Appariement de questions/réponses

Félix Jamet

April 6, 2018

Contents

1 Structures de données et algorithmes (`semeval_struct.py`)

1.1 Imports

```
import re
import operator
```

1.2 Arbres

1.2.1 Transformation en profondeur

Il s'agit ici d'appliquer récursivement une fonction à chacune des feuilles d'un arbre.

```
def transformtree_deep(func, tree):
    """Transform a tree by applying a function to its leaves.

    Parameters
    ----------
    func : function
        The function that will transform each leaf.

    tree : recursive dict
        The tree to transform.

    Returns
    -------
    out : recursive dict
        The transformed tree.
    """
    return {
        key: transformtree_deep(func, value)
        if isinstance(value, dict)
        else func(value)
        for key, value in tree.items()
    }
```

1.2.2 Transformation sur N niveaux

L'inconvénient de `transformtree_deep` est qu'elle parcourt l'arbre entièrement, tandis qu'il serait parfois utile de ne parcourir que les N premiers niveaux.

```python
def transformtree_n(func, tree, n):
    """Transform a tree up to a maximum depth by applying a function to its leaves.

    Once the maximum depth is reached, the function is applied, even if it is not a leaf.

    Parameters
    ----------
    func : function
        The function that will transform each leaf, as well as the final depth.

    tree : recursive dict
        The tree to transform.

    n : int
        The maximum depth

    Returns
    -------
    out : recursive dict
        The transformed tree.
    """
    if n > 0:
        return {
            key: transformtree_n(func, value, n - 1)
            if isinstance(value, dict)
            else func(value)
            for key, value in tree.items()
        }
    else:
        return { key: func(value) for key, value in tree.items() }
```

1.2.3 Transformation générique

Une sémantique intéressante pour la transformation d'arbre est de considérer que transformer un arbre jusqu'au niveau -1 revient à le transformer sans limite de profondeur.

```python
def transformtree(func, tree, n=-1):
    if n < 0:
        return transformtree_deep(func, tree)
    else:
        return transformtree_n(func, tree, n)
```

1.3 Tris

1.3.1 Dictionnaire

Par défaut, le tri sur les entrées d'un dictionnaire se fait sur les clés. On propose la fonction `sorted_items` pour faire un tri sur les valeurs.

```
def sorted_items(dictionary, key=operator.itemgetter(1), reverse=False):
    return sorted(dictionary.items(), key=key, reverse=reverse)
```

1.3.2 Tri naturel

Il n'y a pas de support natif dans python pour trier des éléments dans l'ordre naturel. La fonction `natural_sort_key` permet d'atteindre ce résultat, lorsque utilisée comme le paramètre `key` d'une fonction de tri.

```
def natural_sort_key(key):
    """Transform a key in order to achieve natural sort.
    from https://blog.codinghorror.com/sorting-for-humans-natural-sort-order/
    """
    def convert(text):
        return int(text) if text.isdigit() else text
    return [convert(c) for c in re.split('([0-9]+)', key)]
```

2 Fonctionnalités de support (`semeval_util.py`)

2.1 Imports

```
1 import pickle
```

2.2 Sauvegarde et chargement d'objets sur le disque

On utilise pickle pour sauvegarder et charger des objets qu'il serait long de reconstruire à chaque execution d'un script.

```
1 def save_object(obj, filename):
2     pickle.dump(obj, open(filename, 'wb'))
3
4 def load_object(filename):
5     return pickle.load(open(filename, 'rb'))
```

3 Traitement du langage naturel (`semeval_taln.py`)

3.1 Imports

```
import os.path
import math
from itertools import chain
from collections import Counter
from semeval_xml import get_semeval_id, get_related_threads, xmlextract
from semeval_util import save_object, load_object
```

La classe `Counter` est une sous-classe de `dict`, permettant de compter les occurences d'une clé. Elle est ici utilisée pour représenter des sacs de mots.

3.2 Analyse des questions par un modèle

Étant donné que les questions sont organisées de manière hiérarchiques, à savoir N questions originales, chacune accompagnée de 10 questions reliées, elles vont être représentées par un arbre.

Chaque question est passée dans un modèle de language, produisant ainsi un document.

La fonction `make_document_tree` permet de construire l'arbre des documents, à partir :

— des questions originales (`original_questions`),
— d'un modèle de language (`model`),
— d'une fonction d'extraction de contenu (`content_extractor`)

```
def make_document_tree(original_questions, model, content_extractor):
    result = {}
    for org in original_questions:
        orgid = get_semeval_id(org)
        result[orgid] = {
            get_semeval_id(rel): model(content_extractor(rel))
            for rel in get_related_threads(org)
        }
        result[orgid]['org'] = model(content_extractor(org))
    return result
```

L'analyse d'une phrase par un modèle étant une opération possiblement coûteuse, les documents liés aux questions vont être sauvegardées sur le disque afin de ne pas avoir à refaire tous les calculs à chaque fois.

```
def make_or_load_document_tree(xml_source, saved_path, model, content_extractor,
 ↪  verbose=False):
    if os.path.isfile(saved_path):
        if verbose:
            print('Loading document tree from', saved_path)
```

```
        result = load_object(saved_path)
        return result
    else:
        if verbose:
            print('Creating document tree. This might take a while...')

        extractor = xmlextract(xml_source)
        result = make_document_tree(
            extractor.get_org_elements(), model, content_extractor)

        if verbose:
            print('Saving document tree to', saved_path)
        save_object(result, saved_path)

        return result
```

3.3 Pondération de termes

TF (*Term Frequency*) et IDF (*Inverse Document Frequency*) sont des mesures permettant de pondérer des termes selon leur importance dans un corpus.

Les documents sont ici manipulés comme des sacs de mots, implémentés ici sous forme de compteurs.

3.3.1 *Term Frequency*

La TF d'un terme correspond à sa fréquence d'apparition dans l'ensemble des documents.

$$TF(terme, document) = \frac{occurences(terme, document)}{taille(document)}$$

où la taille d'un document correspond au nombre de termes qu'il contient.

Plutôt que de calculer la TF d'un terme dans un document à chaque fois que nécessaire, la TF de tous les termes d'un document est stockée dans un dictionnaire.

```
def term_frequencies(bag):
    documentlen = sum(bag.values())
    return {
        term: occurrences / documentlen
        for term, occurrences in bag.items()
    }
```

3.3.2 *Inverse Document Frequency*

L'IDF d'un terme est proportionnelle à l'inverse du nombre de documents dans lesquels il apparaît. Elle se base sur la DF (*Document Frequency*), correspondant au nombre de document dans lesquels un terme apparaît.

$$DF(terme, corpus) = \|\{doc / doc \in corpus \wedge terme \in doc\}\|$$

$$IDF(terme, corpus) = log\left(\frac{taille(corpus)}{DF(terme, corpus)}\right)$$

De la même manière que pour la TF, l'IDF de tous les termes du corpus est stockée dans un dictionnaire.

```python
def document_frequencies(corpus):
    result = Counter()
    for document in corpus:
        result.update({term for term in document})
    return result


def inverse_document_frequencies(corpus, DF=None):
    if DF == None:
        DF = document_frequencies(corpus)
    return {term: math.log2(len(corpus)/docfreq)
            for term, docfreq in DF.items()}
```

3.3.3 *Term Frequency - Inverse Document Frequency*

La TF-IDF d'un terme correspond à une combinaison de sa TF et de son IDF :

$$TF\text{-}IDF(terme, document, corpus) = TF(terme, document) * IDF(terme, corpus)$$

La TF-IDF d'un terme est implémentée comme une fonction utilisant des dictionnaires TF et IDF passés en paramètres.

```python
def tf_idf(term, termfreq, inversedocfreq, out_of_corpus_value):
    """Term Frequency - Inverse Document Frequency of a term using dictionaries.

    If the term is not in the inverse document frequency dictionary, this function will
    use the argument out_of_corpus_value.

    Parameters
    ----------
    term : str
        The term.

    termfreq : dict
        The term frequencies of the document.

    inversedocfreq : dict
        The inverse document frequencies of the corpus.

    Returns
    -------
    out : float
        The TF-IDF value of the term.
    """
    if term not in termfreq:
        return 0
    if term not in inversedocfreq:
        return out_of_corpus_value

    return termfreq[term] * inversedocfreq[term]
```

1. Sacs de mots Le score TF-IDF d'un sac de mots correspond à la somme des valeurs TF-IDF de ses éléments :

$$score_{TF\text{-}IDF}(sac, document, corpus) = \sum_{terme \in sac} TF\text{-}IDF(terme, document, corpus)$$

```
def tf_idf_bow(bag, termfreq, inversedocfreq, out_of_corpus_value):
    return sum(tf_idf(term, termfreq, inversedocfreq, out_of_corpus_value) *
     ↪  occurences
                for term, occurences in bag.items())
```

2. Similarité de documents Le score de similarité entre deux documents correspond au score TF-IDF du sac de mots qu'ils forment.

$$similarité(doc_a, doc_b, corpus) = score_{TF\text{-}IDF}(SAC_{doc_a} \cap SAC_{doc_b}, SAC_{doc_a} \cup SAC_{doc_b}, corpus)$$

Où SAC_{doc} est le sac de mots de doc.

Les documents doc_a et doc_b sont considérées comme ayant un rôle symmétriques, c'est pourquoi le sac de mots envoyé à $score_{TF\text{-}IDF}$ est leur union.

```
def tf_idf_bow_scorer(bag_maker, doca, docb, inversedocfreqs, out_of_corpus_value):
    baga = bag_maker(doca)
    bagb = bag_maker(docb)
    intersection = baga & bagb
    termfreq = term_frequencies(baga + bagb)
    return tf_idf_bow(intersection, termfreq, inversedocfreqs, out_of_corpus_value)
```

Où `bag_maker` est une fonction retournant un sac de mots.

4 Évaluation des approches (`semeval_executable.py`)

4.1 Imports

```
#!/usr/bin/env python3
from itertools import product, combinations
import spacy
from spacy.lang.en.stop_words import STOP_WORDS
from semeval_struct import *
from semeval_util import *
from semeval_xml import get_semeval_content
from semeval_taln import *
```

4.2 Scores

Les scores sont stockés dans un arbre construit à partir de l'arbre des documents. `compute_relqs_scores` calcule les scores de similarité d'un noeud de l'arbre des documents, en attribuant à chaque question relié son score obtenu en comparaison avec sa question originale.

```
def compute_relqs_scores(orgqnode, scorer):
    return {relid: scorer(orgqnode['org'], orgqnode[relid])
            for relid in orgqnode.keys() - {'org'}}
```

`make_score_tree` transforme le premier niveau d'un arbre de documents en lui appliquant `compute_relqs_scores` associé à la fonction de scoring recue en paramètre.

```
def make_score_tree(document_tree, scorer):
    return transformtree(
        lambda x: compute_relqs_scores(x, scorer),
        document_tree,
        0
    )
```

4.2.1 Écriture des fichiers de prédiction

Semeval fournit un script permettant de noter les performances d'une approche. Ce script prend en entrée un fichier de prédiction dont chaque ligne correspond à une question reliée et est formatée de la manière suivante :

```
orgq_id relq_id 0  score true
```

Les troisième et cinquième colonnes sont sans intérêt pour cette tâche.

Le fichier de prédiction est destiné à être ensuite comparé à un fichier de référence de Semeval, afin d'évaluer les performances du système.

La fonction `write_scores_to_file` permet de générer ce fichier de prédiction. Les résultats sont triés sur le tas, pour correspondre à l'ordre du fichier de références.

```
def write_scores_to_file(scores, filename):
    """Write a semeval score tree to a prediction file.

    Parameters
    ----------
    scores : dict of dict of float
        The scores to write.

    filename : str
       The name of the output file.
    """
    linebuffer = [(orgid, relid, str(0), str(score), 'true')
                  for orgid, relqs in scores.items()
                  for relid, score in relqs.items()]

    linebuffer.sort(key=lambda x: natural_sort_key(x[1]))

    with open(filename, 'w') as out:
        out.write('\n'.join(['\t'.join(el) for el in linebuffer]))
```

4.3 Dimensions orthogonales d'une approche

Plusieurs dimensions orthogonales sont envisagées pour appareiller des questions. Ces dimensions sont combinées les unes avec les autres, en faisant un produit cartésien, formant ainsi une approche.

4.3.1 Modèle de langage

Un seul modèle de langage est utilisé.

```
models = {
    'spacy_en': spacy.load('en')
}
```

4.3.2 Corpus

Les approches sont testées sur les données 2016 et 2017 de Semeval.

```
corpuses = {
    '2016': 'SemEval2016-Task3-CQA-QL-test-input.xml',
    '2017': 'SemEval2017-task3-English-test-input.xml',
}
```

4.3.3 Extraction de contenu

Deux manières d'extraire du contenu sont envisagées. Elles se différencient au niveau de l'extraction du contenu des questions reliées. La première extrait uniquement le sujet et le corps d'une question, tandis que la seconde extrait également les commentaires des questions reliées.

```
extractors = {
    'questions': lambda x: get_semeval_content(x).lower(),
   # 'questions_with_comments': get_semeval_content_with_relcomments
}
```

Ces fonctions sont fournies dans le fichier `semeval_xml.py`.

4.3.4 Filtrage des mots

Les mots d'un sac de mots peuvent être filtrés ou non selon un prédicat.

```
MAPPSENT_STOPWORDS = set(open('stopwords_en.txt', 'r').read().splitlines())

def isnotstopword(word):
    return word not in STOP_WORDS

def isnotstopword2(word):
    return word not in MAPPSENT_STOPWORDS

filters = {
    'gtr2': lambda word: len(word) > 2,
    'nostopwords': isnotstopword2,
    'nofilter': lambda x: True,
}
```

4.3.5 Construction des sacs de mots

Les sacs de mots sont construits à l'aide de deux fonctions. La première est une fonction d'extraction de caractéristique, qui étant donné un token, renvoie la caractéristique désirée de celui-ci. La deuxième est une fonction d'extraction de phrase, qui étant donné un document, renvoie un itérable contenant des mots.

Chaque méthode de construction de sacs de mots utilise ces deux fonctions.

```
def extracttext(tok):
    return tok.text

def extractlemma(tok):
    return tok.lemma_

def extractlabel(ent):
    return ent.label_ if hasattr(ent, 'label_') else None

def getentities(doc):
    return doc.ents

wordextractors = {
    'text': extracttext,
    'lemma': extractlemma,
    'label': extractlabel,
}
```

```

sentenceextractors = {
    'entities': getentities,
    'document': lambda x: x,
}

bowmakers = {
    'named_entities_text': ('text', 'entities'),
    'named_entities_label': ('label', 'entities'),
    'tokens': ('text', 'document'),
    'lemmas': ('lemma', 'document'),
}

def getbowmakerfunctions(key):
    return (wordextractors[bowmakers[key][0]], sentenceextractors[bowmakers[key][1]])
```

Les fonctions associées aux éléments de `bowmakers` sont destinés à être passés à la fonction `createbowmaker`, retournant une fonction permettant de construire un sac de mots selon les modalité voulues.

```
def createbowmaker(wordextractor, sentenceextractor, filters):
    def bowmaker(document):
        return Counter(
            list(filter(lambda x: all(f(x) for f in filters),
                   map(wordextractor, sentenceextractor(document))))
            )

    return bowmaker
```

4.3.6 Création des approches

Les arbres des documents sont précalculés pour éviter de répéter cette opération coûteuse.

```
training_file = 'SemEval2016-Task3-CQA-QL-train-part1.xml'

training_doctree = make_or_load_document_tree(
    training_file,
    'train_2016_part1.pickle',
    models['spacy_en'],
    get_semeval_content,
    verbose=True
)

doctrees = {
    '_'.join((model, corpus, extractor)): make_or_load_document_tree(
        corpuses[corpus],
        '_'.join((model, corpus, extractor) )+ '.pickle',
        models[model],
        extractors[extractor],
        verbose=True
    )
    for model, corpus, extractor in product(models, corpuses, extractors)
}
```

La fonction `nonemptypartitions` est utilisée pour combiner les filtres.

```
def nonemptypartitions(iterable):
    for i in range(1, len(iterable) + 1):
        for perm in combinations(iterable, i):
            yield perm

def join_predicates(iterable_preds):
    def joinedlocal(element):
        for pred in iterable_preds:
            if not pred(element):
                return False
        return True
    print('joining', *(pred for pred in iterable_preds))
    return joinedlocal

filters_partition = list(nonemptypartitions(set(filters) - {'nofilter'}))

filters_partition.append(('nofilter',))
```

Les approches sont crées en faisant le produit cartésien des dimensions envisagées.

```
approches = list(product(doctrees, bowmakers, filters_partition))
```

4.4 Exécution des approches

Chacune des approches précédemment générées est exécutée et les scores produits sont écrits dans les fichiers correspondants.

```
def getpredfilename(doctree, bowmaker, filterspartition):
    return '_'.join((doctree, bowmaker, *filterspartition, 'scores.pred'))


# inversedocfreqs = transformtree(
#     lambda wordextractor: inverse_document_frequencies(
#         [[wordextractor(tok) for tok in doc]
#          for org in training_doctree.values()
#          for doc in org.values()]
#     ),
#     wordextractors
# )

inversedocfreqs = {
    wordex + '_' + sentex: inverse_document_frequencies(
        [[wordextractors[wordex](tok) for tok in sentenceextractors[sentex](doc)]
         for org in training_doctree.values()
         for doc in org.values()]
    )
    for wordex, sentex in bowmakers.values()
}

out_of_corpus_value = max(inversedocfreqs['text_document'].values())

for doctree, bowmaker, filterspartition in approches:
    wordex, sentex = bowmakers[bowmaker]
```

```
    bowmakerfunc = createbowmaker(wordextractors[wordex], sentenceextractors[sentex],
                                  [filters[filterkey] for filterkey in filterspartition])

    scores = make_score_tree(
        doctrees[doctree],
        lambda a, b: tf_idf_bow_scorer(
            bowmakerfunc, a, b,
            inversedocfreqs[wordex + '_' + sentex], out_of_corpus_value)
    )

    prediction_file = getpredfilename(doctree, bowmaker, filterspartition)
    print('writing scores to', prediction_file)
    write_scores_to_file(scores, prediction_file)
```

5 Résultats

Le script shell suivant est utilisé pour extraire le score MAP d'un fichier de prédiction :

```
prediction=$1

if echo $prediction | grep --quiet "2016"
then
    reference=scorer/SemEval2016-Task3-CQA-QL-test.xml.subtaskB.relevancy
else
    reference=scorer/SemEval2017-Task3-CQA-QL-test.xml.subtaskB.relevancy
fi

python2 scorer/ev.py $reference $prediction | grep "^MAP" | sed 's/ \+/;/g' | cut -f 4 -d
 ↪  ';'
```

Année	Sac de mots	Filtres	Score MAP
2016	Lemmes	≤ 2, Mots outils	0.7559
2016	Lemmes	≤ 2	0.7500
2016	Textes des entités nommées	≤ 2	0.7493
2016	Textes des entités nommées	Mots outils	0.7493
2016	Textes des entités nommées	≤ 2, Mots outils	0.7493
2016	Textes des entités nommées	Pas de filtre	0.7493
2016	Tokens	≤ 2, Mots outils	0.7460
2016	Tokens	≤ 2	0.7409
2016	Lemmes	Mots outils	0.7395
2016	Lemmes	Pas de filtre	0.7352
2016	Tokens	Mots outils	0.7288
2016	Tokens	Pas de filtre	0.7223
2016	Étiquettes des entités nommées	≤ 2	0.7079
2016	Étiquettes des entités nommées	Mots outils	0.7079
2016	Étiquettes des entités nommées	≤ 2, Mots outils	0.7079
2016	Étiquettes des entités nommées	Pas de filtre	0.7079
2017	Lemmes	≤ 2	0.4629
2017	Tokens	≤ 2	0.4515
2017	Lemmes	Pas de filtre	0.4492
2017	Lemmes	Mots outils	0.4484
2017	Lemmes	≤ 2, Mots outils	0.4483
2017	Tokens	≤ 2, Mots outils	0.4392
2017	Tokens	Pas de filtre	0.4366
2017	Tokens	Mots outils	0.4300
2017	Étiquettes des entités nommées	≤ 2	0.4153
2017	Étiquettes des entités nommées	Mots outils	0.4153
2017	Étiquettes des entités nommées	≤ 2, Mots outils	0.4153
2017	Étiquettes des entités nommées	Pas de filtre	0.4153
2017	Textes des entités nommées	≤ 2	0.4083
2017	Textes des entités nommées	Mots outils	0.4083
2017	Textes des entités nommées	≤ 2, Mots outils	0.4083
2017	Textes des entités nommées	Pas de filtre	0.4083

Année	Score MAP baseline
2016	0.7475
2017	0.4185