SOUS-FORÊTS;1;CHEVILLON;SOULTZ-SOUS-FORÊTS;;
2;nous souhaitons voyager de MINIAC-MORVAN vers THAON-LES-VOSGES;1;MINIAC-MORVAN;THAON-LES-VOSGES;;
3;j'aimerais aller de SAINT-HILAIRE-DE-VILLEFRANCHE vers DARCEY;1;SAINT-HILAIRE-DE-VILLEFRANCHE;DARCEY;;
4;je songe faire un trajet depuis OUGES vers MARESQUEL-ECQUEMICOURT;1;OUGES;MARESQUEL-ECQUEMICOURT;;
5;nous songeons faire un trajet de SAINT-JUST-EN-CHAUSSÉE vers AILLY-SUR-NOYE;1;SAINT-JUST-EN-CHAUSSÉE;AILLY-SUR-NOYE;;
6;j'envisage de faire le trajet de MERREY vers HERBLAY;1;MERREY;HERBLAY;;
7;je pense partir depuis LEVAL vers ARPAJON;1;LEVAL;ARPAJON;;
8;je pense partir de PERPIGNAN à SAVIGNY-SUR-ORGE;1;PERPIGNAN;SAVIGNY-SUR-ORGE;;
9;je souhaite partir depuis LA ROCHE-SUR-FORON vers BUSSY-LETTRÉE;1;LA ROCHE-SUR-FORON;BUSSY-LETTRÉE;;
10;nous anticipons de partir de BOURGNEUF-EN-RETZ à LIMERAY;1;BOURGNEUF-EN-RETZ;LIMERAY;;
11;je souhaite partir de OSTRICOURT vers SOISSONS;1;OSTRICOURT;SOISSONS;;
12;nous projetons de faire un trajet depuis VITRE vers PRUNIERS-EN-SOLOGNE;1;VITRE;PRUNIERS-EN-SOLOGNE;;
13;nous songeons voyager depuis BACOUEL-SUR-SELLE vers ÉGLY;1;BACOUEL-SUR-SELLE;ÉGLY;;
14;j'anticipe de faire un trajet de PORTA vers LAUDUN;1;PORTA;LAUDUN;;
15;nous programmons de partir de SERDINYA vers MALESHERBES;1;SERDINYA;MALESHERBES;;
```

En conclusion, nous avons élaboré un dernier générateur qui, grâce à un scraper, extrait aléatoirement des phrases duWikipédia français pour élargir la diversité des phrases non valides dans notre dataset.

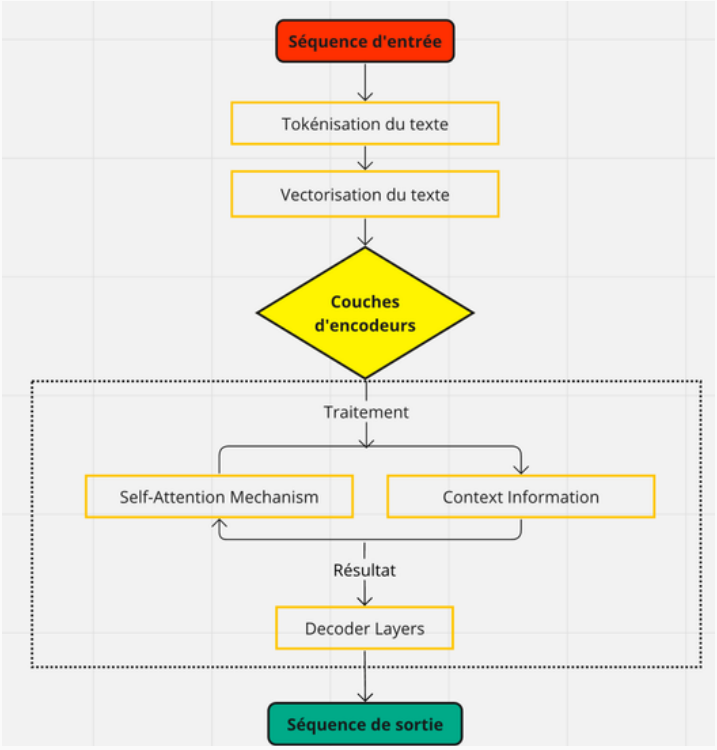
```python
class cScrapper:
    def __init__(self):···

    def __add_csv_row(self, sentence):···

    def __add_sentence(self, list_paragraph):···

    def __add_sentence_from_specific_word(self, list_paragraph):···

    def __add_sentence_without_specific_word(self, list_paragraph):···

    def __is_Valid_Link(self, link):···

    def __is_Valid_Sentence(self, sentence):···

    def __get_Indice_Link(self,size):···

    def __get_link(self, list_link):···

    def start(self, number):···

    def write_csv(self):···
```

Named Entity Recognition

Pour la détection des entités nommées, nous avons choisi une approche basée sur les transformers. Dans cette optique, nous avons orienté notre choix vers la bibliothèque spaCy, reconnue pour ses solutions robustes, ainsi que sa facilité d'utilisation et d'intégration dans un processus de fine-tuning.

Transformers

Les transformers sont des réseaux de neurones profonds largement employés dans le domaine du traitement du langage naturel en raison de leurs performances remarquables.
Fondés sur un **système d'attention**, ils cherchent à préserver les interdépendances entre les mots en se concentrant sur un **mécanisme d'attention** qui évalue le degré de connexion entre deux éléments de deux séquences.
Contrairement à d'autres modèles, qui se fondent davantage sur les dépendances entre les vectorisations des éléments de séquences, les transformers privilégient une approche plus contextuelle et adaptable.



- Séquence d'entrée :
  - Séquence d'entrée à traiter (dans notre càs, se sera la demande de l'utilisateur en texte).

- Tokénisation et vectorisation :
  - La phrase est d'abord **"tokénisée",** pour ensuite être représentée en **"vecteurs", qu'on appelle "Word Embeddings".**

- Encoder Layers :
  - Les couches d'encodeur traitent les "words embeddings" (vecteurs) en prenant compte le sens de chaque mot, par rapport aux autres.

- Context Information :
  - Fournit une représentation enrichie de chaque mot qui inclut sa relation avec tous les autres mots dans la phrase.

- Self-Attention Mechanism :
  - Permet à chaque mot de la phrase d'influencer la représentation des autres, mettant en évidence les relations importantes.

- Decoder Layers :
  - Les couches de décodeur génèrent la séquence de sortie à partir des informations contextuelles fournies par l'encodeur.

- Output Sequence :
  - La séquence de sortie produite par le Transformer, basée sur la séquence d'entrée et le traitement effectué par l'encodeur et le décodeur.

Quelques explications avec exemple :

- 1. Séquence d'entrée
  - Exemple : "Je voudrais aller à Paris, depuis Toulouse"

- 2. Tokenisation :
  - Processus : Divise le texte en unités plus petites (**tokens**) pour être traitées.
  - Exemple : ["Je", "voudrais", "aller", "à", "Paris", ",", "depuis", "Toulouse"]
  - La **tokénisation** est en fait bien plus complexe que celà, ajoutant des caractères spéciaux pour le traitement, mais pour l'exemple, restons sur cette représentation simple

- 3. Vectorisation (Word Embeddings) :
  - Processus : Convertit chaque **token** en un **vecteur** numérique.
  - Exemple de **vecteurs** simplifiés :
    - "Je" → [0.2, 0.1]
    - "voudrais" → [0.4, 0.3]
    - "aller" → [0.5, 0.3]
    - "à" → [0.1, 0.3]
    - "Paris" → [0.6, 0.8]
    - "," → [0.0, 0.0]
    - "depuis" → [0.4, 0.2]
    - "Toulouse" → [0.7, 0.9]
  - Les **vecteurs** réels seraient beaucoup plus grands, mais ceux-ci sont simplifiés pour l'exemple.

- 4. Couches d'encodeurs (avec Self-Attention intégrée) :
  - Processus : Sur chacune des couches du réseau de neurones, chaque **vecteur** de mot est transformé en prenant en compte le contexte fourni par tous les autres mots grâce au mécanisme de **Self-Attention**.
  - Exemple simplifié :
    - Pour "Paris", le mécanisme de **self-attention** note l'importance relative de "aller à" par rapport à "Paris", renforçant la représentation **vectorielle** de "Paris" en tant que destination.
    - Similairement, "Toulouse" gagne en importance contextuelle en relation avec "depuis", signalant qu'il s'agit du point de départ.

- 5. Context Information :
  - Processus : Chaque vecteur de mot est maintenant une représentation riche en informations contextuelles.
  - Exemples simplifiés :
    - Le vecteur pour "Paris" porte maintenant des informations contextuelles indiquant qu'il est associé à une action de déplacement vers un lieu.
    - Le vecteur pour "Toulouse" porte des informations indiquant qu'il est le point de départ du déplacement.

- 6. Decoder Layers :
  - Processus : Dans le cadre d'un transformers qui doit générer par exemple un texte explicatif en retour, les "decoders layers" sont utilisés. Dans le cadre notre projet, pour la NER, les "Decoder Layers" pourraient être remplacés par une couche de classification qui utilise les vecteurs contextuels pour identifier et classer les entités.
  - Exemple simplifié :
    - Les vecteurs de "Paris" et "Toulouse" seraient interprétés par la couche de classification pour les étiqueter respectivement comme "destination" et "origine".

- 7. Séquence de sortie :
  - Processus : La classification de chaque mot est assemblée pour former la sortie finale.
  - Exemple de sortie :
    - "Je" → Aucune entité
    - "voudrais" → Aucune entité
    - "aller" → Aucune entité
    - "à" → Aucune entité
    - "Paris" → Destination
    - "," → Aucune entité
    - "depuis" → Aucune entité
    - "Toulouse" → Origine
  - En utilisant ces étiquettes, le système NER peut facilement extraire que l'utilisateur souhaite se déplacer de Toulouse à Paris.

NER Sans Transformers

Un système NER sans l'utilisation de modèles Transformers, fonctionne similairement au niveau des étapes, néanmoins il ne se base pas sur un mécanisme d'attention saisissant le contexte global des mots ; mais plutôt sur des caractéristiques statiques extraites du texte.

Il identifie les entités principalement grâce à des marqueurs linguistiques superficiels comme la capitalisation des noms propres, la syntaxe, et des mots-clés contextuels.

De ce fait il reconnaitra parfaitement dans la phrase suivante "Je voudrais aller à Paris, depuis Toulouse", Paris et Toulouse comme étant des locations.
Néanmoins il ne saura pas définir correctement le lieu de départ et de destination.

Une solution possible, serait d'entrainer un NER pour qu'il reconnaisse les mots-clés contextuels comme "depuis, à partir de" suivis d'une ville étant le lieu de départ.
Et pareillement pour la destination.

SpaCy

SpaCy, une bibliothèque Python spécialisée dans le traitement automatique du langage, se démarque par son caractère open source et son accessibilité, comme indiqué précédemment. Son architecture repose sur la création de pipelines constituées de modules téléchargeables ou personnalisables, offrant la flexibilité d'ajouter ou de retirer facilement des composants en fonction de la tâche de NLP souhaitée.

License : MIT

Mise en place

En suivant le principe énoncé juste au-dessus, il est possible de télécharger depuis le site de SpaCy un fichier de configuration contenant des paramètres par défaut pour les divers éléments de la pipeline que l'on souhaite intégrer. Ces composants peuvent être des modèles pré-entraînés provenant de Hugging Face ou même des modèles existant exclusivement en local. Une fois le fichier obtenu à l'aide des commandes spécifiées, il devient alors possible d'entraîner cette configuration en utilisant un jeu de données préalablement traité.

Choix des composants

Nous avons opté pour la construction de notre pipeline en intégrant deux composants clés : le **transformer Camembert** et le **NER de SpaCy**.

Camembert, en tant que modèle transformer pré-entraîné avec des phrases françaises, offre une capacité exceptionnelle à saisir les relations sémantiques et contextuelles. Ce choix s'est révélé particulièrement pertinent pour notre projet.

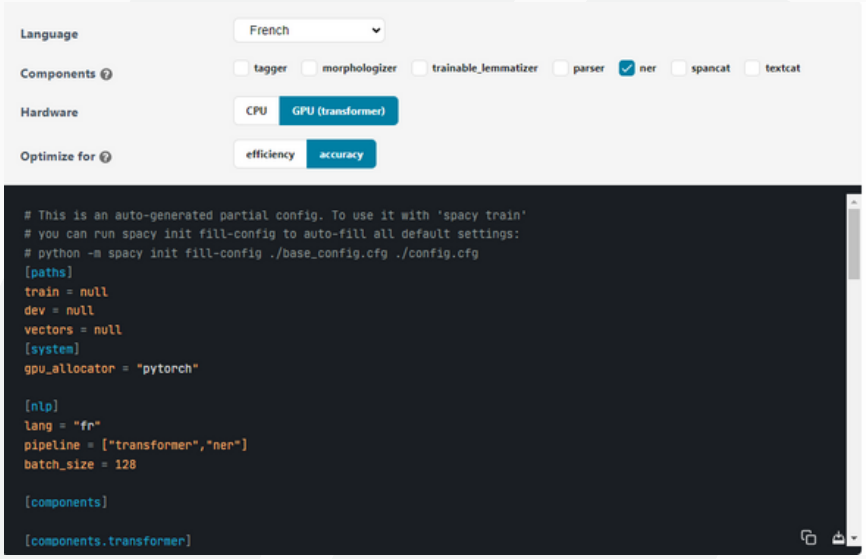
Associé au NER de SpaCy, spécialisé dans la reconnaissance des entités, et spécifiquement entraîné pour identifier les villes d'origine et de destination dans une phrase, le NER se concentre de manière ciblée sur l'identification des entités cruciales pour notre cas d'utilisation. Pendant ce temps, Camembert se focalise sur la richesse contextuelle du langage. Cette synergie entre les deux composants renforce l'efficacité de notre pipeline, permettant une extraction précise des informations nécessaires à la détermination d'itinéraires entre les villes.

Implémentation

Configuration système : RTX 3050 Laptop
CUDA: 11.8
cuDNN: 8.x

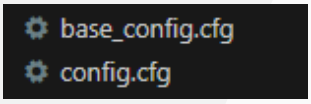
Installation : *pip install -U pip setuptools wheel*
pip install -U 'spacy [cuda11x]''
pip install spacy-transformers
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

Pour obtenir le fichier de configuration, il suffit de renseigner les options souhaitées, notamment :



Pour compléter le fichier **base_config.conf** et obtenir le fichier **config.conf** il suffit d'utiliser cette commande spaCy :
python -m spacy init fill-config ./config/base_config.cfg ./config/config.cfg

Comme mentionné précédemment, après l'exécution de cette commande, l'attente est d'avoir ces deux fichiers :



Prétraitement des données

Pour entraîner la pipeline sur notre jeu de données, il est nécessaire d'apporter une modification à chaque phrase afin de les formater selon une structure particulière :

```
("j'aimerais me rendre à toulouse depuis bordeaux.", [(22, 30, "ARRIVAL"),(38, 46, "ORIGIN")])
```

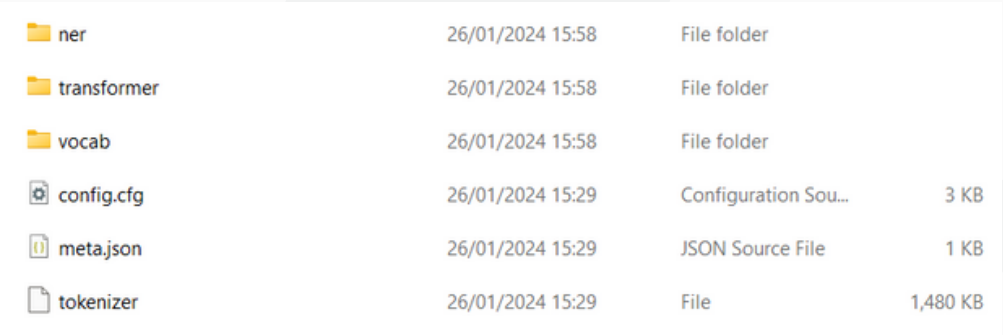
Ainsi, nous avons développé un algorithme qui, pour chaque phrase, la formate en spécifiant les indices de début et de fin des villes indiquant une destination, s'il y en a. Ensuite, l'algorithme convertit la phrase en minuscules pour uniformiser la casse. Une fois cette première étape accomplie, l'ensemble du jeu de données est divisé en trois ensembles distincts : un ensemble d'entraînement, un ensemble de test et un ensemble de validation (**train**, **test** et **val**).
Ensuite, pour chacun de ces trois sous-ensembles, nous appliquons une fonction dont le code est fourni par spaCy. Cette fonction permet de transformer une phrase avec les indices des mots recherchés dans un format spécifique pour le modèle, puis de sauvegarder l'ensemble résultant dans un fichier .spacy.

```
class cGenerate_Sentence:
    def __init__(self): ···

    def __read_csv(self): ···

    def __formate_data(self): ···

    def __write_csv(self): ···

    def setUp(self): ···


c_generate_sentence = cGenerate_Sentence()
c_generate_sentence.setUp()

train_data, test_data = train_test_split(c_generate_sentence.data_formated, test_size=0.2)
train_data, val_data = train_test_split(train_data, test_size=0.25)

def create_spacy_data(data, file_name):
    nlp = spacy.blank("fr")
    db = DocBin()
    for text, annotations in data:
        doc = nlp(text)
        ents = []
        for start, end, label in annotations['entities']:
            span = doc.char_span(start, end, label=label)
            if span is not None:
                ents.append(span)
        doc.ents = ents
        db.add(doc)
    db.to_disk(file_name)

create_spacy_data(train_data, "./data/train.spacy")
create_spacy_data(val_data, "./data/val.spacy")
create_spacy_data(test_data, "./data/test.spacy")
```

Entrainement

Après avoir formaté les données correctement et complété le fichier de configuration, nous sommes prêts à démarrer le processus d'entraînement en utilisant la commande suivante :
python -m spacy train ./config/config.cfg --output ./output/ --paths.train ./data/train.spacy --paths.dev ./data/val.spacy --gpu-id 0

À la fin du processus d'entraînement, nous obtenons ainsi un répertoire renfermant les informations relatives à notre pipeline.



ner	26/01/2024 15:58	File folder	
transformer	26/01/2024 15:58	File folder	
vocab	26/01/2024 15:58	File folder	
config.cfg	26/01/2024 15:29	Configuration Sou...	3 KB
meta.json	26/01/2024 15:29	JSON Source File	1 KB
tokenizer	26/01/2024 15:29	File	1,480 KB

Utilisation

Une fois que le modèle est entraîné, son utilisation est simple : il suffit de le charger dans un programme Python en utilisant les fonctions fournies par la bibliothèque. Ensuite, on fournit au modèle la phrase à traiter, et ce dernier renverra un objet contenant les informations obtenues lors du passage du modèle.
Voici un exemple d'application dans notre projet:

```python
import spacy

nlp = spacy.load("output/model-best")


while(1):
    test_sentences = input()
    if(test_sentences == "stop"):
        break

    doc = nlp(test_sentences)
    if(len(doc.ents) == 2):
        ent_1 = doc.ents[0]
        print(f"Entité nommée : {ent_1.text} | Type : {ent_1.label_}")

        ent_2 = doc.ents[1]
        print(f"Entité nommée : {ent_2.text} | Type : {ent_2.label_}")

    else:
        print("aucun itinéraire trouvé !")
```

Path Finding

Introduction

Comme pour la section sur le speech recognition, cette partie n'était pas la plus importante. L'objectif était de sélectionner un algorithme capable de déterminer le meilleur chemin entre plusieurs points, chacun ayant un poids associé. Dans notre contexte, les points sont représentés par des gares de trains et les poids entre ces points correspondent à la durée d'un trajet entre deux stations. Nous avons choisi d'utiliser l'algorithme de Dijkstra pour cette tâche, étant donné qu'il était particulièrement recommandé dans notre cas.

Mise en place

Les composants nécessaires à la construction du graphe pour l'algorithme de Dijkstra nous ont été fournis à travers divers documents. Ces fichiers ont renseigné les détails sur les différentes gares, notamment leurs emplacements et les durées des trajets entre elles. Une fois les éléments et le graphe établis, nous avons associées les gares à leur villes respectives . Cela nous permet de fournir en entrée deux noms de villes et d'obtenir en sortie plusieurs informations, telles que la liste des villes par lesquelles le trajet s'effectue, la liste des gares traversées, les temps entre chaque gare, ainsi que la durée totale du trajet.

Implémentation

Initialement, nous avons créé deux classes pour lire et stocker les informations présentes dans les différents fichiers, ainsi que les données nécessaires à l'implémentation pratique de l'algorithme.

```
#FILES
class cTimeTables:
    def __init__(self): ···

    def setUp(self): ···

    def __format_data(self): ···

    def __format_name(self,name): ···

    def __read_csv(self): ···

    def toString(self): ···

class cTrainStation_Localisation:
    def __init__(self): ···

    def setUp(self): ···

    def __format_data(self): ···

    def __read_csv(self): ···

    def toString(self): ···
```

Par la suite, nous avons élaboré une classe visant à lier chaque gare à ses itinéraires potentiels.

```
class cTrainStation:
    def __init__(self,name="name"):···

    def add_trajet(self,trip_id,end,time):···

    def get_time(self,destination):···
```

En troisième lieu, nous avons mis en place une classe intégrant les fonctionnalités de l'algorithme de Dijkstra.

```
#ALGO
class cDijkstra:
    def __init__(self):···

    def setUp(self, matrix):···

    def __dijkstra_shortest_path(self, start):···

    def dijkstra_shortest_path_with_route(self, start, end):···
```

Enfin, nous disposons d'une classe qui centralise les classes mentionnées précédemment et établit les liens nécessaires entre elles, facilitant ainsi leur utilisation.

```
# MAIN CLASS
class cDijkstra_API:
    def __init__(self):···

    #SET UP CLASS
    def setUp(self):        ···


    def __add_train_station(self, trip_id, end, start, time):···

    def __create_matrix(self):···

    def __fill_list_train_station(self):···

    def __init_dijkstra(self):···


    #API CALL
    def __get_indice_from_name(self, name):···


    def __get_city_by_name(self, list_name):···

    def __get_name_by_index(self, list_index):···

    def __get_pos_by_name(self, list_name):···

    def __get_time_by_index(self, list_index):···

    def __get_total_time_by_index(self, list_index):···


    def find_path(self, name_start, name_end):···
```

Exemple d'utilisation

```python
# ----------------------------------------------------
#               EXEMPLE UTILISATION
# ----------------------------------------------------

#CREATION INSTANCE
c_dijkstra = cDijkstra_API()

#INITIALISATION
c_dijkstra.setUp()

#UTILISATION
response = c_dijkstra.find_path("bordeaux", "ToulOuse")

#RESPONSE
print(response["list_name"])
# --> ['gare de bordeaux-st-jean', 'gare de marmande', 'gare de agen', 'gare de toulouse-matabiau']

print(response["list_pos"])
# --> [('44.82653979', '-0.55619406'), ('44.50306728', '0.16811469'), ('44.20797206', '0.62090454'),

print(response["list_city"])
# --> ['bordeaux', 'marmande', 'agen', 'toulouse']

print(response["list_time"])
# --> [67, 36, 81]

print(response["total_time"])
# --> 184
```

Workflow

En intégrant les divers éléments exposés précédemment, nous avons élaboré des flux de travail qui connectent les différents algorithmes. Nous disposons ainsi de deux workflows fonctionnels conçus pour résoudre la problématique donnée : permettre à un utilisateur, que ce soit par écrit ou via la reconnaissance vocale en énonçant une phrase spécifiant un itinéraire de la ville d'origine à la ville d'arrivée, d'obtenir un cheminement valide en utilisant le réseau ferroviaire

Solution non visuelle

La première solution proposée ne comporte pas d'interface graphique. Il s'agit d'un script qui enchaîne les étapes de reconnaissance vocale, de traitement du langage naturel (NLP) et de recherche d'itinéraire, fournissant en sortie un ensemble d'informations sur le chemin recommandé.

Ce script se décompose en plusieurs étapes, notamment le chargement des bibliothèques, la configuration d'un journal (logger) pour consigner les requêtes effectuées, le chargement et la préparation des différentes classes requises, l'utilisation du convertisseur de discours en texte (speech-to-text), l'utilisation du modèle NER (Named Entity Recognition), l'utilisation de l'algorithme de Dijkstra, et enfin, la visualisation des résultats.

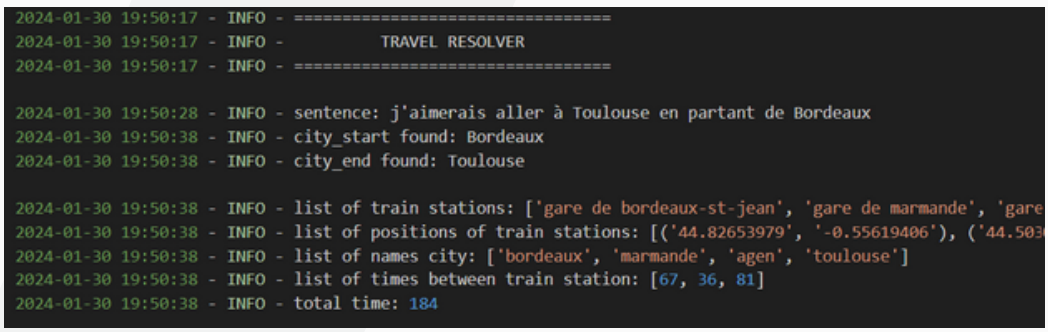
Utilisation

Lancer le programme avec la commande suivante :

python workflow.py

Puis suivre les instructions.

Logs générés dans un dossier du même nom



Solution visuelle

Dans la deuxième solution, nous avons incorporé une interface homme-machine (IHM) aux programmes, facilitant l'exécution de toutes les tâches requises telles que la conversion de la parole en texte (speech-to-text), la reconnaissance d'entités nommées (NER) et l'application de l'algorithme de Dijkstra. Cette interface visuelle offre également la possibilité de visualiser l'itinéraire sur une carte, avec la position des gares traversées. De plus, des sélecteurs sont disponibles pour lister les différentes villes disponibles, permettant ainsi de choisir la ville de départ et d'arrivée.

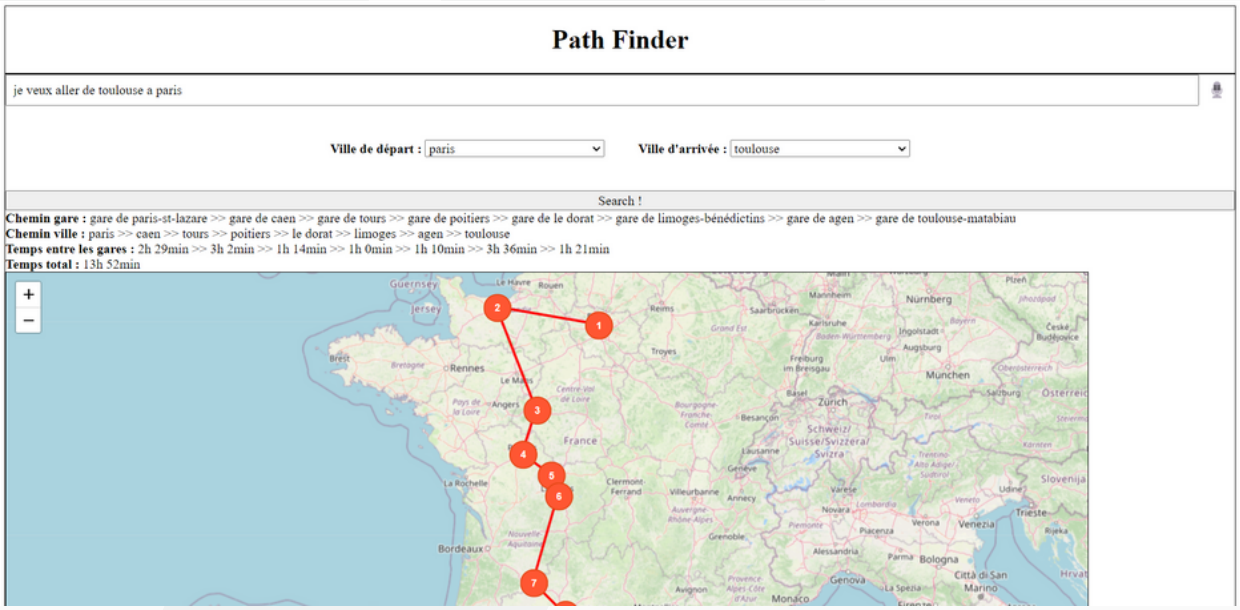
Le mécanisme mis en place est similaire à la première solution : les bibliothèques nécessaires sont importées, et les classes sont initialisées. Cependant, la séquence d'actions diffère légèrement. Lorsqu'une action est effectuée sur l'interface, celle-ci interroge un serveur (contenant les différentes classes mentionnées précédemment) et exécute la tâche demandée, que ce soit l'appel du modèle ou l'appel de l'algorithme de Dijkstra. Une fois la réponse du serveur reçue, l'interface se charge de la visualiser.

Utilisation

Lancer le programme avec la commande suivante :

python API.py

Puis ouvrer la page html contenue dans le chemin suivant ./src/web .



Essaies et Résultats

Tous les modèles qui seront présentés ultérieurement ont été entraînés avec le même jeu de données. Ce jeu de données comprend un ensemble de 900 phrases, dont 50% contiennent un itinéraire et 50% n'en contiennent pas. De plus, pour chaque ensemble d'entraînement et de test, la répartition a été la même : 70% pour l'entraînement, 15% pour la validation et 15% pour le test.

Classification de texte

Initialement, nous avons débuté en élaborant un modèle de classification de texte. L'objectif sous-jacent de ce modèle était de déterminer si une phrase contenait ou non un itinéraire, permettant ainsi d'affirmer ou d'infirmer cette présence. C'est ainsi que nous avons conçu ce modèle.

Configuration

Tokenizer : camembert-base
model : camembert-base
Epoch : 7
Batch size: 16

Metrics

```
Matrice de confusion :
[[71  0]
 [ 0 63]]

Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        71
           1       1.00      1.00      1.00        63

    accuracy                           1.00       134
   macro avg       1.00      1.00      1.00       134
weighted avg       1.00      1.00      1.00       134
```

Conclusion

À première vue, on pourrait penser que le modèle a maîtrisé la distinction entre les deux catégories de classe (phrases valides et phrases non valides). Cependant, lorsqu'il est testé avec des phrases aléatoires, il peut parfois détecter des itinéraires, entraînant ainsi des faux positifs.
Cependant, nous n'avons pas approfondi cette solution. Nous avons recentré notre attention sur le problème fondamental qui consistait à différencier les villes d'origine et les villes de destination.

NER sans transformers

Dans le premier modèle de reconnaissance d'entités nommées (NER) que nous avons construit. Notre pipeline a été élaboré en intégrant les modules tok2vec et ner proposés par spaCy. Cette séquence d'opérations permet initialement de fragmenter la phrase en tokens. Ensuite, chaque token est converti en une représentation vectorielle, capturant ainsi les informations relatives aux mots. Une fois cette première étape accomplie, les représentations vectorielles sont transmises au module NER, qui se charge d'assigner des étiquettes à chaque mot, générant en sortie une structure indiquant les entités nommées détectées.

Configuration

lang = "fr"
Pipeline : ["tok2vec","ner"]
Tok2vec : spacy.Tok2Vec.v2
Ner : spacy.TransitionBasedParser.v2
Tokenizer : spacy.Tokenizer.v1
Batch size: 1000
Optimizer : Adam.v1
Patience : 1600
Max_steps : 20000

Metrics

```
================================== Results ==================================

TOK      100.00
NER P    96.22
NER R    95.42
NER F    95.82
SPEED    292


============================== NER (per type) ==============================

              P       R       F
ORIGIN    97.46   95.83   96.64
ARRIVAL   95.00   95.00   95.00
```

NER avec transformers

Dans le dernier modèle de reconnaissance d'entités que nous avons développé, nous avons choisi d'intégrer les transformers. Parallèlement au modèle NER précédemment décrit, nous utilisons une pipeline, mais composée cette fois ci des modules "transformer" et "ner" de spaCy. Le processus global reste similaire, où le premier module fragmente la phrase en tokens et les transforme en représentations vectorielles pour capturer les informations des mots. La différence majeure réside dans la capacité du modèle à saisir le contexte du token dans l'ensemble de la phrase. L'étape du NER reste inchangée, attribuant des étiquettes à chaque mot et produisant en sortie une structure indiquant les entités nommées détectées.

Configuration

lang = "fr"
Pipeline : ["transformer","ner"]
transformer : spacy-transformers.TransformerModel.v3 (camembert-base)
Ner : spacy.TransitionBasedParser.v2
Tokenizer : spacy.Tokenizer.v1
Batch size: 128
Optimizer : Adam.v1
Patience : 1600
Max_steps : 20000

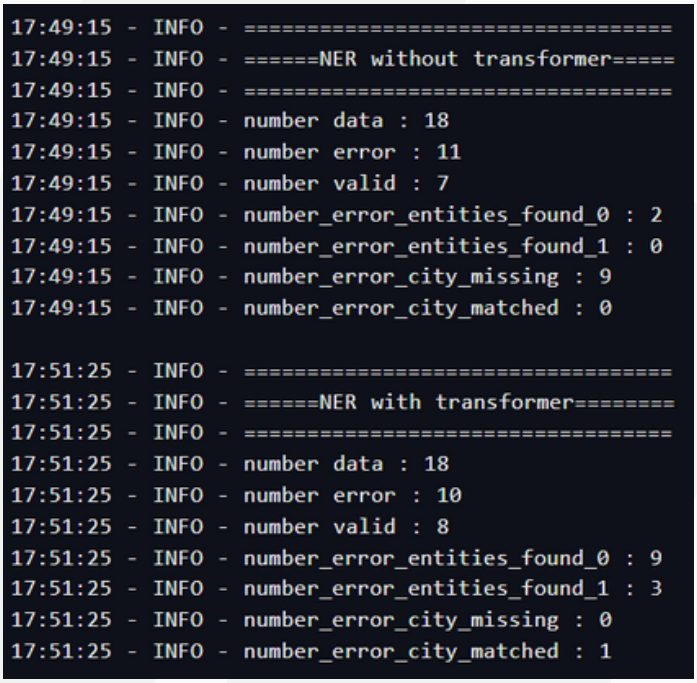
Metrics

```
================================== Results ==================================

TOK     100.00
NER P   91.70
NER R   96.67
NER F   94.12
SPEED   140


============================== NER (per type) ==============================

              P       R       F
ORIGIN    95.90   97.50   96.69
ARRIVAL   87.79   95.83   91.63
```

Comparaison des modèles

Pour évaluer et comparer les différents modèles de reconnaissance d'entités nommées (NER) obtenus, nous avons développé un script. Ce script lit un fichier CSV en entrée, contenant un ensemble de phrases que les modèles n'ont jamais rencontrées, en particulier du point de vue syntaxique, en raison de la complexité de la génération de telles phrases.

Les résultats obtenus sont les suivants :



number_error_entities_found_0 : Indique le nombre de phrase dont le modèle n'a trouvé aucune entités nommées.

number_error_entities_found_1 : Indique le nombre de phrase dont le modèle a trouvé une seule entité nommée.

number_error_city_missing : Indique le nombre de phrase dont le modèle à identifié les deux villes faisant partie de la même catégorie.

number_error_city_matched : Indique le nombre de phrase dont le modèle à inversé la ville d'origine et la ville de départ.

Conclusion

Ces métriques soulignent que le modèle qui n'utilise pas les transformers semble mieux détecter les noms de villes, mais il a du mal à contextualiser si la ville mentionnée est d'origine ou de destination, souvent regroupant les villes dans la même catégorie. En revanche, le modèle utilisant les transformers éprouve des difficultés à localiser les villes dans la phrase, mais il évite de classer les villes observées dans la même catégorie.

Cela met en lumière le fait que lorsqu'on s'éloigne du contexte dans lequel le modèle a été entraîné, des difficultés apparaissent. Les options d'amélioration incluraient l'enrichissement du jeu de données avec des exemples plus variés et l'exploration itérative du nombre de phrases générées afin d'approcher au mieux une généralisation optimale du modèle à partir d'exemples, notamment pour mieux remplir son rôle sur des tournures de phrases qu'il n'aurait jamais vues.