tp_l8dn003_20_04

March 22, 2020

TP n°4: distance de Levenshtein

La distance de Levenshtein, ou distance d'édition, est une façon de mesurer à quel point deux mots se ressemblent. L'idée de base est d'attribuer un poids à chacune des opérations qui peuvent permettre de passer d'un mot à un autre (suppression ou insertion d'une lettre, interversion de deux lettres, remplacement d'une lettre par une autre...). La distance est calculée à partir de ce nombre d'opérations (éventuellement pondérées). Cette notion est très utilisée pour faire de la correction orthographique, pour gérer les tokens inconnus dans de nombreuses applications de TAL, pour faire de la morphologie computationnelle, *etc*. L'article Wikipedia https://fr.wikipedia.org/wiki/Distance_de_Levenshtein est une bonne source d'information. On y donne même une version de l'algorithme, mais je propose plutôt une approche progressive pour mieux appréhender les aspects algorithmiques du problème: on va implémenter différentes fonctions de mesure de distance qui en faisant différentes hypothèses simplificatrices.

Prise en compte (simplifiée) des substitutions

1. Ecrire une fonction `nb_subst(u,v)` qui renvoie le nombre de substitutions d'un caractère qui distinguent u de v. Par hypothèse les deux chaînes sont de même longueur (sinon on renvoie -1), et la seule opération mesurée est le remplacement d'un caractère par un autre.
2. Cette valeur dépend bien sûr du nombre de lettres du mot. On peut essayer de prendre cela en compte en renvoyant le ratio de caractères modifiés par rapport au nombre total de caractères: la distance maximale est alors à 1 (tous les caractères sont différents). Implémenter cette fonction `dist_subst(u,v)`.
3. Cette fonction est une fonction *partielle* : elle ne permet pas de définir une distance entre deux chaînes quelconques, mais seulement entre deux chaînes de même longueur. Pour passer à une mesure totale, on peut proposer que si les deux chaînes ne sont pas de même longueur, on fait comme si on rallongeait artificiellement la plus courte en ajoutant des caractères toujours différents de leur correspondant dans la chaîne la plus longue. Le nombre d'opérations peut être interprété comme *le nombre de caractères à modifier ou à supprimer pour passer de la chaîne la plus longue à la plus courte*. Comme précédemment, on va renvoyer un ratio plutôt qu'un nombre brut. Implémenter cette fonction `subst_totale(u,v)`.
4. Pour tester les fonctions :
    1. Faites un jeu d'essai (paires de mots) qui correspondent aux différents cas de figure que vous pouvez anticiper. Augmentez systématiquement ce jeu d'essai, en vérifiant bien sûr que vos prédictions sont correctes.
    2. Prennez plusieurs centaines de mots-formes dans un texte (pas trop quand-même: 200 c'est bien) de la façon dont on a procédé dans les TP précédents, et calculez les distances 2 à 2: regardez quels sont les mots les plus proches (mais quand-même différents), etc.

Une fois vos tests réalisés, vous constaterez que, sans surprise, les mots les plus proches (mais différents) sont les paire de mots les plus longs qui ne diffèrent que d'une seule lettre. En effet, à cause de la normalisation effectuée, une différence d'un seul caractère est comptée comme plus petite si le mot a plus de lettres. Cela peut correspondre à une certaine intuition, mais dans la pratique la distance proposée par Levenshtein ne procède pas à une normalisation.

Autre approche : prise en compte de séquences absentes

Ecrire une fonction qui va s'appliquer à deux chaînes de caractères telles que la seconde est obtenue à partie de la première en ôtant une (seule) portion *contiguë*. La fonction `coupe(u,v)` doit renvoyer la taille de la coupe qui a été pratiquée dans u pour obtenir v.
Formellement: $\exists a, b, w$ t.q. $u = awb$ & $v = ab$, où a, b et w sont des chaînes de caractères non vides.
La fonction doit renvoyer -1 si la condition n'est pas vérifiée (0 signifie que u==v).

Indice algorithmique: il faut chercher d'une part le préfixe commun, d'autre part le suffixe commun: si ces deux sous-chaînes sont non vides, il ne reste plus qu'à compter la taille de w.

Implémentation de la distance de Levenshtein standard

Les deux questions précédentes permettent de se faire une (petite) idée de la complexité de la version complète de l'algorithme. La troisième partie du TP va consister à implémenter l'algorithme, en partant de la version proposé dans la page wikipedia. C'est un algorithme de programmation dynamique: l'idée est de stocker le plus possible de résultats intermédiaires pour ne les faire qu'une seule fois. La base de l'algorithme est donc la table qui stocke ces informations.

1. Essayer de faire tourner l'algo sur quelques exemples à la main pour comprendre son esprit général. Les exemples de la page peuvent aider.
2. Adapter en python l'algorithme de la page. On trouve facilement sur internet des bouts de code qui implémentent l'algorithme, mais j'espère que vous avez compris le peu d'intérêt qu'il y a à recopier du code si on ne comprend pas ce qu'il fait.
3. Installer (avec anaconda ou pip) le package python-levenshtein, ce qui va vous permettre de comparer votre version à l'implémentation fournie par cette bibliothèque.

[1]:
```
import Levenshtein as lev
```

[4]:
```
lev.distance("mamaizon", "mison")
```

[4]: 4