MASTER AIC
UNIVERSITÉ PARIS-SACLAY
2018-2019

SECOND-ORDER HMM FOR TYPOS CORRECTION

TC4/ ALGORITHMES D'INFÉRENCE ET D'APPRENTISSAGE À GRANDE ÉCHELLE

CEDRIC DES LAURIERS
MIRWAISSE DJANBAZ

1. Introduction

Ce travail vise à corriger les fautes de frappe sans dictionnaire à l'aide d'un jeu d'entraînement et d'un jeu test tous deux labellisés, d'un HMM et de l'algorithme de Viterbi afin de restituer les séquences d'observations les plus probables.

Plus précisément le jeu de données est labellisé par le caractère réel, et l'observation est le caractère constaté :
Par exemple, pour les jeux de données train10, test10, train20 et test20 avec respectivement 10 et 20% d'erreurs de substitution, les mots se présentent ainsi :

Pour le mot 'other' on a :
[('o', 'o'), ('t', 't'), ('h', 'h'), ('e', 'e'), ('f', 'r')]

Ici l'erreur est le 'f' ('f' l'observation) à la place de 'r' ('r' l'état, le caractère réel).

Dans un premier temps, nous verrons comment traiter les erreurs de substitution avec un HMM d'ordre 1 et 2, puis nous traiterons les erreurs d'insertion, de suppression et enfin nous finirons par une approche non supervisée pour supprimer les fautes de frappe.

2. Correction des erreurs de substitution à l'aide du HMM d'ordre 1

Voir détails du code dans le notebook question1.pynb

L'HMM d'ordre 1 permet de prendre en compte l'état à l'instant t-1.
Pour entraîner un HMM à l'ordre 1, il faut construire différentes matrices : la matrice de transition (état à état : (27,27) : 26 + 1 unk). Ici, les états sont simplement les caractères en label dans le jeu de données, une matrice d'observation qui associe les différentes observations à leurs états respectifs, et enfin une matrice d'initiation répertoriant les états qui débutent les séquences.
Pour illustrer le passage ci-dessus :
[('observation1', 'initiation1'), ('observation2', 'état2') ...]
initiation1 étant un état aussi

Pour construire ces 3 matrices il suffit de compter les tuples ou les états avec des dictionnaires par exemple.

Dans le même temps, on mappe les index des matrices obtenues avec des dictionnaires pour pouvoir récupérer les valeurs des états ou des observations.
Ensuite, l'algorithme de Viterbi se charge de renvoyer la séquence d'observation la plus probable à l'aide des différentes matrices obtenues par le HMM.

Résultats :

Erreur (caractères) %	do_nothing	HMM d'ordre 1 + Viterbi
test10	10.17 %	7.67 %
test20	19.40 %	15.01 %

Pour les parties 2 à 5 voir le notebook q2_q5.pynb

3. Correction des erreurs de substitution à l'aide d'un HMM d'ordre 2

Même principe que dans le HMM d'ordre 1, sauf que cette fois ci nous voulons aussi prendre en compte l'état à t-2 pour capter un peu plus de contexte et donc faire moins d'erreur.
On construit donc une matrice de transition à l'ordre 2 (état_t, (état_t_2, état_t_1)).
Rappelons ici qu'un état est une lettre de l'alphabet, les dimensions de cette matrice sont donc (26+1, 27*27), le +1 vient du unk. Une fois la matrice de transition obtenue on applique Viterbi.

Remarque : on conserve la matrice de transition d'ordre 1 pour les caractères en 2ème position de séquence par exemple.

Résultats :

Erreur (caractères) %	do_nothing	HMM d'ordre 2 + Viterbi
test10	10.17 %	4.60 %
test20	19.40 %	9.26 %

4. Correction des erreurs d'insertion à l'aide d'un HMM d'ordre 2

Pour cette partie, nous avons utilisé un corpus propre en y ajoutant 10% d'erreur d'insertion. Pour signifier une erreur insertion, nous mettons la valeur d'état = '<ins>'. Comme ci-dessous :

Exemple : [('a', 'a'), ('c', 'c'), ('c', 'c'), ('o', 'o'), ('u', 'u'), ('n', 'n'), ('t', 't')]
devient : [('a', 'a'), ('c', 'c'), ('c', 'c'), ('w', '<ins>'), ('o', 'o'), ('u', 'u'), ('n', 'n'), ('t', 't')]

Il suffit donc juste d'ajouter la valeur '<ins>' au vocabulaire pour pouvoir prédire les insertions.
Matrice de transition 2 : (28, 28*28)
Résultats :

Erreur (caractères) %	do_nothing	HMM d'ordre 2 + Viterbi
test10	9.09 %	5.04 %

5. Correction des erreurs de suppression à l'aide d'un HMM d'ordre 2

Pour cette partie, nous avons dû modifier un peu la structure des données afin d'adapter le HMM aux erreurs de suppression.
Pour cela, nous avons dû former des couples de lettres dans chaque mot dans la mesure du possible (si le nombre de lettres contenues dans le mot n'est pas pair).

Par exemple :

mot pair
[(f,f), ('o','o'), ('r','r'), ('m','m')]
devient [('fo', 'fo'), ('rm', 'rm')]

mot impair
[('t','t'), ('h','h'), ('e', 'e')]
devient [('th', 'th'), ('e', 'e')]

On y incorpore un certain nombre d'erreurs aléatoirement par exemple :

[('vi', 'vi'), ('ol', 'ol'), ('en', 'en'), ('ce', 'ce')]
devient [('i', 'vi'), ('ol', 'ol'), ('en', 'en'), ('ce', 'ce')]
Ici, la lettre supprimée est ‘v’. L’objectif est donc que l’algorithme prédit le couple de lettre ‘vi’.

La faiblesse de cette méthode est l’augmentation du nombre d’états et d’observations, et donc du temps de calcul (grandes matrices), sinon cela marche plutôt bien visiblement.
Matrice de transition 2 : ((27×27), $(27*27)^2$).

Erreur (caractères) %	do_nothing	HMM d'ordre 2 + Viterbi
test	7.42 %	3.10 %

6. Correction de mot à l’aide de l’apprentissage non supervisé

Nous avons ici utilisé la distance de Levenshtein pour déterminer ou non s’il y a une faute dans le mot, la correction ne se fait plus au niveau caractère (ou ensemble de caractère) mais au niveau du mot.

*“La **distance de Levenshtein** est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d’une chaîne à l’autre.”*
Wikipedia

Par exemple la distance de Levenshtein entre “chien” et “chion” est 1 puisqu’il suffit qu’on substitue ‘o’ par ‘e’ dans ‘chion’ pour obtenir ‘chien’.

Pour que cette méthode fonctionne, nous avons besoin d’un texte correctement écrit (le jeu d'entraînement). Plus celui-ci sera gros, plus la taille du vocabulaire sera élevée, plus les corrections seront meilleures (en contrepartie d’un temps de calcul plus élevé à cause du parcours du vocabulaire). Ensuite, il nous faut un texte dans la même langue à corriger. Pour que cette méthode marche (ou ait un intérêt), il faut qu’il y ait des mots en commun dans les 2 textes.

Les paramètres à déterminer *(empiriquement, via grid search par exemple : créer un jeu de validation alors)* sont la distance maximum de Levenshtein au-delà duquel on ne modifie pas

le mot (ici nous la fixons à 1 = max_Cost) et la taille de la liste des mots ayant une distance inférieure à max_Cost (de même ici nous la fixons à 1).

Si les deux conditions sont respectées, alors on modifie le mot par un autre mot présent dans le vocabulaire généré par le texte d'entraînement ayant la plus faible distance de Levenshtein avec celui-ci.

Pour le jeu d'entraînement on utilise le texte train10 mais sans erreur avec seulement les labels.
Pour le jeu de test, on utilise seulement les observations de test 10

Mot bien écrit %	do_nothing	Levenshtein
test10	62.89 %	71.28 %

Cela marche relativement bien, mais ce genre de méthode qui corrige au niveau du mot peut vite avoir un temps de calcul trop long puisque l'on parcourt pour chaque mot du texte à corriger tout le vocabulaire, à cela vient s'ajouter le calcul des distances. Pour un texte gros et / ou un vocabulaire trop gros, cette approche est vite trop coûteuse en termes de calcul.