TD6._sujet

February 15, 2023

1 TD6 N-gram

1.1 Partie 1 n-grams introduction

Les premières approches statistiques construisent des modèles à partir des fréquences, pas seulement des mots ou des tokens mais des n-grammes.

On veut utiliser la fonction de comparaison pour explorer les n-grammes présents dans nos données. On reprend 3 bouts des textes dans DinG. Tourner le code suivant pour obtenir `text_tokenized`.

[ ]:
```
from nltk.tokenize import RegexpTokenizer

ding1 = "0091 R   Euh bon après de toute façon on va à la limite on va faire 1␣
 ↪tour pour euh alors oui si par contre au départ on va devoir à tour de rôle␣
 ↪placer notre première ville de départ plus 1 route euh adjacente à notre␣
 ↪ville"
ding2 = "0119 R   Euh faut que tu places aussi une route dans la direction qui␣
 ↪t'intéresse euh alors tout ce qui est construit normalement ne peut plus␣
 ↪être euh déconstruit ouais on peut pas détruire quelque chose par contre␣
 ↪ouais il faut peut-être qu'on attende un tout petit peu parce que c'est on␣
 ↪peut considérer que c'est le début du jeu"
ding3 = "0133 R   Alors ouais pour bien faire donc là au le premier tour de jeu␣
 ↪ça va être vraiment ça ça va être on va chacun mettre 2 moins on met une␣
 ↪ville chacun puis une deuxième ville avec une route adjacente et ensuite à␣
 ↪chaque tour on commence par lancer les dés ensuite on peut faire du commerce␣
 ↪euh et ensuite on fait nos achats et quand on a fini bah c'est le suivant␣
 ↪qui va lancer les dés et euh les les on va distribuer les ressources qui␣
 ↪correspondent et ensuite on va euh on va faire les différentes actions"

my_text = ding1.split(' ', 2)[-1].strip() + ' ' + \
ding2.split(' ', 2)[-1].strip() + ' ' + ding3.split(' ', 2)[-1].strip()

regex_tokenizer = RegexpTokenizer("[a-zA-Z'`éèî]+")
text_tokenized = regex_tokenizer.tokenize(my_text)
```

1.1.1 Exo1 Explorer ngrams dans NLTK

Créer une variable de type `list` et qui s'appelle `trigrams`. Obtenir tous les tri-grammes dans `text_tokenized` avec la méthode `ngrams` et padding à gauche et à droite avec le symbole "_".

"padding" une forme spéciale de masquage où les étapes masquées se trouvent au début ou à la fin d'une séquence. Par exemple, les premiers 3-grams seront sous forme :

```
('_', '_', 'Euh'),  ('_', 'Euh', 'bon'),  ('Euh', 'bon', 'après')
```

Stocker dans le variable `trigrams` et afficher le résultat.

Attention : padding se fait toujours après tokenization. Essayer cette méthode en mettant `my_text` au lien de `text_tokenized`, observer les résultats.

[ ]:
```
from nltk.util import ngrams
```

[ ]:
```
# TODO

trigrams = pass
trigrams
```

Maintenant obtenir les unigrammes, les bigrammes, les trigrammes et les quadgrammes dans `text_tokenized`. Nommer les "unigrams", "bigrams", "trigrams", "quadgrams".

Ensuite créer une variable `merged_bigrams` pour stocker les strings de bigrammes en mettant un espace au milieu. Par exemple pour le bigramme ("le", "jeu"), vous obtenez "le jeu". Afficher la variable `merged_bigrams`.

[ ]:
```
# TODO

unigrams = pass
bigrams = pass
trigrams = pass
quatregrams = pass

merged_bigrams
```

1.1.2 Exo2 Trouver la fréquence des ngrams avec `FreqDist`

Importer FreqDist, ensuite l'appliquer sur la liste des bigrams que vous venez de créer. Stoker les 5 bigrams les plus fréquents dans un variable `bigram_top5`, afficher les bigrams ainsi que le nombre d'occurrence.

[ ]:
```
# TODO

from nltk import FreqDist
```

Maintenant faire un affichage (plot) de `bigram_top5`.

[ ]:
```
# TODO
```

1.1.3 Exo3 Explorer `everygrams`

Importer `everygrams` depuis NLTK, lire le documentation de cette méthode et appliquer sur `text_tokenized`. Pour cela, écrire une fonction `ngram_generator` qui prend comme entrée 3

arguments : - une séquance des mots - longueur minimale du ngram - longueur maximale du ngram

et comme sortie une liste des tous les ngrams satisfaisant.

Stocker la sortie de votre fonction dans une variable `my_ngram` et afficher les occurrences de tous les n-grams (n=[1,4]) sous forme de dictionnaire : - clé : ngram - valeur : occurrence

Par exemple :

`{'du jeu': 1, 'du': 2, ...}`

[ ]:
```
# TODO

from nltk import everygrams
```

1.2 Partie 2 Entraîner un modèle du langue avec n-gram

Dans cette partie, on va regarder une application spécifique en utilisant des n-grams. Nous allons entraîner un modèle MLE (Maximum Likelihood Estimator).

Le module est proposé dans NLTK. On va utiliser ce module et créer un modèle 3-gram.

[ ]:
```
from nltk.lm import MLE
```

Faire tourner le code suivant pour charger les données. Il va créer un texte nommé “language-never-random.txt” dans votre répertoire.

[ ]:
```
import os
import requests
import io

if os.path.isfile('language-never-random.txt'):
    with io.open('language-never-random.txt', encoding='utf8') as fin:
        text = fin.read()
else:
    url = "https://gist.githubusercontent.com/alvations/
 ↪53b01e4076573fea47c6057120bb017a/raw/
 ↪b01ff96a5f76848450e648f35da6497ca9454e4a/language-never-random.txt"
    text = requests.get(url).content.decode('utf8')
    with io.open('language-never-random.txt', 'w', encoding='utf8') as fout:
        fout.write(text)
```

1.2.1 Exo1 Préparation des données d’entraînement

Créer une variable `tokenized_text` qui stocke le `text` tokenisé. Pour cela, vous aller :

- séparer d’abord des phrases et stocker votre résultat dans une variable `sent_tokenize`
- et puis tokenzise des mots dans chaque sentence (vous pouvez faire un loop pour cela) et stocker le résultat dans une variable `tokenized_text`. Le `tokenized_text` est une list de lists, chaque sous-list contient des mots (i.e. une phrase). Par exemple, les deux premiers élements seront quelques choses comme :

```
[['language', 'is', 'never', ',', 'ever', ',', 'ever', ',', 'random', 'adam',
'kilgarriff', 'abstract', 'language', 'users', 'never', 'choose', 'words',
'randomly', ',', 'and', 'language', 'is', 'essentially', 'non-random', '.'],
['statistical', 'hypothesis', 'testing', 'uses', 'a', 'null', 'hypothesis', ',',
'which', 'posits', 'randomness', '.']]
```

Vous êtes libres de choisir les tokenizers disponibles dans NLTK. N'oublier pas de les importer avant.

Print les deux premiers éléments dans `tokenized_text` et afficher le nombre de phrases dans `tokenized_text`.

[ ]:
```
import re
from nltk.tokenize import ToktokTokenizer
from nltk.lm.preprocessing import padded_everygram_pipeline

# TODO


# uncomment for test
# print(tokenized_text[:2])
# print(len(tokenized_text))
```

Avec le `tokenized_text`, on va créer les données d'entraînement avec un pipeline dans NLTK : `padded_everygram_pipeline`.

Etudier la documentation de cette méthode : https://www.nltk.org/api/nltk.lm.preprocessing.html#nltk.lm.prepro

Fournir le texte toknized et le nombre de n-grams. Pour les résultats (output), nommer-les `train_data` et `padded_sents`.

[ ]:
```
# TODO
```

1.2.2 Exo2 Adapter (fit) le modèle avec les données d'entraînement, explorer le vocabulaire du modèle

Utiliser la méthode `fit` dans `model`, vous obtenez donc le modèle entraîné (simple non ?).

Combien de mots dans le vocabulaire de ce modèle ? Puisque vocab est un attribut du modèle, vous pouvez regarder directement avec `model.vocab`.

[ ]:
```
# TODO


# Uncomment for test
# print(model.vocab)
```

Le vocabulaire nous aide à gérer des mots inconnus par un string `UNK` (unknown). Si on recherche le vocabulaire inconnu, le modèle va automatiquement remplacer ce mot par `UNK`.

La méthode `lookup` de vocab nous aide à faire la recherche. Regarder le doc pour `nltk.lm.vocabulary.Vocabulary`.

Tester avec la phrase "language is never random random ." et "language is never random lah ."

[ ]:
```
# TODO
```

1.2.3 Exo3 Compter des occurrences, calculer des probabilités

Combien de 3-grams existent-il dans ce modèle ? Tester avec `counts` et afficher le résultat.

[ ]:
```
# TODO
```

Pour un certain mot ou une séquence des mots, on pourrait compter son occurrence avec la méthode `counts`.

Tester avec : - un unigram, compter l'occurrence de 'language', 'rah' - un bigram, compter l'occurrence de **'is'|'language'** - un trigram, compter l'occurrence de **'never'|'language is'**

[ ]:
```
# TODO
```

Le but d'un modèle de langue est de pouvoir calculer la probabilité d'une séquence des mots. Pour cela, on pourrait utiliser la méthode `score`.

Tester avec : - un unigram, la probabilité du mot 'language' - un bigram, la probabilité d'avoir la séquence **"language is"** - un trigram, la probabilité d'avoir la séquence **"language is never"**

[ ]:
```
# TODO
```

1.2.4 Exo4 Si vous avez du temps (Question bonus) :

Quelle est la phrase la plus probable dans votre modèle ? - "language is not random" - "language is never random"

Aller chercher dans le text original, est-ce que ces deux phrases existent-elles ?

Si oui, pourqoui vous obtenez ces probabilités ?

Comment pourriez-vous améliorer votre modèle de langue pour obtenir la bonne probabilité ?

[ ]:
```
# TODO
```

1.2.5 /! Submission instructions

You will need to submit this lab on Arche before 9:59am on **Friday, 3rd March**. Submit either a `.py` or an `.ipynb` file containing the functions you wrote for all the exercises and name it `td6_firstname_lastname.py` or `td6_firstname_lastname.ipynb` accordingly, where `firstname` should be your first name and `lastname` should be your last name (e.g. Jane Doe's submission should be called `td6_jane_doe.py` or `td6_jane_doe.ipynb`).