

TP Levenshtein et Classification

Sylvain Gault

8 septembre 2024

1 Introduction

Ce TP est à réaliser par groupe de 3 ou 4 en Python sous l'environnement de développement de votre choix. Il sera à rendre le Dimanche 07/10/2024 à 13h37 UTC. Il sera à rendre sur Teams dans le devoir « *TP note* ». Vous nommerez vos fichiers du noms des membres du groupe.

Vous rendrez un rapport en **PDF** contenant au moins les réponses aux questions ainsi que toute description nécessaire à la reproduction de vos résultats. **Numérotez les questions** auxquelles vous répondez dans votre rapport et incluez au moins une réponse pour chaque question. Pas de réponse = pas de points. N'oubliez pas de lister **les noms** de tous les membres de votre groupe.


Le but de ce TP est d'implémenter des algorithmes classiques de correction orthographique et de classification.

2 Levenshtein

Exercice 1 Lenvenshtein simple

Le but de cet exercice est d'implémenter un algorithme de Levenshtein simple. Où les coûts sont 1 pour l'insertion et suppression, et 2 pour la substitution.

1. Écrivez un programme qui prend comme argument sur la ligne de commande un mot à corriger.
2. Écrivez une fonction `levenshtein` qui prend en argument 2 mots. Elle sera complétée dans les questions suivantes pour renvoyer la distance de Levenshtein.
3. Votre fonction `main` va devoir appeler cette fonction avec le mot passé en argument, et un mot avec lequel le comparer.
4. Dans votre fonction `levenshtein`, définissez une liste de liste qui contiendra les valeurs de la grille. Remplissez-la de 0 pour le moment.
5. Remplissez la première ligne et la première colonne de votre grille. Ces valeurs devraient être triviales.
6. Itérez sur toutes les paires de lettres et définissez la valeur de votre grille en fonction des valeurs de la grille déjà calculées. N'oubliez pas que deux lettres identiques ont un coût de 0.
7. Renvoyez la valeur de grille qui correspond à la distance de Levenshtein des deux chaînes.
8. Testez votre fonction sur quelques paires de mots. Testez des mots courts pour vous assurer du bon fonctionnement de votre code.

9. Testez aussi sur des mots de plus en plus longs afin d'évaluer la performance de votre programme.
10. Récupérez le dictionnaire de mots français `french.txt`  sur Teams et utilisez-le pour comparer le mot de la ligne de commande avec tous les mots du dictionnaire. Affichez seulement le mot le plus proche.
11. (*Bonus*) Utiliser la classe `PriorityQueue` du module `queue` afin de garder les 10 mots du dictionnaire les plus proches.

Exercice 2 Liste d'édition

Dans cet exercice, on va conserver et afficher les opérations d'édition nécessaires à transformer le mot de la ligne de commande en mot corrects.

1. Modifiez votre fonction `levenshtein` pour créer une nouvelle liste de liste qui contiendra au départ des chaînes vides.
2. Modifiez votre code pour que lors que vous remplissez la valeur d'une case de la grille de valeurs, vous gardiez aussi (dans la nouvelle grille) d'où vient cette valeur. Vous mettrez "D" s'il aura fallu supprimer un caractère de la chaîne d'entrée, "I" s'il aura fallu insérer, "S" s'il aura fallu substituer, "-" si les caractères étaient identiques. En cas d'ambiguïté, préférez toujours les substitutions aux insertions ou suppressions.
3. À la fin de la fonction, utilisez cette nouvelle grille afin de reconstituer le chemin en partant de la fin. Ce chemin indique les opérations à effectuer pour transformer la première chaîne en la deuxième.
4. Remettez cette liste à l'endroit. Vous pourrez utiliser la méthode `reverse` des listes.
5. Retournez cette liste en plus de la distance.
6. Testez votre code sur quelques exemples simples, puis compliqués.

Exercice 3 (*Bonus*) Levenshtein pondéré


Le but de cet exercice est de d'implémenter un algorithme de Levenshtein qui prend en compte le fait que certaines erreurs sont plus courantes que d'autres.

1. Copiez votre fonction `levenshtein` en `levenshteinw`. Ajoutez 3 arguments optionnels (initialement à `None`) à cette nouvelle fonction. Ces arguments donnent les poids d'insertion, suppression et substitution.
2. Dans votre nouvelle fonction, définissez ces valeurs à 1, 1 et 2 s'ils sont à `None`.
3. Testez sur quelques exemples intéressants avec de nouveaux poids. Vous pourrez construire ces exemples sans utiliser le dictionnaire.
4. Modifiez votre fonction `levenshteinw` pour qu'elle utilise non pas une valeur constante pour les poids d'insertion / suppression / substitution, mais une valeur qui dépend de la lettre insérée / supprimée, ou de la paire de lettre substituées. Vous définirez un dictionnaire par défaut qui vaut 1 pour les deux premiers, et 2 pour les substitutions. Le comportement de votre code ne devrait pas changer pour le moment.
5. Définissez dans votre fonction `main` un dictionnaire qui définit quelles lettres sont le plus susceptibles d'être insérées. On supposera que les lettres sur le bord du clavier sont le plus susceptibles d'être insérées.

6. Idem, mais pour les poids de suppression. On supposera que les voyelles sont plus susceptibles d'être oubliées.
7. Idem pour les paires de lettres. Pour l'exercice, vous mettrez toutes les valeurs égales à 2, et quelques valeurs avec un poids inférieur.
8. Testez votre code sur quelques cas bien choisis pour montrer que votre levenshtein pondéré fonctionne bien.

3 Classifieur Bayésien Naïf

Exercice 4 Classification Bayésienne

Le but de cet exercice est de créer un classifieur Bayésien basique. Vous aurez besoin du fichier `SMSSpamCollection.tsv` . Ce fichier est constitué de deux colonnes séparées par une tabulation. La première colonne indique `spam` ou `ham` pour dire si le SMS est du spam. Le SMS est sur le reste de la ligne.

1. Créez un programme qui lit le fichier et crée deux listes. Une liste contenant les spam, une liste contenant les SMS réguliers.
2. Combien de SMS de chaque catégorie avez-vous ? Quel est la probabilité d'avoir un spam ?
3. Concaténez vos listes en deux grands méga-documents séparés par une espace.
4. Dans la vraie vie, on utiliserait une tokenisation plus intelligente. Ici, on va simplement garder uniquement les lettres en remplaçant tous les autres caractères par des espaces. Mettez ce code dans une fonction `preprocess`.
5. Dans votre fonction `preprocess`, remplacez les suite de 2 espaces ou plus par un seul caractère espace puis mettez tout en minuscule.
6. Séparez maintenant vos deux méga-documents en une liste de tokens en séparant sur les espaces. Votre fonction `preprocess` est maintenant complète.
7. Dans une fonction `bayes_train`, créez le vocabulaire en récupérant l'intégralité des mots connus, spam ou non-spam.
8. Comptez le nombre d'occurrence de chaque mot séparément pour les spams et les non-spams. Vous pourrez utiliser la classe `Counter` si vous le souhaitez. Ces décomptes de mots et le vocabulaire constituent votre modèle. Retournez-les à la fin de votre fonction `bayes_train`.
9. Comment calculer $\hat{P}(w_i|c_j)$ dans votre programme ?
10. Écrivez une fonction `prob_word` qui calculera cette probabilité.
11. Écrivez une fonction `prob_class` qui calcule $\hat{P}(c|d)$ la probabilité d'une classe donnée (spam ou non-spam) étant donné un document (un SMS).
12. En utilisant la fonction précédente, écrivez une fonction `is_spam` qui détermine si un SMS donné est un spam ou non. Testez-la sur quelques exemples.
13. En utilisant des documents d'un seul mot, déterminez quels sont les mots de votre vocabulaire les plus susceptibles d'être détectés comme spam ou comme non-spam.
14. De manière à évaluer votre classifieur, entraînez-le sur seulement 75% des SMS du dataset initial. Les 25% restant seront utilisé pour tester.

15. Testez sur les 25% de données inutilisées pendant l'entraînement. Prétraitez les SMS un à un avec votre même fonction `preprocess` puis passez-les à votre fonction `is_spam`.
16. Quel est le taux de succès de votre classifieur ?
17. (*Bonus*) En fonction de vos résultats, calculez la précision et le rappel (recall) de votre classifieur.