TP4 : Génération de texte avec des chaînes de Markov

D'après un TP de Y. Dupont

L'objectif de ce TP est de faire de la génération de texte après avoir calculé un modèle de langue sur un ensemble de textes de votre choix.

Exercice 0 : préparer les données

Si vous ne voulez pas travailler sur les textes de Proust et de Zola fournis sur le moodle (800 000 mots environ), créez votre propre corpus d'entraînement.

Récupérez sur le site du projet Gutenberg, ou ailleurs, les textes des auteurs que vous souhaitez. Vous pouvez travailler dans la langue que vous préférez, merci de me signaler si vous ne travaillez pas sur le français.

Il faut télécharger les textes en **plain text UTF-8**.

Nettoyez ces fichiers afin de n'avoir plus que le texte du roman (il faut retirer du contenu au début et à la fin de chaque fichier).

Implémentez la fonction :

```
def doc_to_word_list(path):
    '''
    paramètre : chemin vers un corpus
    returns : liste de liste
        - liste de phrases
        - chaque phrase est une liste de tokens
    '''
    with open(path) as input_stream:
        # compléter en utilisant les outils de découpage en phrases et en
tokens du TP2.
```

Vous pouvez aussi utiliser le tokéniseur "custom" suivant qui fonctionne à peu près :

```
tokenizer = nltk.RegexpTokenizer(r"([A-Z][A-Z0-9.]+|[0-9]+[,.][0-9]+|
[cdjlmnst]'|qu'|[\w'-]+|\S)")
```

Résultat attendu :

```
[['Adaptant', 'aux', 'humbles', 'ambitions', 'de', 'cette', 'femme', ',',
"l'", '
```

```
instinct', ',', 'le', 'désir', ',', "l'", 'industrie', ',', "qu'", 'il',
'avait'
, 'toujours', 'eus', ',', 'il', "s'", 'était', 'ingénié', 'à', 'se',
'bâtir', ',
', 'fort', 'au-dessous', 'de', "l'", 'ancienne', ',', 'une', 'position',
'nouvel
le', 'et', 'appropriée', 'à', 'la', 'compagne', 'qui', "l'", 'occuperait',
'avec
', 'lui', '.'],
['Or', 'il', "s'", 'y', 'montrait', 'un', 'autre', 'homme', '.'] ...
```

Créez une variable `corpus` contenant l'ensemble des documents sous forme de liste de liste de tokens, en utilisant par exemple la fonction `extend`.

```
corpus = []
corpus.extend(doc_to_word_list("doc1.txt"))
corpus.extend(doc_to_word_list("doc2.txt"))
```

Exercice 1 : chaîne de Markov de premier ordre (unigrammes)

Compter les transitions

Écrivez une fonction `count_unigram_transitions` qui, étant donné un corpus, compte les différentes transitions d'un mot à l'autre dans le corpus.

Le résultat prendra la forme d'un dictionnaire à deux niveaux pour un mot M_{t}

- Le premier contient le mot M_{t}
- le second niveau contient le nombre d'occurrences de M_{t+1} après M_{t}

```
def count_unigram_transitions(corpus):
    '''
    paramètre : corpus (séquence de séquence de tokens)
    returns : dictionnaire à deux niveaux contenant pour chaque mot le
nombre d'apparitions pour chaque mot suivant possible.
    '''
    transitions = {}
    # compléter
    return transitions
```

Résultat attendu :

```
{
'Ma' : {'mère': 15, "grand'mère": 16, 'tante': 4, 'position': 1, 'femme':
4, 'parole': 6, 'vie': 1, 'mauvaise': 1, 'grand-mère': 3, 'belle': 2,
'sensibilité': 1, 'chambre': 1, 'déception': 1, 'plus': 1, 'pauvre': 6,
'foi': 13, 'chère': 7, 'pince': 1, 'tête': 2, 'soeur': 1, 'fille': 1,
```

```
'bonne': 1
},
'quand' : {'il': 228, 'ils': 68, 'on': 135, 'je': 95, 'elle': 167, ',': 20,
"c'": 4, 'un': 18, 'ma': 3, 'cette': 4, 'le': 43, 'à': 4, 'la': 39, 'nous':
41, ...
},
...
}
```

Des effectifs aux probabilités

Écrivez une fonction `probabilify` qui, transforme, pour chaque mot, les effectifs de transition en probabilité d'apparition.

La somme des probabilités des transitions partant d'un même mot doit sommer à 1 (ou presque 1, les nombres étant petits, des imprécisions peuvent arriver).

```
def probabilify(comptes_transitions):
    '''
    paramètre : dictionnaire de transitions
    returns : dictionnaire à deux niveaux contenant pour chaque mot la
probabilité d'apparition pour chaque mot suivant possible.
    '''
    # Compléter
```

Résultat attendu :

```
{
' Ma ' : {'mère': 0.16853932584269662, "grand'mère": 0.1797752808988764,
'tante': 0.0449438202247191, 'position': 0.011235955056179775, 'femme':
0.0449438202247191, 'parole': 0.06741573033707865, 'vie':
0.011235955056179775, 'mauvaise': 0.011235955056179775, 'grand-mère':
0.033707865168539325, ...
},
'quand' : {'il': 0.18181818181818182, 'ils': 0.05422647527910686, 'on':
0.1076555023923445, 'je': 0.07575757575757576, 'elle': 0.1331738437001595,
...
},
...
}
```

Créer une chaîne de Markov d'ordre 1

Définissez comme suit une chaîne de Markov :

```
def markov_chain_unigram(corpus):
    transitions = count_unigram_transitions(corpus)
```

```
    return probabilify(transitions)
```

```
markov_chain = markov_chain_unigram(corpus)
```

Générer des phrases avec votre chaîne de Markov

Écrivez une fonction `generate_unigram` qui, étant donné un token de départ, génère une phrase.

On considère que nous sommes à la fin d'une phrase dès que l'on atteint une ponctuation forte ('.', '?', '!', etc.), et on génère au maximum `NB_MOTS_MAXI`.

Pour générer le mot suivant, on prendra systématiquement le mot **le plus probable** étant donné le mot précédent.

Afin de vérifier le bon fonctionnement de la fonction, vous générerez d'abord une phrase commençant par `Si`.

```
ponctuation = # compléter
NB_MOTS_MAXI = # compléter
def generate_unigram(markov_chain, start_token):
    '''
    Paramètres :
    - chaîne de Markov
    - token de départ
    Returns : void.
    Affiche la phrase générée.
    '''
    maximum = NB_MOTS_MAXI
    token = start_token
    # compléter
    while token not in ponctuation and maximum > 0:
        # compléter
```

```
generate_unigram(markov_chain, "Si")
```

Améliorer la génération

Comme vous avez pu le constater, la génération est assez… décevante. Pour améliorer la génération automatique, nous allons ajouter un peu d'aléatoire.

Créez une fonction `generate_unigram_alea` pour y ajouter un argument `n_best` qui permet de choisir non pas le mot **le plus probable**, mais un mot au hasard **parmi les `n_best` plus probables**.

```
def generer_unigramme_alea(markov_chain, start_token, n_best=1):
    '''
    Paramètres :
    - chaîne de Markov
    - token de départ
    - nombre de tokens dans lesquels choisir le token suivant

    Returns : void.
    Affiche la phrase générée.
    '''
```

Testez pour différentes valeurs de `n_best` :

```
generer_unigramme_alea(markov_chain, "Si", 2)
```

```
generer_unigramme_alea(markov_chain, "Si", 10)
```

Exercice 2 : chaîne de Markov d'ordre 2 (bigrammes)

Nous avons fini par avoir une génération acceptable, mais elle demeure encore un peu incohérente. Afin d'améliorer la cohérence de la génération des mots, nous allons utiliser un contexte plus grand afin de générer les mots.

Reprennez les fonctions précédentes (à part `probabilify` qui restera la même) afin d'utiliser non pas un mot, mais deux mots en contexte.

Si deux mots sont utilisés comme contexte, ils doivent être séparés par une espace.

```
def compter_transitions_bigrammes(corpus):
    transitions = {}

    # compléter

    return transitions
```

```
def chaine_markov_bigramme(corpus):
    # compléter
```

```
markov_chain = chaine_markov_bigramme(corpus)
```

```
def generate_bi(markov_chain, start_token):
    maximum = NB_MOTS_MAXI
    token = start_token
    prevs = ["", ""]
    # compléter
```

```
generate_bi(markov_chain, "Si")
```

```
def generate_bi(markov_chain, start_token, n_best=1):
    maximum = NB_MOTS_MAXI
    token = start_token
    prevs = ["", ""]
    # compléter
```

```
generate_bi(markov_chain, "Si", 2)
```

```
generate_bi(markov_chain, "Si", 10)
```

Exercice 3 : chaîne de Markov d'ordre arbitraire

Nous y sommes presque ! Nous commençons à avoir une génération de qualité acceptable.

Afin de rendre votre programme plus générique et plus puissant, réécrivez les fonctions précédente afin qu'elles puissent créer des chaînes de Markov de n'importe quel ordre.

L'ordre de la chaîne de Markov devra être rajouté aux fonctions.

```
def count_transitions(corpus, ordre):
    # compléter
```

```
def markov_chain(corpus, ordre):
    # compléter
```

```
markov_chain = markov_chain(corpus, 3)
```

```
def generate(markov_chain, ordre, start_token):
    # compléter
```

```
generate(markov_chain, 3, "Si")
```

```
def generate_alea(markov_chain, ordre, start_token, n_best=1):
    # compléter
```

```
generate(markov_chain, 3, "Si", 2)
```