Basic Tagger Parser

Hermes Martinez
Timothée Mickus

Présentation générale

Le projet consiste en l'implémentation de deux outils : un POS-tagger et un analyseur statistique en dépendances. Les deux outils devaient être conçus de manière à ce qu'ils puissent être utilisés ensemble. Nous avons décidé de baser les systèmes de pondération sur des réseaux neuronaux afin qu'il puissent avoir en commun des couches de neurones, ce qui nous garantissait que l'output du tagger pouvait être utilisé par l'analyseur.

Le tagger utilise comme système de pondération un réseau neuronal LSTM qui prend en entrée une séquence de mots et prédit en sortie une séquence de POS tags. Nous avons inclus une couche de word embeddings déjà générée par l'algorithme word2vec appris du français et une option du réseau pour l'utiliser et ré-entraîner les vecteurs de mots à chaque époque ou bien de générer sa propre couche d'embeddings depuis zéro. L'utilisation d'une matrice de words embeddings pré-entraînée nous permet aussi d'avoir un tag plus précis pour les mots inconnus. L'architecture LSTM permet en outre de se baser sur les mots précédents pour la prédiction de la catégorie du discours. Ce qui fait que les mots qui ne sont ni dans le corpus d'entraînement ni pré-entraînés dans le modèle de word embeddings, bien que mappés sur le même vecteur de mots, sont assignés à une catégorie selon leur contexte.

Nous avons étudié, pour le tagger, différentes architectures et différents algorithmes de descente de gradient. Nous avons aussi joué sur la taille de la mémoire des transitions LSTM, le nombre d'époques, et le nombre de neurones dans les couches. Le paramètre qui a eu le plus d'impact a été l'algorithme de descente de gradient, RMSProp s'étant révélé le plus à même de donner des résultats satisfaisants (95 % d'accuracy). Le nombre d'époques, au delà d'un certain seuil, ne joue en revanche qu'un rôle faible.

Nous avons décidé d'implémenter un analyseur en arc standard, et avons choisi comme algorithme d'exploration des solutions une recherche en faisceau. Cet analyseur peut se noter <S, B, A>. Il est initialisé avec un mot qui symbolise la racine dans la pile, l'ensemble des tokens de la phrase dans le buffer, et un ensemble vide pour les arcs.

Le système de transition est formellement décrit par :

Shift : $\langle S, w|B, A\rangle \rightarrow \langle S|w, B, A\rangle$

Right-arc : $\langle S|x|y, B, A\rangle \rightarrow \langle S, B, A \cup \{x \rightarrow y\}\rangle$

Left-arc : $\langle S|x|y, B, A\rangle \rightarrow \langle S, B, A \cup \{y \rightarrow x\}\rangle$

On ajoute aussi terminate pour symboliser un arbre entièrement analysé : <w,ø, A>

Et on dénote l'initialisation comme : <ø, a … n, ø>

La méthode d'exploration des solutions est une recherche par faisceau, c'est-à-dire que nous n'étudions que les transitions pour lequel le préfixe de dérivation a été parmi les meilleurs à l'étape précédente. Ces meilleurs sont sélectionnés comme ceux ayant un meilleur score selon le modèle de pondération. Le système de pondération est implémenté en réseau neuronal.

Architecture du projet

Les directives qu'il fallait suivre pour ce projet indiquaient qu'il devait être séparé en trois fichiers .py, ce que nous avons fait.

Le fichier **corpus.py** présente un utilitaire pour lire et manipuler des corpus au format CONLL. Il possède plusieurs fonctions :

- load(filename), qui permet de lire un fichier .conll et d'en construire une représentation sous forme de listes : le corpus est une liste de phrases, chaque phrase est une liste de mots, et chaque mot correspond à une liste de colonnes selon le format CONLL. Le keyword randomize permet de mélanger aléatoirement les phrases du corpus.
- split(filename), permet de séparer et de sauvegarder en sous-fichier un fichier .conll dont le nom est passé en paramètre. Le keyword proportions permet de paramétriser la séparation ainsi que les extensions qui permettent de caractériser les osu fichier. Le keyword randomize permet de mélanger aléatoirement les phrases du corpus.
- extract(corpus) permet de manipuler une représentation python du corpus pour en extraire des informations. Le keyword columns permet de préciser quelles colonnes doivent être extraites. Cette fonction renvoie une liste pour chaque colonne requise, contenant la liste des valeurs correspondantes pour chaque séquence dans le corpus.
- extract_features_for_depency(filename) permet de renvoyer la chaîne de caractère attendue par le parser (cf. infra) à partir d'un nom de fichier .conll.
- read_embeddings(filename) permet de renvoyer un dictionnaire de mot : représentation vectorielle à partir d'un fichier de word embeddings déjà pré-entrainé

Il contient en outre un dictionnaire _COL, qui permet de relier les index aux colonnes selon le format CONLL. Il est préférable de ne pas le modifier puisque les fonctions d'extraction d'information en dépendent.

Le fichier **tagger.py** contient l'implémentation d'un tagger neuronal. Il possède une classe NNTagger, laquelle contient les méthodes suivantes :

- train(filename) permet d'initialiser et d'entraîner le modèle vectoriel. En tant qu'effet de bord, il construit aussi les mappings pour les embeddings de mots et les vecteurs one-hot des POSTags.
- test(filename), permet d'évaluer un modèle déjà entraîné sur un corpus de test.
- predict(sentences) permet de catégoriser en partie du discours une liste de phrases. Elle renvoie les phrases taguées au même format que la méthode extract de corpus.py
- save() , permet de sauvegarder les informations essentielles du tagger (son réseau neuronal, ainsi que le vocabulaire qui permet de convertir le langage naturel en entier tel qu'attendu par le réseau). Les keywords permettent de spécifier avec quel nom enregistrer ces informations
- load(), qui est une méthode de classe, permet de renvoyer un tagger contenant uniquement les informations minimales nécessaires à son emploi à partir d'une sauvegarde effectuée à l'aide de la méthode save()

Le fichier **parser.py** contient l'implémentation d'un analyseur syntaxique en dépendance à l'aide de l'algorithme arc_standard, utilisant une recherche par faisceau, et dont le système de pondération est un réseau neuronal. Il contient les deux classes DepencyTree et DependencyParser. La classe DependencyTree permet d'obtenir une représentation syntaxique pour une phrase donnée ; elle contient les méthodes suivantes :

- read_tree(istr), méthode de classe qui pour un flux de caractères contenant les informations nécessaires renvoie la représentation en tant que DependencyTree associée.
- accurracy(other), qui permet de calculer à quel point deux représentations syntaxiques divergent l'une de l'autre.
- N(), qui donne le nombre de tokens couvert par l'arbre.

La classe DependencyParser permet d'instancier un analyseur syntaxique par arc standard, dont la méthode d'exploration des solutions est par faisceaux, et le système de pondération par réseau neuronal. Elle contient les méthodes suivantes :

- oracle_derivation(dependency_tree) permet de renvoyer pour un arbre déjà annoté la séquence d'actions que l'analyseur devrait effectuer pour arriver au même résultat
- les fonctions shift(config, tokens), leftarc(config, tokens), rightarc(config, tokens) et terminate(config, tokens) permettent de traiter l'évolution de la pile, du buffer, et du score associé pour les actions de l'analyseur S, L, R et T respectivement.
- parse_one(sentence) donne l'analyse en dépendance de la phrase passée en paramètre selon l'analyseur. Le keyword beam_size permet de manipuler la taille du faisceau pour l'exploration de solution. Le keyword get_beam permet d'obtenir le faisceau de solutions explorées au lieu de renvoyer la seule analyse.
- encode(sequence, tokens) permet de transformer une séquence d'indices au format nécessaire pour le système de pondération
- score(config, action, tokens) permet de scorer de l'action selon la configuration donnée.
- test(dataset) permet de tester l'analyseur sur le jeu d'exemples passés en paramètres. Le keyword beam_size permet de paramétrer la taille du faisceau d'exploration de solutions
- train(dataset, tagger) permet d'initialiser et d'entraîner le système de pondération. Le tagger passé en paramètre correspond à un tagger neuronal (cf. supra) ; le modèle neuronal de pondération de l'analyseur possède en effet plusieures couches en commun avec le réseau neuronal du tagger. Le keyword epochs permet de paramétrer le nombre d'époques pour l'entraînement du réseau neuronal.

L'ensemble de ce projet et l'historique de son évolution depuis le début peut être trouvé sur l'adresse git: https://github.com/Sachka/BasicTaggerParser . Ce répertoire .git contient en outre les différentes ébauches que nous avons menées, y compris une tentative d'intégration des annotations en partie du discours dans l'input du système de pondération du parser (l'architecture est cohérente et le script correct, mais les résultats ne suivent pas).

Dans l'ensemble nous avons obtenu une très bonne précision avec le réseau LSTM de notre tagger (entre 95% et 98% de précision), mais le parsing s'est révélé une tâche plus difficile à satisfaire. En particulier, si nous avons pu mettre au point un système de pondération qui semble efficace, et qui indique correctement la prochaine action selon la configuration actuelle de l'analyseur, nous n'avons cependant pas su l'adapter à la méthode d'exploration de solutions que nous avons choisie: les arcs formés ne correspondent au mieux qu'à 32% aux arcs de références. Ceci est probablement dû au fait que notre algorithme d'entraînement prenait en compte les configurations une à une au lieu d'étudier l'entière séquence d'actions à mener.