Algorithms for Speech and NLP TD 3 : Construction d'un système de normalisation des tweets

Pirashanth RATNAMOGAN
MVA ENS Paris-Saclay
pirashanth.ratnamogan@ens-paris-saclay.fr

1. But du TD

Ce troisème TD a pour but de créer un système capable de normaliser des tweets. En effet, lorsqu'on utilise un algorithme de traitement du langage, il est très important de procéder à un débruitage des données afin de donner des données pertinentes à nos algorithme. Dans ce TD, je me suis concentré sur certaines tâches de normalisation : retirer les informations considérées comme non pertinentes, corriger l'orthographe des mots, transformer les formes "one-to-many", retirer les caractères non-ascii, et enfin uniformiser les caractères en les mettant en minuscule.

2. Le système de normalisation des tweets

2.1. Preprocessing des tweets

Afin de traiter efficacement les tweets, un ensemble d'opérations élémentaires est effectué sur les tweets. Le système créé commence par mettre tout le tweet en minuscule. Ensuite, puisque le système est adapté aux tweets en anglais, il a été décidé de retirer tous les caractères non-ascii ce qui permet notamment de retirer les smilleys dont on peut difficile tirer de l'information. Les adresses url sont également retirés. Un dictionnaire convertissant les 70 acronymes les plus courants en leur version complète a également été crée. Enfin,j'ai créé un outil de Tokenization adapté aux opérations que je voulais réaliser sur les tweets en m'appuyant sur les outils de la librairie `re`. Enfin une option permet de retirer ou non tout ce qui commence par # ou @ (spécifique aux tweets).

2.2. Des métriques dans l'espace des mots

2.2.1 Des métriques basées sur les caractères

Les distances de Levenshtein

On a vu en cours qu'afin de travailler sur la correction orthographique la distance de Levenshtein ainsi que la distance de Damereau-Levenshtein étaient très utiles. En effet, celles-ci permettent de mesure le nombre de caractères à changer afin de passer d'un string à un autre. On a vu qu'un mot et sa version mal orthographiée sont en général très proche avec la métrique de Levenshtein. Les distances de Levenshtein et de Damereau-Levenshtein ont été implémenté dans le fichier `FormalSimilarityComponents.py`. On a vu que la plupart des fautes orthographiques sont dans une Levenshtein distance de 2, on ne proposera donc la correction d'un mot que par un mot qui est à une Levenshtein distance inférieure à 2 du mot original.

L'utilisation d'un noyau qui s'applique sur les chaînes de caractères

Afin de compléter la notion de distance de Levenshtein vu en cours, on peut utiliser le String kernel [1], qui est un noyau très populaire pour comparer des strings (entre 0 et 1, plus la valeur est grande plus les strings comparés sont proches). En adaptant un code disponible sur internet https ://github.com/timshenkao/StringKernelSVM, j'ai implémenté ce noyau qui me servira dans l'architecture de mon système.

2.2.2 Une métrique basée sur la sémantique : Utilisation de Glove

Des tests sur mon système m'ont permis de voir que l'utilisation de la distance basée sur l'alignement des caractères composant les strings comparés n'était pas suffisante seule. Par exemple 'freack' et 'breack' sont à une Levenshtein de 1 alors qu'ils ne sont sémantiquement pas lié. Afin de pallier à ce problème, on peut utiliser la notion vue en cours de word embedding. Ainsi, j'offre la possibilité dans mon modèle d'indiquer le chemin vers les "Twitter pretrained word vectors" de Glove afin d'obtenir une représentation des deux mots comparés. L'utilisation de la cosine similarity permettra de comparer les mots dans cet embedding. L'utilisation du vocabulaire pretrained de Glove me permet également d'avoir une liste de mots, un vocabulaire qui me permet de voir si le mot que je teste existe ou non. Si le mot n'est pas dans la liste des mots dans Glove, il est très probable que ce mot soit mal orthographié. Si le mot est une entité nommée alors il y a peu

de chance que l'on propose un mot à la correction avec une Levenshtein distance inférieure à 2 alors il ne sera pas changé même s'il n'est pas dans le vocabulaire Glove.

2.3. Context2vec : utiliser l'information sur le contexte

Ce qu'on a vu dans ce qui précède c'est que si l'on a une faute d'orthographe dans un mot il existe des méthodes pour trouver le mot plus adapté dans une liste de mots données pour sa correction. Pour avoir cette liste, il nous a été demandé d'utiliser `context2vec` [2]. Sans rentrer dans le détail cet outil propose des mots qui seraint adaptés à un contexte donné en entrée. En retirant le mot mal orthographié de la phrase originale, on a donc une liste de mots qui peuvent potentiellement remplacer ce mot.

2.4. Le système globale

Maintenant que l'on a toutes les briques du système on peut décrire comment celles-ci sont imbriquées. Etant donné un tweet à normaliser on effectue les opérations suivantes :

1. On effectue le preprocessing décrit (élmination des adresses, des smilleys, texte mis en minuscule, possibilité d'élimination des @ et #) et on tokenize le tweet
2. On parcourt tous les mots du tweets, si ce ne sont pas des caractère spéciaux (éléments twitter token après @ ou # ou ponctuation), on retire le mot du tweet, on prend son contexte et context2vec nous propose une liste de mots que l'on pourrait mettre à la place du mot retiré.
3. Pour chaque mots du tweet, on utilise la Levenshtein distance et le String Kernel qui extraient une sous-liste de mots proposés par context2vec proche du mot original (**dans tous les cas au moins à une Levenshtein distance inférieure à 2**).
4. Pour chaque mots, on utilise les "pretrained word-vectors" de Glove pour voir si le mot original est dans le vocabulaire et si un des mots proposés à la fin de l'étape précédente est proche dans l'espace décrit par Glove (cosine similarity). En fonction du résultat soit on remplace le mot par un des mots proposés soit on le conserve tel qu'il était à l'origine.

3. Les tests

3.1. Quelques exemples

Afin de voir l'efficacité de mon modèle j'ai utilisé certains résultats issus du Corpus Bataclan fournis et d'autres crées manuellement. J'ai mis en italique l'exemple original et en gras la normalisation proposée :

1 – *RT @blckliquor : *kkk murders ppl**
rt @blckliquor :* kkk murders people *
2 – *I will tyr to do my best*
i will try to do my best.
3 – *Can ya come asap*
can ya come as soon as possible ?
4 – *RT @RT_com : #Paris txi drivers turned off their meters, took people home for free - reports https ://t.co/UUjfMTCXsM https ://t.co/1TicKMbNJy*
rt @rt_com : #paris taxi drivers turned off their beers, took people home for free- reports
5 – *RT @Imkjac : The Paris attacks are not #political They are not #religious – the people behind them are just pure #evil #ParisAttacks #Batac. . .*
rt @imkjac : the panic attacks are not #political they are not #religious- - the people behind them are just pure #evil #parisattacks #batac

Comme on peut le voir le résultat est honnête, on peut aussi décider de retirer les entités spécifiques aux tweet (mot après # ou @). Les phrases 1, 2 et 3 sont relativement bien normalisées (même si on laisse quelques one-to-many). On peut regretter la transformation de "meters" en "beers" et de "Paris" en "panic" dans les phrases 4 et 5.

3.2. Avantages et défauts du système

Comme ce système a été fait en moins d'une semaine il comporte évidemment des lacunes. Le fait de ne pas avoir entrainé sur les données que l'on veut utiliser les word vectors ainsi que context2vec nous renvoie des absurdités. Ainsi alors que "Paris" est très répété dans le corpus Bataclan, il est changé par "panic" car "Paris" n'est pas renvoyé en résultat par context2vec. On voit également que les one-to-many pas présent dans le dictionaire sont mal traités ("kkk"). On peut regretter le fait que le placement de la ponctuation n'est pas parfait malgré des effortst. Le temps de calculs de l'algorithme est aussi à améliorer car il ne tiendrait pas le temps réel. On terminera par relever les bons résultats obtenus, la correction orthographique dans des contextes favorable : "txi" en "taxi" et "tyr" en "try" notamment. Pour améliorer le résultat, on pourrait effectuer plusieurs choses : entrainer context2vec et Glove sur des données pertinentes, améliorer l'heuristique de choix des bons correctifs, se baser sur plus de données labélisées à la main pour repérer les erreurs les plus courantes.

Références

[1] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. 2002.

[2] O. Melamud, J. Goldberger, and I. Dagan. context2vec : Learning generic context embedding with bidirectional lstm. 2016.