TP TAL SMT

Youna Froger, Alice Gydé, Mathis Quais

16 Décembre 2024

1 Introduction

Voici le lien du dépôt GitHub où se trouve le code de ce TP :
https://github.com/younafroger/TAL$_H NM2$

L'objectif de ce TP était de terminé le code vu en cours, afin d'entraîner notre modèle de traduction automatique. Pour cela, nous l'avons adapté afin qu'il utilise 3 corpus de texte disctints :
- "fr-train.conll" qui est le corpus d'entraînement
- "fr-dev.conll" qui le corpus de validation, semblable au premier, mais qui n'est pas utilisé pour l'entraînement
- "fr_test_withtags.conll" qui est le corpus ettiqueté qui permet de tester notre entraînement en ayant des données validée et taguée.

Nous nous sommes appuyé sur l'aide de l'IA (ChatGPT) afin de remettre en forme le code correctement et afin d'optimiser la partie "entraînement" car nous avons rencontré des problèmes de memoire dû a notre materiel informatique qui n'était pas assez performant pour cette tâche, et de la documentation de SpaCy pour l'implémentation de l'entraînement.

2 Analyse du code

Le code commence avec l'importation des bibliothèques nécessaire.

Dans un premier temps, la fonction process prend en entrée le texte brut et extrait de celui-ci les entités présentents annotées (position et tags).
Puis, la fonction load_conll charge les fichiers .conll et limite la taille des échantillon (paramètre max_sample) afin d'optimiser le code pour qu'il fonctionne sur nos machines. En sortie, nous avons une liste de tuples (texte et annotation).

```
def process(txt, liste):
    tokens = txt.split("\n")
    start = 0
    entities = list()
    phrase = list()
    for t in tokens:
        if len(t) > 0 and t[0] != "#":
            word, _, _, tag = t.split(" ")
            end = len(word) + start
            entities.append((start, end, tag))
            phrase.append(word)
            start = end + 1
    liste.append((" ".join(phrase), {"entities": entities}))
    return liste

def load_conll(file_path, max_samples=None):
    data = []
    with open(file_path, 'r', encoding='utf-8') as file:
        txt = file.read()
        phrases = txt.split("\n\n")
        for i, phrase in enumerate(phrases):
            if max_samples and i >= max_samples:
                break
            data = process(phrase, data)
    return data
```

Ensuite, on initialise le modèle Spacy, en en créant un vide pour le texte en français. On ajoute un composant ner (Names ENtity Recognition) afin de débuter la reconnaissance d'entités automatique dans les textes. Enfin, on déclare des classes d'entités (ArtWork, Artist...).

```
nlp = spacy.blank("fr")
lookups = Lookups()
lookups.add_table("lemma_lookup", {})
lookups.add_table("lemma_rules", {})
lookups.add_table("lemma_exc", {})
```

```
lookups.add_table("lemma_index", {})
nlp.vocab.lookups = lookups

ner = nlp.add_pipe("ner", last=True)
new_labels = ["ArtWork", "Artist", "VisualWork", "WrittenWork"]
for label in new_labels:
    ner.add_label(label)
```

Puis, on prépare nos données d'entraînement. A l'aide d'une boucle, on parcourt nos données dans le corpus fr_train.

```
train_data = []
for text, annotations in fr_train:
    doc = nlp.make_doc(text)
    example = Example.from_dict(doc, annotations)
    train_data.append(example)
```

Pour la partie entraînement, nous avons rencontré quelques difficultés, notamment de lancement sur nos machines. Pour l'entraînement, on a testé plusieurs nombre d'époques (Epochs) et nous avons remarqué que le modèle stagne en performance à partir de 40 Epochs (nombre de fois où il s'entraîne). Ensuite, nous avons dû réduire nos données en plusieurs batchs afin d'optimiser notre consommation de mémoire. La fonction nlp.update() initialise un taux d'abandon à chaques tours d'entraînement afin d'éviter le sur-entraînement, qui peut avoir des effets négatifs sur le résultat. Pour chaque époques, on affiche le loss "la perte" et l'objectif et de se rapporcher d'une valeur basse. Ce modèle d'entraînement est finalement sauvegardé via la fonction nlp.to_disk.

```
optimizer = nlp.begin_training()
epochs = 40
batch_size = 32

for i in range(epochs):
    losses = {}
    batches = [train_data[i:i + batch_size] for i in range
    (0, len(train_data), batch_size)]
    for batch in batches:
        nlp.update(batch, drop=0.5, losses=losses)
    print(f"Epoch {i + 1}/{epochs}, Losses: {losses}")


nlp.to_disk("ner_model")
```

Après avoir entraîné notre modèle, on passe à l'évaluation de ce dernier. Tout d'abord, en parcours les données du corpus "fr_test", on compare les données prédites par le modèle précédement, afin celles qui sont annotées.

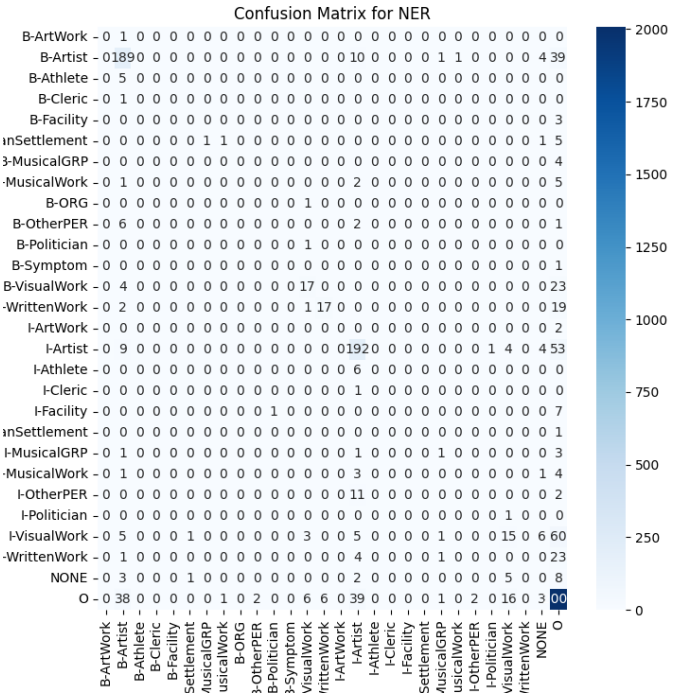
Afin d'évaluer les résutlats, on génère une matrice de confusion avec la fonction confusion_matrix. En plus, on réalise un rapport de classification, qui affiche des métriques utiles à l'évaluation d'un modèle (métriques déjà abordées précédement telles que la précisions, le rappel ou le F1-SCORE).

```
def evaluate_ner(model, test_data):
    true_labels = []
    predicted_labels = []
    all_labels = set()

    for text, annotations in test_data:
        doc = model(text)
        true_entities = [(start, end, label) for start, end,
        label in annotations["entities"]]
        predicted_entities = [(ent.start_char, ent.end_char, ent.label_)
        for ent in doc.ents]

        for true_entity in true_entities:
            start, end, label = true_entity
            true_labels.append(label)
            all_labels.add(label)

            matching_predicted = [ent[2] for ent in
            predicted_entities if ent[:2] == (start, end)]
            if matching_predicted:
                predicted_labels.append(matching_predicted[0])
            else:
                predicted_labels.append("NONE")

        for predicted_entity in predicted_entities:
            if predicted_entity[:2] not in [(ent[0],
            ent[1]) for ent in true_entities]:
                predicted_labels.append(predicted_entity[2])
                true_labels.append("NONE")
                all_labels.add(predicted_entity[2])

    all_labels = sorted(all_labels | {"NONE"})
    cm = confusion_matrix(true_labels, predicted_labels, labels=all_labels)

    plt.figure(figsize=(8, 8))
    sns.heatmap(cm, annot=True, cmap="Blues",
    xticklabels=all_labels, yticklabels=all_labels, fmt="d")
    plt.title("Confusion Matrix for NER")
    plt.xlabel("Predicted Labels")
    plt.ylabel("True Labels")
```

```
plt.savefig("confusion_matrix.png")


print("\nClassification Report:")
print(classification_report(true_labels,
predicted_labels, labels=all_labels))
```

3 Conclusion et analyse des résultats

Voici le résultat de notre matrice de confusion.



Notre modèle de traduction automatique présente beaucoup de difficulté. On s'apperçoit que nos données sont souvent qualifié dans la classe 0, ce qui signifie qu'il ne reconnait pas les entités qui ont étiquettes des classes plus spécifiques telles qu'Artiste, VisualWork etc. En augmentant le nombre d'époques (initialement à 5), les prédictions ont été améliorées (à la base, quasiment toutes les prédictions étaient sur 0).

Pour améliorer nos résultats, il serait intéressant de pouvoir lancer notre script sur un plus grand échantillon de nos jeux de données (ici limité à 500 données pour l'entrainement, et 200 pour les deux autres).