TD1 IFT 6285 - Traitement automatique des langues naturelles

Auteur: Adel Nabli

Abstract

Pour résoudre la problématique de ce TD, nous avons implémenté une méthode basée sur le lissage Kneser Ney pour trigramme. Chaque mot manquant est extrapolé en retenant les tokens vu à l'entraînement les plus probables sachant le bigramme précédent. Nous appliquons cette méthode de 2 manières différentes: la première en parcourant le texte dans le sens de la lecture, la deuxième en parcourant le texte dans le sens inverse. Les mots les plus probables après mise en commun des 2 distributions sont retenus. Nous pondérons ensuite leurs masses de probabilité en prenant en compte leur tag et en calculant la probabilité d'avoir ledit tag à cet emplacement de la phrase. Nous retenons enfin les 3 tokens les plus probables suite à ces calculs. Cette méthode nous fournit une *accuracy* de 0.423 sur le texte de test *u30.en*.

1. Motivations

Le but de l'exercice que nous avons à résoudre peut se résumer à trouver, pour chaque mot manquant, le mot qui nous paraît être le plus probable sachant le contexte:

$$\underset{tokens \in vocabulaire}{\text{argmax}} \quad p(tokens|contexte) \tag{1}$$

Pour cela, il nous faut donc trouver une bonne définition du *contexte*, une distribution p qui nous paraît adaptée au problème considéré et préciser le *vocabulaire* que nous allons utiliser.

- Pour le *vocabulaire* sur lequel nous allons générer les nouveaux mots, on peut se poser la question de l'intérêt de générer de nouvelles formes jamais vues dans le corpus d'entraînement (par apprentissage analogique par exemple). Pour se donner une idée de la pertinence d'une telle démarche, on peut calculer *l'indice de richesse du vocabulaire* du corpus d'entraînement ($R = \frac{V}{N}$ où V est le nombre de vocables différents et N la longueur du texte), calculer son *indice de gini* et afficher la *courbe de Lorenz*.
On trouve ainsi un indice $R = 0.003$ (faible) et un gini de 0.6859 (élevé) pour le corpus d'entraînement anglais. Ces deux valeurs laissent à penser que nous sommes face à un corpus de texte à la **richesse de vocabulaire très pauvre** et où **les mots les plus fréquents représentent une grosse proportion des occurences totales**, ce qui nous permet de conclure que se **restreindre au vocabulaire vu en entraînement** devrait suffire pour couvrir la majorité des cas de mots manquants dans le corpus de test.

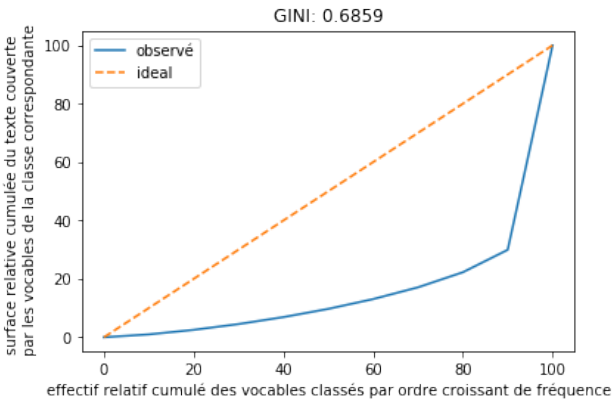

Figure 1. Courbe de concentration des mots du corpus en fonction de la fréquence des vocables

- Concernant la définition du *contexte*, nous pouvons essayer d'analyser les caractéristiques du texte que nous regardons lorsque l'on s'essaye soit même à la tâche. Suite à cela, nous pouvons définir plusieurs échelles de contexte:

  - A l'échelle du *proche voisinage* d'un trou, pour trouver un mot qui convient, on s'intéresse à la fois à la *grammaire* de l'expression (est-ce un verbe, un nom, un adjectif... qu'il faut mettre ?) ainsi qu'aux "*expressions naturelles*" qui viennent intuitivement à

l'esprit au vu des quelques mots entourant le trou.

- A l'échelle de *la phrase*, on regarde si le mot trouvé génère bien un texte ayant un *sens* cohérent.

- A l'échelle du *paragraphe*, on peut regarder si la phrase complète bien le message passé et si aucun contresens n'est créé.

Cependant, on remarque que plus l'échelle du contexte est globale, plus les caractéristiques à étudier deviennent abstraites (qu'est ce qu'un "*sens cohérent*" par exemple ?) et complexes à modéliser. Par soucis de simplicité, nous allons donc nous concentrer dans cet exercice sur le contexte donné par le *proche voisinage* du trou à compléter, à savoir les **quelques mots entourant le trou**.

- Ainsi, pour la distribution p, nous voulons utiliser une fonction qui nous permettra de prendre en compte ce contexte proche, ainsi qu'une fonction qui arrivera à générer des mots "*rarement vu*" en entraînement vu que cela représente la majorité de notre vocabulaire (voir *Fig.1*). Un modèle qui pourrait convenir est donc le modèle *bi-lstm*. De plus, ce modèle pourrait, si suffisamment de données il y a, capturer des information du contexte à l'échelle de la phrase. Cependant, par curiosité, par soucis de *temps d'entraînement* et pour apprendre à mieux l'apprivoiser, nous nous sommes plutôt penché sur le modèle de **Kneser-Ney** car très démocratisé en tant que "baseline" dans la communauté. Ce modèle permet bien d'apprendre statistiquement certaines *expressions naturelles* et permet de plus un *lissage* bienvenu au vu de nos données parcimonieuses. Nous allons détailler dans la partie suivante nos choix de modélisation pour implémenter ce modèle.

2. Choix de Modélisation

Le modèle de Kneser-Ney permet d'attribuer une masse de probabilité à un token ω_n sachant le $(n-1)gram$ précédent ce token dans la phrase considérée. Par exemple, en appliquant la formule de lissage, si nous sommes dans le cas d'une phrase "$\omega_1\omega_2$ <unk/>", pour un token donné ω appartenant au vocabulaire d'entraînement, nous avons:

$$p_{KN}(\omega|\omega_1\omega_2) = \frac{\max\{c(\omega_1\omega_2\omega) - D, 0\}}{c(\omega_1\omega_2)} + \frac{D}{c(\omega_1\omega_2)} N_{1+}(\omega_1\omega_2\bullet) \frac{\max\{c(\omega_1\omega_2\omega) - D, 0\}}{c(\omega_2)} + \frac{D}{c(\omega_2)} N_{1+}(\omega_2\bullet) \frac{N_{1+}(\bullet\omega)}{N_{1+}(\bullet\bullet)}$$

Avec:

- D la constante *discount* (des modèles plus raffinés utilisent différentes valeures de D suivant les cas).
- c la fonction *count* qui compte le nombre d'occurrence d'un $ngram$ donné en argument.
- N_{1+} la fonction qui compte le nombre de $ngram$ différents de la forme spécifiée qu'il existe.

Ainsi, pour un modèle de *Kneser-Ney* d'ordre n, le contexte se résume au $(n-1)gram$ précédent le trou. Or, ce n'est pas exactement ce que nous avons défini comme étant le contexte proche que nous voulions étudier. En effet, notre volonté est d'étudier les quelques mots qui *entourent* le trou et non pas seulement ceux qui le précède car nous croyons qu'il y a également de l'information contextuelle utile à droite du trou. Ainsi, nous avons choisit d'implémenter un modèle de Kneser-Ney qui apprend et prédit dans le sens de lecture usuelle et de le compléter avec le même modèle mais qui a apprit et qui prédit des tokens en parcourant les phrases du corpus dans le sens inverse. Nous espérons ainsi, en combinant les deux distributions apprises, créer artificiellement un contexte "$(n-1)gram_{gauche} \cup (n-1)gram_{droite}$".

Or, d'après (1), on cherche:

$$\underset{\omega \in vocab_{train}}{\operatorname{argmax}} \; p(\omega|contexte)$$

Ce qui revient alors à chercher:

$$\underset{\omega \in vocab_{train}}{\operatorname{argmax}} \; p(\omega|(n-1)gram_{gauche} \cup (n-1)gram_{droite})$$

et donc à maximiser pour $\omega \in vocab_{train}$:

$$\lambda \, p_{KN}(\omega|(n-1)gram_{gauche}) + (1-\lambda) p_{KN}(\omega|(n-1)gram_{droite})$$

avec $\lambda = \dfrac{p((n-1)gram_{gauche})}{p((n-1)gram_{gauche}) + p((n-1)gram_{droite})}$.

Nous avons choisi d'étudier plus classiquement le cas des **trigramme**, ce qui revient à faire l'hypothèse que le token manquant est indépendant de tous ceux éloignés de plus de 2 positions de lui. Ainsi, pour chaque trou $\omega_1^{gauche}\omega_2^{gauche}$<unk/>$\omega_2^{droite}\omega_1^{droite}$ dans le corpus test, nous sortons 2 ensembles de tokens qui sont, pour $sens \in \{gauche, \; droite\}$, les:

$$\underset{\omega \in vocab_{train}}{\operatorname{argmax}} \; p_{KN}(\omega|\omega_1^{sens}\omega_2^{sens})$$

Pour simplifier les calculs, **nous ferons l'hypothèse dans toute la suite que** $\lambda = 0.5$, à savoir que le modèle de langue rend tout autant probable le bigram de gauche que celui de droite. Cela permettra simplement de sommer les deux distributions pour extraire les tokens résultants les plus probables.

Avec un tel modèle, nous pouvons ainsi espérer pouvoir apprendre quelques expressions usuelles. Cependant, au vu de la parcimonie de nos données d'apprentissage, nous pouvons nous interroger quant à la capacité de notre modèle à inférer une grammaire correcte en même temps. C'est pourquoi on se propose à aider l'apprentissage de la grammaire en créant un nouveau modèle Kneser-Ney, mais qui se spécialise dans l'inférence des *tags*. Ainsi, on projette l'espace des mots de notre corpus (de dimension "nombre de mots différents" si on utilise une représentation *bag of words*) dans un espace plus petit (l'espace des *tags*), ce qui diminue le *nombre de classe existantes* mais augmente le *nombre de représentants par classe* dans le corpus d'entraînement.

En entraînant un troisième modèle Kneser-Ney sur un tel corpus transformé, on peut s'attendre à mieux pouvoir inférer les tags que les mots eux-même. Cette information "plus sûres" peut ainsi nous aider à guider notre choix dans la phase finale *d'extraction des 3 tokens les plus probables* parmi tout ceux pré-sélectionnés précédemment en pondérant la probabilité de chaque token par celle de son tag. Finalement, on maximise pour $\omega \in vocab_{train}$:

$$\Big[p_{KN}(\omega|\omega_1^g\omega_2^g,\ tag) + p_{KN}(\omega|\omega_1^d\omega_2^d,\ tag)\Big]p(tag)$$

Reste alors la question de l'implémentation d'une telle méthode, des résultats qu'elle engendre et celle de la validité de nos hypothèses.

3. Implémentation, expériences et résultats

3.1. Baseline: Kneser-Ney simple

Nous voulons implémenter notre méthode en langage python car c'est le langage avec lequel nous nous sentons le plus à l'aise. Nous nous sommes ainsi demandé si le modèle Kneser-Ney avait été codé dans la librairie `nltk`. A première vue, c'est bien ce qu'il semblait (dans `nltk > probability > KneserNeyProbDist`), cependant, en regardant le code source, nous nous sommes rendus compte que ce qui avait été implémenté était en fait un modèle de *backoff* qui autorise à certaines combinaisons de se voir attribuer une probabilité nulle si jamais vues dans le corpus d'entraînement (tel que codé dans `nltk`, $p_{KN}(\omega|\omega_1\omega_2) = 0$ si le bigram $\omega_1\omega_2$ n'a jamais été vu à l'entraînement).
Ne trouvant pas d'autre bibliothèques où la méthode aurait été codée correctement, nous nous sommes attelé à la coder nous même (voir *Notebook*). Cela ressemble à l'algorithme $Algo.1$ ci-contre.

Algorithm 1 Baseline Kneser-Ney

Input: tokenized train and test corpus, discount D
Compute and store $\forall\ tokens, bigrams, trigrams \in corpus_{train}$ the values of $c(\omega_1\omega_2\omega)$, $c(\omega_1\omega_2)$, $N_{1+}(\omega_1\omega_2\bullet)$, $c(\omega)$, $N_{1+}(\omega_2\bullet)$, $N_{1+}(\bullet\omega)$, $N_{1+}(\bullet\bullet)$.
for every `<unk/>` in the test corpus
  for every ω in the train corpus
    compute $p_{KN}(\omega|\omega_1\omega_2)$
  store the set of most probable ω for this `<unk/>`
  replace the `<unk/>` by the most probable ω in the test corpus to be able to compute the next missing word if it's less than 2 positions away
return all the sets of ω

3.1.1. Remarques d'ordre pratique

- Afin d'avoir de meilleurs résultats, nous avons ajouté un token `<s>` en début et fin de chaque phrase pour faire apprendre les structures les plus probables aux bords. Avoir le même token en début et en fin de phrase ne dérange pas ici car de toute manière nous balayons la phrase soit de gauche à droite, soit de droite à gauche en connaissant à chaque fois le moment de son début et son moment de fin (les phrases du corpus de test ont déjà cela de fixé).
- Ne voulant pas pouvoir prédire une "fin" ou un "début" de phrase car cela est déjà connu dans le corpus test, nous ne retournons jamais ce token.
- Afin de pouvoir calculer chaque valeur de N_{1+} rapidement, nous avons utilisé une structure d'arbre pour ranger les comptes de trigram et bigram du corpus d'entraînement (voir *Notebook*).
- Nous utilisons une valeur de discount $D = 0.75$ systématiquement.

3.1.2. Résultats

- Le calcul de toutes les fonctions c et N_{1+} se fait en $1min20$, chaque prédiction met $0.38s$ a être calculée pour les textes anglais et $1.26s$ pour les textes finnois.
- Nous avons, avec ce baseline, les résultats suivants:

$accuracy(u30.en) = 0.30$
$accuracy(u30.fi) = 0.17$

L'$accuracy$ étant calculée de la manière suivante:

$$accuracy = \frac{|\ \omega_{true} \in\ (\omega^{(1)}, \omega^{(2)}, \omega^{(3)})\ retournes\ |}{nombre\ de\ predictions\ faites}$$

Il semblerait donc que le finnois soit bien plus difficile à prédire que l'anglais, ce qui semble cohérent vu

la différence de structure de ces langues. Voyons maintenant si nous pouvons améliorer ces résultats en utilisant les méthodes imaginées.

3.2. Kneser-Ney aller-retour

Nous voulons trouver pour chaque $\omega_1^g\omega_2^g$<unk/>$\omega_2^d\omega_1^d$ du corpus de test:

$$\underset{\omega \in vocab_{train}}{\operatorname{argmax}} \Big[p_{KN}(\omega|\omega_1^g\omega_2^g) + p_{KN}(\omega|\omega_1^d\omega_2^d)\Big]$$

Or, nous savons déjà calculé $\underset{\omega \in vocab_{train}}{\operatorname{argmax}} p_{KN}(\omega|\omega_1^g\omega_2^g)$ (partie précédente). Il nous suffit donc de trouver $\underset{\omega \in vocab_{train}}{\operatorname{argmax}} p_{KN}(\omega|\omega_1^d\omega_2^d)$. Pour cela, il suffit d'inverser l'ordre des mots de chaque phrase du corpus d'entraînement, et d'appliquer l'algorithme déjà implémenté (en inversant également l'ordre des mots de chaque phrase du corpus de test), puis de faire la correspondance entre les ω trouvés en parcourant les phrases à l'envers et ceux trouvé en les parcourant à l'endroit. La distribution de probabilité finale étant la somme des deux calculées.

3.2.1. RÉSULTATS

Nous avons, avec ce modèle, les résultats suivants:

$accuracy(u30.en) = 0.411$
$accuracy(u30.fi) = 0.24$

Ainsi, avec cette méthode, nous augmentons de plus de 10% l'$accuracy$ vis à vis du baseline pour l'anglais, et seulement 6% pour le finnois. Notre hypothèse comme quoi de l'information contextuelle est aussi présente à droite des tokens manquants <unk/> est bien vérifiée.

3.3. Intégration des tags dans la prédiction

Nous faisons l'hypothèse que prédire les tags des mots manquants nous ajoutera de l'information et nous permettra d'améliorer notre prédiction en la guidant vers les mots "aux bons tags". Nous cherchons ainsi à maximiser pour $\omega \in vocab_{train}$:

$$\Big[p_{KN}(\omega|\omega_1^g\omega_2^g,\ tag) + p_{KN}(\omega|\omega_1^d\omega_2^d,\ tag)\Big]p(tag)$$

Pour tagger un texte, nous pouvons utiliser un HMM et une base de donnée de texte pré-taggés tel que *Penn treebank* pour l'anglais et *Finish treebank* pour le finnois. Nous avons utilisé l'outil **`treetagger`** qui fait exactement cela pour tagger nos corpus d'entraînement et de test.

3.3.1. REMARQUES D'ORDRE PRATIQUE

- L'outil **`treetagger`** utilise un jeu de 58 tags.
- Nous faisons d'abord l'opération de tagger tous les corpus (cela se fait en dehors de notre programme python, via l'interface graphique de **`treetagger`**), puis nous importons dans notre programme les fichiers générés par le logiciel.
- Pour chaque prédiction ω que nous faisons pour un token manquant, nous avons besoin d'avoir accès à son tag si nous voulons ensuite pondérer sa probabilité par celle de son tag. Pour cela, nous prenons le tag le plus fréquent pour ce token dans le corpus d'entraînement.
- Nous voulons avoir accès à la probabilité du tag du token manquant $p(tag)$. Pour cela, nous utilisons l'approximation $p(tag_\omega) \simeq p_{KN}(tag_\omega|tag_{\omega_1}tag_{\omega_2})$ en se disant que notre prédiction, même si imparfaite, est plus "sûre" que celle du token ω, et qu'elle est donc susceptible d'apporter de l'information à notre prédiction finale. Ainsi, nous devons encore utiliser l'*algo.1*, mais cette fois sur les corpus de tags.

3.3.2. RÉSULTATS

En implémentant la méthode, nous nous rendons compte qu'il y a différente manières de l'appliquer: faut-il d'abord utiliser (**méthode 1**) notre *Kneser-Ney "sens de lecture", puis Kneser-Ney "inverse", sommer les probabilités puis multiplier par celles Kneser-Ney "tag"*, ou plutôt (**méthode 2**) faire cela dans l'ordre *Kneser-Ney "sens de lecture" multiplié par Kneser-Ney "tag sens de lecture", puis Kneser-Ney "inverse" multiplié par Kneser-Ney "tag sens inverse",et enfin sommer* ?
Les deux méthodes ne sont pas identiques car à chaque étape, par soucis de mémoire et de temps de calcul, nous ne retournons pas pour chaque <unk/> toute la distribution sur tous les tokens possibles, mais seulement les probabilités pour un petit sous ensemble de tokens les plus probables. De même pour la distribution sur les tags.

Méthode 2: $accuracy(u30.en) = 0.35$

Face à ce résultat, en baisse par rapport aux précédents (et par manque de temps), nous avons directement réfléchi à une manière d'améliorer notre prédiction sans regarder tout de suite ce que donnait la **méthode 1**. En effet, en s'intéressant aux tags proposés par **`treetagger`**, on s'est rendu compte qu'il était encore possible de réduire la dimension de l'espace dans lequel on voulait faire nos prédictions car nous pouvions regrouper plusieurs tags ensembles (ex: $VBN,\ VBZ,\ VBP \rightarrow VERB$). En utilisant la correspondance `tag` $\rightarrow$ `pos` de **spacy** nous avons pu établir un dictionnaire de traduction de tags et ainsi réduire notre espace à seulement **15 tags**.
Dans ce nouvel espace, nous avons les résultats suivants:

Méthode 2: $accuracy(u30.en) = 0.40$
Méthode 1: $accuracy(u30.en) = 0.42$

Ainsi, la **méthode 1** semble meilleure que la seconde. De plus, en réduisant encore la taille de notre espace de tag, nous avons pu réussir à établir une probabilité $p(tag)$ suffisamment bonne pour bien **ajouter de l'information** par rapport à nos meilleurs prédictions précédentes.

3.3.3. Expériences

Nous avons également essayé de voir quelles étaient les performances d'un simple MLE pour l'estimation de tags (voir *Notebook*). En effet, avec un espace de seulement 15 tags, nous nous demandions si ce modèle plus simple que le Kneser-Ney n'arriverait pas à avoir des performances comparables où meilleures. Nous avons trouvé que le modèle Kneser-Ney restait notre meilleur modèle pour cela.

4. Discussions autour de la méthode

Pour arriver à nos résultats, nous avons fait un ensemble d'hypothèses. Pour les vérifier, nous nous sommes principalement tourné vers le corpus *u30.en* par soucis de rapidité de la prédiction vis à vis du finnois (et par manque de temps vis à vis des autres corpus anglais). Il serait donc intéressant d'étudier les performances de notre méthode la plus aboutie sur ces autres corpus.

Nous avons également seulement utilisé la métrique $accuracy$ comme jauge à nos modèles. En effet, notre volonté d'utiliser le lissage Kneser-Ney comme méthode nous a amené à pouvoir prédire une distribution de probabilité sur tous les tokens du corpus d'entraînement pour tous les mots manquants rencontrés. Ainsi, parler de "précision" ou de "recall" n'a pas de sens ici. Cependant, nous pouvons toujours nous demander si nous n'aurions pas pu instaurer un "seuil minimal" pour les probabilités prédites, en-deçà duquel nous jugerions que nos prédictions ne sont pas pertinentes et donc nous retournerions rien. Pour répondre à cette question, nous pouvons regarder les valeurs des probabilités prédites pour les tokens ω_{true} que nous avons réussi à retourner dans les triplets $(\omega^{(1)}, \omega^{(2)}, \omega^{(3)})$.

Ainsi, la *Fig.2* nous montre que nous pourrions mettre un seuil aux alentours de 10^{-3} en deçà duquel il ne sert à rien de retourner les mots prédits si nous cherchons à prédire le bon mot.

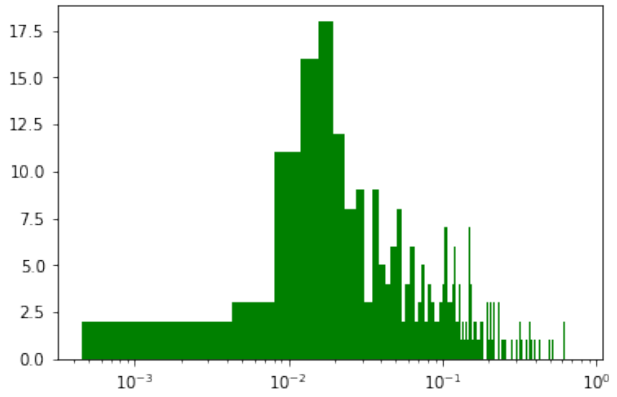

Figure 2. Distributions des valeurs de probas prédites pour les tokens ω_{true} anglais pour notre meilleur méthode sur *u30.en*, échelle logarithmique en abscisse

5. Bibliography

1. http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/
2. https://spacy.io/usage/linguistic-features
3. https://courses.washington.edu/hypertxt/csar-v02/penntable.html
4. Labbé Dominique. Une mesure de la richesse du vocabulaire : l'indice de Gini. In: Mots, n 15, octobre 1987. Comment nommer?Barbares - Berbères. Islam. Arbre de la liberté. Economia. Les juifs de Cagayous. Sig(is)mund. pp. 171-185.
5. https://www.nltk.org/_modules/nltk/probability.html
6. https://west.uni-koblenz.de/sites/default/files/BachelorArbeit_MartinKoerner.pdf