NLP Workshop

19-09-2024

Haeji Yun

Dans le but d'analyser l'expérience des voyageurs Airbnb à Paris, nous avons appliqué les techniques NLP sur les 1,794,006 commentaires voyageurs rédigés dans 47 langues différentes.

Les données proviennent de *Inside Airbnb*, un site web d'investigation qui rapporte et visualise les données extraites d'Airbnb.

Ci-dessous sont quelques exemples de commentaires :

```
1: Superbe découverte que cet appartement d Hugo , parfaitement situé en plein cœur
du Marais . Il est très bien équipé , y compris la cuisine qui vous permet de
préparer des repas et magnifiquement meublé <br/>Le lit est très confortable et
la douche parfaite . Etant au dernier étage, il est super lumineux avec une
belle vue sur les toits de Paris . Supermarché en bas de l immeuble et plein de
petits restaurants qu Hugo nous a gentiment recommandés  . Arrivée autonome, ce
qui est très pratique .  Je vous recommande sans hésiter l appartement d Hugo
que vous apprécierez très certainement et nous serons ravis de revenir

2: El apartamento es mejor que en las fotos. Todo es nuevo y esta impecable. Paty y
su familia fueron muy amables siempre dispuestos a contestar y ayudar y me senti
muy acompañada por ellos en todo momento (yo viajo sola) . El barrio tiene de
todo y esta cerca de todo. Se puede volver de noche sola tranquila porque hay un
bares en las esquinas y siempre hay gente en la calle. TIene diferentes medios
de transporte cercanos. Sin duda volveria a este simpatico lugar!

3: Merci à Fabienne pour sa réactivité et son accueil chaleureux. Très beau
logement !
```

Les technqiues NLP appliquées comprennent :
- La détection de langues
- La traduction de langues non-anglophones en anglais
- L'analyse de sentiment
- La classification multi-labels

1. Détection de Langue

La détection de la langue permet d'identifier la langue dans laquelle chaque commentaire est rédigé et d'appréhender la diversité des voyageurs Airbnb à Paris à travers la répartition des langues utilisées.

Pour la détection de langue, nous avons utilisé la librairie *langdetect*, qui prend en charge 55 langues.

> **langdetect**
>
> La librairie *langdetect* est un port direct de la librairie *language-detection* de Google, convertissant le code de Java en Python.

Il y a deux façons de détecter la langue avec la librarie. Nous allons appliquer les deux méthodes sur un commentaire Airbnb:

```
commentaire = "Tout s'est bien déroulé. Merci bien."
```

La première méthode utilise la fonction *detect* pour déterminer la langue du texte. Cette fonction retourne le code de langue correspondant au texte fourni. Pour le commentaire *"Tout s'est bien déroulé. Merci bien."*, la langue détectée est 'fr', ce qui signifie que le commentaire est en français.

```
from langdetect import detect
detect(commentaire)
```

```
'fr'
```

La deuxième méthode consiste à prédire les probabilités pour les langues les plus probables à l'aide de la fonction *detect_langs*. Pour notre commentaire, la probabilité qu'il soit en français est très proche de 1. Le nombre de langues retournées dépend de la pertinence et de la probabilité de chaque langue pour le texte donné.

```
from langdetect import detect_langs
detect_langs(commentaire)
```

```
[fr:0.9999983865073611]
```

Dans le projet, nous avons utilisé la première méthode avec la fonction *detect*.

Puisque nous appliquons la fonction sur un grand volume de données, il est important de prévoir les erreurs potentielles afin d'éviter l'interruption de l'exécution:

- La fonction *detect* accepte uniquement des chaînes de caractères en entrée. Si une entrée non textuelle est fournie, elle lèvera une TypeError avec le message "expected str, got Type".
- Les émojis, la ponctuation et les caractères spéciaux étant des chaînes de caractères valides en Python, ils ne provoqueront pas de TypeError, mais la fonction ne pourra pas détecter la langue, entraînant le message d'erreur "Error: No features in text".

Voici quelques exemples d'erreurs que nous pouvons rencontrer sur tels commentaires:

	comments	error message
0	NaN	expected string or bytes-like object, got 'float'
1	!	No features in text.
2	:)	No features in text.
3	.	No features in text.
4	10	No features in text.

Pour détecter les erreurs, nous pouvons utiliser l'exception *LangDetectException* qui signale les cas où la fonction *detect* ne parvient pas à identifier une langue. Dans le projet, nous avons considéré toutes les exceptions comme une langue inconnue.

Nous avons créé une fonction qui détecte la langue en gérant les exception avec le bloc *try-except*. Ce bloc permet d'intercepter les erreurs et de les traiter sans interrompre brusquement le programme:

- Le bloc *try* contient le code à exécuter : si l'entrée est une chaîne de caractères, la fonction *detect* est appliquée, sinon elle est classée comme langue inconnue.
- Si une exception se produit, l'exécution passe directement au bloc *except*, qui traite l'erreur en la classant comme langue inconnue.

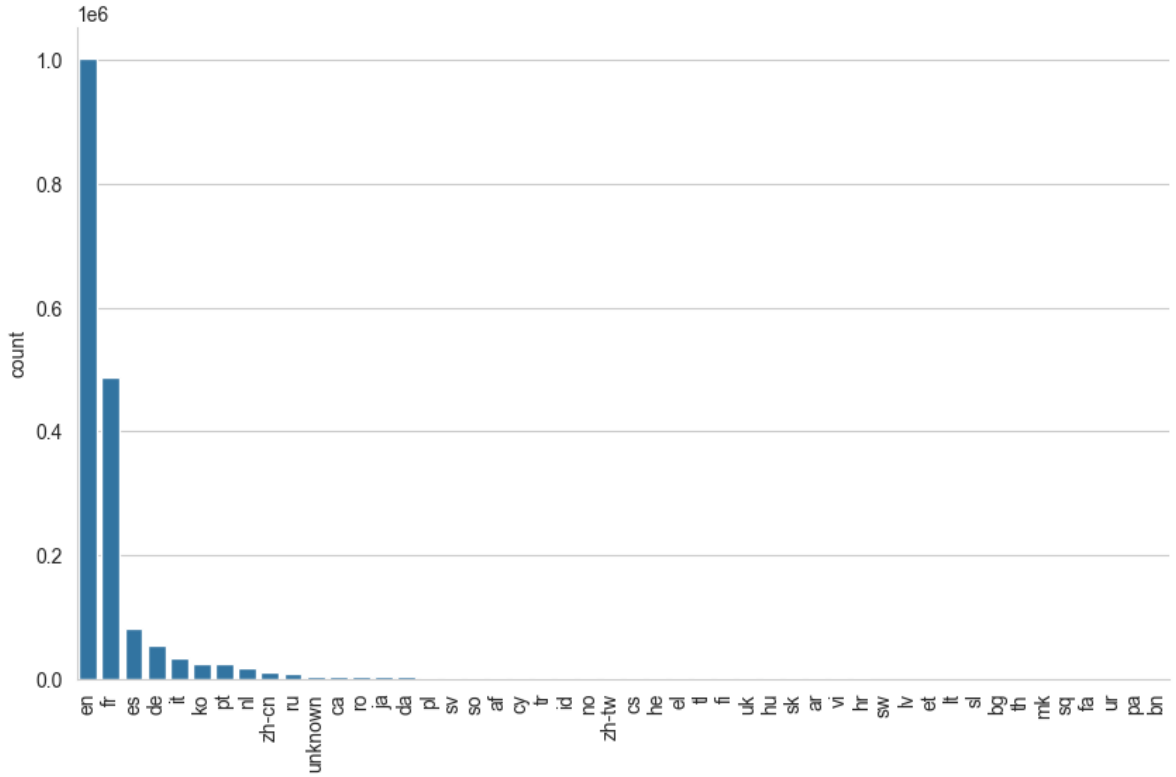
```
def detect_language(text):
    try:
        if isinstance(text, str):
            return detect(text)
        else:
            return "unknown"
    except LangDetectException:
        return "unknown"
```

Nous observons bien tous les exceptions sont détectées comme langue inconnue.

```
df_bis['language'] = df_bis['comments'].apply(detect_language)
df_bis[['comments', 'language']]
```

	comments	language
0	NaN	unknown
1	!	unknown
2	:)	unknown
3	.	unknown
4	10	unknown
5	Tout s'est bien déroulé. Merci bien.	fr

Le graphique de répartition des langues detectées ci-dessous montre que plus de la moitié des commentaires sont rédigés en anglais. De plus, 90% des commentaires sont rédigés en 5 langues : l'anglais, le français, l'espagnol, l'allemand et l'italien.



2. Traduction

Il existe plusieurs bibliothèques Python permettant de traduire du texte. Nous avons opté pour *deep-translator* en raison de sa simplicité d'utilisation, sans nécessiter de configurations d'API. Il s'agit d'une bibliothèque open-source qui donne accès à plusieurs traducteurs populaires tels que Google Translate, DeepL, Yandex, Pons, Microsoft Translator, etc.

Nous utilisons le traducteur GoogleTranslator de la bibliothèque, à initialiser en précisant la langue d'origine et la langue cible. Par exemple, pour traduire le commentaire *"Tout s'est bien déroulé. Merci bien."*, nous pouvons initialisé le traducteur en précisant le français comme langue source avec le paramètre *source='fr'* et l'anglais comme langue cible avec le paramètre *target='en'*. Ensuite, en appliquant le traducteur au commentaire avec la méthode *translate*, nous obtenons sa version anglaise "Everything went well. Thank you very much.".

```
from deep_translator import GoogleTranslator
translator = GoogleTranslator(source='fr', target='en')
translator.translate(commentaire)
```

```
'Everything went well. Thank you very much.'
```

Il est également possible d'initialiser le traducteur pour qu'il détecte automatiquement la langue d'origine en utilisant le paramètre *source='auto'*. GoogleTranslator prend en charge 133 langues différentes.

```
from deep_translator import GoogleTranslator
translator = GoogleTranslator(source='auto', target='en')
translator.translate(commentaire)
```

```
'Everything went well. Thank you very much.'
```

Les principales erreurs à prévoir concernent les entrées nulles ou les entrées trop longues. GoogleTranslator accepte 5000 caractères maximum. Si l'entrée est nulle ou trop longue, il renvoie le message d'erreur "text must be a valid text with maximum 5000 characters, otherwise it cannot be translated".

Dans le projet, nous avons créé une fonction de traduction capable de gérer ces erreurs :
- Le traducteur n'est pas appliqué lorsque le commentaire est vide.
- Le traducteur n'est pas appliqué aux commentaires trop longs. Seuls 0,0024 % des commentaires dans les données dépassent la limite de caractères. Si une plus grande partie des données est affectée, la troncation du texte serait une solution.
- Trois tentatives de traduction sont effectuées lorsqu'une exception de traduction est levée, pour éviter une interruption due à des erreurs de réseaux ou de serveur.

```
def translate(text, dest='en', retries=3, max_length=4500):
    if pd.isna(text) or (isinstance(text, str) and text.strip() == ''):
        return text
    if len(text) > max_length:
        return "[TEXT TOO LONG - SKIPPED]"
    for _ in range(retries):
        try:
            translator = GoogleTranslator(source='auto', target=dest)
            translated = translator.translate(text)
            return translated
        except Exception as e:
            print(f"\nTranslation error: {e}, retrying in 3 seconds...")
            time.sleep(3)
    return "[TRANSLATION FAILED]"
```

Nous avons également créé une fonction permettant de sauvegarder régulièrement la progression des traductions. Cette fonction crée un fichier pour enregistrer la traduction lorsque le fichier de sauvegarde n'existe pas, et ajoute les nouvelles traductions au fichier si celui-ci existe déjà. Comme le jeu de données est relativement volumineux avec 800.000 entrées, cette méthode permet d'éviter la perte de données déjà traduites en cas d'arrêt imprévu.

```
def save_progress(df, filename):
    if not os.path.exists(filename):
        df.to_csv(filename, mode='w', index=False)
    else:
        df.to_csv(filename, mode='a', header=False, index=False)
    print(f"Progress saved to {filename}")
```

Pour traduire l'ensemble des commentaires, nous effectuons et sauvegardons la traduction par intervalle de 1000 commentaires. Nous avons également activé une barre de progression pour suivre l'avancement de la traduction.

```
from tqdm import tqdm
save_interval = 1000
total_rows = len(df)
df['comments_en'] = None
for start in range(0, total_rows, save_interval):
    end = min(start + save_interval, total_rows)
    tqdm.pandas(desc=f"Processing rows {start}to {end}")
    df.loc[start:end,'comments_en']=df.loc[start:end,'comments'].progress_apply(translate)
    save_progress(df.iloc[start:end], f'translation_progress.csv')
df.to_csv('df_translation.csv', index = False)
```

3. Analyse de Sentiment

Pour l'analyse de sentiment, nous avons utilisé le modèle **bert-base-multilingual-uncased-sentiment** de librairie *transformers*. C'est un modèle BERT affiné pour l'analyse de sentiment sur des avis de produits en six langues : anglais, néerlandais, allemand, français, espagnol et italien. Le modèle prédit le sentiment de l'avis en nombre d'étoiles de 1 à 5.

> **i** modèles BERT
>
> **BERT** (Bidirectional Encoder Representations from Transformers) est un modèle de langage pré-entraîné développé par Google, utilisant une architecture basée sur les transformers. Il est pré-entraîné sur une grande quantité de données textuelles (comme Wikipédia et BookCorpus).
> Le modèle **bert-base-multilingual-uncased** est une version plus petite de BERT, pré-entraînée sur des textes dans 104 langues différentes sans distinction de majuscules/minuscules.
> Le modèle **bert-base-multilingual-uncased-sentiment** est une version affinée du modèle original bert-base-multilingual-uncased. Il a été spécifiquement ajusté pour l'analyse de sentiment.

Puisque le modèle est capable de traiter directement des textes dans ces six langues, il est possible d'appliquer le modèle sur des textes en anglais, néerlandais, allemand, français, espagnol ou italien, sans recourir à une étape de traduction. Cela permet de préserver la précision des prédictions en prenant en compte les nuances linguistiques propres à chaque langue.

Tokenization
Les modèles basés sur des transformers ne peuvent pas avoir en entrée des chaînes de texte brut. Le texte doit être d'abord tokenisé et encodé sous forme de vecteur numérique.

Nous pouvons réaliser cela grâce à la classe *AutoTokenizer* qui permet de charger rapidement le tokenizer associé à un modèle pré-entraîné.

Nous pouvons initializer le tokenizer avec la méthode *from_pretrained()* en lui précisant le modèle, puis l'appliquer directement sur le texte avec la méthode *encode()*. Nous obtenons la séquence finale des identifiants de tokens.

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(
  'nlptown/bert-base-multilingual-uncased-sentiment')
tokenizer.encode(commentaire)
```

```
[101, 12666, 161, 112, 10182, 11338, 43800, 119, 83595, 11338, 119, 102]
```

Ce que fait le tokenizer du modèle bert-base-multilingual-uncased-sentiment

-Il convertit le texte en minuscules.
-Il découpe le texte d'entrée en tokens.
-Des tokens spéciaux [CLS] et [SEP] sont ajoutés à la séquence tokenisée.
La plupart des modèles de transformeurs nécessitent des tokens spéciaux pour marquer le début et la fin des phrases. Dans le cas des modèles de type BERT, un token [CLS] est ajouté au début, et un token [SEP] est ajouté à la fin.
-Chaque token est remplacé par son ID correspondant dans le vocabulaire du tokenizer.

Le format retourné par défaut est une liste Python, ce qui n'est pas adapté pour BERT, qui accepte au maximum 512 tokens sous forme de tenseur. Pour préparer le texte dans le bon format, nous devons ajuster certains paramètres lors de l'initialisation du tokenizer:
- *max_length* : fixe la longueur maximale des séquences à encoder à 512 tokens.
- *truncation* : active la troncation des séquences dépassant 512 tokens avec *truncation=True*.
- *return_tensors* : retourne les données en format tenseur avec soit *return_tensors='pt'* pour PyTorch, soit *return_tensors='tf'* pour TensorFlow.

```
tokens=tokenizer.encode(commentaire,max_length=512,truncation=True,return_tensors='pt')
tokens
```

```
tensor([[  101, 12666,   161,   112, 10182, 11338, 43800,   119, 83595, 11338,
           119,   102]])
```

Ainsi nous obtenons les ID de tokens en tenseurs.

Modèle

Maintenant que le texte est au bon format, nous pouvons l'entrer dans le modèle. La classe *AutoModelForSequenceClassification* permet de charger automatiquement des modèles pré-entraînés et ajustés pour les tâches de classification de séquences spécifiques.

Nous initialisons le modèle **bert-base-multilingual-uncased-sentiment** avec la méthode *from_pretrained()*, en spécifiant son nom. Ensuite, nous appliquons le modèle sur le texte que nous avons converti en tenseurs.

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained(
  'nlptown/bert-base-multilingual-uncased-sentiment')
result = model(tokens)
result.logits
```

```
tensor([[-2.3824, -1.8174,  0.9035,  1.7424,  1.3676]],
       grad_fn=<AddmmBackward0>)
```

La sortie du modèle est constituée de logits — des scores de prédiction non normalisés pour chaque classe. Nous sélectionnons ensuite l'index avec le score le plus élevé pour déterminer la prédiction du modèle. Comme les index sont compris entre 0 et 4 (alors que les classes vont de 1 à 5), nous ajoutons 1 à l'index pour obtenir la classe correspondante.

Dans notre exemple, la classe prédite est 4, ce qui signifie que le score de sentiment prédit est 4.

```
int(torch.argmax(result.logits))+1
```

```
4
```

Pour appliquer la tokenisation et utiliser le modèle **bert-base-multilingual-uncased-sentiment** sur notre jeu de données entier, nous avons créé une fonction qui exécute toutes ces étapes :

```
def sentiment_score(text):
    tokens=tokenizer.encode(text,max_length=512,truncation=True,return_tensors='pt')
    with torch.no_grad(): #Disable gradient calculation
        result = model(tokens)
    return int(torch.argmax(result.logits))+1
df['sentiment'] = df['comments_en'].apply(sentiment_score)
```

Puisque nous effectuons la prédiction des sentiments sur un grand jeu de données entraînant un temps d'exécution long, nous avons opté à sauvegarder les prédictions à intervalles réguliers de 1000 commentaires afin d'éviter toute perte de résultats en cas d'arrêt imprévu. Nous avons également activé une barre de progression pour suivre l'avancement :

```
save_interval = 1000
total_rows = len(df)

for start in range(0, total_rows, save_interval):
    end = min(start + save_interval, total_rows)
    tqdm.pandas(desc=f"Processing rows {start}-{end}")
    df.loc[start:end,'sentiment']=df.loc[start:end,
    'comments_en'].progress_apply(sentiment_score)
    save_progress(df.iloc[start:end], f'sentiment_progress.csv')
df.to_csv('df_sentiment.csv', index = False)
print("\nAll rows processed!")
```

4. Multi-label Classification

Avec la classification zero-shot, nous cherchons à classer les commentaires selon les cinq thèmes principaux des avis Airbnb: la localisation, la communication, le lit, l'appartement et le quartier. Pour cela, nous utilisons le modèle **bart-large-mnli**, une version de BART affinée sur le jeu de données MNLI.

> **i** BART & jeu de données MNLI
>
> **BART** (Bidirectional and Auto-Regressive Transformer) est un modèle transformeur développé par Facebook AI Research. Il est conçu pour être très polyvalent et peut à la fois comprendre et générer du texte. Il est utilisé dans des tâches comme la synthèse de texte, la traduction automatique, et les réponses aux questions.
> Le **jeu de données MNLI** (Multi-Genre Natural Language Inference) est un vaste ensemble de données conçu pour les tâches d'inférence en langage naturel, où un modèle doit déterminer la relation entre deux phrases: une prémisse et une hypothèse. Ce jeu de données contient des textes issus de divers genres, tels que la fiction, les documents gouvernementaux ou les dialogues parlés, permettant ainsi aux modèles entraînés sur MNLI de s'adapter à différents types de langage. MNLI est largement utilisé pour évaluer la capacité des modèles à effectuer des tâches de raisonnement, et il sert de base pour affiner des modèles comme BART ou RoBERTa pour la classification zero-shot et d'autres tâches d'inférence en langage naturel.

Nous pouvons charger le modèle à l'aide de la classe *pipeline*, qui gère l'ensemble du processus, de la tokenisation à la prédiction, et fournit des résultats prêts à l'emploi. Ainsi, lorsqu'un modèle est initialisé avec la classe *pipeline*, nous pouvons l'utiliser directement sur du texte brut sans passer par l'étape de tokenisation comme nous avons fait pour l'analyse de sentiment.

Nous initialisons le modèle **bart-large-mnli** avec *pipeline* en spécifiant son nom. Pour utiliser le modèle sur le texte, nous lui fournissons le texte et les labels. Le paramètre *multi-label=True* (par défaut, *multi-label=False*) permet d'activer la prédiction multi-classe.

Le modèle retourne un score pour chaque label. Les scores, compris entre 0 et 1, représentent la probabilité que le texte corresponde à chaque label, et sont triés par ordre décroissant.

```
from transformers import pipeline

labels = ['bed', 'apartment', 'neighborhood', 'location', 'communication']

model_zs = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
model_zs(commentaire, candidate_labels=labels, multi_label = True)
```

```
{'sequence': "Tout s'est bien déroulé. Merci bien.",
```

```
 'labels': ['communication', 'neighborhood', 'apartment', 'location', 'bed'],
 'scores': [0.9855712056159973,
  0.8129251003265381,
  0.7709853649139404,
  0.4758288264274597,
  0.38189730048179626]}
```

Dans notre exemple, la probabilité que le commentaire porte sur le label *communication* est la plus élevée, avec un score de 0.98 par rapport aux autres classes.

Lorsque le paramètre *multi_label* n'est pas activé, le modèle adopte une approche de classification exclusive. Le modèle choisit un seul label qui correspond le mieux au texte.

```
model_zs(commentaire, candidate_labels=labels)
```

```
{'sequence': "Tout s'est bien déroulé. Merci bien.",
 'labels': ['communication', 'neighborhood', 'apartment', 'bed', 'location'],
 'scores': [0.6694537401199341,
  0.10473016649484634,
  0.09397656470537186,
  0.08557500690221786,
  0.04626457393169403]}
```

Dans le projet, nous avons créé deux fonctions:
- Une fonction de classification multi-label qui retourne un score pour chaque label.
- Une fonction de sélection de labels qui choisit les labels ayant des scores supérieurs à 0,9, ou le label avec le meilleur score si aucun label n'a un score supérieur à 0,9.

```
def zero_shot(text):
    return model_zs(text, candidate_labels=labels, multi_label = True)

def extract_label(output):
    labels = output['labels']
    scores = output['scores']
    best_labels = [label for label, score in zip(labels, scores) if score >= 0.9]
    return best_labels if best_labels else [labels[0]]
```

Comme pour la traduction et la prédiction de sentiment, nous avons sauvegardé les prédictions à intervalles réguliers de 1000 commentaires avec une barre de progression, afin d'éviter toute perte de résultats en cas d'arrêt imprévu.

```
df['label_classification'] = None

save_interval = 1000
total_rows = len(df)

for start in range(0, total_rows, save_interval):
    end = min(start + save_interval, total_rows)
    tqdm.pandas(desc=f"Processing rows {start}-{end}")
    df.loc[start:end, 'label_classification'] = df.loc[start:end,
    'comments_en'].progress_apply(zero_shot)
    save_progress(df.iloc[start:end], f'zeroshot_progress.csv')

df['labels'] = df['label_classification'].apply(extract_label)

df.to_csv('df_final.csv', index = False)
print("\nCompleted!")
```

Full code
https://github.com/haejiyun/airbnb-reviews/blob/main/airbnb_nlp.py

Streamlit
https://airbnb-guest-reviews.streamlit.app/