الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي

Université Badji Mokhtar - Annaba
Badji Mokhtar – Annaba University

جامعة باجي مختار – عنابـــة

Faculté : Technologie
Département : Informatique
Domaine : Mathématiques- Informatique
Programme : Informatique
Spécialisation : Systèmes Informatique et Décisions

Rapport de Travaux Pratiques

Module : Web Mining

Analyse des sentiments en utilisant le prompt engineering, les embeddings et le fine-tuning avec la base de données HARD (Hotel Arabic Reviews Dataset)

Réalisé par :

- **Nom :** Ghouaoua
- **Prénom :** Insaf

Enseignante : Dr.Legrini Samira

Année Universitaire : 2024/2025

1. Introduction

L'analyse de sentiment est un domaine clé du traitement automatique du langage naturel (NLP) qui vise à extraire et classifier les opinions exprimées dans les textes, souvent en termes de polarité : positive, négative ou neutre. Cette technologie est largement utilisée dans divers domaines, tels que le marketing, la gestion de la réputation en ligne, et l'expérience utilisateur, en particulier dans des industries comme celle de l'hôtellerie, où les avis clients jouent un rôle crucial.

Ce rapport se concentre sur l'analyse de sentiments appliquée aux avis d'hôtels en arabe, en utilisant la base de données **HARD (Hotel Arabic-Reviews Dataset)**. Cette base contient des avis rédigés en arabe, une langue qui présente des défis uniques pour le NLP en raison de sa richesse morphologique, de sa structure complexe, et de la rareté des ressources adaptées.

L'objectif principal de ce travail est de comparer trois approches distinctes pour l'analyse de sentiment :

1. **Prompt Engineering**, qui exploite des modèles de langage avancés pour générer des prédictions sans nécessiter un entraînement sur mesure.
2. **Embeddings**, qui utilisent des représentations vectorielles des textes pour des classifications supervisées.
3. **Fine-tuning**, qui consiste à ajuster un modèle pré-entraîné sur une tâche spécifique en utilisant les données annotées du jeu de données.

À travers ce rapport, nous décrivons en détail les étapes nécessaires, depuis le prétraitement des données jusqu'à l'évaluation des performances des différentes méthodes, afin d'identifier l'approche la plus adaptée à cette tâche.

2. Développement

1. Prétraitement des données

Le prétraitement des données est une étape cruciale dans tout projet de traitement automatique du langage naturel (TALN). Elle permet d'améliorer la qualité des données en éliminant les éléments inutiles, standardisant les formats, et rendant ainsi les données exploitables pour les modèles d'apprentissage. Voici les étapes détaillées du prétraitement effectuées, accompagnées d'explications.

1.1. Chargement et étiquetage des données

```python
import pandas as pd

data = pd.read_excel('/content/balanced-reviews.xlsx')

# Ajout d'une colonne 'Sentiment' basée sur la note
def label_sentiment(rating):
    if rating >= 4:
        return 'Positive'
    elif rating == 3:
        return 'Neutral'
    else:
        return 'Negative'

data['Sentiment'] = data['rating'].apply(label_sentiment)

# Affichage des données étiquetées
print(data[['rating', 'review', 'Sentiment']].head())
```

Explication :

- Les données de critiques d'hôtels sont importées à partir d'un fichier Excel.
- Une nouvelle colonne, Sentiment, est créée pour catégoriser les critiques en *positive*, *négative*, ou *neutre* selon leur note. Cela simplifie l'analyse en réduisant la complexité des données.

1.2. Suppression du texte non-arabe

```python
import re

def remove_non_arabic(text):
    arabic_pattern = re.compile(r'[^\u0600-\u06FF\s]')
    return arabic_pattern.sub('', text)

data['Cleaned_Review'] = data['review'].apply(remove_non_arabic)

# Affichage des critiques nettoyées
print(data[['review', 'Cleaned_Review']].head())
```

Explication :

- Cette étape filtre les caractères non-arabes pour garantir que seules les parties en langue arabe sont conservées.
- Elle est essentielle pour travailler uniquement sur les textes utiles, excluant les symboles, chiffres, et autres éléments non pertinents.

1.3. Suppression de la ponctuation et des caractères spéciaux

```python
import string
import re

def remove_punctuation(text):
    arabic_punctuation = ['،', '؛', '؟', '٪', '…', '_', '(', ')', '[', ']', '{', '}', '<', '>']
    all_punctuation = string.punctuation + ''.join(arabic_punctuation)

    # Supprimer la ponctuation (arabe + générale)
    text = text.translate(str.maketrans('', '', all_punctuation))
    return text

data['Cleaned_Review_No_Punctuation'] = data['Cleaned_Review'].apply(remove_punctuation)

# Afficher les résultats
print(data[['Cleaned_Review', 'Cleaned_Review_No_Punctuation']].head())
```

Explication :

- Cette étape supprime tous les signes de ponctuation (générale et arabe) pour éviter qu'ils n'influencent les analyses futures.
- Les critiques sont ainsi transformées en texte brut.

1.4. La suppression des lettres répétées excessives

```python
import re

def remove_repeated_letters(text):
    # Remplacer 3 répétitions ou plus par 2
    text = re.sub(r'(.)\1{2,}', r'\1\1', text)
    return text

data['Normalized_No_Repeated_Letters'] = data['Cleaned_Review_No_Punctuation'].apply(remove_repeated_letters)

# Afficher les résultats
print(data[['Cleaned_Review_No_Punctuation', 'Normalized_No_Repeated_Letters']].head())
```

Explication :

- Les répétitions excessives de lettres (ex. "رااائع" devient "رائع") sont limitées à deux occurrences pour standardiser les textes.
- Cela est particulièrement utile dans les critiques où les répétitions sont souvent utilisées pour exprimer une émotion.

1.5. Standardisation des mots dialectaux en arabe littéraire

```python
# Dictionnaire de transformation du dialecte koweïtien vers l'arabe standard
dialect_to_standard_dict = {
    "شلون": "كيف",
    "زين": "جيد",
    "شفيك": "ما بك",
    "ليش": "لماذا",
    "وين": "أين",
    "شنو": "ماذا",
    "خوش": "ممتاز",
    "وايد": "كثير",
    "الحين": "الآن",
    "يبي": "يريد",
    "عشان": "لأن",
    "شوف": "انظر",
    "ماكو": "لا يوجد",
    "قاعد": "يفعل حالياً",
    "ماعليه": "لا بأس",
    "إنتي": "أنتِ",
    "إنت": "أنت",
    "أبوي": "أبي",
    "يمه": "أمي",
    "مالت عليك": "عار عليك",
    "كلش": "أبداً",
}

def dialect_to_standard(text):
    for dialect_word, standard_word in dialect_to_standard_dict.items():
        text = text.replace(dialect_word, standard_word)
    return text

data['Standardized_Review'] = data['Normalized_No_Repeated_Letters'].apply(dialect_to_standard)

# Afficher les résultats
print(data[['Normalized_No_Repeated_Letters', 'Standardized_Review']].head())
```

Explication :

- Un dictionnaire est utilisé pour convertir les mots du dialecte koweïtien en arabe standard. Cela garantit une homogénéité dans les données linguistiques.
- Cette étape facilite l'analyse ultérieure en réduisant la variabilité lexicale due aux différents dialectes.

1.6. Normalisation du texte arabe

```python
def normalize_arabic(text):
    text = text.replace('آ', 'ا')  # Remplacer 'آ' par 'ا'
    text = text.replace('أ', 'ا')  # Remplacer 'أ' par 'ا'
    text = text.replace('َ', '')   # Supprimer 'fatha'
    text = text.replace('ً', '')   # 'tanwin fatha'
    text = text.replace('ُ', '')   # 'damma'
    text = text.replace('ٌ', '')   # 'tanwin damma'
    text = text.replace('ِ', '')   # 'kasra'
    text = text.replace('ٍ', '')   # 'tanwin kasra'
    text = text.replace('ْ', '')   # 'sukun'
    text = text.replace('ّ', '')   # 'shadda'
    return text

data['Normalized_Review'] = data['Standardized_Review'].apply(normalize_arabic)

# Afficher les résultats
print(data[['Standardized_Review', 'Normalized_Review']].head())
```

Explication :

- Remplacer 'آ' et 'أ' par 'ا' : En arabe, 'آ' et 'أ' sont des variantes d'écriture de la lettre 'ا' (alif). Pour simplifier et uniformiser le texte, ces caractères sont remplacés par la forme de base 'ا'.
- Supprimer les voyelles courtes et les signes diacritiques (fatha, damma, kasra, tanwin, sukun, shadda) : Ces signes sont utilisés en arabe pour indiquer la prononciation des voyelles et d'autres modifications phonétiques. Cependant, dans une analyse de texte, ces signes ne sont pas toujours nécessaires, car ils n'affectent pas le sens du mot. Leur suppression permet de rendre le texte plus uniforme pour les modèles de traitement de texte.
- Le texte normalisé est ainsi épuré des caractères qui ne contribuent pas à son sens, facilitant ainsi l'analyse par des modèles NLP (traitement du langage naturel).

1.7. Sauvegarde des données dans un fichier Excel

```python
import pandas as pd
from google.colab import files

output_data = data[['no', 'Hotel name', 'rating', 'user type', 'room type', 'nights', 'Normalized_Review', 'Sentiment']]

print(output_data.head())

output_filename = '/content/hotel_reviews_preprocessed.xlsx'
output_data.to_excel(output_filename, index=False)

# Télécharger le fichier
files.download(output_filename)

print("Le fichier Excel a été sauvegardé sous le nom 'hotel_reviews_preprocessed.xlsx'.")
```

Explication:

- Les données prétraitées sont sauvegardées dans un fichier Excel afin de les utiliser plus tard.

Conclusions :

Ces étapes de prétraitement ont permis d'obtenir des critiques d'hôtels nettoyées, homogénéisées et prêtes à être utilisées dans les étapes suivantes du projet, notamment l'analyse et la modélisation.

Référence :

Lien vers le code complet "Prétraitement_des_données" sur Google Colab:

https://colab.research.google.com/drive/1C-z1YMChoyDa3mWpCE5ytCqtFMK6aYSO?usp=sharing

2. Méthodes d'analyse de sentiment

2.1. Prompt Engineering :

1. Utilisation de ChatGPT pour le prompt engineering :

Nous avons commencé par utiliser **ChatGPT-4o mini** pour générer un prompt afin d'analyser les sentiments des avis d'hôtel. Le prompt que nous avons utilisé était :
"Please analyze the sentiment of hotel reviews in the provided Excel file. The file contains columns such as 'Hotel name', 'rating', 'user type', 'room type', 'nights', and 'Normalized_Review'. For each review in the 'Normalized_Review' column, classify the sentiment as positive, neutral, or negative based on the content of the review. Also, consider the 'rating' column to refine the sentiment analysis. Once you've analyzed the reviews, provide the sentiment classification for each review by adding a new column "predicted_sentiment" in the Excel file that I provided."

ChatGPT a répondu en nous expliquant la structure du fichier et en détaillant les étapes nécessaires pour effectuer l'analyse de sentiment. Il a procédé à l'analyse des avis et a ajouté une nouvelle colonne intitulée predicted_sentiment avec les classifications des sentiments pour chaque avis dans le fichier Excel.

Lien vers chat de ChatGPT-4o mini:
https://chatgpt.com/share/675dc5bd-79f0-800d-bec9-3eba0b29e27b

2. Traitement, comparaison des résultats et évaluation des performances:

```python
import pandas as pd
from sklearn.metrics import classification_report

file_path = '/content/testgpt.xlsx'
df = pd.read_excel(file_path)

print(df.columns)


actual_sentiment = df['Sentiment']
predicted_sentiment = df['predicted_sentiment']

actual_sentiment = actual_sentiment.str.upper()
predicted_sentiment = predicted_sentiment.str.upper()

report = classification_report(actual_sentiment, predicted_sentiment, target_names=['POSITIVE', 'NEGATIVE'])

print(report)
```

Après avoir reçu le fichier Excel mis à jour par ChatGPT, contenant la colonne **predicted_sentiment**, nous avons comparé les résultats obtenus avec les **sentiments réels** des avis (colonne **Sentiment**) et les **sentiments prédits** par ChatGPT. À l'aide de Python et de la bibliothèque **pandas**, nous avons chargé le fichier Excel et comparé ces deux colonnes pour obtenir une évaluation des performances de l'analyse de sentiment réalisée par ChatGPT.
Nous avons utilisé **scikit-learn** pour générer un rapport de classification en comparant les valeurs réelles de sentiment (colonne **Sentiment**) et les valeurs prédites par ChatGPT (colonne **predicted_sentiment**).

3. Interprétation des résultats:

```
Index(['no', 'Hotel name', 'rating', 'user type', 'room type', 'nights',
       'Normalized_Review', 'Sentiment', 'predicted_sentiment'],
      dtype='object')
              precision    recall  f1-score   support

    POSITIVE       1.00      1.00      1.00        18
    NEGATIVE       1.00      1.00      1.00        81

    accuracy                           1.00        99
   macro avg       1.00      1.00      1.00        99
weighted avg       1.00      1.00      1.00        99
```

Les résultats montrent que **la précision, le rappel et le score F1 sont tous de 1.00** pour chaque catégorie (positif et négatif), ce qui signifie que les prédictions de ChatGPT sont parfaitement alignées avec les sentiments réels des avis.
L'**exactitude** de 1.00 indique que toutes les prédictions étaient correctes, ce qui montre que ChatGPT a bien analysé les sentiments des avis.
Les moyennes **macro** et **pondérées** parfaites indiquent que le modèle de ChatGPT a traité les avis de manière équilibrée, sans biais entre les deux catégories.

Conclusion :

En résumé, l'analyse de sentiment réalisée par ChatGPT a donné des résultats parfaits, avec une correspondance complète entre les sentiments réels et les prédictions pour les avis d'hôtel. Cela démontre que l'approche utilisée est efficace pour l'analyse des sentiments, même sans utiliser de modèles d'apprentissage automatique. Les résultats sont cohérents et fiables pour les données de ce fichier.

Référence :

Lien vers le code complet "Classification_report_prompt" sur Google Colab:

https://colab.research.google.com/drive/1xWcM6wzKFOuiM8GpesvYMPzRAgLDoNAS?usp=sharing

2.2. Embeddings :

1. Tokenization et Suppression des Stopwords

1.1 Tokenization

```python
import pandas as pd
from nltk.tokenize import word_tokenize
import nltk

file_path = '/content/hotel_reviews_preprocessed.xlsx'
data = pd.read_excel(file_path)

data['Tokenized_Review'] = data['Normalized_Review'].apply(word_tokenize)

# Afficher les résultats
print(data[['Normalized_Review', 'Tokenized_Review']].head())
```

Explication:

- La tokenisation a été effectuée à l'aide de la bibliothèque NLTK, avec la fonction word_tokenize qui découpe les avis en mots individuels.

1.2 Suppression des Stopwords

```python
from nltk.corpus import stopwords
import nltk

# Importe des stopwords de NLTK
arabic_stopwords = set(stopwords.words('arabic'))

# Liste des stopwords supplémentaires
additional_stopwords = {
    'و', 'في', 'على', 'من', 'إلى', 'عن', 'هذا', 'هذه', 'ذلك', 'تلك',
    'هو', 'هي', 'أنا', 'أنت', 'أنتم', 'هم', 'هن', 'كان', 'كانت', 'لك',
    'له', 'لها', 'لنا', 'عند', 'عليه', 'عليها', 'فيها', 'بين', 'هناك',
    'هنا', 'إذا', 'لكن', 'أو', 'بل', 'ثم', 'إن', 'أن', 'ألا', 'إلا',
    'حتى', 'كل', 'أي', 'بعض', 'بعد', 'قبل', 'أمام', 'خلف', 'تحت', 'فوق',
    'مع', 'لذلك', 'ذلك', 'مازال', 'قد',
    'عندما', 'مثل', 'إليها', 'بها', 'بهم', 'منهم', 'عليهم', 'بكم', 'أيضا'
}
# Fusionner les listes
arabic_stopwords.update(additional_stopwords)

def remove_stopwords(tokens):
    return [word for word in tokens if word not in arabic_stopwords]

data['Filtered_Review'] = data['Tokenized_Review'].apply(remove_stopwords)

# Afficher les résultats
print(data[['Tokenized_Review', 'Filtered_Review']].head())
```

Explication:

- Les stopwords (mots comme "إلى", "تلك", "و", etc.) ont été supprimés pour améliorer la qualité des données. Pour ce faire, nous avons utilisé une liste de stopwords en arabe provenant de NLTK et l'avons complétée avec des mots courants supplémentaires.

2. Génération des embeddings

Dans cette étape, nous avons exploré trois approches différentes pour générer des embeddings à partir des avis d'hôtel en arabe. Les embeddings sont des représentations numériques des mots, permettant de capturer leurs relations sémantiques. Les trois méthodes utilisées sont **AraBERT**, **Word2Vec (CBOW)** et **FastText (Skip-Gram)**.

2.1 Génération des Embeddings avec AraBERT

```python
from transformers import AutoTokenizer, AutoModel
import torch

# Charger le tokenizer et le modèle
tokenizer = AutoTokenizer.from_pretrained('aubmindlab/bert-base-arabertv02')
model = AutoModel.from_pretrained('aubmindlab/bert-base-arabertv02')

# Fonction pour générer des embeddings
def generate_embedding(text):
    # Tokenisation et encodage
    inputs = tokenizer(text, return_tensors='pt', truncation=True, padding=True, max_length=512)
    with torch.no_grad():
        outputs = model(**inputs)
    embedding = outputs.last_hidden_state[:, 0, :].squeeze().numpy()
    return embedding

data['Embedding'] = data['Filtered_Review'].apply(lambda tokens: generate_embedding(" ".join(tokens)))

# Afficher les résultats
print(data[['Filtered_Review', 'Embedding']].head())
```

Explication:

- Nous avons utilisé **AraBERT**, un modèle préentraîné pour la langue arabe, pour générer ces embeddings. Le processus implique de transformer chaque revue en un vecteur dense représentant sa signification.

2.2 Génération des Embeddings avec Word2Vec (CBOW)

```python
from gensim.models import Word2Vec

# Entraîner un modèle Word2Vec CBOW
word2vec_cbow_model = Word2Vec(data['Tokenized_Review'], vector_size=100, window=5, sg=0, min_count=1)

# Générer des embeddings
def generate_word2vec_cbow_embedding(tokens):
    embeddings = [word2vec_cbow_model.wv[word] for word in tokens if word in word2vec_cbow_model.wv]
    return sum(embeddings) / len(embeddings) if embeddings else [0] * 100

data['Embedding'] = data['Filtered_Review'].apply(generate_word2vec_cbow_embedding)
```

Explication:

- Nous avons utilisé le modèle Word2Vec dans sa configuration **Continuous Bag of Words (CBOW).** Il est entraîné sur les avis tokenisés pour apprendre les relations entre les mots. Dans le mode CBOW, le modèle prédit un mot cible en se basant sur son contexte, c'est-à-dire les mots qui l'entourent.
- Pour chaque avis, nous avons calculé la moyenne des vecteurs de tous les mots présents dans l'avis. Si un mot n'était pas dans le vocabulaire du modèle, il était ignoré.

2.3 Génération des Embeddings avec FastText (Skip-Gram)

```python
from gensim.models import FastText

# Entraîner un modèle FastText Skip-Gram
fasttext_sg_model = FastText(data['Tokenized_Review'], vector_size=100, window=5, sg=1, min_count=1)

# Générer des embeddings
def generate_fasttext_sg_embedding(tokens):
    embeddings = [fasttext_sg_model.wv[word] for word in tokens if word in fasttext_sg_model.wv]
    return sum(embeddings) / len(embeddings) if embeddings else [0] * 100

data['Embedding'] = data['Filtered_Review'].apply(generate_fasttext_sg_embedding)
```

Explication:

- Nous avons utilisé le modèle **FastText** avec l'architecture **Skip-Gram,** il est entraîné sur les avis tokenisés. Contrairement à Word2Vec, FastText représente chaque mot comme un ensemble de sous-mots (par exemple, "hotels" serait représenté par "hot", "ote", "tel", etc.). Cela permet au modèle de générer des vecteurs même pour des mots rares ou inconnus.
- Nous avons calculé la moyenne des vecteurs de tous les mots présents dans chaque avis.

2.5 Enregistrement des Embeddings dans un fichier excel

```python
from google.colab import files

def format_embedding(embedding):
    return '[' + ', '.join(f'{x:.10f}' for x in embedding) + ']'

# Création 'un DataFrame
output_data = data[['no', 'Hotel name', 'rating', 'user type', 'room type', 'nights', 'Normalized_Review', 'Sentiment']]
output_data['Embedding'] = data['Embedding'].apply(format_embedding)

print(output_data.head())

# Enregistrement des données dans un fichier Excel
output_filename = '/content/hotel_reviews_embeddings.xlsx'
output_data.to_excel(output_filename, index=False)

files.download(output_filename)

print("Le fichier Excel a été sauvegardé sous le nom 'hotel_reviews_embeddings.xlsx'.")
```

Explication:

- Pour chacune des méthodes, une fois les embeddings générés, nous les avons ajoutés à un fichier Excel afin qu'ils soient prêts à être utilisés pour l'entraînement du modèle.

3. Entraînement d'un Classificateur

Dans cette étape, nous avons entraîné trois méthodes différentes pour la classification des sentiments : **CNN (Convolutional Neural Network), LSTM (Long Short-Term Memory),** et **RNN (Recurrent Neural Network)**. Chaque méthode exploite des architectures neuronales spécifiques adaptées aux caractéristiques séquentielles des données textuelles.

3.1 Recurrent Neural Network (RNN)

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

file_path = '/content/hotel_reviews_embeddings.xlsx'
data = pd.read_excel(file_path)

# Préparer les données
X = data['Embedding']
y = data['Sentiment']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Définir le mapping des labels
label_mapping = {'Negative': 0, 'Neutral': 1, 'Positive': 2}
y_train = [label_mapping[label] for label in y_train]
y_test = [label_mapping[label] for label in y_test]

# Dataset PyTorch
class SentimentDataset(Dataset):
    def __init__(self, embeddings, labels):
        self.embeddings = [eval(emb) for emb in embeddings]
        self.labels = labels

    def __len__(self):
        return len(self.embeddings)
```

```python
    def __getitem__(self, idx):
        embedding = torch.tensor(self.embeddings[idx], dtype=torch.float32)
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return embedding, label

train_dataset = SentimentDataset(X_train, y_train)
test_dataset = SentimentDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16)

# Définir le modèle RNN
class SentimentRNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers, dropout=0.2):
        super(SentimentRNN, self).__init__()
        self.rnn = nn.RNN(input_dim, hidden_dim, num_layers=n_layers, batch_first=True, dropout=dropout, nonlinearity='tanh')
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # RNN Forward Pass
        out, hidden = self.rnn(x)
        # Fully Connected Layer using the last hidden state
        out = self.fc(hidden[-1])
        return self.softmax(out)

# Hyperparamètres
input_dim = len(eval(X_train.iloc[0]))
hidden_dim = 128
output_dim = 3  # Nombre de classes
n_layers = 2
```

```
dropout = 0.2

model = SentimentRNN(input_dim, hidden_dim, output_dim, n_layers, dropout)

# Définir la fonction de perte et l'optimiseur
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Entraînement du modèle
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    total_loss = 0
    for embeddings, labels in train_loader:
        embeddings = embeddings.unsqueeze(1)  # Ajouter une dimension pour batch_first
        optimizer.zero_grad()
        outputs = model(embeddings)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {total_loss / len(train_loader):.4f}")
```

Explication:

- Préparation des données : Nous avons extrait les embeddings et les labels, converti les sentiments en valeurs numériques, puis divisé les données en ensembles d'entraînement et de test.
- Création du dataset : Nous avons défini une classe SentimentDataset pour structurer les embeddings et les labels en tenseurs exploitables par PyTorch.
- Chargement des données : Nous avons transformé les ensembles d'entraînement et de test en loaders (DataLoader) pour un traitement par mini-lots.
- Définition du modèle : Nous avons conçu un modèle RNN (SentimentRNN) avec une couche fully connected pour classifier les sentiments et une activation softmax pour calculer les probabilités.
- Configuration du modèle : Nous avons utilisé la Cross-Entropy Loss comme fonction de perte et Adam comme optimiseur avec des hyperparamètres définis.
- Entraînement : Nous avons entraîné le modèle sur 10 époques, mis à jour les poids après chaque lot et affiché la perte moyenne pour chaque époque.

3.2 Convolutional Neural Network (CNN)

```
    def __getitem__(self, idx):
        embedding = torch.tensor(self.embeddings[idx], dtype=torch.float32)
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return embedding, label

train_dataset = SentimentDataset(X_train, y_train)
test_dataset = SentimentDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16)

# Définir le modèle CNN
class SentimentCNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SentimentCNN, self).__init__()
        # Reduced kernel size to 1 to match input dimension
        self.conv1 = nn.Conv1d(in_channels=input_dim, out_channels=128, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=128, out_channels=64, kernel_size=1)
        self.pool = nn.MaxPool1d(1) # Reduced pooling kernel size for consistency
        self.fc1 = nn.Linear(64, 32)  # Changed input size to 64
        self.fc2 = nn.Linear(32, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.transpose(1, 2)  # Permuter les dimensions pour le CNN
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return self.softmax(x)
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

file_path = '/content/hotel_reviews_embeddings.xlsx'
data = pd.read_excel(file_path)

# Préparer les données
X = data['Embedding']
y = data['Sentiment']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Définir le mapping des labels
label_mapping = {'Negative': 0, 'Positive': 1}
y_train = [label_mapping[label] for label in y_train]
y_test = [label_mapping[label] for label in y_test]

# Dataset PyTorch
class SentimentDataset(Dataset):
    def __init__(self, embeddings, labels):
        self.embeddings = [eval(emb) for emb in embeddings]
        self.labels = labels

    def __len__(self):
        return len(self.embeddings)
```

```python
# Hyperparamètres
input_dim = len(eval(X_train.iloc[0]))  # Dimension des embeddings
output_dim = 2  # Binaire : Positif ou Négatif

model = SentimentCNN(input_dim, output_dim)

# Définir la fonction de perte et l'optimiseur
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Entraînement du modèle
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    total_loss = 0
    for embeddings, labels in train_loader:
        embeddings = embeddings.unsqueeze(1)
        optimizer.zero_grad()
        outputs = model(embeddings)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {total_loss / len(train_loader):.4f}")
```

Explication:

- Nous avons utilisé les CNN pour capturer des motifs locaux dans les données textuelles. Bien qu'ils soient couramment associés aux images, ils peuvent également être adaptés aux séquences de texte.
- Dans notre cas, des convolutions 1D ont permis de détecter des relations spécifiques entre les mots dans les embeddings.
- Grâce aux couches de convolution et de pooling, nous avons extrait des caractéristiques discriminantes qui facilitent la classification.

3.3 Long Short-Term Memory (LSTM)

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Charger les données
file_path = '/content/hotel_reviews_embeddings.xlsx'
data = pd.read_excel(file_path)

# Préparer les données
X = data['Embedding']
y = data['Sentiment']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Définir le mapping des labels
label_mapping = {'Negative': 0, 'Positive': 1}
y_train = [label_mapping[label] for label in y_train]
y_test = [label_mapping[label] for label in y_test]

# Dataset PyTorch
class SentimentDataset(Dataset):
    def __init__(self, embeddings, labels):
        self.embeddings = [eval(emb) for emb in embeddings]
        self.labels = labels

    def __len__(self):
        return len(self.embeddings)
```

```
    def __getitem__(self, idx):
        embedding = torch.tensor(self.embeddings[idx], dtype=torch.float32)
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return embedding, label

train_dataset = SentimentDataset(X_train, y_train)
test_dataset = SentimentDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16)

# Définir le modèle LSTM
class SentimentLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers=1):
        super(SentimentLSTM, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # LSTM retourne toutes les sorties et la dernière sortie cachée
        _, (hidden, _) = self.lstm(x)
        # On utilise la dernière couche cachée pour la classification
        hidden = hidden[-1]
        x = self.fc(hidden)
        return self.softmax(x)

# Hyperparamètres
hidden_dim = 128  # Nombre de dimensions dans les états cachés de LSTM
num_layers = 1  # Nombre de couches LSTM
input_dim = len(eval(X_train.iloc[0]))
```

```
output_dim = 2  # Binaire : Positif ou Négatif

# Instancier le modèle
model = SentimentLSTM(input_dim, hidden_dim, output_dim, num_layers)

# Définir la fonction de perte et l'optimiseur
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Entraînement du modèle
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    total_loss = 0
    for embeddings, labels in train_loader:
        embeddings = embeddings.unsqueeze(1)
        optimizer.zero_grad()
        outputs = model(embeddings)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {total_loss / len(train_loader):.4f}")
```

Explication:

- Nous avons opté pour les LSTM, un type de réseau de neurones récurrents (RNN), afin de gérer les séquences longues et les dépendances à long terme.
- Ce modèle est particulièrement adapté pour le traitement des textes, car il conserve le contexte des mots précédents lors de l'analyse des phrases.
- Nous avons utilisé les états cachés finaux des couches LSTM pour prédire les sentiments.

4. Entraînement d'un Classificateur

3.4 Évaluation du model

```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Évaluation du modèle
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for embeddings, labels in test_loader:
        embeddings = embeddings.unsqueeze(1)
        outputs = model(embeddings)
        preds = torch.argmax(outputs, dim=1)
        all_preds.extend(preds.numpy())
        all_labels.extend(labels.numpy())

# Rapport de classification
print(classification_report(all_labels, all_preds, target_names=['Negative',  'Positive']))
```

Explication:

- Nous avons évalué chaque combinaison d'embedding **(AraBERT, Word2Vec, et FastText)** et de classificateur **(RNN, CNN, et LSTM)** à l'aide des métriques de précision, rappel, F1-score et précision globale (accuracy). L'analyse est effectuée pour deux classes : **Positive** et **Negative**.

Les résultats obtenus :

1. Résultats pour AraBERT

a. AraBERT avec RNN

	precision	recall	f1-score	support
Negative	0.92	0.89	0.90	1149
Positive	0.86	0.89	0.87	851
accuracy			0.89	2000
macro avg	0.89	0.89	0.89	2000
weighted avg	0.89	0.89	0.89	2000

Explication:

- **Précision globale (accuracy) :** 89%
- **Analyse :**
  - Le rappel pour la **Negative** et **Positive** (0.89).
  - Meilleur F1-score pour la classe **Negative** (0.90) par rapport à **Positive** (0.87).

b. AraBERT avec CNN

	precision	recall	f1-score	support
Negative	0.96	0.84	0.90	1149
Positive	0.82	0.95	0.88	851
accuracy			0.89	2000
macro avg	0.89	0.89	0.89	2000
weighted avg	0.90	0.89	0.89	2000

Explication:

- **Précision globale (accuracy) :** 89%
- **Analyse :**
  - Rappel plus élevé pour la classe **Positive** (0.95), indiquant une meilleure détection des avis positifs.
  - Léger avantage en F1-score par rapport à RNN.

c. AraBERT avec LSTM

	precision	recall	f1-score	support
Negative	0.89	0.93	0.91	1149
Positive	0.90	0.85	0.88	851
accuracy			0.90	2000
macro avg	0.90	0.89	0.90	2000
weighted avg	0.90	0.90	0.90	2000

Explication:

- **Précision globale (accuracy) :** 90%
- **Analyse :**
  - Meilleure performance globale grâce à un équilibre entre précision et rappel.
  - F1-score le plus élevé pour les deux classes.

2. Résultats pour Word2Vec

a. Word2Vec avec RNN

	precision	recall	f1-score	support
Negative	0.81	0.87	0.84	1149
Positive	0.80	0.73	0.76	851
accuracy			0.81	2000
macro avg	0.81	0.80	0.80	2000
weighted avg	0.81	0.81	0.81	2000

Explication:

- **Précision globale (accuracy) :** 81%
- **Analyse :**
  - Performance modérée, avec une précision légèrement supérieure pour la classe **Negative** (0.87).

b. Word2Vec avec CNN

	precision	recall	f1-score	support
Negative	0.81	0.89	0.85	1149
Positive	0.82	0.72	0.77	851
accuracy			0.81	2000
macro avg	0.82	0.80	0.81	2000
weighted avg	0.82	0.81	0.81	2000

Explication:

- **Précision globale (accuracy) :** 81%
- **Analyse :**
  - Performance similaire à RNN, mais un rappel légèrement inférieur pour la classe **Positive** (0.72).

c. Word2Vec avec LSTM

	precision	recall	f1-score	support
Negative	0.78	0.91	0.84	1149
Positive	0.85	0.66	0.74	851
accuracy			0.80	2000
macro avg	0.82	0.79	0.79	2000
weighted avg	0.81	0.80	0.80	2000

Explication:

- **Précision globale (accuracy) :** 80%
- **Analyse :**
  - Rappel pour la classe **Negative** (0.91) plus élevé, mais précision globale légèrement inférieure.

3. Résultats pour FasText

a. FasText avec RNN

	precision	recall	f1-score	support
Negative	0.85	0.94	0.89	1149
Positive	0.91	0.78	0.84	851
accuracy			0.87	2000
macro avg	0.88	0.86	0.87	2000
weighted avg	0.88	0.87	0.87	2000

Explication:

- **Précision globale (accuracy) :** 87%
- **Analyse :**
  - Très bon rappel pour la classe **Negative** (0.94), mais F1-score légèrement plus faible pour **Positive** (0.84).

b. FasText avec CNN

	precision	recall	f1-score	support
Negative	0.86	0.94	0.90	1149
Positive	0.90	0.79	0.84	851
accuracy			0.88	2000
macro avg	0.88	0.86	0.87	2000
weighted avg	0.88	0.88	0.87	2000

Explication:

- **Précision globale (accuracy) :** 88%
- **Analyse :**
  - Légère amélioration par rapport à RNN, avec un F1-score équilibré entre les deux classes.

c. FasText avec LSTM

	precision	recall	f1-score	support
Negative	0.89	0.90	0.90	1149
Positive	0.86	0.85	0.86	851
accuracy			0.88	2000
macro avg	0.88	0.88	0.88	2000
weighted avg	0.88	0.88	0.88	2000

Explication:

- **Précision globale (accuracy) :** 88%
- **Analyse :**
  - Performances proches de CNN, mais F1-score légèrement plus constant entre les deux classes.

Comparaison générale :

- **Meilleur embedding :**
  - AraBERT offre les meilleures performances globales avec des scores plus élevés dans toutes les métriques.
- **Meilleur classificateur :**
  - LSTM est le plus performant dans la majorité des combinaisons, grâce à sa capacité à capturer les dépendances contextuelles.
- **Recommandation :**
  - **Combinaison AraBERT avec LSTM** offre la meilleure précision globale (90%) et le meilleur équilibre entre les classes.

Test du model

Dans cette partie, nous avons testé toutes les combinaisons d'*embeddings* et de *classificateurs* que nous avons déjà mises en œuvre, en prenant l'exemple d'**AraBERT** avec **RNN**.

```python
from transformers import AutoTokenizer, AutoModel
import torch

model_name = "aubmindlab/bert-base-arabertv2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
embedding_model = AutoModel.from_pretrained(model_name)

def get_embedding(text):
    tokens = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=128)
    with torch.no_grad():
        outputs = embedding_model(**tokens)
    return outputs.last_hidden_state[:, 0, :].squeeze(0).numpy()

def predict_sentiment(model, text):
    embedding = get_embedding(text)
    embedding_tensor = torch.tensor(embedding, dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    model.eval()
    with torch.no_grad():
        output = model(embedding_tensor)
        prediction = torch.argmax(output, dim=1).item()
    label_mapping = {0: "Negative", 2: "Positive"}
    return label_mapping[prediction]

while True:
    avis = input("Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : ")
    if avis.lower() == 'quit':
        print("Sortie du programme.")
        break

    classe = predict_sentiment(model, avis)
    print(f"L'avis est classé comme : {classe}")
```

Explication:

- Nous avons chargé le modèle AraBERT pour l'extraction des embeddings, ainsi que le tokenizer associé, afin de convertir les avis en texte en tokens que le modèle peut traiter.
- Grâce à la fonction get_embedding, nous extrayons les représentations numériques (embeddings) des avis en utilisant AraBERT, en récupérant l'embedding du token [CLS], qui résume le texte.
- La fonction predict_sentiment prend ces embeddings extraits et les passe à notre modèle LSTM (celui que nous avons entraîné) pour prédire si l'avis est positif, négatif ou neutre.
- Nous avons conçu une boucle interactive qui permet à l'utilisateur de saisir des avis en arabe. Tant que l'utilisateur ne tape pas "quit", le programme prédit et affiche le sentiment de chaque avis. Si l'utilisateur tape "quit", le programme se termine.

Résultats

```
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : استمتعت بكل لحظة في هذا الفندق. الإفطار كان لذيذًا والخدمات ممتازة
L'avis est classé comme : Positive
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : الفندق قديم، يحتاج إلى تجديد
L'avis est classé comme : Negative
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : فندق جيد
L'avis est classé comme : Positive
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) :  سيء للغاية
L'avis est classé comme : Negative
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : التكييف ضعيف للغاية
L'avis est classé comme : Negative
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : موقع الفندق ممتاز، قريب من كل شيء
L'avis est classé comme : Positive
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) : الحمام ملوث، والخدمة سيئة
L'avis est classé comme : Negative
Entrez un avis en arabe pour prédire le sentiment (ou tapez 'quit' pour quitter) :
```

Explication:

- Le modèle utilise le texte saisi par l'utilisateur pour prédire si l'avis est positif ou négatif.
- Les avis exprimant des sentiments favorables (comme "**فندق جيد**", "**موقع الفندق ممتاز**"، "**قريب من كل شيء**") sont classés comme "Positive", tandis que ceux exprimant des plaintes ou des aspects négatifs (comme "**التكييف ضعيف للغاية**", "**سيء للغاية**", "**الحمام ملوث، والخدمة سيئة**") sont classés comme "Negative".

Conclusion

Dans cette étape, nous avons préparé les données en les tokenisant et en supprimant les stopwords, puis nous avons généré des embeddings en utilisant différentes méthodes, notamment **AraBERT**, **Word2Vec** et **FastText**. Ensuite, nous avons entraîné plusieurs modèles de classification, tels que **RNN**, **CNN**, et **LSTM**, pour classer les sentiments des avis.

Après l'entraînement, nous avons évalué les performances de chaque combinaison d'embedding et de classifieur en testant leur capacité à prédire les sentiments des avis à l'aide des données de test. Les prédictions obtenues ont été comparées aux étiquettes réelles à l'aide d'un rapport de classification, fournissant des métriques telles que la précision, le rappel et le score F1 pour chaque classe (négatif, positif). Enfin, nous avons effectué un test pratique en transformant un texte brut en embeddings avec une des méthodes utilisées, puis en appliquant un modèle entraîné pour classer le sentiment du texte.

Références :

Lien vers le code complet "Embeddings_Arabert_RNN" sur Google Colab:

https://colab.research.google.com/drive/1sdmtag8FYhHNDo9AlGP5t1f5uI8YeH8r?usp=sharing

Lien vers le code complet "Embeddings_Arabert_CNN" sur Google Colab:

https://colab.research.google.com/drive/1o2icbnt55k7Jz_S7fwCrn4YzTR_lDHAw?usp=sharing

Lien vers le code complet "Embeddings_Arabert_LSTM" sur Google Colab:

https://colab.research.google.com/drive/1ubQjdkenymfccYHGdTIBG_z8v9ijiNnA?usp=sharing

Lien vers le code complet "Embeddings_ Word2Vec_RNN" sur Google Colab:

https://colab.research.google.com/drive/1oLKZ4dYTgiD92gy7wTXO9uHS04GHQKHa?usp=sharing

Lien vers le code complet "Embeddings_ Word2Vec_CNN" sur Google Colab:

https://colab.research.google.com/drive/17BgBXg77_8N7l_kS6WLehQgwzq74skmv?usp=sharing

Lien vers le code complet "Embeddings_ Word2Vec_LSTM" sur Google Colab:

https://colab.research.google.com/drive/1C1H1-a5jRC9CDnjSOn9FlJ60dRe3jU_1?usp=sharing

Lien vers le code complet "Embeddings_ FasText_RNN" sur Google Colab:

https://colab.research.google.com/drive/1tExp7CFC42VxLHVZtxKwDf7vwFQ7cKXo?usp=sharing

Lien vers le code complet "Embeddings_ FasText_CNN" sur Google Colab:

https://colab.research.google.com/drive/19EDKSbC29DmyNqCOMtYcJOwJT8kDewQK?usp=sharing

Lien vers le code complet "Embeddings_ FasText_LSTM" sur Google Colab:

https://colab.research.google.com/drive/1tLpeBt0BFw-C1jath9F5QaqNgQOIRvPR?usp=sharing

2.1. Fine-tuning :

1. FineTuning avec AraGPT2

1. L'importation des bibliothèques

```
import pandas as pd
from transformers import GPT2TokenizerFast, GPT2ForSequenceClassification, Trainer, TrainingArguments
from arabert.aragpt2.grover.modeling_gpt2 import GPT2ForSequenceClassification as ArGPT2ForSequenceClassification
from arabert.preprocess import ArabertPreprocessor
from torch.utils.data import Dataset
```

2. Chargement des données

```
data = pd.read_excel('/content/hotel_reviews_preprocessed.xlsx')
print(data.head())
```

Explication:

- Nous avons chargé les données **prétraitées** des avis d'hôtel depuis un fichier Excel à l'aide de pandas. Cela nous a permis de prévisualiser rapidement les premières lignes du jeu de données pour vérifier son contenu.

3. Préparation des données pour le fine-tuning

```
X = data['Normalized_Review']
y = data['Sentiment']

# Mapper les sentiments (Positive -> 1, Negative -> 0)
y = y.map({'Positive': 1, 'Negative': 0})
```

Explication:

- Nous avons extrait les colonnes pertinentes du dataset : les avis normalisés (pour l'entrée du modèle) et les sentiments (pour les étiquettes). Les sentiments ont été ensuite convertis en étiquettes numériques : "Positive" devient 1 et "Negative" devient 0.

4. Chargement du modèle pré-entraîné AraGPT2

```
MODEL_NAME = 'aubmindlab/aragpt2-base'
arabert_prep = ArabertPreprocessor(model_name=MODEL_NAME)
model = ArGPT2ForSequenceClassification.from_pretrained(MODEL_NAME, num_labels=2)
tokenizer = GPT2TokenizerFast.from_pretrained(MODEL_NAME)

# Assigner un token de padding
tokenizer.pad_token = tokenizer.eos_token

# Définir pad_token_id dans la configuration du modèle
model.config.pad_token_id = tokenizer.pad_token_id
```

Explication:

- Nous avons utilisé un modèle pré-entraîné d'**AraGPT2** de l'AUBMindLab, qui est une version arabophone de **GPT-2**. Nous avons également préparé le préprocesseur **ArabertPreprocessor** pour transformer les données d'entrée selon les spécificités de ce modèle.
- Le modèle a été ajusté pour accepter un token de padding (utilisation de eos_token ou d'un nouveau token), ce qui est essentiel pour traiter des séquences de longueur variable.

5. **Crétion d'une classe personnalisée pour le dataset**

```python
class TextDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = self.labels[idx]
        return item

    def __len__(self):
        return len(self.labels)
```

Explication:

- Nous avons créé une classe personnalisée TextDataset qui nous permet de manipuler efficacement les données tokenisées et les labels correspondants.

6. **Préparation des données d'entrée pour l'entraînement**

```python
# Tokeniser les données
encodings = tokenizer(list(X), truncation=True, padding=True, max_length=256)
train_dataset = TextDataset(encodings, list(y))
```

Explication:

- Nous avons ensuite tokenisé les avis d'hôtel, en les tronquant et en les complétant pour qu'ils aient tous une longueur de 256 tokens, et créé le dataset d'entraînement.

7. **Configuration les arguments d'entraînement**

```python
training_args = TrainingArguments(
    output_dir="./aragpt2-finetuned",
    num_train_epochs=10,
    # Le taux d'apprentissage
    learning_rate=5e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    # Évaluer après chaque époque
    evaluation_strategy="epoch",
)
```

Explication:

- Nous avons défini des arguments d'entraînement via **TrainingArguments**. Nous avons opté pour un taux d'apprentissage de **5e-5**, **10** époques d'entraînement, et une taille de batch de 16.

8. Création de l'objet Trainer

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=train_dataset,
    tokenizer=tokenizer
)
```

Explication:

- L'objet Trainer a été créé pour orchestrer l'entraînement du modèle en utilisant les données et les paramètres d'entraînement définis.

9. Lancement du fine-tuning

```
trainer.train()
```

Explication:

- Nous avons lancé l'entraînement du modèle en utilisant la méthode train() de l'objet Trainer. Cette étape a permis d'ajuster les poids du modèle en fonction de notre jeu de données spécifique.

10. Évaluation sur les données de test

```
from sklearn.metrics import classification_report
import torch

test_data = pd.read_excel('/content/hotel_reviews_test.xlsx')
X_test = test_data['Normalized_Review']
y_test = test_data['Sentiment'].map({'Positive': 1, 'Negative': 0})

encodings_test = tokenizer(list(X_test), truncation=True, padding=True, max_length=256)
test_dataset = TextDataset(encodings_test, list(y_test))

# Faire des prédictions sur le jeu de test
predictions = trainer.predict(test_dataset)

predicted_labels = predictions.predictions.argmax(axis=-1)  # Indices des classes prédites

# Afficher le classification_report
print(classification_report(y_test, predicted_labels, target_names=["Negative", "Positive"]))
```

Explication:

- Nous avons chargé un jeu de données de test pour évaluer les performances du modèle fine-tuné. Les avis ont été tokenisés, et un dataset de test a été créé.
- Nous avons ensuite utilisé la méthode predict() du Trainer pour générer des prédictions, qui ont été comparées aux étiquettes réelles.
- Un rapport de classification a été généré pour évaluer la performance du modèle.

Résultat obtenue :

	precision	recall	f1-score	support
Negative	0.95	0.94	0.94	1190
Positive	0.91	0.93	0.92	809
accuracy			0.93	1999
macro avg	0.93	0.93	0.93	1999
weighted avg	0.93	0.93	0.93	1999

Explication:

- **Precision (Précision)**

  - La précision mesure la capacité du modèle à ne prédire comme positives que les instances réellement positives.
    - **Classe Negative** : La précision est de **0.95** (95%), indiquant que 95% des prédictions pour la classe *Negative* sont correctes.
    - **Classe Positive** : La précision est de **0.91** (91%). Cela montre que le modèle est aussi performant pour cette classe malgré une légère baisse par rapport à la classe *Negative*.

- **Recall (Rappel)**

  - Le rappel mesure la capacité du modèle à identifier correctement toutes les instances positives réelles.
    - **Classe Negative** : Le rappel est de **0.94**, ce qui signifie que 94% des exemples négatifs réels ont été correctement prédits.
    - **Classe Positive** : Le rappel est de **0.93**, montrant une bonne capacité à retrouver les exemples positifs.

- **F1-Score**

  - L'**F1-Score** combine précision et rappel pour donner une mesure équilibrée.
    - **Classe Negative** : L'F1-Score atteint **0.94**.
    - **Classe Positive** : L'F1-Score est de **0.92**.
      Ces résultats montrent une performance homogène entre les deux classes.

- **Support**

  - Le *support* représente le nombre d'échantillons dans chaque classe :
    - **Classe Negative** : **1190** exemples.
    - **Classe Positive** : **809** exemples.
      Bien que la classe *Negative* soit plus représentée, le modèle parvient à maintenir un bon équilibre de performance.

- **Accuracy (Exactitude)**

  - L'**accuracy** globale est de **0.93** (93%), ce qui indique que 93% des prédictions du modèle sont correctes sur l'ensemble des **1999** échantillons.

- **Macro Average et Weighted Average**

  - **Macro avg** : La moyenne des métriques de précision, rappel et F1-Score entre les classes, sans pondération. Ici, elle est de **0.93** pour chaque métrique.
  - **Weighted avg** : La moyenne pondérée des métriques, tenant compte de la taille de chaque classe. Les valeurs restent également à **0.93**.

11. Test du model

```python
from transformers import AutoTokenizer, GPT2ForSequenceClassification

model_name = "/content/drive/MyDrive/aragpt2-finetuned"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = GPT2ForSequenceClassification.from_pretrained(model_name)
print("Le modèle et le tokenizer ont été chargés avec succès !")
```

```python
import torch

def predict_sentiment(review):
    # Prétraiter le texte
    new_review_cleaned = arabert_prep.preprocess(review)
    inputs = tokenizer(new_review_cleaned, return_tensors="pt", padding=True, truncation=True, max_length=256)

    # Faire une prédiction
    with torch.no_grad():
        model.eval()  # Modele mode évaluation
        outputs = model(**inputs)

    # Calculer la prédiction
    logits = outputs.logits
    predicted_class = torch.argmax(logits, dim=-1).item()

    # Return sentiment prédit
    sentiment = "Positive" if predicted_class == 1 else "Negative"
    return sentiment

while True:
    new_review = input("Entrez un avis (ou tapez 'quit' pour quitter) : ")

    if new_review.lower() == 'quit':
        print("Au revoir!")
        break

    sentiment = predict_sentiment(new_review)

    print(f"Sentiment prédit : {sentiment}")
```

Explication:

- Nous avons chargé le modèle pré-entraîné (aragpt2-finetuned) et le tokenizer pour la classification des séquences.
- Nous avons nettoyé l'avis en utilisant la fonction arabert_prep.preprocess avant de le tokeniser (le transformer en indices numériques).
- Nous avons utilisé le modèle pour prédire le sentiment de l'avis (positif ou négatif).
- Nous avons déterminé la classe prédite en fonction des logits générés par le modèle.
- Nous avons permis à l'utilisateur de saisir un avis pour obtenir la prédiction du sentiment.
- Nous avons ajouté une option "quit" permettant de quitter la boucle.

Résultat

```
Entrez un avis (ou tapez 'quit' pour quitter) : استمتعت بكل لحظة في هذا الفندق. الإفطار كان لذيذًا والخدمات ممتازة. سأعود بالتأكيد
Sentiment prédit : Positive
Entrez un avis (ou tapez 'quit' pour quitter) : الفندق قديم، يحتاج إلى تجديد
Sentiment prédit : Negative
Entrez un avis (ou tapez 'quit' pour quitter) : موقع الفندق ممتاز، قريب من كل شيء
Sentiment prédit : Positive
Entrez un avis (ou tapez 'quit' pour quitter) :
```

Explication:

- Le modèle utilise le texte saisi par l'utilisateur pour prédire si l'avis est positif ou négatif.
- Les avis exprimant des sentiments favorables (comme "**فندق جيد**", "**موقع الفندق ممتاز**"، " **قريب من كل شيء**") sont classés comme "Positive", tandis que ceux exprimant des plaintes ou des aspects négatifs (comme "**الحمام ملوث، والخدمة سيئة**", " **سيء للغاية**", "**التكييف ضعيف للغاية**") sont classés comme "Negative".

Conclusion

Nous avons réussi à fine-tuner un modèle AraGPT2 pour effectuer une classification binaire des sentiments sur des avis d'hôtel en arabe. Nous avons adapté les données et le modèle à nos besoins spécifiques et avons configuré les paramètres d'entraînement de manière appropriée pour maximiser la performance du modèle. L'évaluation a permis de tester l'efficacité du fine-tuning.

Les prochaines étapes pourraient inclure l'optimisation des hyperparamètres, l'utilisation de données de validation pour éviter le surapprentissage, ou l'intégration du modèle dans une application réelle pour des prédictions en temps réel.

Référence

Lien vers le code complet " FineTuning_AraGPT2" sur Google Colab:

https://colab.research.google.com/drive/14grADCN5izYQU0ucXJNdutjLELfnaDN7?usp=sharing

2. FineTuning avec CAMelBert

12. L'importation des bibliothèques

```
import pandas as pd
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments
from peft import get_peft_model, LoraConfig, TaskType
from torch.utils.data import Dataset
```

13. Chargement des données

```
data = pd.read_excel('/content/hotel_reviews_preprocessed.xlsx')
```

Explication:

- Nous avons chargé les données **prétraitées** des avis d'hôtel depuis un fichier Excel à l'aide de pandas.

14. Préparation des données pour le fine-tuning

```
X = data['Normalized_Review']
y = data['Sentiment']

# Mapper les sentiments (Positive -> 1, Negative -> 0)
y = y.map({'Positive': 1, 'Negative': 0})
```

Explication:

- Nous avons extrait les colonnes pertinentes du dataset : les avis normalisés (pour l'entrée du modèle) et les sentiments (pour les étiquettes). Les sentiments ont été ensuite convertis en étiquettes numériques : "Positive" devient 1 et "Negative" devient 0.

15. Chargement du modèle CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment

```
MODEL_NAME = 'CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment'

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSequenceClassification.from_pretrained(MODEL_NAME, num_labels=3, ignore_mismatched_sizes=True)
# Préparer QLoRA
lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS,  # Classification de séquence
    inference_mode=False,
    r=16,
    lora_alpha=32,
    lora_dropout=0.1,
    target_modules=["query", "key", "value"],
)
model = get_peft_model(model, lora_config)
```

Explication:

- Nous avons chargé le modèle **CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment** pour la classification de sentiments en arabe, en utilisant le tokenizer et le modèle pré-entraîné, avec trois labels (positif, négatif, neutre).Le modèle a été ajusté pour accepter un token de padding (utilisation de eos_token ou d'un nouveau token), ce qui est essentiel pour traiter des séquences de longueur variable.
- Nous avons configuré **QLoRA** en ajustant les paramètres comme le rang et le dropout, afin d'adapter le modèle pré-entraîné de manière efficace et légère pour la classification.
- Nous avons intégré la configuration LoRA au modèle via la fonction get_peft_model, permettant un fine-tuning rapide tout en conservant de bonnes performances pour l'analyse des sentiments.

16. Crétion d'une classe personnalisée pour le dataset

```python
class TextDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)
```

Explication:

- Nous avons créé une classe personnalisée TextDataset qui nous permet de manipuler efficacement les données tokenisées et les labels correspondants.

17. Préparation des données d'entrée pour l'entraînement

```python
# Tokeniser les données
encodings = tokenizer(list(X), truncation=True, padding=True, max_length=256)
train_dataset = TextDataset(encodings, list(y))
```

Explication:

- Nous avons ensuite tokenisé les avis d'hôtel, en les tronquant et en les complétant pour qu'ils aient tous une longueur de 256 tokens, et créé le dataset d'entraînement.

18. Configuration les arguments d'entraînement

```python
training_args = TrainingArguments(
    output_dir="./camelbert-finetuned-qlora",
    num_train_epochs=10,  # Nombre d'époques
    learning_rate=1e-4,  # Taux d'apprentissage pour QLoRA
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    evaluation_strategy="epoch",  # Évaluation à chaque époque
    save_strategy="epoch",  # Sauvegarde à chaque époque
    fp16=True,  # Utiliser le calcul en précision mixte
    save_total_limit=2,
    push_to_hub=False,  # Ne pas envoyer les modèles sur le Hub
)
```

Explication:

- Nous avons défini des arguments d'entraînement via **TrainingArguments**. Nous avons opté pour un taux d'apprentissage de **5e-5**, **10** époques d'entraînement, et une taille de batch de 16.

19. Création de l'objet Trainer

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    eval_dataset=dataset,
    tokenizer=tokenizer,
)
```

Explication:

- L'objet Trainer a été créé pour orchestrer l'entraînement du modèle en utilisant les données et les paramètres d'entraînement définis.

20. Lancement du fine-tuning

```
trainer.train()
```

Explication:

- Nous avons lancé l'entraînement du modèle en utilisant la méthode train() de l'objet Trainer. Cette étape a permis d'ajuster les poids du modèle en fonction de notre jeu de données spécifique.

21. Évaluation sur les données de test

```
from sklearn.metrics import classification_report
import torch

test_data = pd.read_excel('/content/hotel_reviews_test.xlsx')
X_test = test_data['Normalized_Review']
y_test = test_data['Sentiment'].map({'Positive': 1, 'Negative': 0})

encodings_test = tokenizer(list(X_test), truncation=True, padding=True, max_length=256)
test_dataset = TextDataset(encodings_test, list(y_test))

# Faire des prédictions sur le jeu de test
predictions = trainer.predict(test_dataset)

predicted_labels = predictions.predictions.argmax(axis=-1)  # Indices des classes prédites

# Afficher le classification_report
print(classification_report(y_test, predicted_labels, target_names=["Negative", "Positive"]))
```

Explication:

- Nous avons chargé un jeu de données de test pour évaluer les performances du modèle fine-tuné. Les avis ont été tokenisés, et un dataset de test a été créé.
- Nous avons ensuite utilisé la méthode predict() du Trainer pour générer des prédictions, qui ont été comparées aux étiquettes réelles.
- Un rapport de classification a été généré pour évaluer la performance du modèle.

Résultat obtenue :

	precision	recall	f1-score	support
Negative	0.95	0.95	0.95	1190
Positive	0.93	0.93	0.93	809
accuracy			0.94	1999
macro avg	0.94	0.94	0.94	1999
weighted avg	0.94	0.94	0.94	1999

Explication:

- **Precision (Précision)**

  - La précision mesure la capacité du modèle à ne prédire comme positives que les instances réellement positives.
    - **Classe Negative** : La précision est de **0.95** (95%), indiquant que 95% des prédictions pour la classe *Negative* sont correctes.
    - **Classe Positive** : La précision est de **0.93** (93%). Cela montre que le modèle est aussi performant pour cette classe malgré une légère baisse par rapport à la classe *Negative*.

- **Recall (Rappel)**

  - Le rappel mesure la capacité du modèle à identifier correctement toutes les instances positives réelles.
    - **Classe Negative** : Le rappel est de **0.95**, ce qui signifie que 94% des exemples négatifs réels ont été correctement prédits.
    - **Classe Positive** : Le rappel est de **0.93**, montrant une bonne capacité à retrouver les exemples positifs.

- **F1-Score**

  - L'**F1-Score** combine précision et rappel pour donner une mesure équilibrée.
    - **Classe Negative** : L'F1-Score atteint **0.95**.
    - **Classe Positive** : L'F1-Score est de **0.93**.
      Ces résultats montrent une performance homogène entre les deux classes.

- **Support**

  - Le *support* représente le nombre d'échantillons dans chaque classe :
    - **Classe Negative** : **1190** exemples.
    - **Classe Positive** : **809** exemples.
      Bien que la classe *Negative* soit plus représentée, le modèle parvient à maintenir un bon équilibre de performance.

- **Accuracy (Exactitude)**

  - L'**accuracy** globale est de **0.94** (94%), ce qui indique que 94% des prédictions du modèle sont correctes sur l'ensemble des **1999** échantillons.

- **Macro Average et Weighted Average**

  - **Macro avg** : La moyenne des métriques de précision, rappel et F1-Score entre les classes, sans pondération. Ici, elle est de **0.94** pour chaque métrique.

- **Weighted avg** : La moyenne pondérée des métriques, tenant compte de la taille de chaque classe. Les valeurs restent également à **0.94**.

22. Test du model

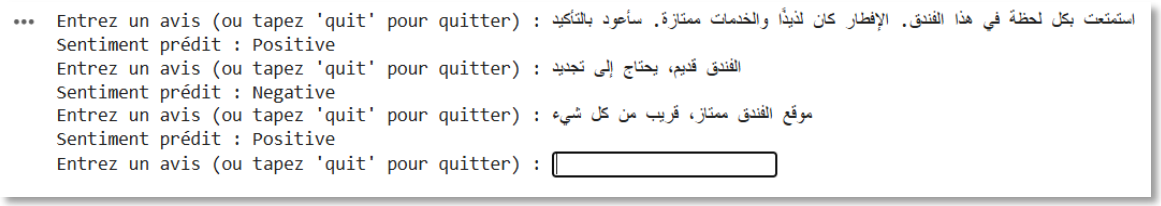
```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Chargement du tokenizer et le modèle depuis Google Drive
model = AutoModelForSequenceClassification.from_pretrained(drive_model_dir)
tokenizer = AutoTokenizer.from_pretrained(drive_model_dir)

print("Modèle et tokenizer chargés depuis Google Drive.")
```

```python
# Vérifier si un GPU est disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
# Fonction pour prédire le sentiment d'une critique
def predict_sentiment(review):
    # Tokeniser la revue
    inputs = tokenizer(review, return_tensors="pt", padding=True, truncation=True)
    inputs = {key: val.to(device) for key, val in inputs.items()}

    # Faire la prédiction
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_class = torch.argmax(logits, dim=-1).item()

    if predicted_class == 1:
        return "Positif"
    else:
        return "Négatif"

while True:
    review = input("Entrez une critique (ou 'quit' pour quitter) : ")

    if review.lower() == 'quit':
        print("Au revoir!")
        break

    sentiment = predict_sentiment(review)
    print(f"Sentiment prédit : {sentiment}")
```

Explication:

- Nous avons chargé le modèle et le tokenizer depuis Google Drive à l'aide des fonctions from_pretrained(), ce qui permet d'utiliser un modèle fine-tuné pour prédire le sentiment des critiques.
- Nous avons vérifié si un GPU est disponible et, si c'est le cas, déplacé le modèle vers le GPU pour accélérer les prédictions.
- Nous avons implémenté une fonction qui prend une revue en entrée, la tokenise, et prédit si le sentiment est **positif** ou **négatif** en utilisant le modèle. Une boucle permet à l'utilisateur de saisir des critiques et d'obtenir la prédiction en continu jusqu'à ce qu'il choisisse de quitter.

Résultat



Explication:

- Le modèle utilise le texte saisi par l'utilisateur pour prédire si l'avis est positif ou négatif.
- Les avis exprimant des sentiments favorables (comme "**فندق جيد**", "**موقع الفندق ممتاز**"، " **قريب من كل شيء**") sont classés comme "Positive", tandis que ceux exprimant des plaintes ou des aspects négatifs (comme "**الحمام ملوث، والخدمة سيئة**", " **سيء للغاية**", "**التكييف ضعيف للغاية**") sont classés comme "Negative".

Conclusion

Nous avons utilisé le modèle **CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment**, déjà fine-tuné pour la classification des sentiments en arabe, et l'avons adapté à notre tâche avec **QLoRA** pour un fine-tuning plus léger. Nous avons préparé les données et optimisé les paramètres d'entraînement pour obtenir un modèle performant. Les prochaines étapes incluent l'optimisation des hyperparamètres et l'intégration du modèle pour des prédictions en temps réel.

Référence

Lien vers le code complet “ FineTuning_CAMelBert” sur Google Colab:

https://colab.research.google.com/drive/1nMToYuCIB6o6CN2eQ6usil5RCgQ74FqH?usp=sharing

3. Analyse des Résultats des Modèles

Comparaison entre les résultats obtenus

1. Prompt Engineering

Méthode	Modèle	Précision (Negative)	Précision (Positive)	Rappel (Negative)	Rappel (Positive)	F1-Score (Negative)	F1-Score (Positive)	Accuracy	Macro Average	Weighted Average
Prompt Engineering	ChatGPT-4o mini	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

2. Fine-Tuning

Méthode	Modèle	Précision (Negative)	Précision (Positive)	Rappel (Negative)	Rappel (Positive)	F1-Score (Negative)	F1-Score (Positive)	Accuracy	Macro Average	Weighted Average
Fine-Tuning	AraGPT2	0.95	0.91	0.94	0.93	0.94	0.92	0.93	0.93	0.93
	CAMelBert	0.95	0.93	0.95	0.93	0.95	0.93	0.94	0.94	0.94

3. Embedding

Méthode	Modèle	Classificateur	Précision (Negative)	Précision (Positive)	Rappel (Negative)	Rappel (Positive)	F1-Score (Negative)	F1-Score (Positive)	Accuracy	Macro Average	Weighted Average
Embedding	AraBERT	RNN	0.92	0.86	0.89	0.89	0.90	0.87	0.89	0.89	0.89
		CNN	0.96	0.82	0.84	0.95	0.90	0.88	0.89	0.89	0.89
		LSTM	0.89	0.90	0.93	0.85	0.91	0.88	0.90	0.90	0.90
	Word2Vec	RNN	0.81	0.80	0.87	0.73	0.84	0.76	0.81	0.80	0.81
		CNN	0.81	0.82	0.89	0.72	0.85	0.77	0.81	0.81	0.81
		LSTM	0.78	0.85	0.91	0.66	0.84	0.74	0.80	0.79	0.80
	FastText	RNN	0.85	0.91	0.94	0.78	0.89	0.84	0.87	0.87	0.87
		CNN	0.86	0.90	0.94	0.79	0.90	0.84	0.88	0.87	0.87
		LSTM	0.89	0.86	0.90	0.85	0.90	0.86	0.88	0.88	0.88

Conclusion

- **Prompt Engineering** : Ce modèle a atteint des résultats parfaits pour toutes les métriques, avec des scores de **1.00** pour la précision, le rappel, le F1-score et l'accuracy. Bien que ces résultats soient impressionnants, ils peuvent être le reflet de la configuration du modèle et de la taille des données. Néanmoins, cette approche semble offrir des performances exceptionnelles dans ce contexte.

- **Fine-Tuning** : Nous avons d'abord effectué un fine-tuning avec **AraGPT2**, obtenant une **accuracy** de 93%, ce qui a montré une amélioration significative par rapport aux modèles de base. Ensuite, nous avons appliqué le fine-tuning sur **CAMeL-BERT**, déjà fine-tuné pour la classification des sentiments en arabe. En utilisant **QLoRA** pour une adaptation plus légère, les performances ont encore été renforcées, avec une précision, un rappel et une **accuracy** très élevés. Le fine-tuning a permis d'équilibrer les résultats des deux classes, montrant ainsi son efficacité pour cette tâche de classification des sentiments.
- **Embedding**:

    - **AraBERT** : Les résultats avec AraBERT varient selon les classificateurs. Le modèle avec LSTM atteint les meilleures performances globales avec une *accuracy* de **0.90**, un F1-score équilibré entre les classes, et une meilleure gestion des déséquilibres entre les classes. Les classificateurs RNN et CNN offrent des performances légèrement inférieures, avec des *accuracies* respectives de **0.88** et **0.89**.

    - **Word2Vec** : Cette approche est légèrement moins performante, avec des *accuracies* autour de **0.80** à **0.81**. Les modèles RNN et CNN montrent des performances similaires, tandis que le modèle LSTM gère moins bien les déséquilibres entre les classes. Cela reflète les limites de Word2Vec pour capturer des représentations complexes comparé à AraBERT ou FastText.

    - **FastText** : FastText démontre de meilleures performances que Word2Vec, avec une *accuracy* allant jusqu'à **0.88**. Les modèles CNN et RNN obtiennent des résultats proches et équilibrés, tandis que LSTM se démarque avec un bon compromis entre précision, rappel et F1-score.

En résumé le fine-tuning avec **AraGPT2** a permis d'atteindre une **accuracy** de 93%, mais l'application du fine-tuning avec **CAMeL-BERT** se distingue comme l'approche la plus efficace, offrant un bon compromis entre performance et généralisation tout en assurant un équilibre entre les classes. Cependant, les modèles basés sur des embeddings spécifiques comme **AraBERT** et **FastText**, combinés avec LSTM ou CNN, montrent également des performances prometteuses, notamment pour des tâches exigeant des représentations riches. Enfin, l'approche de prompt engineering, bien qu'idéale dans des conditions spécifiques, nécessite des ajustements pour une généralisation plus large.

4. Conclusion

Dans ce travail pratique, nous avons exploré différentes techniques d'analyse des sentiments appliquées à des données de critiques d'hôtels en arabe, en utilisant trois approches distinctes : le **prompt engineering**, les **embeddings** associés à différents classificateurs, et le **fine-tuning**.

Les résultats montrent que chaque approche présente des avantages et des limitations :

- **Prompt Engineering** : Cette approche a donné des résultats parfaits avec une précision, un rappel, un **F1-score** et une **accuracy** de 1.00. Ces résultats exceptionnels peuvent être attribués à la spécificité des données ou à la configuration utilisée. Cependant, cette méthode pourrait offrir des résultats particulièrement pertinents dans des tâches similaires avec des données parfaitement adaptées.
- **Embeddings avec Classificateurs** : Nous avons testé plusieurs embeddings (AraBERT, Word2Vec, FastText) associés à des classificateurs comme RNN, CNN et LSTM. Les meilleurs résultats ont été obtenus avec **AraBERT** et le classificateur **LSTM**, atteignant une **accuracy** de 0.90. **FastText** a montré des performances solides avec une **accuracy** allant jusqu'à 0.88, tandis que **Word2Vec** a montré des résultats légèrement inférieurs, soulignant ses limites pour capturer des représentations complexes. Bien que les performances globales aient été bonnes, les différences de précision et de rappel entre les classes **Positive** et **Negative** montrent la nécessité d'améliorer l'équilibre des modèles.
- **Fine-Tuning** : Après un fine-tuning avec **AraGPT2**, nous avons obtenu une **accuracy** de 93%, ce qui montre l'efficacité du fine-tuning sur des modèles préexistants. Puis, nous avons fine-tuné **CAMeL-BERT** pour une meilleure précision sur les données arabes. Cette approche a permis d'équilibrer les performances entre les deux classes avec des **précision**, **rappel** et **F1-score** élevés et cohérents, offrant ainsi une meilleure généralisation et un compromis idéal entre performance et robustesse.

En somme, ces différentes techniques d'analyse des sentiments nous ont permis de comprendre l'impact du choix de la méthode et des paramètres sur les performances des modèles. Les modèles basés sur des embeddings riches comme **AraBERT** et **FastText**, combinés à des classificateurs avancés comme **LSTM** ou **CNN**, offrent des alternatives prometteuses. Le **fine-tuning** s'est révélé être l'approche la plus robuste pour des applications réelles, tandis que le **prompt engineering** pourrait donner des résultats exceptionnels dans des contextes très spécifiques.

Ce travail pratique nous a permis de mieux comprendre les différentes stratégies d'analyse des sentiments et leur impact sur les performances des modèles, tout en mettant en évidence l'importance de l'ajustement des paramètres et du choix de la méthode pour obtenir des résultats optimaux.

Références des outils et modèles utilisés

- Google Colab : https://colab.google/
- Le model bert-base-arabertv2 : https://huggingface.co/aubmindlab/bert-base-arabertv2
- Le model aragpt2-base : https://huggingface.co/aubmindlab/aragpt2-base
- Le model CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment : https://huggingface.co/CAMeL-Lab/bert-base-arabic-camelbert-da-sentiment
- Wandb.ai : https://wandb.ai/home