Étiqueteur *super-senses* avec *transformers* pré-entraînés

PSTAL - TP 4 - Carlos Ramisch

L'objectif de ce TP est de *développer* et *évaluer* un système capable de prédire des étiquettes sémantiques appelées ***super-senses*** (colonne `frsemcor:noun`). Ces étiquettes indiquent, pour chaque nom, sa catégorie sémantique gros-grain, par exemple : *surprise* est un sentiment (étiquette `Feeling`), *jambe* est une partie du corps (`Body`), *avocat* est une personne (`Person`) ou un aliment (`Food`) selon le contexte, etc. Nous utiliserons des représentations vectorielles appelées **embeddings contextuels**, issues de modèles *transformers* pré-entraînés.

1 Étiquetage en *super-senses*

L'étiquetage du sens des mots est une tâche historiquement difficile en TAL. [1] Les premiers modèles, peu performants, essayaient de prédire, parmi tous les sens listés dans un dictionnaire, le plus approprié dans le contexte. Ciaramita & Johnson (2003) ont été les premiers à proposer de simplifier cette tâche à l'aide de catégories plus génériques, car la granularité trop fine des dictionnaires a longtemps été une difficulté en WSD.

Les données que nous utiliserons sont issues du projet FR-SemCor, qui a annoté le corpus Sequoia en *super-senses* uniquement pour les noms (Barque et al. 2020). [2] Donc, les mots qui nous intéressent sont les noms communs, les noms propres et les numéros ; ceux dont la `upos` $\in$ {`NOUN`, `PROPN`, `NUM`}. Le système doit prédire une des 24 étiquettes *super-sense* nominales possibles, ou alors prédire `*` pour les mots n'ayant pas de *super-sense* annoté. [3] Il s'agit, comme les TP précédents, d'une tâche d'étiquetage de séquences, par exemple :

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}
$x =$	*Quelle*	*surprise*	*!*	*Arturo*	*arrive*	*en*	*Chine*	*au*	*moment*	*de*	*la*	*fête*
	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓
$y =$	`*`	`Feeling`	`*`	`Person`	`*`	`*`	`Institution`	`*`	`Time`	`*`	`*`	`Act`
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}

2 Bibliothèque *transformers*

Le système repose sur des embeddings extraits d'un modèle de langage *transformer* bidirectionnel du type "BERT". Ces embeddings sont optimisés pour prédire un mot en fonction de son contexte (modèle de langage masqué, ou MLM). Comme le contexte et le sens sont corrélés selon l'hypothèse distributionnelle, nous supposons qu'il est pertinent de prédire les *super-senses* à partir de ces embeddings contextuels.

Généralement, on ignore la couche MLM, et on garde le "mille-feuilles" en dessous. Une nouvelle phrase à encoder passe dans les N couches d'auto-attention et, à la dernière (`last_hidden_state`), on obtient un embedding d-dimensionnel pour chaque token de l'entrée. Sur ces embeddings, on ajoute une couche de décision (p.ex. linéaire) spécifique à notre tâche, qui sera apprise. On parle de **fine-tuning** quand l'apprentissage de la couche de décision retro-propage les gradients vers l'intérieur du *transformer* et modifie les embeddings. Nous utiliserons les embeddings du *transformer* de manière statique : on parle alors d'**embeddings contextuels**.

La bibliothèque *transformers* de HuggingFace contient des modules pratiques pour ce TP. De plus, des centaines de modèles pré-entraînés sont disponibles sur `https://huggingface.co/models`. Ces modèles contiennent l'architecture du réseau et les valeurs de tous les paramètres : embeddings, matrices d'auto-attention, couches intermédiaires, etc. De plus, les modèles se téléchargent automatiquement dès la première utilisation !

Les modèles ont été pré-entraînés sur des corpus dans une ou plusieurs langues. Certains sont uniquement francophones, comme `almanach/camembert-*` et `flaubert/flaubert_*` Alors que d'autres modèles tels que `google-bert/bert-base-multilingual-*` et `distilbert/distilbert-base-multilingual-cased` incluent le français parmi une centaine de langues vues à l'entraînement. Pour le développement, `distilbert` est preferable par sa relative légèreté. Utilisez `camembert` et `google-bert` quand le développement sera fini.

1. Cette tâche est aussi appelée *désambiguisation lexicale* ou WSD, de l'anglais *word sense disambiguation*
2. Nous utilisons une version simplifiée, voir `sequoia/README.md` pour les détails.
3. L'étiquette `*` est prédite dans 2 situations : soit le mot n'est pas un nom, soit il l'est, mais son annotation est manquante.

3 Préparation des données

Chaque modèle de langage a son propre tokéniseur, p.ex. `CamembertTokenizer`. Pour instancier le tokéniseur à partir du nom du modèle, on utilise `AutoTokenizer.from_pretrained`. La phrase à tokéniser est une liste de mots avec `is_split_into_words=True`. Pour obtenir un tenseur en sortie, `return_tensors='pt'` :[4]

```
>>> from transformers import AutoTokenizer, AutoModel
>>> tokenize = AutoTokenizer.from_pretrained("almanach/camembert-base")
>>> tok_sent = tokenize(["Arturo","devra","retenter","demain"], is_split_into_words=True, return_tensors='pt')
>>> print(tok_sent['input_ids'])
tensor([[    5,  2082, 10394,  2613,   343,  3889,   108,  2385,     6]])
```

La sortie du tokéniseur peut être donnée au modèle, qui génèrera un embedding ($d = 768$) par token :

```
>>> model=AutoModel.from_pretrained("almanach/camembert-base")
>>> emb_sent = model(**tok_sent)['last_hidden_state'][0]
>>> print(emb_sent.shape)
torch.Size([9, 768])
```

Alignement (sub-)tokens↔mots Notez que, pour 4 mots, nous obtenons 9 embeddings, correspondant aux 9 sub-tokens générés par le tokéniseur. Cependant, les annotations de *super-sense* sont alignées aux mots, et non pas aux sub-tokens. P.ex. nous ne pourrions pas créer un `DataLoader` avec 9 entrées correspondant à 4 sorties, il faut les aligner ! La fonction `tok_sent.word_ids()` permet d'obtenir l'alignement entre les mots et les sub-tokens :[5]

```
>>> tok_sent.word_ids()
[None, 0, 0, 1, 2, 2, 2, 3, None]
```

Par exemple, le mot d'indice 2 contient 3 sub-tokens, dans les positions 4, 5 et 6 de `tok_sent['input_ids']`. Votre fonction d'alignement doit donc parcourir les *mots* de la phrase et, pour chaque mot ayant le `upos` $\in$ {`NOUN`, `PROPN`, `NUM`}, trouver le(s) embedding(s) correspondants à l'aide de `word_ids`. Créez ensuite un seul embedding pour ce mot en moyennant les embeddings contextuels de ses sub-tokens. Cet embedding sera associé à l'étiquette super-sense du mot. Figez les embeddings avec `torch.no_grad()` lors de l'appel du modèle. Pensez à transformer en indices les étiquettes de *super-sense*, car `transformers` ne le fait pas pour nous.

4 Classifieur d'embeddings contextuels

Si la préparation des données est complexe et laborieuse, le classifieur, lui, est assez facile à programmer. Il s'agir d'un MLP composé de une ou plusieurs couches linéaires (`nn.Linear`) suivies de fonctions d'activation non-linéaires (p.ex. `nn.ReLu`). La première couche dense prend en entrée l'embedding contextuel du mot à classifier. Sa dimension dépend du modèle pré-entraîné utilisé (`camembert`, `distilbert`, ...) et peut être obtenue avec `config.hidden_size`. La dernière couche aura comme fonction d'activation le softmax (implicite, comme d'habitude). Sa dimension est égale au nombre d'étiquettes de *super-sense* à prédire, plus 1 position correspondant à la prédiction de `*`. Il n'est pas nécessaire de faire du padding comme on ferait dans RNN.

5 Entraînement, prédiction et évaluation

L'entraînement requiert un `DataLoader` associant embeddings contextuels et étiquettes *super-sense*. À chaque *epoch*, calculez la loss et l'*accuracy* sur le `.dev`. Comme nous n'avons pas de padding, il n'y a pas besoin de masquage. Sauvegardez le modèle, le vocabulaire des étiquettes et les hyper-paramètres avec `torch.save`. Le nom du modèle pré-entraîné est un hyper-paramètre important à sauvegarder. Pour la prédiction, mettre `*` dans la colonne `frsemcor:noun` de tous les mots n'ayant pas une des 3 POS cible. Pour les autres, obtenir les embeddings contextuels alignés, les passer dans le classifieur et choisir l'étiquette de score maximal.

Pour évaluer les prédictions, utilisez le script `accuracy`, déjà utilisé lors des TP précédents, avec les options :

- `--tagcolumn frsemcor:noun` Indique la colonne prédite (12ème), qui s'appelle `frsemcor:noun`.
- `--upos-filter NOUN PROPN NUM` Nous ne voulons pas considérer les * prédites sur les mots n'ayant pas les POS cible (déterminants, verbes, etc.) Cela gonflerait artificiellement le score d'évaluation. Dette option permet de calculer la *accuracy* uniquement pour les mots dont la POS nous intéresse.
- `--train` permet de calculer la *accuracy* sur les OOV, ce qui peut être utile pour comparer des modèles.

option pour évaluer uniquement sur les noms, numéros et noms propres.

4. Avec `convert_ids_to_tokens` on lit `['<s>', '_Art', 'uro', '_devra', '_re', 'tent', 'er', '_demain', '</s>']`.
5. Cette fonction n'est pas disponible pour `flaubert`, raison pour laquelle ce modèle est déconseillé.

6 Travail à effectuer

Vous devez écrire deux scripts : `train_postag.py` pour l'entraînement du modèle, et `predict_ssense.py` pour la prédiction des *super-senses*. Le code du classifieur (Section 4) doit être partagé par les deux scripts. L'évaluation des prédictions sera effectuée par le script fourni `lib/accuracy.py` en utilisant les options `-upos-filter NOUN PROPN NUM` et `-tagcolumn frsemcor:noun`, comme expliqué en Section 5.

7 Extensions

Plus proche voisin Une méthode alternative pour effectuer la classification consiste à créer des embeddings "prototypes" pour chaque *super-sense*. Ces prototypes sont la moyenne de tous les embeddings ayant ce *super-sense* dans le corpus d'entraînement. En inférence, on obtient l'embedding contextuel du mot à classifier, puis on lui affecte le *super-sense* du prototype le plus proche (plus proche voisin, ou 1-NN). Cette méthode est expliquée en page 253 (Section 11.3.1) de Jurafsky & Martin (2024) et dans les 2 articles y cités. Implémentez et comparez cette méthode à celle fondée sur le MLP décrite ci-dessus.

Fine-tuning Au lieu de figer les représentations, on peut *fine-tuner* les embeddings à la tâche de prédiction de *super-senses*. Cela donne souvent de meilleures résultats. Pour cela, il faudra déplacer l'obtention des embeddings à l'intérieur de la fonction `forward` du modèle. Il est aussi important d'avoir un taux d'apprentissage adapté, essayez `lr=2e-5`, ou testez le *linear warmup*, souvent utilisé dans cette situation. Le fine-tuning étant très lourd, vous aurez probablement besoin de GPU. Alternativement, vous pouvez figer toutes les couches du modèle (`requires_grad=False`) sauf les k les plus hautes, avec $k = 1$ ou 2, par exemple. Comparez les résultats et le temps d'entraînement avec/sans *fine-tuning* des modèles `camembert`, `distilbert`, et `bert-multilingual`.