Université Pierre-Mendès-France
Sciences sociales & humaines

Technologies mobiles pour la reconnaissance vocale des langues africaines

Rapport de projet professionnel
UE de Génie logiciel

Nom : GAUTHIER
Prénom : Elodie

UFR SHS - IMSS

Master 2 Professionnel

Spécialité : Ingénierie de la Cognition, de la Création et des Apprentissages

Parcours : Double Compétence en Informatique et Sciences sociales

Sous la direction de Laurent Besacier

Année universitaire 2013-2014

Résumé

Ce rapport décrit le travail effectué durant ces deux mois de projet professionnel. La majeur partie du temps a été consacrée à la création d'un système de reconnaissance automatique de la parole, pour la langue hausa. Nous décrivons les différents processus qui nous ont menés au système final. A partir de données audio et de leurs transcriptions orthographiques collectées au Cameroun par Schlippe et al. et d'un dictionnaire acoustique réalisé par GlobalPhone (mise en forme réadaptée pour nos expérimentations), nous avons créé notre propre modèle de langage.
Grâce à la boîte à outils Kaldi, nous avons entrainé notre système sur un corpus d'apprentissage. Puis, nous avons lancé le décodage sur notre corpus de développement. Nous obtenons un taux d'erreur mots de 16,19% sur le modèle acoustique à base de SGMMs. Ensuite, nous avons récupéré le modèle de langage de GlobalPhone, nous lui avons enlevé la casse et nous avons lancé le décodage. Nous obtenons un WER de 9,93%. Enfin, nous avons créé un modèle de langage à partir de l'interpolation des deux précédents. Nous avons obtenus un WER de 10,03%. Ces résultats sont très encourageants pour la suite.
Plus tard, nous aimerions installer le décodeur de parole de Kaldi sur des téléphones mobiles Android.

Mots-clés : reconnaissance vocale, Kaldi, Android, appareils mobiles

Abstract

This report describes the work done during these two months of professional project. The major part of the time was devoted to the creation of an automatic speech recognition for the Hausa language. We describe the differents processes that led us to the fnal system. From audio and orthographic transcriptions collected in Cameroon by Schlippe et al. and acoustic dictionary created by GlobalPhone (shaping rehabilitated for our experiments), we created our own language model.
With the Kaldi toolkit, we trained our system on a training set. Then we started decoding on our development set. The system achieves a word error rate of 16.19% on the acoustic model based on SGMMs. Then we got the language model of GlobalPhone, we removed the case and we started decoding. We get a *WER* of 9.93%. Finally, we create a language model from the interpolation of the previous two. We obtained a WER of 10.03%. These results are very encouraging for the future.
Next, we would like to install the speech decoder Kaldi on Android smarphones.

Keywords : automatique speech recognition, Kaldi, Android, mobile devices

Table des matières

Introduction ... **6**
Chapitre 1 – Contexte du projet ... **7**
1.1. Le Laboratoire d'Informatique de Grenoble ... 7
1.2. L'équipe GETALP ... 7
1.2.1 Le projet ALFFA ... 8
1.3 Le Hausa ... 8
Chapitre 2 – La reconnaissance automatique de la parole ... **10**
2.1. Principes généraux ... 10
2.1.1 L'extraction des paramètres acoustiques ... 10
2.1.2 L'étape de modélisation ... 11
2.1.2.1 Le modèle acoustique ... 11
2.1.2.2 Le modèle de langage ... 12
2.2. La boîte à outils Kaldi ... 12
Chapitre 3 – Le corpus de travail ... **13**
Chapitre 4 – Les expérimentations ... **14**
4.1 Mise en forme des données brutes ... 14
4.2 Initialisation des données ... 17
4.3 Les modèles de langage ... 19
4.3.1 Définition ... 19
4.3.2 Préparation de notre modèle de langage ... 20
4.3.3 Les autres modèles de langages ... 21
4.3.3.1 Le modèle de langage GlobalPhone ... 21
4.3.3.2 Le nouveau modèle interpolé ... 21
4.4 Les modèles acoustiques ... 22
4.4.1 Définition ... 22
4.4.2 Préparation ... 22
Chapitre 5 – Bilan technique ... **27**
5.1 Résultats ... 27
5.1.1 La perplexité ... 27
5.1.2 Le taux d'erreurs mots ... 29
5.2 Difficultés rencontrées ... 31
5.3 Critiques ... 31
5.4 Améliorations et perspectives ... 31
Chapitre 6 – Bilan personnel ... **33**
Bibliographie ... **34**
Webographie ... **34**
Table des figures ... **35**
Annexes ... **36**

Introduction

D'après l'UNESCO [Unesco, 2010], « le nombre de langues parlées en Afrique va de 1 000 à 2 500, selon les estimations et les définitions. Les États monolingues n'existent pas et les langues traversent les frontières sous forme de configurations et de combinaisons différentes. Le nombre de langues varie entre deux et trois au Burundi et au Rwanda, à plus de 400 au Nigeria. ». C'est dans ce contexte que se situe le développement de technologies vocales sur mobiles pour les langues africaines : les systèmes de reconnaissance vocale et de synthèse vocale peuvent aider les personnes analphabètes à communiquer mais peuvent aussi être utilisées pour des applications ludiques. De plus, l'adjonction d'un système de traduction automatique peut faciliter les échanges commerciaux.

Ce projet professionnel s'inscrit dans le cadre du projet ANR Blanc ALFFA, qui a pour but de développer des technologies vocales pour mobiles en Afrique. Durant ce projet, nous nous sommes intéressé à une grande langue véhiculaire d'Afrique de l'Ouest : le hausa. Pour créer notre système de reconnaissance, nous avons utilisé la boîte à outils de reconnaissance de la parole Kaldi.

Ce rapport est découpé en 6 chapitres. Le premier chapitre présente le contexte du projet ; le deuxième explique les principes de fonctionnement d'un système de reconnaissance automatique de la parole ; le troisième décrit le corpus de travail que nous avons exploité ; le quatrième expose les scripts créés et les expérimentations menées ; le cinquième présente les résultats que nous avons obtenus et, enfin, le sixième est un bilan personnel sur le travail mené au cours de ces deux mois de projet professionnel. Le planning de travail est visible en annexe 10.

Chapitre 1 – Contexte du projet

Ce projet professionnel a été réalisé au sein du Laboratoire d'Informatique de Grenoble et plus précisément au sein de l'équipe GETALP.

1.1. Le Laboratoire d'Informatique de Grenoble

Le Laboratoire d'Informatique de Grenoble (LIG) se situe sur le campus universitaire de Saint-Martin d'Hères. Il est axé sur cinq thématiques de recherche :

- Génie des Logiciels et des Systèmes d'Information ;
- Méthodes Formelles, Modèles et Langages ;
- Systèmes Interactifs et Cognitifs ;
- Systèmes Répartis, Calcul Parallèle et Réseaux ;
- Traitement de Données et de Connaissances à Grande Echelle.

Les défis du LIG – communs à ces cinq axes de recherche – sont : la diversité et la dynamicité des données, des services, des dispositifs d'interaction et des contextes d'usage imposent l'évolution des systèmes et des logiciels pour en garantir des propriétés essentielles telles que leur fiabilité, performance, autonomie et adaptabilité.

1.2. L'équipe GETALP

Le GETALP[1] (Groupe d'Étude en Traduction Automatique/Traitement Automatisé des Langues et de la Parole) est une équipe parmi les 22 présentes du Laboratoire d'informatique de Grenoble. Elle rassemble des informaticiens, linguistes, phonéticiens, traducteurs et traiteurs de signaux.

Les travaux de recherche du GETALP sont pluridisciplinaires et participent à la création d'une informatique « ubilingue ». C'est pourquoi les domaines de recherche sont variés et font partie aussi bien du domaine de l'informatique, que des sciences du langage.

---

[1] http://getalp.imag.fr/

Parmi les 9 axes de recherche de l'équipe, mon travail s'insère dans celui de la « Traduction et transcription automatiques de la parole ».

1.2.1 Le projet ALFFA

Le projet ALFFA (*African Languages in the Field: speech Fundamentals and Automation*) est un projet ANR blanc qui a débuté en octobre 2013 et qui est financé sur 4 ans. Ce projet vise à proposer, à terme, des micro services vocaux pour les téléphones mobiles en Afrique (par exemple, un service de téléphone pour consulter le prix des matières premières ou pour fournir des rapports voix des systèmes d'information) mais également des outils de traitement automatique de la parole afin d'aider les linguistes de terrain à décrire les langues et à les analyser.
4 organismes participent à ce projet : le Laboratoire d'Informatique de Grenoble (i.e. : LIG), le Laboratoire d'Informatique d'Avignon (i.e. : LIA), le laboratoire Dynamique du langage à Lyon (i.e. : DDL) et Voxygen une jeune entreprise bretonne spécialisée en synthèse vocale. C'est donc un projet interdisciplinaire puisqu'il réunit non seulement des informaticiens et spécialistes en traitement du signal mais également des linguistes de terrain et des phonéticiens.

1.3 Le Hausa

Le hausa fait partie de la famille des langues chamito-sémitiques. Plus précisément, le hausa est la langue la plus parlée des langues tchadiques [Vycichl, 1990]. Il est la langue officielle du nord Nigeria (30 millions de locuteurs environ) et est une langue nationale du Niger (9 millions de locuteurs environ) mais est aussi parlé au Ghana, au Benin, au Cameroun, au Togo, au Tchad et au Burkina Faso [Koslow, 1995]. Le hausa est considéré comme une langue véhiculaire d'Afrique de l'Ouest et d'Afrique centrale : il est parlé dans de multiples grandes villes commerciales telles que Dakar, Abidjan, Lomé, Ouagadougou ou encore Bamako.

Environ un quart des mots du hausa proviennent de l'arabe mais la langue a aussi été influencée par le français. Le hausa peut être écrit avec l'orthographe arabe depuis le début du 17è siècle : ce système d'écriture est appelé *'ajami.* Cependant, l'orthographe officielle est basée sur l'alphabet latin et est appelé *boko.* Ce système d'écriture a été imposé par les anglais lors de la colonisation, dans les années 30.

Le *boko* est formé de 22 caractères provenant de l'alphabet anglais (i.e. : A/a, B/b, C/c, D/d, E/e, F/f, G/g, H/h, I/i, J/j, K/k, L/l, M/m, N/n, O/o, R/r, S/s, T/t, U/u, W/w, Y/y, Z/z) plus ɓ, ɗ, ƙ (respectivement transcrits par B/b, D/d, et K/k dans les journaux) et ' (qui représente l'arrêt glottal). De plus, il existe 3 tons différents en Hausa. Par exemple, chacune des 5 voyelles /a/, /e/, /i/, /o/ et /u/ peuvent avoir un ton bas, haut ou descendant. Ces tons sont décrits par des diacritiques (respectivement, par l'accent grave, l'accent aigu et l'accent circonflexe) dans les dictionnaires. Mais encore, les voyelles peuvent varier en durée (i.e. : voyelle longue ou courte) et cette variation à une incidence sur la signification du mot.

Chapitre 2 – La reconnaissance automatique de la parole

2.1. Principes généraux

La reconnaissance automatique de la parole (i.e. : « RAP ») consiste à transcrire un signal acoustique en une suite de mots écrits. Par la suite, le système est capable d'analyser et d'interpréter ces séquences afin de prendre une décision face à ce qu'il a reconnu.

La construction d'un système de reconnaissance automatique de la parole peut être décomposée en deux parties: l'extraction de paramètres acoustiques – les MFCCs (i.e. : *Mel-Frequency Cepstrum Coefficients* ou coefficients Mel cepstraux) – et l'étape de modélisation. Cette dernière comprend deux étapes : le modèle acoustique et le modèle de langage, tous deux créés à partir de HMMs (i.e. : *Hidden Markov Model* ou chaînes de Markov cachées).

Ci-dessous, nous pouvons voir un schéma du processus de reconnaissance :

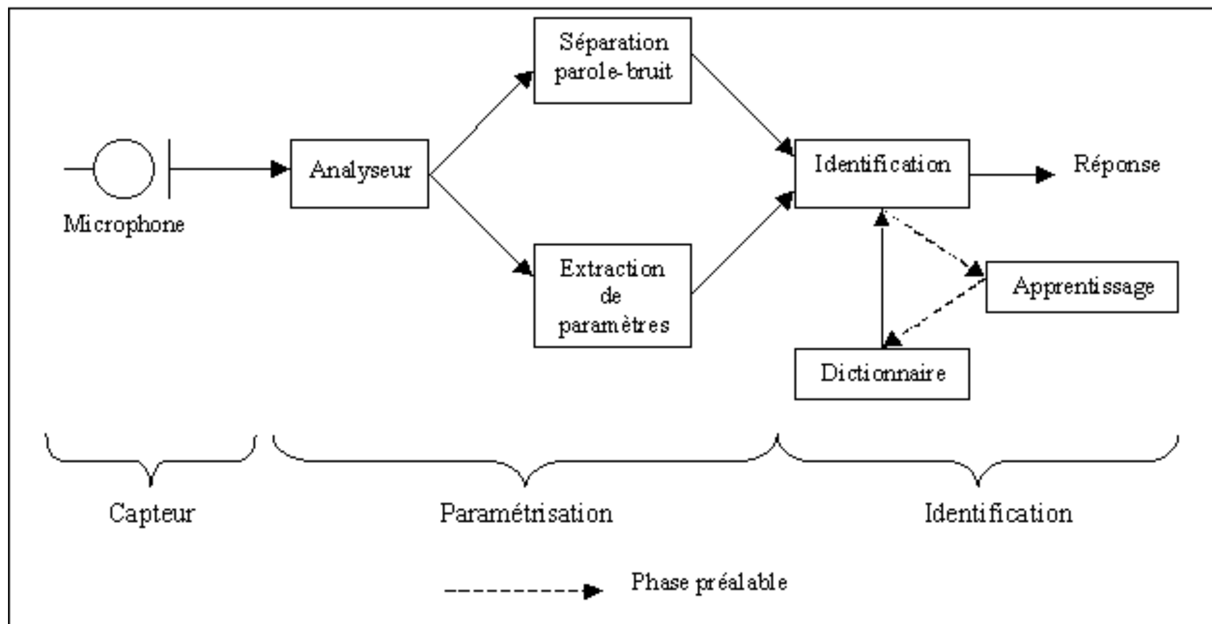


Figure 1 Architecture d'un système de reconnaissance de la parole

2.1.1 L'extraction des paramètres acoustiques

La première étape d'un système de reconnaissance de la parole est d'extraire des caractéristiques qui permettent de discerner les composants du signal audio qui sont

pertinents pour l'identification du contenu linguistique, en rejetant les autres informations contenues dans ce signal (i.e. : le bruit ambiant, les émotions, etc.).
Les sons émis par un être humain sont représentés par la forme du tractus vocal, incluant la langue, les dents, les lèvres, etc. Par conséquent, en déterminant précisément cette forme, il est possible d'identifier le phonème produit.
Les MFCCs permettent de représenter avec justesse l'enveloppe du spectre de puissance à court-terme. Ce sont ces coefficients que nous allons donc extraire, puisque la forme du conduit vocal se manifeste dans cette enveloppe. Ainsi, le système de reconnaissance ne se sert pas directement du signal sonore qu'il reçoit en entrée : il exploite la représentation paramétrique de ce signal, c'est-à-dire que le signal acoustique est découpé en vecteurs de paramètres acoustiques (i.e. : les coefficients Mel cepstraux).
Cette étape d'extraction permet de représenter digitalement des signaux d'entrée et de faciliter le traitement des étapes d'apprentissage et de décodage. Plus le nombre de paramètres est important, plus l'étape d'apprentissage sera performante.

2.1.2 L'étape de modélisation

2.1.2.1 Le modèle acoustique

Le modèle acoustique, représenté par des chaînes de Markov cachées, provient d'un entraînement réalisé à partir d'enregistrements de corpus oraux.

Selon les caractéristiques du locuteur – accent, âge, condition physique, sexe, etc. –, la prononciation des phonèmes est différente. Afin de modéliser ce phénomène, ce sont en général des HMMs à 3 états qui sont utilisés. À chaque état correspond un modèle de mélange gaussien (i.e. : GMM) qui permet, lors de l'étape de décodage, d'émettre une hypothèse sur la réalisation du phonème. Dans le cas des modèles acoustiques en contexte, chaque phonème possède un modèle par contexte existant.

Après l'étape d'apprentissage, le système décode acoustiquement de nouveaux enregistrements : le modèle acoustique attribue, sur les nouvelles données, des scores de vraisemblance « donnée apprise – donnée inconnue » s'appuyant sur les modèles HMMs appris. Les scores permettent d'extraire des mots grâce à un dictionnaire de prononciation. Ce dictionnaire contient les mots présents dans le corpus oral utilisé (ou

plus) suivis de leur prononciation phonétique ainsi que toutes les réalisations acoustiques possibles d'un phonème en fonction du contexte (allophones).

2.1.2.2 Le modèle de langage

Les suites de mots obtenues lors du décodage sont analysées avec un modèle de langage. Ce modèle est réalisé à partir des transcriptions du corpus oral. Il permet d'estimer la probabilité qu'une séquence de mots existe dans la réalité et dépend directement de la langue à reconnaître ainsi que de la grammaire de cette langue. Afin d'être généré, le modèle de langage s'appuie sur le dictionnaire de prononciation précédemment cité.

2.2. La boîte à outils Kaldi

Kaldi est une boîte à outils open source pour la reconnaissance de la parole, qui permet de construire un système de reconnaissance. Il est écrit en C++, sous licence Apache v2.0. Kaldi se veut simple d'utilisation, le plus générique et le plus flexible possible pour que l'utilisateur puisse personnaliser son code comme il le désire.
Ci-dessous, nous présentons un schéma simplifié de la structure de Kaldi. Les modules de la bibliothèque de Kaldi dépendent de deux bibliothèques externes : les bibliothèques d'algèbre linéaire (i.e. : BLAS/LAPACK) et la bibliothèque qui permet d'intégrer des transducteurs à états finis (i.e. : OpenFST). La classe « décodable » fait le pont entre ces deux librairies externes. Enfin, les modules plus bas dépendent d'un ou plusieurs modules du dessus.

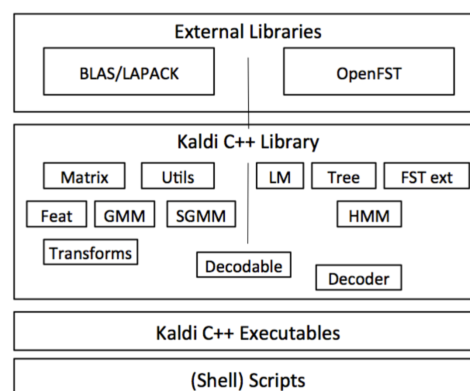


Figure 2 Schéma simplifié de la structure de Kaldi

Chapitre 3 – Le corpus de travail

Concernant les données audio, nous avons exploité celles enregistrées par Tim Schlippe [Schlippe, 2012]. Ces données ont initialement été collectées dans le but de développer un système de reconnaissance de la parole continue à grand vocabulaire.
Tim Schlippe et son équipe ont extrait des phrases depuis différents sites web en Hausa et les ont mises en forme dans le but d'être exploitables pour le modèle de langue. Puis, il les ont faites lire à 103 locuteurs natifs du Cameroun (i.e. : 33 hommes et 69 femmes de 16 à 60 ans). Les données ont été collectées selon le même schéma que dans GlobalPhone[2]. Au total, ce sont 7 895 expressions qui ont été enregistrées. Ces enregistrements ont été réalisés avec un micro-casque Sennheiser 440-6, dans un environnement sans bruit. Les données ont été échantillonnées à 16kHz, en 16-bit et encodées au format PCM.
Ci-contre, le tableau détaillant le découpage du corpus oral transcrit pour le Hausa.

Ensemble	Hommes	Femmes	#occurrences	#tokens	dureé
Apprentissage	24	58	5863	40k	6 h 36 m
Développement	4	6	1021	6k	1 h 02 m
Evaluation	5	5	1011	6k	1 h 06 m
Total	33	69	7895	52k	8 h 44 m

Figure 4 : Détail du corpus oral transcrit pour le Hausa

Pour ce qui est du dictionnaire de prononciation, nous avons utilisé celui créé par Globalphone. Il contient 42 662 entrées dont 42 079 mots (des mots peuvent avoir plusieurs prononciations). Il répertorie les 33 phonèmes de la langue (i.e. : 26 consonnes, 5 voyelles et 2 diphtongues).
Le Hausa est une langue à tons et les voyelles peuvent varier en durée. Dans un premier temps, nous avons préféré construire un dictionnaire de prononciation le plus générique possible : nous n'avons pas considéré les variations de longueur des voyelles ni les différents tons de la langue.

[2] GlobalPhone est un corpus multilingue de données orales et écrites, disponible pour 20 langues depuis ELRA (http://catalog.elra.info)

Chapitre 4 – Les expérimentations

4.1 Mise en forme des données brutes

Pour créer le modèle acoustique, nous avons dû transformer les fichiers audio au format *Analog to Digital Converted* (*.adc*) en fichiers *Waveform audio file format (.wav*). Pour ce faire, nous avions pensé à utiliser l'outil SoX mais il ne reconnaissait pas le format d'entrée. Nous avons cherché d'autres convertisseurs mais aucun ne permettait de faire cette conversion. Finalement, nous avons décidé d'écrire un script en python nommé `adc2wav.py` [Annexe 1] qui convertit les fichiers ADC en WAVE en suivant la même arborescence que les fichiers sources.

Un fichier au format ADC ne contient aucun en-tête, alors qu'un fichier WAVE en contient un d'une taille de 44 octets. L'en-tête d'un fichier WAVE est composé de trois blocs. Le type de données de chaque bloc est identifié par un *four-character code* (FOURCC). Un FOURCC est un entier non signé codé sur 32-bit, créé par la concaténation des 4 caractères ASCII. Il sert à identifier le type de chaque bloc.

Les blocs sont composés comme suit :

1) bloc de définition du format RIFF (i.e. : *Resource Interchange File Format*) d'une taille de 16 octets composé de 3 descripteurs :
   - le FOURCC «RIFF» (codé sur 4 octets)
   - la taille du fichier moins 8 octets (codé sur 4 octets)
   - le FOURCC «WAVE» codé en ASCII (codé sur 4 octets)
2) bloc de définition du format des données audio, d'une taille de 22 octets :
   - le FOURCC «fmt » (codé sur 4 octets)
   - la taille de l'en-tête restante (codé sur 2 octets)
   - le format audio (codé sur 2 octets)
   - le nombre de canaux (codé sur 2 octets)
   - la fréquence d'échantillonnage (codé sur 4 octets)
   - le taux d'échantillonnage fixe (fréquence d'échantillonnage * nombre de canaux * bits par échantillon) (codé sur 4 octets)

- la taille d’un échantillon (nombre de canaux * bits par échantillon/8) (codé sur 2 octets)
- le nombre de bits utilisés pour le codage de chaque échantillon (codé sur 2 octets)

3) bloc de définition des données
   - le FOURCC «data» (codé sur 4 octets)
   - la taille des données (nombre de canaux * nombres d’échantillons * bit par échantillons/8) (codé sur 4 octets)
   - les données

Nous avons donc, grâce au module `wave` de python, construit un en-tête qui spécifie que les fichiers audio sont au format WAVE. Nous avons initialisé les constantes utiles pour nous telles que le nombre de canaux utilisés (enregistrements mono donc 1), la taille d’un échantillon (1*16/8=2) ainsi que la fréquence d’échantillonnage des enregistrements (16 000Hz). Ensuite, nous avons appelé des méthodes qui nous ont permises de créer l’en-tête puis d’écrire à la suite les données des fichiers ADC.
Nous avons obtenu un fichier WAVE de la même taille que le fichier ADC + 44 octets. A l’aide du logiciel Praat, nous avons vérifié que le signal acoustique et tous les paramètres audio restent inchangés, ce qui est le cas.

Pour le modèle de langage, nous avons écrit des scripts python qui transforment les données brutes du corpus GlobalPhone. En effet, nous avons mis en forme les fichiers sources afin de pouvoir utiliser les scripts shell de l’équipe GETALP déjà créés pour d’autres données. Ces scripts servent à créer des fichiers nécessaires à l’apprentissage et au décodage dans Kaldi ; ils sont les suivants :

- `GPDictHAscript.py` [Annexe 2] pour la création du fichier `lexicon.txt` qui est une mise en forme du dictionnaire de prononciation source nommé `GPDictHA.txt`.

  Les particularités de la langue (tons et longueurs des voyelles) ont été codées dans le dictionnaire original. Cependant, pour nos premiers tests, nous n’en avons pas tenu compte. Nous avons ainsi généré un lexique suivi d’une transcription phonémique complètement épurée, sans précision sur la réalisation phonétique des phonèmes.

Aussi, nous avons transformé les lexèmes du dictionnaire en minuscule. En effet, la casse pourrait créer des doublons « artificiels », ce qui engendrerait des ambiguïtés lors de l'étape de reconnaissance si nous ne la rendons pas insensible.

- Par la suite, nous nous sommes servis de ce fichier de sortie (`lexicon.txt`) pour en extraire tous les phonèmes de la langue. Nous avons lancé une commande shell dans le Terminal (voir ci-dessous) qui permet de découper le fichier afin d'obtenir une liste de tous les phonèmes du dictionnaire puis nous les avons triés en supprimant les doublons. Nous avons redirigé le résultat de ces traitements successifs dans un fichier que nous avons nommé `nonsilence_phones.txt` :

```
cat lexicon.txt | cut -f2- | tr ' ' '\n' | sort –u > nonsilence_phones.txt
```

- `GPDictHAscriptTag.py` [Annexe 3] pour la création d'un nouveau dictionnaire de prononciation à partir du dictionnaire original, mis en forme selon notre protocole pour être lu par Kaldi, mais avec prise en compte des particularités de la langue (les tons et la durée des voyelles) originellement *taggées*.

- `scriptTranscription.py` [Annexe 4] pour la création du fichier `transcriptionHA.txt` qui recense, ligne par ligne, toutes les transcriptions orthographiques des locuteurs entre balise `<s> </s>`, et les associe au nom de leur fichier audio correspondant entre parenthèse. Dans ce script, nous utilisons le module `glob`. Il nous permet d'utiliser la fonction `glob.glob(chemin)` qui liste le contenu d'un répertoire avec appel récursif aux sous-répertoires.

Comme dit précédemment, nous nous sommes ensuite servis de scripts shell qui retraitent les fichiers créés en amont afin de les rendre exploitables par le décodeur Kaldi. Nous avons modifier les scripts afin qu'ils soient adaptés pour nos données.

Pour nos expérimentations, nous avons besoin de trois dossiers :

- « `train` » (entrainement) : contient les données d'apprentissage, c'est-à-dire les données sur lequel le système sera entrainé ;
- « `dev` » (développement) : contient les données expérimentales, c'est-à-dire les données qui serviront à tester les performances des modèles, en réalisant des

tests unitaires au fur et à mesure des résultats (i.e. : techniques de lissage en modifiant certains paramètres, etc.) ;

- « `test` » (évaluation) : contient les données qui permettront de réaliser des tests finaux, une fois que les résultats sur le corpus de développement seront satisfaisants.

Il est important que les données expérimentales soient totalement différentes des données d'apprentissage afin d'avoir un modèle robuste en sortie.

Pour la séparation des données, nous avons utilisé le même protocole que Tim Schlippe et al. [Schlippe, 2012] : 10 locuteurs pour le développement, 10 locuteurs pour les tests et le reste pour l'entraînement.
Comme la sélection des données pour chaque dossier n'a pas été réalisée dans l'ordre du numéro de locuteur, nous avons créé un script python nommé `sepTrain-Test-Dev.py` [Annexe 5] afin de copier les dossiers (et fichiers audio contenus) dans les bons répertoires. Aussi, ce script permet de changer simplement les noms de dossiers des locuteurs afin de séparer les données différemment si nous décidons de modifier le protocole. De plus, nous avons dû séparer les transcriptions orthographiques correspondantes. En sortie, nous avons donc 3 fichiers de transcriptions (un qui correspond aux données audio du `train`, un qui correspond aux données audio du `dev` et enfin un qui correspond aux données audio du `test`. Pour réaliser cette tâche, nous avons écrit 3 scripts : `trsTrain.py` [Annexe 6], `trsDev.py` [Annexe 7] `et` `trsTest.py` [Annexe 8].

4.2 Initialisation des données

Ci-dessous, nous avons détaillé le premier script `01_init_datas_eg_hausa.sh` qui permet de créer tous les fichiers qui serviront par la suite aux scripts de création du modèle de langage et aux scripts de création des modèles acoustiques du Hausa.

Cette première ligne du script permet de créer le fichier `text` qui nous a servi par la suite à la génération du modèle de langage.

```
cat trsTrain.txt | awk 'BEGIN {FS="("} {print $2 " "tolower($1)}' | sed 's/)//g' | sed 's/<s>//g' | sed 's/<\/s>//g' | sort | uniq > text
```

Grâce à la commande awk nous avons appliqué des traitements sur le fichier trsTrain.txt : nous avons découpé le fichier à partir de la parenthèse ouvrante puis nous avons inversé les deux champs de l'enregistrement courant au moyen des variables $1 (représente la transcription orthographique) et $2 (représente le nom du fichier audio correspondant), tout en précisant que nous voulions que $1 soit en minuscule (pour les mêmes raisons citées dans le paragraphe précédent). Ensuite, nous avons appliqué la commande sed qui permet, ici, de rechercher/remplacer un motif précis. Une fois ces traitements effectués, nous avons redirigé la sortie vers un nouveau fichier que nous avons nommé text.

Les lignes suivantes permettent de créer le fichier utt2spk à partir du fichier text. Ce fichier est utile à la création du modèle acoustique.

```
cat text | awk 'BEGIN {FS=" "} {print $1}' > toto1
cat toto1 | sed -e 's/HA//g' | cut -d'_' -f1 | sed '/^$/d' > toto2
paste toto1 toto2 >utt2spk
rm toto1 toto2
```

Ici, nous avons manipulé le fichier en utilisant l'espace comme séparateur de champ puis enregistré la sortie dans un fichier nommé toto1. Ensuite, nous avons repris le fichier toto1 pour récupérer la chaîne de caractères dont nous avions besoin pour le fichier final dans une sortie appelée toto2. Enfin, nous avons concaténé les entrées de toto1 et toto2 pour créer le fichier utt2spk. Ce dernier est nécessaire au script qui générera les paramètres acoustiques pertinents par la suite. Il contient le nom de chaque fichier audio suivi du nom du dossier auquel il appartient.

L'étape suivante du script appelle un script perl qui génèrera un fichier nommé spk2utt à partir du fichier utt2spk précédemment créé. Ci-dessous la ligne qui permet de réaliser ceci :

```
/home/gauthier/kaldi/utils/utt2spk_to_spk2utt.pl utt2spk | sort -k1 > spk2utt
```

Cette ligne applique donc le script Perl utt2spk_to_spk2utt.pl sur le fichier utt2spk et le trie à partir de la position 1. Ceci nous a permis de générer un nouveau fichier qui contient le nom du dossier des enregistrements audio d'un locuteur suivi des noms de

tous les fichiers audio qu'il contient (séparés par un espace). Le retour chariot permet de séparer les listes par dossier.

Enfin, la dernière étape du script consiste en la création de vecteurs MFCC. Ces paramètres acoustiques sont les plus utilisés en reconnaissance automatique de la parole. Nous avons extrait ces paramètres à partir des fichiers WAVE. Les lignes suivantes permettent de réaliser cette étape :

```
pushd ~/kaldi/data
for dir in $TRAIN_DIR $EXP_DIR
      do
            steps/make_mfcc.sh --nj 4 $dir log  mfcc
            steps/compute_cmvn_stats.sh $dir log mfcc
      done
popd
```

Comme la procédure est la même pour chaque dossier (« `train` », « `test` » et « `dev` »), nous avons écrit une boucle qui effectuera le même traitement pour chaque dossier. Cette boucle remplace la variable `$dir` par chacun des noms de dossiers – dont les chemins sont indiqués dans les variables `$TRAIN_DIR` et `$EXP_DIR` – afin d'aller chercher les bons fichiers dans chaque dossier puis d'enregistrer tous les fichiers de sortie au bon endroit.

4.3 Les modèles de langage

4.3.1 Définition

Un modèle de langage est une distribution de probabilités associée à une séquence de mot. Le but d'un modèle de langage est d'associer une probabilité à toute suite de mots et de participer aux choix des candidats pour la suite de phrases à reconnaître.

Pour créer un modèle de langage, il existe deux approches : les modèles stochastiques (issus de la théorie de l'information) et les modèles à base de grammaires formelles (faisant appel à des connaissances linguistiques et grammaticales).

Dans ce projet, nous allons créer un modèle de langage stochastique. En effet, notre but final étant la création d'un système de reconnaissance automatique de la parole sur

terminaux mobile, notre système devra reconnaître de la parole continue. Or, les modèles à base de grammaires formelles ne permettent pas de décrire le langage naturel de manière exhaustive : les grammaires sont fermées et n'autorisent, par exemple, pas les phrases incorrectes. En revanche, les modèles stochastiques sont beaucoup plus ouverts et acceptent toutes les phrases du langage, mêmes les phrases syntaxiquement incorrectes, ce qui est intéressant pour reconnaître de la parole spontanée. De plus, un modèle stochastique fonctionne sur des mesures statistiques : il est fondé à partir d'un vaste corpus, et les calculs qui le génèrent sont automatiques (contrairement aux grammaires qui sont des travaux longs et couteux).

4.3.2 Préparation de notre modèle de langage

Tout d'abord, nous avons créé un nouveau fichier nommé text.trs à partir du fichier text (précédemment tiré des données d'apprentissage) en utilisant les commandes suivantes :

```
cat text | cut -d " " -f2- | sed 's/^ //' > text.trs
```

Ce nouveau fichier .trs contient toutes les transcriptions des locuteurs (sans les ID associés) utilisées comme données d'apprentissage.

Ce fichier créé, nous avons édité puis lancé le script `03_LM_hausa.sh` qui permet de générer le modèle de langage du Hausa. Ce modèle est généré grâce à la boîte à outils SRILM. SRILM est développé par le laboratoire STAR (i.e. : *The Speech Technology and Research Laboratory*) du SRI International. Il permet la création et l'utilisation de modèles de langage à n-grammes pour représenter et traiter des textes.

Pour créer le modèle de langage, nous utilisons la commande `ngram-count` de la boîte à outils SRILM. `ngram-count` génère et manipule des n-grammes, et estime des modèles de langage n-grammes à partir desdits n-grammes. Pour notre projet, nous utiliserons un ordre de 3 pour obtenir un modèle trigramme. Le programme s'appuie sur les transcriptions des données d'apprentissage (i.e. : `text.trs`) pour créer un fichier de comptage. Par la suite, les chiffres qui en résultent sont utilisés pour la construction d'un modèle de langage n-grammes au format ARPA. Ce format est le standard, introduit par Doug Paul puis par le MIT et utilisé par les décodeurs pour les modèles n-grammes.

En sortie du script `03_LM_hausa.sh`, nous avons obtenu un fichier au format FST nommé `G.fst`. Ce fichier décrit la grammaire de la langue, sous forme de transducteur à états finis. Il est généré à partir du modèle de langage (i.e. : le fichier au format ARPA) précédemment créé.

4.3.3 Les autres modèles de langage

4.3.3.1 Le modèle de langage GlobalPhone

Nous avons récupéré le modèle de langage du Hausa nommé HAU.3gram.lm créé par GlobalPhone afin de pouvoir comparer les résultats avec le nôtre. C'est un modèle trigramme à plus grand vocabulaire (41k).
Le modèle de langage était sensible à la casse. Etant donné que nous avions rendu le nôtre insensible - tout comme nos transcriptions - nous avons fait de même pour ce modèle de langage grâce à une commande `awk` et sa fonction `tolower()`. En sortie, nous avons donc un fichier nommé `HAU.3gram_Tolower.arpa` qui est le modèle de langage de GlobalPhone sans la casse.

4.3.3.2 Le nouveau modèle interpolé

Afin d'avoir un modèle de langage amélioré, nous avons interpolé celui que nous avions généré avec ceux de Globalphone (celui avec casse et celui où nous avons enlevé la casse).
Ces deux nouveaux modèles ont été générés grâce à la commande `ngram`. Nous les avons nommé `combine_text_HAU.3gram_Tolower.arpa` (modèle sans casse) et `combine_text_HAU.3gram.arpa` (modèle avec casse). Nous avons attribué un poids de 0.5 pour que l'interpolation se fasse de manière équitable sur chacun des modèles.
Ensuite, nous avons appliqué des tests de perplexité sur ces deux nouveaux modèles de langages.

4.4 Les modèles acoustiques

4.4.1 Définition

Les modèles acoustiques proviennent de différents entraînements du système, réalisés à partir des données sonores du dossier `train`. Le système utilise le dictionnaire acoustique de GlobalPhone au moment du décodage. Cependant, nous avons écrit un script qui modifie automatiquement ce type de document car la mise en forme des données n'était pas conforme au standard d'un dictionnaire acoustique pour l'outil Kaldi.

4.4.2 Préparation

Une fois le modèle de langage obtenu, nous avons adapté et lancé les scripts qui permettent de créer les modèles acoustiques, d'apprendre à Kaldi le hausa ainsi que de tester les données après l'apprentissage. Ces scripts sont les suivants :

- `04_train_mono.sh` ce script permet de créer un modèle acoustique unigramme à partir de HMMs. C'est donc un modèle indépendant du contexte (i.e. : 1 phone, 1 réalisation).
- `04a_train_triphone_hausa.sh` : ce script permet de créer un modèle acoustique trigramme à partir de HMM, c'est-à-dire que le système tient compte du contexte gauche et droit (i.e. : 1 phone, plusieurs réalisations).
- `04b_train_MLLT_LDA_hausa.sh` : ce script une combinaison des méthodes MLLT (i.e. : *Maximum Likelihood Linear Transform*) et LDA (i.e. : *Linear Discriminant Analysis*) qui permettent d'affiner le modèle triphone grâce à la transformation de paramètres acoustiques [Psutka, 2007].
- `04c_train_SAT_FMLLR_hausa.sh` : ce script permet de créer un modèle acoustique adapté au locuteur à partir de fMLLR (i.e. : *feature-space Maximum Likelihood Linear Regression*) et un autre en contexte indépendant [Leggetter, 1995] [Gales, 2000].
- `04d_train_MMI_FMMI_hausa.sh` : ce script permet de créer trois modèles acoustiques dépendants du locuteur à partir de MMI (i.e. : *Maximum Mutual Information*) et de fMMI (i.e. : *feature-space Maximum Mutual Information*). Ces techniques permettent de déterminer le degré de dépendance de deux énoncés [Povey, 2008].

- `04e_train_sgmm.sh` : ce script permet de créer deux nouveaux modèles acoustiques à partir des SGMMs (i.e. : *Subspace Gaussian Mixture Models*). Les SGMMs permettent de regrouper des états phonétiques qui partagent la même structure dans des sous-espaces. Ce type de modèle acoustique permet une représentation plus compacte et donne, par conséquent, de meilleurs résultats par rapport à un modèle classique à base de GMMs [Povey, 2010].

Au total, nous avons obtenu 10 modèles acoustiques.

Dans chacun de ces scripts, une fois que les données ont été apprises, nous avons testé le système avec les données réservées au développement. Cette étape est appelée « décodage ». A la fin de cette étape, nous avons calculé un score appelé *Word Error Rate* (*WER*) qui nous permet d'établir des statistiques, dans le but de déterminer les paramètres optimaux pour le système de reconnaissance automatique de la parole.

4.5 Autres travaux

Initialement, nous avions prévu d'intégrer du développement mobile dans le projet professionnel, en portant la boîte à outils de reconnaissance de parole Kaldi sur smartphone. En effet, à terme, nous aimerions faire de la reconnaissance automatique de la parole directement depuis le téléphone de manière autonome et hors-ligne (i.e. : lancer le décodage sur le terminal mobile lui-même). Néanmoins, suite à nos recherches, nous nous sommes aperçus que ce travail serait trop long et trop compliqué voire même peut-être pas encore réalisable dans l'immédiat. En interrogeant la communauté d'utilisateurs de Kaldi, nous avons compris que la boîte à outils intègre une librairie non compatible avec Android, mais aussi que nous aurions des problèmes de gestion de la mémoire : le décodage est une étape longue et coûteuse.

Afin de découvrir le développement mobile et notamment l'environnement Android, nous avons alors décidé d'installer et de configurer sur deux smartphones (dotés de deux versions différentes d'Android) de l'équipe GETALP une calculatrice créée par la

société Voxygen[3]. Cette calculatrice a pour but principal de lire les nombres ainsi que le résultat du calcul que l'utilisateur aura tapé, au moyen de synthèses vocales. Voxygen en a intégrées 5 : 2 en français (i.e. : voix Moussa et voix Agnès), 1 en anglais (i.e. : voix Paul), 1 en hausa (i.e. : voix Adama) et 1 en wolof (i.e. : voix Pap). Toutefois, cette application incorpore d'autres fonctionnalités :

- faire des calculs (en appuyant sur les chiffres et les opérateurs) ;
- lire les opérandes et les opérateurs du calcul (lecture automatique ou en appuyant sur le bouton « lire ») ;
- lire le résultat (lecture automatique ou en appuyant sur le bouton « lire ») ;
- jouer à un quizz (en appuyant sur le bouton représenté par un point d'interrogation) ;
- découvrir les boutons de la calculette (en appuyant sur le bouton représentant une loupe) ;
- revenir au mode calculatrice (en appuyant sur le bouton représentant des calculs);
- accéder à la table des additions (en appuyant sur le bouton représentant des additions) ;
- accéder à la table des multiplications (en appuyant sur le bouton représentant des multiplications) ;
- changer la langue de la voix de synthèse (choix circulaire en appuyant sur le bouton représentant un haut-parleur) ;
- tout effacer (en appuyant sur le bouton représenté par une gomme) ;
- effacer un caractère (en appuyant sur le bouton représenté par une flèche retour).

Ces fonctionnalités de l'application sont décrites en annexe, au moyen d'un diagramme UML de cas d'utilisation [Annexe 9].

[3] Voxygen est une start-up française créée en 2011. Cette société est spécialisée dans la création de voix de synthèse.

4.5.1 Installation et configuration

Pour installer cette application, nous avons d'abord configuré les téléphones mobiles grâce aux outils du développeur pour activer le débogage USB. Cette option offre la possibilité d'utiliser *Android Debug Bridge*, un programme client-serveur qui fournit la commande `adb` permettant de communiquer en ligne de commande avec le mobile connecté à l'ordinateur.

Ensuite, nous avons installé sur l'ordinateur les outils nécessaires au développement mobile sous Android :

- le JDK (i.e. : *Java Development Kit*) : c'est un package qui contient le JRE (i.e. : *Java Runtime Environment* , nécessaire pour exécuter les applications Java) mais aussi des outils pour compiler et déboguer le code ;
- un environnement de développement intégré (i.e. : *Integrated Development Environment* ou IDE), pour pouvoir développer des applications Android. Nous aurions pu utiliser Eclipse associé au bundle ADT (i.e. : *Android Developer Tools*, indispensable pour développer des applications Android) mais nous avons opté pour Android Studio, créé par Google. Il a été spécialement conçu pour le développement d'application Android et sera prochainement l'outil favori des développeurs Android, à l'instar d'Eclipse+ADT aujourd'hui. En effet, Android Studio intègre par défaut le SDK (i.e. : *Software Development Kit)* d'Android et se base sur deux produits :
    - *IntelliJ IDEA*, un IDE plus rapide qu'Eclipse sur certaines problématiques liées aux Play services ;
    - *Gradle*, un outil dédié à la construction et la compilation, au test et au déploiement dans un environnement de développement.

La calculatrice de Voxygen tournera sur 2 terminaux mobiles différents : un Motorola sous Android 2.3.6 et un Wiko sous Android 4.1.2. Après avoir connecté les téléphones, nous avons ajouté les chemins vers le répertoire `platform-tools` du SDK (dans lequel se trouve la commande `adb`) dans notre PATH afin d'accéder à la commande depuis n'importe quel endroit du terminal. Nous avons tapé la commande `adb devices` qui permet de lister la liste des appareils connectés à l'ordinateur. Cette commande

fournit les numéros d’identification (i.e. : ID) des appareils suivis de l’état de la connexion (hors-ligne, connecté, non détecté).
Grâce à cette commande, nous avons récupéré les ID des deux *smartphones* et lancé l’installation de l’application (i.e. : fichier avec extension *.apk*) grâce à la commande adb –s <ID de l’appareil> install –s <chemin de l’application .apk>. Le premier –s spécifie que l’on va indiquer le numéro d’identification de l’appareil et le second indique que l’installation devra se faire sur un support externe (carte micro sd en général).
Après avoir installé l’application, nous avons utilisé la commande adb –s <ID de l’appareil> push <chemin de l’application .obb> <chemin de copie du fichier>/<nom du package>/<fichier.obb>. Cette commande permet de copier le fichier *.obb*. C’est un fichier de configuration qui contient tous les paramètres de l’application et les voix de synthèse utilisées. Le nom du package permet de relier le fichier de configuration *.obb* au fichier *.apk* (qui représente l’application en elle-même).
Enfin, nous avons ouvert notre IDE et lancé l’outil *Android Device Manager*. Cet outil permet de voir les appareils connectés depuis une interface graphique (plutôt qu’en ligne de commandes) et gérer les fichiers présents sur le téléphone. Cela nous a permis de vérifier que les fichiers *.apk* et *.obb* s’étaient copiés au bon endroit.

A la première ouverture de l’application, une barre de progression indique le chargement des différentes voix. A la fin, l’application est prête à être utilisée.

Chapitre 5 – Bilan technique

5.1 Résultats

5.1.1 La perplexité

L'évaluation extrinsèque, *in-vivo*, d'un modèle de langage prend beaucoup de temps. Pour pallier ce problème, il est possible d'évaluer la qualité d'un modèle de langue de façon intrinsèque, grâce à la perplexité.

Cette mesure permet d'estimer le pouvoir de discrimination d'un modèle entre l'ensemble des mots de son lexique. En d'autres termes, le meilleur modèle de langage est celui qui prédit le mieux le corpus de développement (dont les données sont inconnues du système). Théoriquement, plus la perplexité est basse, meilleur est le modèle (i.e. : plus sa capacité de prédiction est élevée).

Nous avons calculé la perplexité de nos modèles de langage sur les données de développement grâce à la commande `ngram` et à l'option `-ppl` de la boîte à outils SRILM.

Le tableau qui figure ci-dessous nous montre le récapitulatif des tests de perplexité sur chaque modèle de langage :

	OOV	**PPL**	**PPL1**
text.arpa *(system2)*	224	115	255
HAU.3gram.lm	78	117	257
HAU.3gram_Tolower.arpa *(system3)*	0	88	183
combine_text_HAU.3gram.arpa	33	96	202
combine_text_HAU.3gram_Tolower.arpa *(system4)*	0	88	182

La colonne « OOV » représente le nombre de mots inconnus, c'est-à-dire les mots qui apparaissent dans le corpus de test mais pas dans le corpus d'entraînement (depuis lequel le modèle de langue a été généré).

La colonne « PPL » représente la perplexité.

La colonne « PPL1» représente la moyenne de la perplexité par mot.

Nous pouvons observer que notre modèle de langage `text.arpa` est celui qui contient le plus de mots hors vocabulaire (i.e. : « OOV »). Ce résultat s'interprète facilement puisque ce modèle découle de notre corpus d'enregistrements de départ que nous avons divisé en trois sous-corpus. Par conséquent, la probabilité pour que le système tombe sur des mots inconnus est plus forte que pour les autres modèles, étant donné que celui-ci a été appris sur un petit corpus.

En ce qui concerne la perplexité, nous obtenons un score de 115, ce qui le classe avant-dernier par rapport aux tests sur les autres modèles.

Pour le modèle de langage original de GlobalPhone, `HAU.3gram.lm`, nous obtenons 78 mots hors vocabulaire, ce qui est meilleur que notre modèle. Ceci s'explique par le fait que le corpus est beaucoup plus grand. En revanche, le score de perplexité est de 117. C'est le score le plus élevé que nous avons. Ce score est probablement dû au modèle qui prend en compte la casse et qui crée par conséquent des ambiguïtés par rapport à nos données de développement qui en sont dépourvues.

Les meilleurs modèles sont les trois autres, qui sont relativement proches en terme de score. Le modèle de GlobalPhone, `HAU.3gram_Tolower.arpa`, pour lequel nous avons enlevé la casse, obtient un score de perplexité de 88, ce qui est bien meilleur que le même modèle avec casse. De plus, nous obtenons désormais 0 mots hors vocabulaire. Ceci confirme donc nos hypothèses de départ concernant les résultats du modèle original.

Finalement, le meilleur modèle de ces tests de perplexité est celui qui provient de l'interpolation de notre modèle de langage initial (i.e : `text.arpa`) avec le modèle de langage de GlobalPhone dépourvu de la casse (i.e : `HAU.3gram_Tolower.arpa`). Il obtient un score de perplexité de 88 et le nombre de mots hors vocabulaire est de 0, ce qui est une prédiction prometteuse pour les résultats du futur décodage sur nos modèles acoustiques.

5.1.2 Le taux d'erreurs mots

Le taux d'erreurs mots (i.e. : *Word Error Rate*) sert à évaluer le taux de reconnaissance du système après décodage. Ce score est calculé à partir de la distance dite de Damerau-Levenshtein [Damerau, 1964] [Levenshtein, 1966] (i.e. : *Damerau-Levenshtein Metric*) qui permet de mesurer la similarité entre deux séquences, à partir d'un alphabet. Cette distance attribue un poids à chaque opération effectuée par le système lorsque ce dernier transforme la première séquence en la seconde.

La première est appelée « référence ». C'est la transcription du signal audio qu'il faut reconnaître. La seconde est appelée « hypothèse ». C'est la solution la plus probable trouvée par le système.

Il existe trois sortes d'opérations : la substitution, la suppression et l'insertion. Ainsi, le système compte le nombre d'opérations effectuées sur l'alphabet (ici, chaque élément de l'alphabet est un mot) et attribue un score pour chacune des séquences de phrases analysées.

Les résultats du décodage sont satisfaisants. Nous avons récapitulé les scores WER obtenus dans le tableau ci-contre :

	system2	**system3**	**system4**
mono	33.99% (wer_11)	24.38% (wer_12)	25.44% (wer_12)
tri1	24.12% (wer_20)	16.75% (wer_20)	16.94% (wer_20)
tri2a	23.61% (wer_20)	16.99% (wer_20)	17.02% (wer_20)
tri2b	22.71% (wer_20)	16.46% (wer_20)	16.80% (wer_20)
tri3b	19.15% (wer_20)	13.25% (wer_19)	13.16% (wer_20)
tri3b.si	23.57% (wer_18)	17.85% (wer_20)	17.67% (wer_19)
tri3b_fmmi_a	27.67% (wer_18, it3)	22.12% (wer_19, it3)	21.96% (wer_20, it3)
tri3b_mmi_b0.1	18.23% (wer_20)	11.89% (wer_20)	12.36% (wer_20)
tri3b_fmmi_indirect	18.80% (wer_20, it3)	12.41% (wer_20, it4)	12.51% (wer_20, it4)
sgmm2_5b2	16.34% (wer_18)	10.04% (wer_17)	10.33% (wer_17)
sgmm2_5b2_mmi_b0.1	16.19% (wer_18, it3.mbr)	9.93% (wer_20, it1)	10.03% (wer_20, it1)

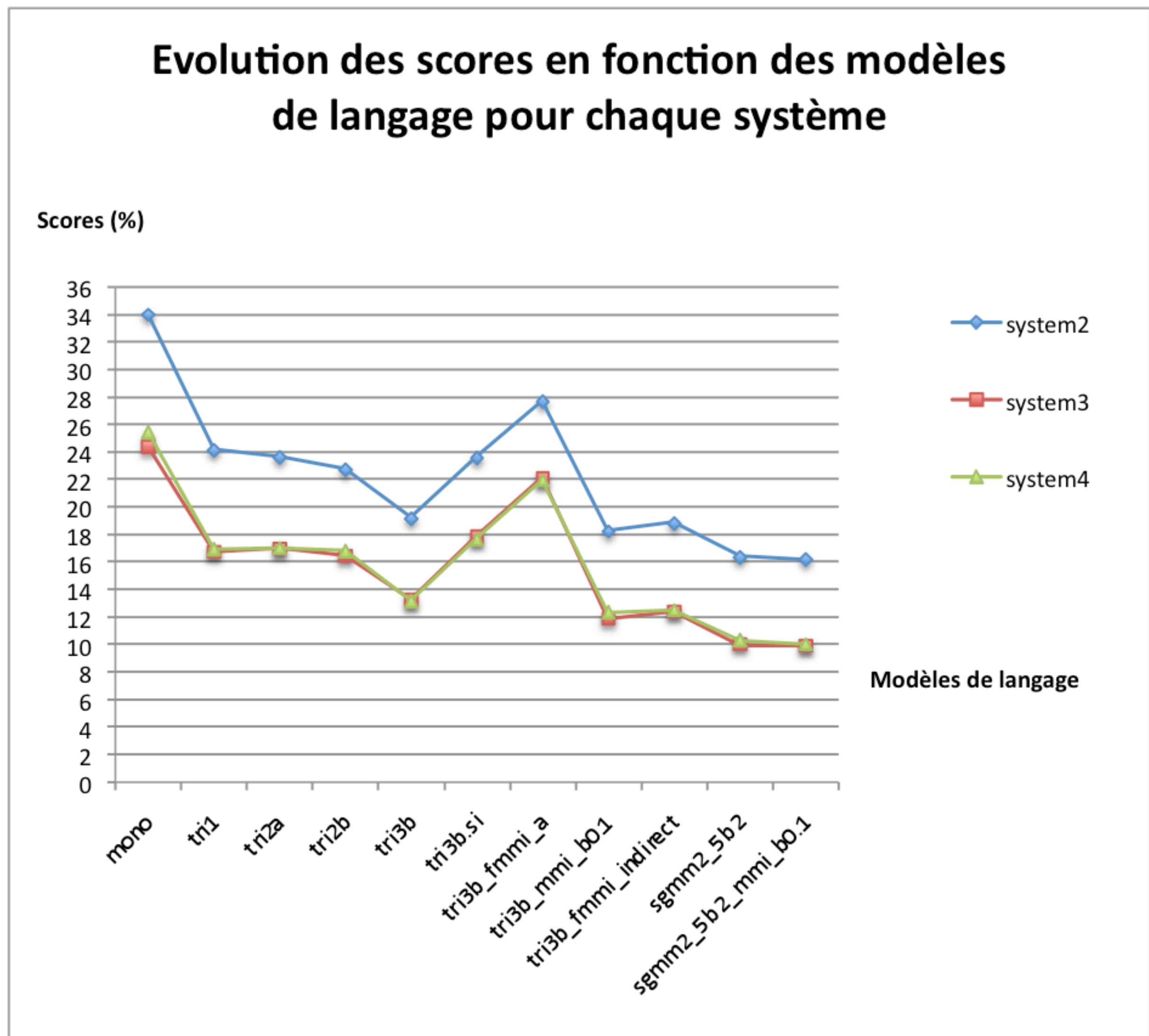


Figure 3 Evolution des scores des systèmes en fonction des modèles acoustiques

« system2 », « system3 », « system4 » sont les différents modèles créés à partir des 3 meilleurs modèles de langue, respectivement `text.arpa`, `HAU.3gram_Tolower.arpa` et `combine_text_HAU.3gram_Tolower.arpa` . La colonne de gauche représente tous les modèles acoustiques.

Nous remarquons que plus les paramètres acoustiques utilisés dans les modèles sont précis, meilleur est le score de reconnaissance. Nous avions émis l'hypothèse que le modèle interpolé serait meilleur, mais, contre toute attente, le modèle le plus performant est celui qui utilise le modèle de langage de GlobalPhone dépourvu de la casse. En effet, avec le modèle acoustique à base de SGMMs, il obtient un taux d'erreurs extrêmement satisfaisant de 9.93 %.

5.2 Difficultés rencontrées

Les difficultés à gérer durant ce projet professionnel ont été :

- les erreurs provenant des droits d'écriture/lecture/exécution ;
- les erreurs provenant de librairies non présentes dans le PATH du serveur ;
- l'imbrication de scripts qui impliquent une succession d'erreurs inattendues dans le planning ;
- la compréhension de scripts écrits par d'autres sans commentaires ni explications sur le code ;
- l'adaptation de scripts très techniques ;
- les problèmes aléatoires des serveurs.

5.3 Améliorations et perspectives

Pour le moment, nos modèles de langage ne contiennent pas les tons ni ne précisent la longueur des voyelles. Nous projetons d'intégrer ces particularités de la langue dans le dictionnaire acoustique et de relancer les étapes d'apprentissage et de décodage après ces modifications.

Aussi, nous aimerions développer une application qui permettrait de faire de la reconnaissance du Hausa sur *smartphone*. L'idéal aurait été de pouvoir porter Kaldi sur un environnement mobile Android. Cependant ce travail est impossible, car les deux environnements ne sont pour l'instant pas directement compatibles : il faudrait reprogrammer Kaldi et adapter toutes les librairies qu'il intègre (BLAS/LAPACK) à l'environnement Android, ce qui est un projet trop ambitieux en terme de coûts, de budget et de temps.

Pour pallier ces contraintes, nous allons stocker nos modèles et le système de reconnaissance sur un *cloud.* L'application mobile transfèrera, via un flux, les données sonores enregistrées depuis le microphone du téléphone à un serveur et appellera le système de reconnaissance. Puis, le système récupèrera ces données audio et traitera les signaux d'entrée. Enfin, le système décodera et l'application transcrira ce que le système aura compris sur l'écran de téléphone de l'utilisateur. Cette application nécessitera donc une connexion à Internet et ne pourra pas fonctionner en local comme imaginé au début. Plus bas, nous avons modélisé le système grâce à un diagramme UML (i.e. : diagramme de séquence).

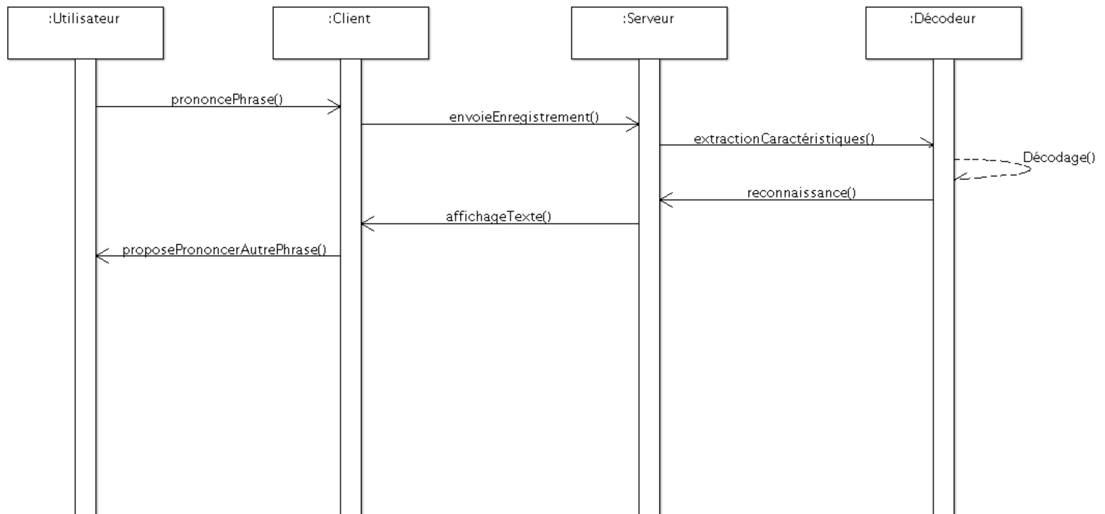


Figure 4 Diagramme de séquence d'une application de reconnaissance vocale

Chapitre 6 – Bilan personnel

J'ai effectué ce stage car il s'inscrit dans le cadre d'un projet interdisciplinaire mêlant informatique et linguistique, mes deux domaines de prédilection.
J'ai pu appliquer mes connaissances en linguistique, acquises lors de mon précédent master en Traitement Automatique de la Langue Ecrite et Parlée. De plus, j'ai pu approfondir mes compétences en informatique acquises durant cette année de master Double Compétence en Informatique et Sciences sociales, notamment en programmation shell, ainsi qu'en Java.
Aussi, j'ai pu apprendre un nouveau langage de programmation, le python. C'est un langage doté d'une grande communauté et qui est très utilisé en traitement automatique de la langue. Cela m'a permis d'appliquer la logique apprise en cours d'algorithmique.

Lors de ces deux mois de projet professionnel, j'ai dû créer des scripts afin de traiter les données en Hausa qui m'étaient fournies, dans le but de les entrainer dans le moteur de reconnaissance Kaldi. J'ai utilisé le langage python car de nombreuses personnes l'utilisent aussi en traitement automatique de la langue naturelle. Il possède de nombreuses librairies utiles au traitement automatique de la langue. De plus, je savais que, si je rencontrais un bogue dans mon script, mes collègues de l'équipe pouvaient m'aider. Dans un autre langage comme Perl, par exemple, cela n'aurait pas été le cas.

A la fin du deuxième mois de projet, j'ai pu découvrir Android. D'abord, nous avions pensé porter Kaldi - la boîte à outils pour la reconnaissance de la parole - sur des terminaux mobiles ; cependant nous nous sommes rendus compte que le temps imparti ne serait pas suffisant. Cette idée fera l'objet du stage qui suivra ce projet professionnel. Puisque le développement mobile et la découverte du système d'exploitation Android était un des objectifs de ce projet professionnel, nous avons décidé d'installer et de configurer sur deux *smartphones* (dotés de deux versions différentes d'Android) de l'équipe GETALP une calculatrice créée par la société Voxygen. Cette application peut être très utile pour le commerce, sur les marchés par exemple, pour les personnes qui ne savent pas lire. Elle sera présentée lors de la conférence annuelle TALN (Traitement Automatique du Langage Naturel), qui se déroule cette année à Marseille, durant la première semaine de juillet 2014.

Bibliographie

Damerau, F., (1964), A technique for computer detection and correction of spelling errors, *Communications of the ACM* (Vol. 7, pp. 659-664).

Gales, M. J. (2000). Cluster adaptive training of hidden Markov models. *Speech and Audio Processing, IEEE Transactions* on, 8(4), 417-428.

Koslow, P., (1995), *Hausaland: the fortress kingdoms*. Chelsea House Publishers.

Leggetter, C. J., & Woodland, P. C. (1995). Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models. *Computer Speech & Language,* 9(2), 171-185.

Levenshtein, V. I., (1966, February). Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady* (Vol. 10, p. 707).

Povey, D., Kanevsky, D., Kingsbury, B., Ramabhadran, B., Saon, G., & Visweswariah, K. (2008, March). Boosted MMI for model and feature-space discriminative training. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference* on (pp. 4057-4060). IEEE.

Povey, D., Burget, L., Agarwal, M., Akyazi, P., Feng, K., Ghoshal, A., ... & Thomas, S. (2010, March). Subspace Gaussian mixture models for speech recognition. *In Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference* on (pp. 4330-4333). IEEE.

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., ... & Vesely, K. (2011, December). The Kaldi speech recognition toolkit. *In Proc. ASRU* (pp. 1-4).

Psutka, J. V. (2007, January). Benefit of maximum likelihood linear transform (MLLT) used at different levels of covariance matrices clustering in ASR systems. In *Text, Speech and Dialogue* (pp. 431-438). Springer Berlin Heidelberg.

Schlippe, T., Komgang Djomgang, E. G., Vu, N. T., Ochs, S., and Schultz, T. (2012), Hausa Large Vocabulary Continuous Speech Recognition. In *SLTU 2012*.

Schultz, T., Vu, N. T., & Schlippe, T. (2013, May). GlobalPhone: a multilingual text & speech database in 20 languages. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference* on (pp. 8126-8130). IEEE.

Stolcke, A., (2002, September). SRILM-an extensible language modeling toolkit. In *INTERSPEECH*.

UNESCO (2010). Pourquoi et comment l'Afrique doit investir dans les langues africaines et l'enseignement multilingue.

Vycichl, W., (1990). Les langues tchadiques et l'origine chamitique de leur vocabulaire. In *Relations interethniques et culture matérielle dans le bassin du lac Tchad: actes du IIIème Colloque MEGA-TCHAD, Paris, ORSTOM, 11-12 septembre 1986* (Vol. 3, p. 33). IRD Editions.

Webographie

http://kaldi.sourceforge.net/

http://fr.openclassrooms.com/informatique/cours/creez-des-applications-pour-android

http://www2.univ-paris8.fr/ingenierie-cognition/master-handi/etudiant/projets/pdamulti/manuel_commande.php

Table des figures

Figure 1 Architecture d'un système de reconnaissance de la parole.. 10

Figure 2 Schéma simplifié de la structure de Kaldi 12

Figure 3 Evolution des scores des systèmes en fonction des modèles acoustiques 30

Figure 4 Diagramme de séquence d'une application de reconnaissance vocale 32

Annexes 1 – Script `adc2wav.py`

```
# -*-coding:Utf-8 -*

######### Script de conversion ##########
#########      ADC to WAVE     ##########
#########################################

import wave,struct,os
import glob

#### Création et initialisation des constantes
nbCanaux = 1 #mono
echtSize = 2 #taille d'un échantillon
freq =  16000 #fréquence d'échantillonnage en Hz

#### Création et ouverture du fichier en écriture

path = 'adc'
pathWAV = "wav"

dirADC = glob.glob(path+'/*')

for i in dirADC:
      loc_ID = i.split('/')[-1]
      os.mkdir('/'.join([pathWAV,loc_ID]))  #création  du  répertoire  du
locuteur

      for file in (glob.glob(i+"/*.adc")):
            fichierADC=open(file) #ouverture du fichier courant en lecture
            fichierWAV=wave.open(pathWAV+file[3:-4]+".wav","w")
      #création&ouverture du fichier WAV en écriture
            #### Copie des données dans le fichier .wav
            fichierWAV.setnchannels(nbCanaux)
            fichierWAV.setsampwidth(echtSize)
            fichierWAV.setframerate(freq)
            fichierWAV.setcomptype('NONE','non compresse')
            fichierWAV.writeframesraw(fichierADC.read())    #écriture    des
données du fichier .adc

            #### Fermeture des fichiers après traitement
            fichierWAV.close()
            fichierADC.close()
      #fin du for file
#fin du for i
```

Annexe 2 – Script `GPDictHAscript.py`

```
# -*-coding:Utf-8 -*

#####################################################
#### Mise en forme du dictionnaire de prononciation ###
#################### SANS les tags ###################
#####################################################

#ouverture du fichier de sortie en écriture
OUT=open("GPDictHAOut.txt","w")

#lecture du fichier source
for line in open("GPDictHA.txt"):
	#séparation entre lexème et sa transcription phonémique
	a, b=line.split(" {{")
	a = a[1:-1]
	OUT.write(a.lower()+"\t") #écriture du lexème
	c=""
	for e in b.split("_")[1:]:
		c+=e.split(" ")[0]+" " #écriture du phonème
	#fin du for
	OUT.write(c[:-1]) #suppression de l'espace après le dernier phonème
de la ligne
	OUT.write("\n")
#fin du for

#fermeture du fichier
OUT.close()
```

Annexe 3 – Script `GPDictHAscriptTag.py`

```
# -*-coding:Utf-8 -*

#### Mise en forme du dictionnaire de prononciation ###
################### AVEC les tags ###################
######################################################

#ouverture du fichier de sortie en écriture
OUT=open("GPDictHATag.txt","w")

#lecture du fichier source
for line in open("GPDictHA.txt"):
	#séparation entre lexème et sa transcription phonémique
	a, b= line.split(" {{")
	a= a[1:-1]
	OUT.write(a.lower()+"\t") #écriture du lexème
	c= ""
	for e in b.split("_")[1:]:
		c+=e.split(" ")[0]+" " #écriture du phonème
		#traitement des tags
		if "L" in e or "S}" in e or "T1" in e or "T2" in e or "T3" in e
or (" S W" in e and len(e)>7):
			c+="_"+e.split(" ")[1]+" " #écriture du tag
		#fin if
	#fin for
	c= c.replace("}","")
	c= c.replace("\n","")
	OUT.write(c[:-1]) #suppression de l'espace après le dernier phonème
de la ligne
	OUT.write("\n")
#fin du for
#fermeture du fichier
OUT.close()
```

Annexe 4 – Script `scriptTranscription.py`

```
# -*-coding:Utf-8 -*

#### Ecrit toutes les transcriptions dans un seul fichier .txt  ###
#### Entrée : 1 fichier de transcription par locuteur ############
#### Sortie : 1 fichier avec toutes les transcriptions ############

import glob
OUT=open("transcriptionHA.txt","w")
for file in sorted (glob.glob("trs/*.trl")):

      for line in open(file):
            if line[0]==";":
                  numLine=line[2:-3]
            else:
                  OUT.write("<s> "+line[:-1]+"</s> "+"("+file[4:-
4]+"_"+numLine+")\n")
```

Annexe 5 – Script `sepTrain-Test-Dev.py`

```
# -*-coding:Utf-8 -*
import glob, os, shutil, time

###### Division du corpus audio vers les dossiers ##
###########     train, dev et test     #############

#######################    PROTOCOLE     ##########################
######## test : 002, 014, 025, 028, 030, 052, 053, 062, 070, 088 ###
######## dev : 018, 031, 034, 038, 046, 047, 050, 055, 058, 072 ####
######## train : reste  ###########################################
###################################################################

test = ["wav/002", "wav/014", "wav/025", "wav/028", "wav/030", "wav/052",
"wav/053", "wav/062", "wav/070", "wav/088"]
dev = ["wav/018", "wav/031", "wav/034", "wav/038", "wav/046", "wav/047",
"wav/050", "wav/055", "wav/058", "wav/072"]
corpus = glob.glob('wav/*')


def union(a, b): #fonction qui retourne l'union de 2 sets
      return list(set(a).union(set(b)))


def difference(a, b): #fonction qui retourne différence entre 2 sets
      return list(set(b).difference(set(a)))

#train = corpus-(test+dev)
reste = union(test,dev) #calcul du reste

#calcul de la différence entre corpus total et reste
train = difference(reste,corpus)
print (">>>>>>>>> Copie des fichiers dans test <<<<<<<<<<<<<<< \n")
for e in test:
      ficTest = glob.glob(e+"/*wav")
      dirName1 = e.split("/")[1] #repertoire locuteur (eg:"002")
```

```
	os.mkdir('/'.join(["test",dirName1])) #création du répertoire du locuteur

	for a in ficTest:

		dirWAV1 = a.split("/")[0] #repertoire "wav"

		ficName1 = a.split("/")[2] #fichier .wav (eg: "HA002_1.wav")

		shutil.copy('/'.join([dirWAV1,dirName1,ficName1]), '/'.join(["test",dirName1,ficName1])) #copie des fichiers

print (">>>>>>>>> FIN copie fichiers dans test <<<<<<<<<<<<< \n")

time.sleep(1)

print (">>>>>>>>> Copie des fichiers dans dev <<<<<<<<<<<<<<< \n")

for f in dev:

	ficDev = glob.glob(f+"/*wav")

	dirName2 = f.split("/")[1]

	os.mkdir('/'.join(["dev",dirName2])) #création du répertoire du locuteur

	for b in ficDev:

		dirWAV2 = b.split("/")[0]

		ficName2 = b.split("/")[2]

		shutil.copy('/'.join([dirWAV2,dirName2,ficName2]), '/'.join(["dev",dirName2,ficName2])) #copie des fichiers

print (">>>>>>>>> FIN copie fichiers dans dev <<<<<<<<<<<<< \n")

time.sleep(1)

print (">>>>>>>>> Copie des fichiers dans train <<<<<<<<<<<<<<< \n")

for g in train:

	ficTrain = glob.glob(g+"/*wav")

	dirName3 = g.split("/")[1]

	os.mkdir('/'.join(["train",dirName3])) #création du répertoire du locuteur

	for c in ficTrain:

		dirWAV3 = c.split("/")[0]

		dirName3 = c.split("/")[1]

		ficName3 = c.split("/")[2]

		shutil.copy('/'.join([dirWAV3,dirName3,ficName3]), '/'.join(["train",dirName3,ficName3])) #copie des fichiers


print (">>>>>>>>> FIN copie fichiers dans train <<<<<<<<<<<<< \n")
```

Annexe 6 – Script `trsTrain.py`

```
# -*-coding:Utf-8 -*

###### sépare les transcriptions correspondant aux données audio #####
###########              du dossier train               ###########
######################################################################

import glob

test = ["002", "014", "025", "028", "030", "052", "053", "062", "070", "088"]
dev = ["018", "031", "034", "038", "046", "047", "050", "055", "058", "072"]
corpus = glob.glob('*')
def union(a, b):
      return list(set(a).union(set(b)))
def difference(a, b):
      return list(set(b).difference(set(a)))

#train = rep-(test+dev)
reste = union(test,dev) #calcul du reste
train = difference(reste,corpus) #calcul de la différence entre corpus total et reste

OUT=open("trsTrain.txt","w")
for file in sorted (glob.glob("trs/*trl")):
      fic = file.split("/")[1]
      if fic[2:-4] in train:
            for line in open("trs/HA"+fic[2:-4]+".trl"):
                  if line[0]==";":
                        numLine=line[2:-3]
                  else:
                        OUT.write("<s> "+line[:-1]+"</s> "+"("+file[4:-4]+"_"+numLine+")\n")
```

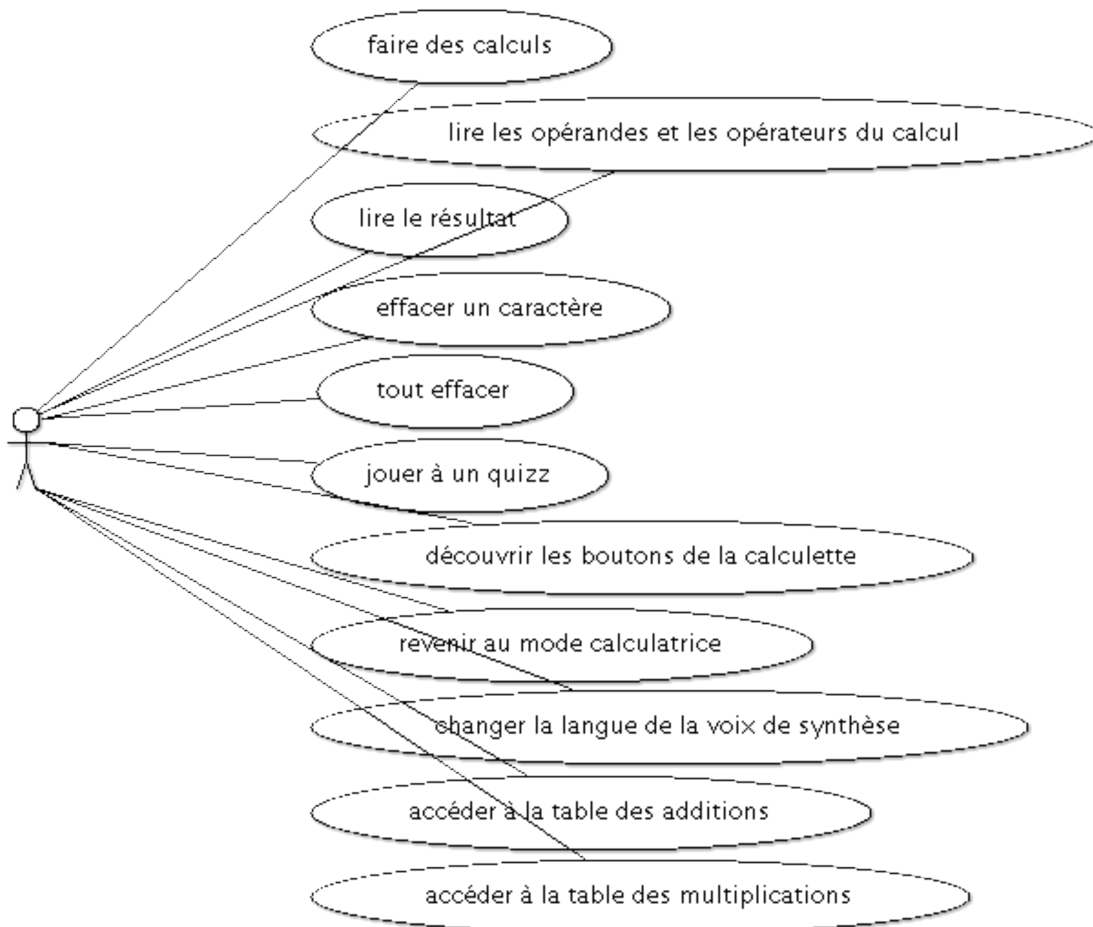
Annexe 7 – Script `trsDev.py`

```
# -*-coding:Utf-8 -*

###### sépare les transcriptions correspondant aux données audio #####
###########               du dossier dev            ################
######################################################################

import glob

dev = ["HA018.trl", "HA031.trl", "HA034.trl", "HA038.trl", "HA046.trl",
"HA047.trl", "HA050.trl", "HA055.trl", "HA058.trl", "HA072.trl"]

OUT=open("trsDev.txt","w")

for file in sorted (glob.glob("trs/*trl")):
	fic = file.split("/")[1]
	if fic in dev:
		for line in open("trs/"+fic):
			if line[0]==";":
				numLine=line[2:-3]
			else:
				OUT.write("<s> "+line[:-1]+"</s> "+"("+file[4:-
4]+"_"+numLine+")\n")
```

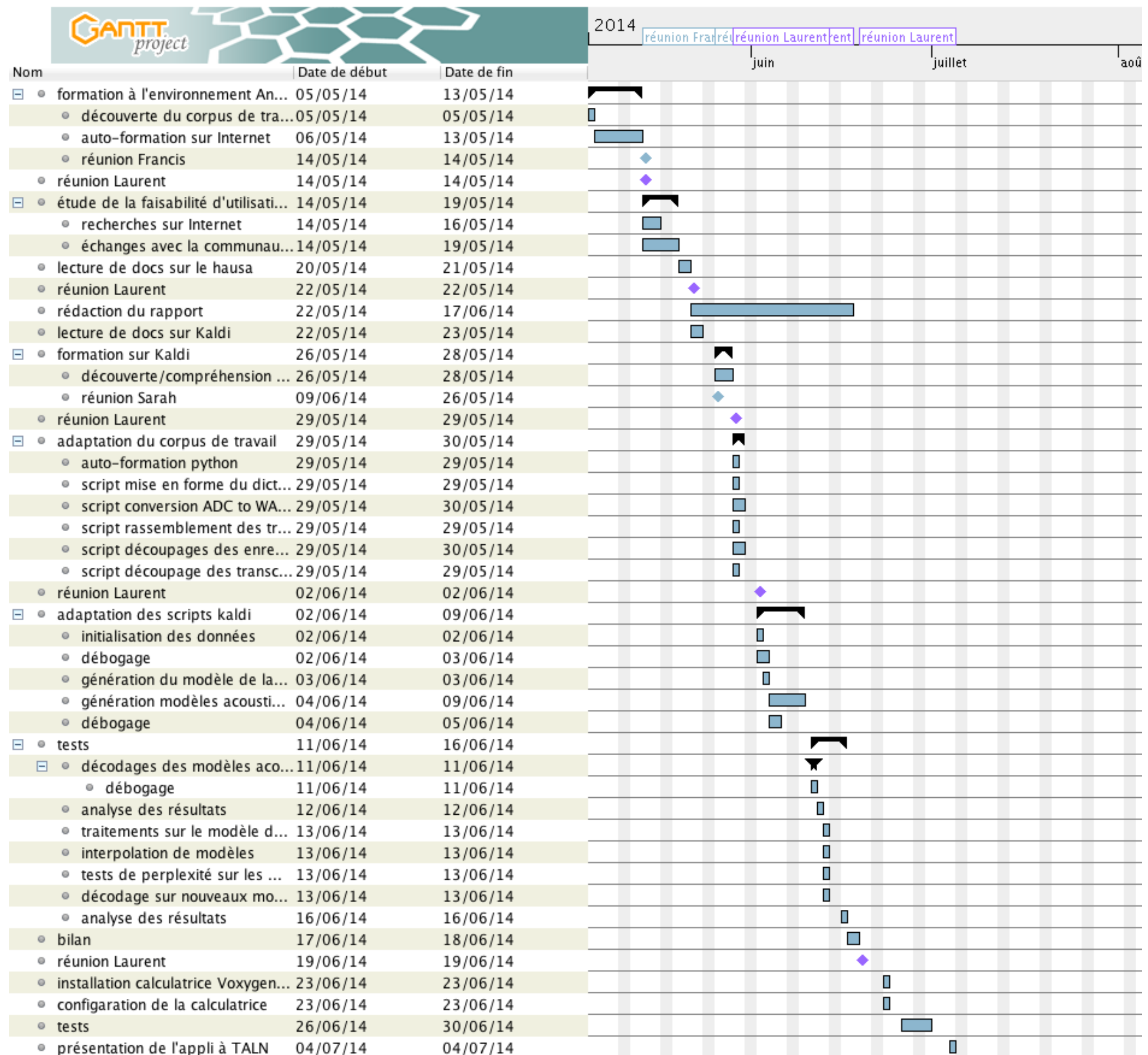
Annexe 8 – Script `trsTest.py`

```
# -*-coding:Utf-8 -*

###### sépare les transcriptions correspondant aux données audio #####
###########               du dossier test                ###########
######################################################################

import glob

test = ["HA002.trl", "HA014.trl", "HA025.trl", "HA028.trl", "HA030.trl", "HA052.trl", "HA053.trl", "HA062.trl", "HA070.trl", "HA088.trl"]

OUT=open("trsTest.txt","w")
for file in sorted (glob.glob("trs/*trl")):
	fic = file.split("/")[1]
	if fic in test:
		for line in open("trs/"+fic):
			if line[0]==";":
				numLine=line[2:-3]
			else:
				OUT.write("<s> "+line[:-1]+"</s> "+"("+file[4:-4]+"_"+numLine+")\n")
```

Annexe 9 – Diagramme de Use-case de la Calculatrice Voxygen



Annexe 10 – Planning de travail



Nom	Date de début	Date de fin
formation à l'environnement An...	05/05/14	13/05/14
découverte du corpus de tra...	05/05/14	05/05/14
auto-formation sur Internet	06/05/14	13/05/14
réunion Francis	14/05/14	14/05/14
réunion Laurent	14/05/14	14/05/14
étude de la faisabilité d'utilisati...	14/05/14	19/05/14
recherches sur Internet	14/05/14	16/05/14
échanges avec la communau...	14/05/14	19/05/14
lecture de docs sur le hausa	20/05/14	21/05/14
réunion Laurent	22/05/14	22/05/14
rédaction du rapport	22/05/14	17/06/14
lecture de docs sur Kaldi	22/05/14	23/05/14
formation sur Kaldi	26/05/14	28/05/14
découverte/compréhension ...	26/05/14	28/05/14
réunion Sarah	09/06/14	26/05/14
réunion Laurent	29/05/14	29/05/14
adaptation du corpus de travail	29/05/14	30/05/14
auto-formation python	29/05/14	29/05/14
script mise en forme du dict...	29/05/14	29/05/14
script conversion ADC to WA...	29/05/14	30/05/14
script rassemblement des tr...	29/05/14	29/05/14
script découpages des enre...	29/05/14	30/05/14
script découpage des transc...	29/05/14	29/05/14
réunion Laurent	02/06/14	02/06/14
adaptation des scripts kaldi	02/06/14	09/06/14
initialisation des données	02/06/14	02/06/14
débogage	02/06/14	03/06/14
génération du modèle de la...	03/06/14	03/06/14
génération modèles acousti...	04/06/14	09/06/14
débogage	04/06/14	05/06/14
tests	11/06/14	16/06/14
décodages des modèles aco...	11/06/14	11/06/14
débogage	11/06/14	11/06/14
analyse des résultats	12/06/14	12/06/14
traitements sur le modèle d...	13/06/14	13/06/14
interpolation de modèles	13/06/14	13/06/14
tests de perplexité sur les ...	13/06/14	13/06/14
décodage sur nouveaux mo...	13/06/14	13/06/14
analyse des résultats	16/06/14	16/06/14
bilan	17/06/14	18/06/14
réunion Laurent	19/06/14	19/06/14
installation calculatrice Voxygen...	23/06/14	23/06/14
configaration de la calculatrice	23/06/14	23/06/14
tests	26/06/14	30/06/14
présentation de l'appli à TALN	04/07/14	04/07/14