Algorithmes du traitement du langage et de la parole (NLP): TD1

Guillaume Petit
g.petit98@gmail.com

February 17, 2020

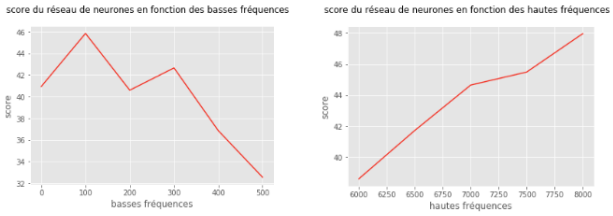
Abstract

J'ai, dans un premier temps, modifié le code pour avoir une répartition équitable des labels au sein de chaque ensemble. J'ai donc augmenté mon ensemble de training à 600 éléments par classe, m'octroyant un bon compromis entre le temps de création de l'ensemble et des bons résultats avec les modèles. J'ai accepté tous les éléments dans l'ensemble de validation et de test.

Pour tester l'impact des paramètres du MFCC et du Melog-FilterBanks, j'utiliserai le réseau de neurones pour voir les scores obtenus. Pour finir, notre modèle a pour but de classifier des mots anglais, donc nous sommes intéressés par les caractéristiques vocales que le MFCC reconnaît mieux que le Mel-filterBanks en théorie.

1 Classification of segmented voice commands

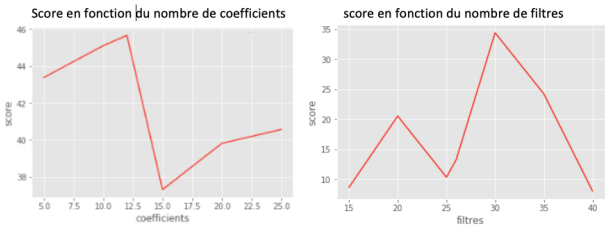
Question 1.1)



Score obtenu en faisant varier les fréquences

Pour trouver les bonnes fréquences, je fais varier les basses fréquences puis les hautes fréquences. Ensuite je fais varier les deux fréquences dans un domaine où j'ai obtenu les meilleurs scores. Finalement, j'obtiens pour les basses fréquences 100 Hz et pour les hautes fréquences 8000 Hz. Sachant que le framerate de nos signaux audio est 16000 et que les hautes fréquences capturent les détails à environ la moitié de cette valeur, cela correspond à la théorie.

Question 1.2) Selon [4], le nombre de filtres varie entre 20 et 40. Je décide donc de vérifier mes scores dans cet ensemble. J'obtiens mon meilleur score (35%) pour 30 filtres. Une fois le nombre de filtres fixé, je cherche le nombre de coefficients cepstraux qui améliore mon score, qui est de 12, comme indiqué dans le cours.



Question 1.3) Notre meilleur résultat est False-False. Les coefficients différentiels ont pour but d'améliorer les résultats mais ont pour effet de doubler ou tripler les dimensions des vecteurs acoustiques. Face au peu d'échantillons que nous avons, nous faisons face au problème de malédiction de la dimensionnalité. Cela peut donc mener à détériorer nos résultats [5]. Cependant, pour une raison inconnue, quand je crée mes ensembles avec la combinaison False-False, je n'obtiens que des vecteurs nuls dans mon training. Je décide donc d'utiliser la seconde meilleure formule 'True-False'.

deltas — deltas deltas	True	False
True	39.34 %	45.66 %
False	44.45 %	48.44 %

Question 1.4)

•**normalisation**: j'ai crée ma fonction normalisation. Les meilleurs scores sont obtenus quand je normalise l'ensemble de training et que je normalise ensuite les deux autres ensembles avec la moyenne et l'écart-type de l'ensemble de training. Toutefois, d'autres normalisations sont possibles, comme normaliser sur l'ensemble des données.

•**augmentation** : j'ai extrait les bruits et créé une fonction pour les ajouter à des sons du training. Cependant, je me suis rendu compte que certains bruits couvraient totalement les voix. Je décide d'instaurer un coefficient pour modéliser le bruit par rapport à l'onde de base. Or certaines ondes de base sont déjà bruitées en partie. Pour éviter d'induire en erreur mon classifieur, je retire les bruits les plus forts. Cela n'a malheureusement pas résolu le problème. Je décide de normaliser le nouveau training. Et cela donne des résultats plutôt convaincants, comme si la normalisation permettait d'effacer en partie le bruit.

MLP	score (en %)	temps (en sec)
classique	45 %	170
normalisées	62 %	17
bruits	4 %	123
normalisées + bruits	60 %	22

Il est évident que la normalisation à un effet positif sur le score. Si j'avais réussi à coefficienter le bruit, le meilleur score aurait été à l'ensemble bruité normalisé.

Question 1.5)

•**régression logistique**: il m'a été impossible de modifier la régression logistique à l'aide de GridSearch. Après 1h30 à runner, la régression logistique ne produisait aucun résultat.

•**MLP**: pour le MLP, j'ai essayé de faire varier l'évolution du taux d'apprentissage et j'ai aussi regardé l'impact du coefficient de normalisation α dans la formule de la perte :

$$\mathcal{L}(x_i, y_i) + \alpha||w||_2$$

•**mon modèle** : j'implémente mon propre réseau de neurones à l'aide de keras. Après avoir étudié différentes architectures et optimiseurs, il s'avère que j'obtiens de bien meilleurs résultats à l'aide d'un optimiseur SGD dont le taux d'apprentissage décroît. Je décide de mettre un arrêt anticipé qui me permet de réduire le temps d'entraînement car à partir d'un moment, le gain de score n'est plus significatif. En fonction des données sur lesquelles j'entraîne mon modèle, l'architecture varie. Je décide de mettre une batch-normalisation à mon modèle entraîné sur des données classiques. Et comme on peut le constater sur le tableau ci-dessous, cela améliore nettement les résultats, bien meilleurs que celui où les données sont normalisées.

Pour les trois modèles, je décide de les entraîner sur trois types de données (classiques, normalisées, normalisées + bruitées). On obtient les résultats suivants :

modèles	classique	normalisées	normalisées + bruit
MLP	62,62 % 188 sec	59.57 % 202 sec	60.53% 282 sec
RL	- % - sec	- % - sec	-% - sec
mon modèle	79 % 223 sec	76,8 % 141 sec	76,7% 132 sec

Question 1.6)

Modèles	classiques	normalisées	normalisées + bruit
score MLP	67.2 %	47.2 %	32.1 %
score mon modèle	77.2 %	75.12 %	74 %

Selon le tableau ci-dessus, le meilleur modèle est celui que j'ai créé sur des données normales. La couche de batch-normalisation est essentielle. Elle accélère le temps d'entraînement et permet de mieux performer que le modèle entraîné sur les données normalisées. A l'aide de la figure 1 (en fin de rapport), on remarque que la classe 'go' est la plus difficile à prédire avec un taux d'erreur de 41,3 %, beaucoup confondue avec la classe 'no'. On remarque que les classes qui se confondent le plus sont des classes parfois très proches ('on' et 'one'), parfois très éloignées ('right' et 'three').

2 Classification of segmented voice commands

Dans cette partie, mon meilleur modèle sera mon réseau de neurones. Il est possible d'utiliser celui entraîné sur les

données normalisées. C'est pour cela que j'ai modifié le code qui permet de générer les postérieures pour pouvoir normaliser les features en option. Après comparaison, je n'ai pas vu de nette amélioration. J'ai donc décidé de garder dans mon code mon modèle classique.

2.1 prédiction de phrases

Question 2.1) La probabilité à priori de chaque mot est égale à une constante car nous avons entraîné notre "discriminator" sur un jeu de données équilibré par classes grâce à la commande "train-labels.count(label) $\leq$ nb-exper-class".
Question 2.2) WER ne peut être inférieur à 0 puisqu'il est le résultat d'opérations de nombres positifs. Il peut être supérieur à 100. Cela sous-entend qu'il a ajouté beaucoup de mots, en plus de s'être trompé. C'est le signe d'une mauvaise prédiction

2.2 entrées indépendantes, greedy search

Question 2.3) L'algorithme greedy a pour sortie : "go marvin one up stop" pour la phrase de référence "go marvin one right stop". Il y a donc 1 substitution. Alors en appliquant la formule, on a $\frac{1+0+0}{5} = 20\%$.

2.3 Modèle de langage

Notre formule de score devient $P(W|X) \approx P(X|W)P(W) \propto P_{one-word}(W|X)P(W)$. Le score sera mis à l'échelle logarithmique pour éviter des problèmes de petites valeurs de la part de l'ordinateur.

Question 2.4) Soit W une séquence de mots, ie $W = (w_1, ..., w_K)$. Pour le modèle Bigram, nous avons :

$$P(W) = \prod_{k=1}^{K} P(w_k|w_{k-1}, ...w_1) = \prod_{k=1}^{K} P(w_k|w_{k-1})$$

.
Question 2.5) **•début de phrase:** En suivant les explications de [1], j'ai décidé de faire commencer mes phrases par le symbole $' < s >'$ pour nous permettre d'avoir le contexte du bigramme du premier mot, ie $P(w_1|' < s >')$.
•vocabulaire: Des 'nan' apparaissent en premier lieu. Il y a deux raisons à cela. D'un côté, les mots 'bed' et 'dog' ne sont pas présents dans la phase d'entraînement. Cependant, $\forall\, w \in$ vocabulaire , $P(dog|w) = P(w|dog) = 0$. De même pour 'bed'.D'un autre côté, certains mots dans l'entraînement ne sont qu'en fin de phrase. On pourrait rajouter une colonne pour un symbole fin de phrase, mais cela me procurerait des problèmes de dimensions. A la place, tous les 'nan' sont mis à 0. Ce n'est pas problématique quand on est sur l'ensemble d'entraînement mais cela peut causer des problèmes si jamais on rencontre des situations différentes dans le test.
•formule : Les lignes de la matrice représentent w_{k-1} et les colonnes w_k. J'ai d'abord compté les occurrences des bigrammes, notées $C(w_{k-1}w_k)$ puis j'ai ensuite divisé par lignes, ie $C(w_{k-1})$, pour obtenir une probabilité.

Question 2.6) Les N-grammes permettent de mieux en mieux de modéliser le corpus de training à mesure que nous augmentons la valeur de N. Les bigrammes ont une cohérence mot-à-mot tandis que les 3/4-grammes commencent à faire ressortir une certaine cohérence dans la phrase. Cependant, la complexité augmente de manière exponentielle avec N.

2.4 Beam-Search

Question 2.7) La complexité de mon algorithme est $O(BLMlog(M))$ et sa complexité de mémoire vaut au plus $O(BM)$ avec B le beam-size, L la longueur de la phrase et M la taille du vocabulaire.

2.5 Viterbi

Question 2.8) En reprenant les notations de [1], avec $v_t(j)$ qui représente la probabilité d'être à l'état j à l'étape t, on a la relation

$$v_t(j) = \max_{1 \leq i \leq M} v_{t-1}(i) a_{ij} b_j(o_t)$$

avec a_{ij} les éléments de la matrice de transition qui permet de passer de l'état précédent i à l'état j et $b_j(o_t)$ la likelihood de l'observation o_t sachant l'état actuel j.

La complexité de Viterbi est $O(LM^2)$ et la complexité de la mémoire est $O(LM)$.

2.6 Résultats des algorithmes

Question 2.9) Viterbi est un algorithme exact. Il explore toutes les possibilités, tandis que beam-search est une approximation gardant les K-meilleures phrases à chaque étape. Cependant, mon beam-search obtient de meilleurs résultats (voir tableau ci-dessous).

	Greedy	BS	Viterbi
Train WER (en %)	27	7.9	9.4
Test WER (en %)	26	6.8	9.2
temps (en sec)	47	46.4	47

En regardant la première matrice de la figure 2 (matrice de transition), on remarque que, pour chaque mot, la majorité de la masse de probabilité de générer le prochain est concentré dans quatre ou cinq classes. Cela réduit les approximations du beam-search de taille 5 et explique qu'il performe aussi bien que Viterbi.

Question 2.10) Je compte les erreurs commises par mes modèles et les résultats sont mis sous forme de matrice, où les lignes représentent les mots prédits et les colonnes les vrais mots. On se reporte à la figure 2 en bas de page pour les résultats.

•**Beam-search:** "tree" est confondu avec "cat" 8 fois, "two" avec "zero" 7 fois et "no" avec "wow" 9 fois.

•**Viterbi:** "tree" est confondu avec "cat" 23 fois, "two" avec "zero" 7 fois et "no" est confondu avec "wow" 9 fois.

En regardant la matrice de transition (1ère matrice de la figure 2), on remarque que ces couples de mots ont généralement les mêmes prédécesseurs, ce qui explique les erreurs commises.

Question 2.11) Pour faire face aux mots peu vus dans les séquences d'entraînement présents dans le test, je décide d'implémenter pour commencer 1-add Laplace smoothing. Cela consiste à rajouter des 1 à chaque compteur pour éviter les 0, ie

$$p(w_i|w_j) = \frac{C(w_j w_i)}{C(w_j)} \text{ devient } p(w_i|w_j) = \frac{C(w_j w_i)+1}{C(w_j)+M}$$

Or, plus notre matrice est sparse, plus cela va impacter les probabilités car une trop grande masse de probabilités est déplacée vers tous les zéros (cf [2], [1]). Je décide donc d'implémenter une méthode plus perfectionnée; Kneser-Ney. Notre probabilité devient :

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)} |\{w; C(w_{i-1}w) > 0\}|$$

$$P_c(w) = \frac{|\{v; C(vw) > 0\}|}{\sum_k |\{v; C(vk) > 0\}|}$$

$$P_{KN}(w_i|w_j) = \frac{\max(C(w_j w_i) - d, 0)}{\sum_v c(w_j v)} + \lambda(w_j) P_c(w_i)$$

On obtient les résultats suivants :

algorithme	1-add laplace	Kneser-Ney
Beam-search WER (en %)	Train : 7.5 Test : 8.5 temps : 47,8 sec	Train : 7 Test : 7 temps : 47,6 sec
Viterbi WER (en %)	Train : 9.1 Test : 8.6 temps : 240 sec	Train : 9 Test : 9 499 sec

On remarque que cela améliore légèrement le wer sur training pour les deux algorithmes et légèrement le wer sur le test pour Viterbi. Cependant, pour une raison inconnue, cela a considérablement augmenté le temps de calcul pour Viterbi.

Pour faire face aux mots inconnus $<$ UNK $>$, on le rajoute à la matrice de transition avec un compteur à 0 et donc sa probabilité sera une distribution uniforme $\frac{\lambda(\epsilon)}{M}$ avec ϵ le string vide.

Question 2.12) Les modèles acoustiques, de prononciation et de langage, ont tous été formés séparément, chacun avec un objectif différent. Des travaux récents dans ce domaine tentent de remédier à ce problème de formation disjointe en concevant des modèles qui sont formés de bout en bout - de la parole directement aux transcriptions. Deux approches principales sont la CTC et les modèles de séquence à séquence avec attention. D'autres modèles, comme [3], propose d'aller plus loin. Contrairement aux approches précédentes, le LAS ne fait pas de suppositions d'indépendance dans les labels et il ne s'appuie pas sur les HMM. Il y a aussi, bien plus connu, les réseaux de neurones récurrents.

3 References

[1] "Speech and Language Processing" de Daniel Jurafsky James H. Martin.

[2] ”NLP Lunch Tutorial: Smoothing” de Bill MacCartney.
[3] ”Listen, Attend and Spell” de William Chan, Navdeep
[4] ”Mel Frequency Cepstral Coefficient (MFCC) tutorial”
[5] ”Sélection de paramètres acoustiques pertinents pour la reconnaissance de la parole” de Abdenour Hacine-Gharbi

Matrice de confusion

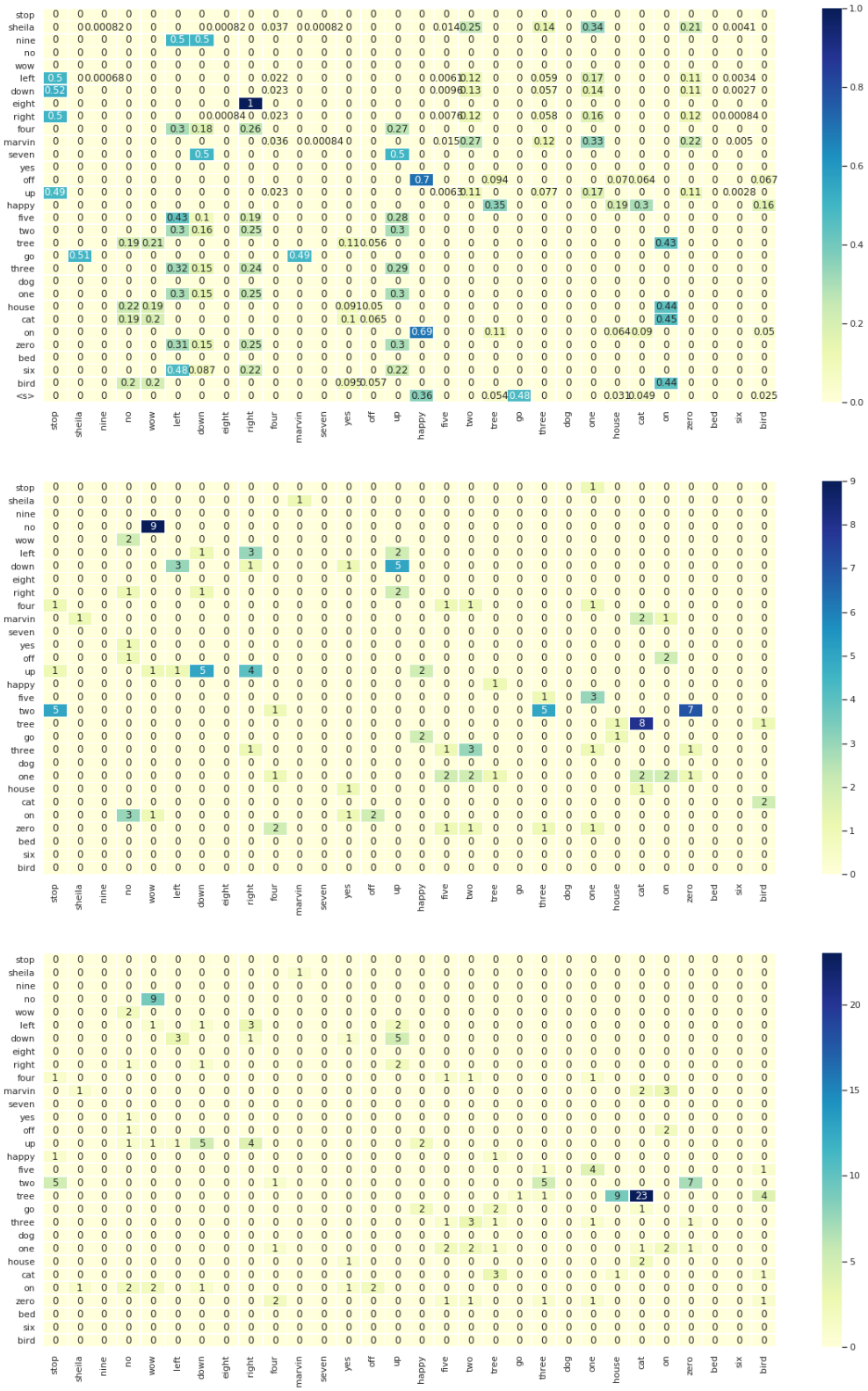


Figure 1: Matrice 1 : transition - Matrice 2: erreur beam-search - Matrice 3: erreur Viterbi