Correction de texte chat par approche de traduction automatique

Mai 2017
ALOUI Cindy et DARY Franck
M1 Informatique

SOMMAIRE

Introduction

De nombreuses entreprises produisent un très grand volume de données textuelles sous forme de 'tchat' c'est-à-dire une discutions instantanée entre un opérateur et un client. Ces entreprises peuvent par exemple être des Fournisseurs d'Accès à Internet, qui proposent à leurs clients des conseils, du service après vente ou une assistance technique au travers d'une messagerie accessible depuis leur site internet.

Le but de ces services n'est évidemment pas de produire ces données (corpus de texte tchat), cependant il serait bénéfique de pouvoir les exploiter puisqu'elles seront produites quoi qu'il advienne. Ce genre de corpus est trop grand pour être exploité par un salarié, alors pour réduire les coûts on utilise traditionnellement des outils de Traitement Automatique du Langage Naturel (TALN). Néanmoins un outil d'analyse automatique n'a pas la compréhension d'un humain, et il a besoin de reconnaître tous ou une très grande majorité des mots du corpus pour être efficace ; et compte tenu du fait que le corpus est généré par des humains qui ne font forcément attention à l'orthographe, et qui font des fautes de frappe : les outils automatiques ne sont pas adaptés au traitement de ce genre de corpus.

Pour rendre ces tchats exploitables, on peut leur appliquer un prétraitement qui consiste à en corriger tous les mots qui ne font pas partie de la langue française.

Lorsqu'un outil automatique travaille sur un texte, il va d'abord le tokeniser c'est à dire remplacer chaque occurrence d'un mot par un code (entier naturel) qui lui est propre.

L'objectif de ce projet est de proposer un programme informatique qui est capable de corriger un grand nombre d'erreurs dans n'importe quel texte en langage naturel, qui pourra ainsi être utilisé pour prétraiter un corpus sur lequel on souhaite ensuite appliquer un outil de TALN.

Il s'agit donc de transformer un texte en un autre, on peut voir ça comme une traduction d'un texte en français incorrect vers un texte en français correct. C'est pourquoi dans le cadre de ce projet nous allons utiliser des techniques issues de la traduction automatique (table de traduction, algorithme de Viterbi), dans le but de déterminer si leur usage est pertinent dans ce cas.

On se limite dans le cadre de ce projet à la correction des erreurs d'orthographe qui sont les plus pénalisantes pour les outils automatiques - en effet même si par exemple un verbe est mal conjugué il garde son sens. On se limite aussi à la langue française, même si le code est écrit de manière non spécifique et peut être utilisé avec toutes les langues qui partagent l'alphabet français.

Dépôt Git distant contenant le code source :

https://github.com/Hartbook/CorrectionTexteChat

Documentation en ligne :

https://hartbook.github.io/CorrectionTexteChat/html/index.html

Exemple bref d'entrée et sortie du programme :

Input :

[00:16:34] _CLIENT_: Bjr, monsieu j'ai oune probeme avec ma livebox : lecran tv est noi.

Output :

[00:16:34] _CLIENT_: Bonjour, monsieur j'ai un problème avec ma livebox : écran tv est noir.

Description détaillée du fonctionnement :

Il y deux modes d'utilisation différents :

- le mode 'buildDatabase' qui est une phase d'apprentissage automatique dans laquelle toutes les ressources nécessaires à la correction sont générées. Cette étape prend du temps, mais ne doit être effectuée qu'une seule fois et permet par la suite de corriger autant de textes tchat qu'on le souhaite.

- le mode 'correct' qui utilise tous les fichiers crées par 'buildDatabase' et qui sert à corriger un texte. La phase d'initialisation prend un certain temps, donc pour ne payer ce coût qu'une fois il est préférable de corriger un grand texte plutôt que de corriger plusieurs petits textes.

Nous allons maintenant détailler le fonctionnement et le travail réalisé sur ces deux modes, de façon chronologique (par rapport à l'exécution du programme).

--buildDatabase :

- Lecture des lexiques :

Le programme commence par lire un ensemble de fichiers passés en argument, ces fichiers représentent le lexique des mots corrects de la langue française qui sera utilisé tout au long du programme pour déterminer si un mot est bien orthographié ou pas.

Pour ce faire on lit les textes, et dès qu'un mot (ensembles de lettres séparés par des caractères non-alphanumériques - tous sauf les lettres, les chiffres, le ` - ` le ` ' ` et le ` _ `) n'appartenant pas déjà au lexique est rencontré, on l'ajoute au lexique en lui associant un code (entier naturel) unique qu'on appellera un token par la suite.

Le lexique est représenté par une table de hachage. Avant de déterminer si un mot appartient ou non au lexique (donc avant l'étape de hachage du mot), il est normalisé : on retire toutes les majuscules et on les remplace par des minuscules. Ce qui permet de ne pas avoir de doublons dans le lexique.

De plus le lexique contient des tokens spéciaux dont les valeurs vont de 0 à 6, ils représentent respectivement : Les mots inconnus (pas dans le lexique), les adresses e-mail, les nombres, les noms propres, les

dates et durées, les adresses web, et les retour à la ligne (jamais utilisé pour tokeniser un texte, il est utile pour construire le modèle 3-grammes). Cela permet de ne pas polluer le lexique avec des mots qui dans tous les cas ne doivent pas être corrigés. Par exemple '13h12' et '14heures' se verront attribuer le même token.

En pratique deux lexiques sont utilisés, l'un ne contiendra que des mots corrects de la langue française (il servira donc à déterminer si un mot est bien orthographié ou pas) et l'autre contiendra à la fois les mots corrects de la langue française (l'intégralité du premier lexique) ainsi que tous les mots incorrects rencontrés pendant l'exécution du programme. A la fin de cette première étape (Lecture des lexiques), les deux lexiques sont identiques (et ne contiennent donc que des mots corrects).

- Construction du modèle 3-grammes :

Maintenant le programme lit un nouvel ensemble de fichiers aussi passés en argument, ces fichiers représentent un grand nombre de textes tchat pas forcément corrigés (corpus d'apprentissage) similaires à ceux que nous souhaitons corriger. Le but étant d'apprendre de ces textes, les probabilités qu'une certaine suite de mots apparaisse dans un texte tchat. Par exemple la suite de mots 'je suis' est plus probable en français que la suite de mots 'je est'. Le programme se limite aux suites de maximum 3 mots, c'est à dire que lorsqu'il lit le corpus d'apprentissage il ne garde que les 3 derniers mots lus en mémoire (comme une fenêtre glissante).

Le corpus d'apprentissage n'étant pas corrigé, il contient des erreurs (mots qui ne sont pas reconnus par le lexique des mots corrects), pour ne pas apprendre à partir d'informations erronées, on ignore les n-grammes qui contiennent un mot incorrect. Comme la fin d'une phrase donne rarement des indices sur le début de la phrase suivante, le programme ne compte pas les n-grammes qui sont à cheval sur deux phrases.

Le corpus d'apprentissage est en général très grand, c'est pourquoi le programme utilise plusieurs threads pour accélérer ce processus (texte découpé en plusieurs morceaux et chaque thread s'occupe d'un morceau).

Une fois que le modèle 3-grammes est généré, il est enregistré sur le disque dans le dossier (chemin relatif à l'exécutable) data/gramsCount/. Pour que sa représentation soit plus concise (ce qui fait gagner un temps précieux à la lecture) il est enregistré au format binaire (ce qui implique

qu'un fichier écrit par une certaine machine doit être lu par la même machine).

Un modèle 3-gramme est représenté par :

- le nombre de 1-grammes (pas uniques) qu'il contient.
- le nombre de 2-grammes (pas uniques) qu'il contient.
- le nombre de 3-grammes (pas uniques) qu'il contient.
- la liste des 1-grammes qu'il contient ainsi que leur nombre d'occurrences.
- la liste des 2-grammes qu'il contient ainsi que leur nombre d'occurrences.
- la liste des 3-grammes qu'il contient ainsi que leur nombre d'occurrences.

Donc si l'on demande au modèle la probabilité d'un n-gramme, il va retourner (nombre d'occurrences de ce n-gramme)/(nombre de n-grammes). Sauf dans le cas où le n-gramme n'a jamais été rencontré lors de l'apprentissage (nombre d'occurrences égal à 0), alors dans ce cas on retourne une probabilité spéciale, basse mais non-nulle.

Les ordinateurs gèrent inefficacement les successions de produits de termes proches de zéro, c'est pourquoi on travaille avec des logarithmes. Le modèle de langage va donc retourner l'opposé du logarithme base *e* (logProb) de la probabilité.

- Construction de la table de traduction :

Le programme va ensuite lire un ensemble de paires de fichiers (fichierIncorrect / fichierCorrect) passés en argument, où fichierIncorrect est un texte en français qui n'a pas encore été corrigé et fichierCorrect est le même texte ayant été corrigé par un francophone. Ce qui implique que les deux textes d'une même paire doivent être alignés (chaque phrase a sa correspondante). Les deux textes (incorrect et correct) de chaque paire sont d'abord tokenisés (le fichier incorrect est tokenisé par le lexique incorrect et vice et versa), lorsqu'un mot inconnu est rencontré par le lexique des mots corrects (donc dans le texte correct) le mot est ajouté à la fois au lexique correct et au lexique incorrect (le lexique correct est un sous ensemble du lexique incorrect comme expliqué plus haut), lorsqu'un mot inconnu est rencontré par le lexique des mots incorrects (donc dans le texte incorrect) le mot est ajouté au lexique des mots incorrects.

A partir de ces paires le programme va apprendre une table de traduction, qui est un objet qui regroupe un ensemble de paires (motIncorrect / motCorrect) et leur attribue un score. Plus ce score est bas, plus le mot incorrect a de chances de se corriger en le mot correct. On détermine ces paires de mots et leur score en appliquant un algorithme d'espérance-maximisation (EM). L'algorithme est itératif, le nombre d'itérations est fixé à 20 (facilement modifiable) ce qui en pratique permet la convergence. C'est une étape lente c'est pourquoi le programme utilise plusieurs threads pour l'accélérer (l'algorithme s'effectue phrase par phrase donc le traitement de chaque phrase se fait en parallèle).

L'algorithme génère plusieurs traductions possibles pour chaque mots, dont certaines ne sont pas du tout pertinentes. C'est pourquoi on supprime les traductions dont le score est inférieur à un certain seuil fixé (déterminé suite à des tests). Pour obtenir le score comme décrit précédemment, on passe la valeur trouvée par l'algorithme (une probabilité entre 0 et 1) en logProb. Ce qui va donner un score entre 0 (traduction très probable) et -ln(seuil) (traduction acceptable la moins probable). La valeur 0 est gênante pour la suite du programme (voir algorithme de Viterbi plus bas), c'est pourquoi on transforme encore une fois les scores. On effectue score = 2*(1+score), ce qui permet d'éviter la valeur 0 (le meilleur score devient 2) et augmente l'écart entre les scores (pour handicaper les mauvais scores).

Une fois la création de la table de traduction terminée, elle est enregistrée sur le disque en format ascii, dans le dossier (chemin relatif à l'exécutable) data/translationTable/. De plus elle est aussi enregistrée dans le même endroit dans un format facilement lisible par un humain, dans une optique de débogage.

- Ecriture des lexiques sur le disque :

Enfin le programme peut enregistrer sur le disque les lexiques qui ont étés augmentés par les étapes précédentes.

Le lexique des mots corrects est enregistré dans le dossier (chemin relatif à l'exécutable) data/lexicon/corrige/.

Le lexique des mots incorrects est enregistré dans le dossier (chemin relatif à l'exécutable) data/lexicon/brut/.

--correct :

- Lectures :

Le programme va lire depuis les fichiers passés en arguments un lexique correct, un lexique incorrect, un modèle 3-gramme et une traduction. Ces fichiers sont ceux générés par 'buildDatabase'.

- Correction du texte :

En premier lieu le texte à corriger est tokenisé (en utilisant le lexique des mots incorrects). Ensuite pour chaque phrase de ce texte, elle est corrigée - chaque phrase est indépendante donc leur correction se fait en parallèle (multithreading).

- Correction d'une phrase tokenisée :
    - Construction du treillis des corrections probables :

Pour chaque mot correct de la phrase, on l'ajoute tel quel au treillis avec une probabilité de 1 (mais peu importe cette valeur).

Pour chaque mot *m* incorrect de la phrase, on va ajouter au treillis (dans la colonne qui correspond à ce mot) un ensemble de traductions probables de ce mot. Ces corrections probables sont générées de deux façons différentes :

- à partir de la table de traduction : pour toutes les paires de mots (*m*,*t*) de logProb *p*, on ajoute *t* au treillis avec la logProb p.

- à partir de la distance de Levenshtein (d'édition) : On parcourt le lexique correct et pour tout mot *t* de ce lexique, si les tailles de *m* et *t* ne sont pas trop différentes (2 caractères de différence au maximum), on calcule une distance de Levenshtein personnalisée de la manière suivante : les substitutions d'une lettre accentuable (a,e,i,o,u) vers une lettre accentuée valent 0.2 et toutes les autres substitutions coutent 1. L'insertion d'un caractère en fin de mot coute 0.3 si une telle insertion ne s'est jamais produite, toutes les autres insertions coutent 1. La suppression d'un caractère en fin de mot coute 0.3 si une telle suppression ne s'est jamais produite, la suppression d'un caractère (` - ` ou ` ' ` ou ` _ `) coute 0.1 et toutes les autres

suppressions coutent 1. Ce qui nous donne une distance *d*, si cette distance est inférieure ou égale à un seuil (2 dans notre cas), on ajoute *t* au treillis avec la logProb *d*+2 (pour que les valeurs soient similaires à celles obtenues par table de traduction).

Si deux corrections identiques de logProbs *p1* et *p2* sont ajoutées treillis en tant que corrections du même mot, alors on en garde qu'un seul exemplaire et ces logProbs se renforcent à l'aide d'une fonction élaborée par nos soins :

min = min(*p1,p2*)

max = max(*p1,p2*)

diff = max - min

nouvelle logProb = min - max(0.5 * (min-diff^2), 0)

Les avantages de cette fonction sont : donne un nombre positif, donne un nombre inférieur ou égal à la plus petite des deux logProbs, au plus les deux logProbs sont proches au plus le résultat est proche de min/2.

- Application de l'algorithme de Viterbi :

Les nœuds du treillis sont des 2-grammes, sauf pour ceux de la première colonne qui sont des 1-grammes.

On applique l'algorithme de Viterbi pour trouver la meilleure traduction de la phrase avec : comme probabilités d'émissions celles obtenues précédemment (lors de la création du treillis) et comme probabilités de transitions les probabilités des n-grammes (premier mot 1-gramme, 2ème mot 2-gramme et à partir du 3ème mot 3-grammes).

- Réintroduction de la syntaxe originelle dans la correction :

Pour l'instant le programme a généré une correction sous forme de suite de phrases corrigées, sans aucune syntaxe. On va donc recopier la syntaxe du document original dans la correction. Pour ce faire on lit le texte d'origine en recopiant la syntaxe dans le document final, et à chaque fois que l'on rencontre un mot on recopie le mot correspondant dans la correction. (voir exemple dans la documentation)

Méthode d'évaluation des résultats obtenus :

Pour s'assurer que notre programme devenait de plus en plus efficace au fur et à mesure que nous implémentions des améliorations, il nous a fallu mettre en place une procédure qui soit capable d'évaluer les corrections les unes contre les autres. On utilise une paire de textes (non-corrigé / corrigé), non-utilisée dans la construction de la table de traduction.

Pour évaluer une correction c du texte : on construit un modèle de langage à partir de la version corrigée (par un humain) de ce texte, puis on calcule la perplexité p de c dans ce modèle de langage. On peut ainsi comparer les perplexités obtenues pour différentes corrections issues de plusieurs versions différentes de notre programme pour déterminer laquelle à fait un meilleur travail. Une autre solution possible aurait été de calculer la distance d'édition d'un texte vers un autre.

Conclusion :

La majorité des erreurs commises par le programme sont des erreurs de conjugaison ou de genre/nombre. Nous estimons que le programme est satisfaisant compte tenu du fait que les phrases gardent globalement leur sens ; et qu'une fois qu'un mot est mis sous forme canonique leur conjugaison et genre/nombre n'importe plus.

L'utilisation de l'algorithme de Levenshtein pour proposer des corrections est très pertinente, la grande majorité des corrections en sont issues. Les constantes utilisées dans cette algorithme déterminent franchement le temps d'exécution du programme, et les valeurs que nous leur avons attribué permettent d'avoir des bonnes corrections dans un temps correct (le fait de relâcher les contraintes augmente beaucoup le temps d'exécution mais n'améliore pas la correction).

L'utilisation de l'algorithme de Viterbi est très pertinente car elle permet l'utilisation du modèle 3-gramme qui est assez puissant pour obtenir une bonne correction dans ce cas de figure.

L'utilisation de la table de traduction pour proposer des corrections est plus anecdotique. Premièrement il est difficile de disposer d'un très grand corpus de paires de textes incorrects / corrects corrigés manuellement, ce qui fait que la table de traduction ne sera pas très fournie. Il est beaucoup plus utile d'écrire directement une table de traduction à la main, dans laquelle on pourra combler les lacunes de l'algorithme de Levenshtein. En effet ce dernier ne proposera jamais de corriger 'bjr' en 'bonjour', donc pour ce genre d'abréviation une bonne table de traduction s'avère utile.

Nous avions aussi envisagé de proposer des corrections issues des similarités phonétiques, mais manquant de temps nous avons préférés nous focaliser sur des valeurs sûres. Nous pensons que si deux mots sont proches phonétiquement ils ont de grandes chances d'être proche au sens de la distance de Levenshtein et donc ce type de correction s'avérerai souvent redondante, et pourrai même introduire des erreurs car en français une même combinaison de lettre peut se prononcer de plusieurs façons différentes. Donc cela aurait pris beaucoup de temps pour un gain faible.

Gestion du projet :

Nous avons travaillé ensemble et pas en parallèle.

Nous avons utilisé git comme gestionnaire de versions, et notre dépôt est hébergé sur github.

Nous avons écrit une documentation qui se trouve aussi sur github.

Le travail a été réalisé sur cette période de temps : du 17/04/17 au 18/05/17 environ 3 jours par semaine, 6 heures par jour.