SCRIBE-CONSEIL

KALDI

SpeechToText

Construire un SRAP fonctionnel en utilisant KALDI pour des traitements NLP postérieurs : installation, Tf-Idf, adaptation au Français & évaluations

Auteur :
CAROLE LAILLER
C.LAILLER@SCRIBE-CONSEIL.COM
06.70.37.38.56

RAPPELS des travaux :

- Première étape : installer Kaldi & se mettre à l'État de l'art avec un système KALDI EN,

- Deuxième étape : les principes d'un nuage de mots (*stop-list*) & du *Tf-Idf*,

- Troisième étape : quelques principes à observer pour adapter la recette EN au Français à travers le corpus *DECODA*,

17 septembre 2019

1 Présentation du projet :

1.1 Sujet

Se trouvent, au sein de ce document, la méthode d'installation pour permettre la construction d'un SRAP robuste et fonctionnel sur de la langue spontanée en Anglais, de nombreuses recommandations pour suivre et comprendre son fonctionnement ainsi que des *tips* pour passer au Français. Ce SRAP, fondé sur une technologie fiable, *i.e.* l'outil libre, KALDI doit fournir une solution de reconnaissance vocale éprouvée en contexte. En outre, une réflexion autour de la construction de nuages de mots et d'un calcul "primaire" de *Tf-Idf* sera menée.

À noter : vous êtes évidemment libres d'utiliser vos outils (éditeurs entre autres) et vos langages de scripts (les exemples données sont en *perl* et *Bash* (*shell*). N'hésitez pas cependant à regarder, à lire, à scruter. Par ailleurs, vous devrez d'abord et avant tout faire preuve d'un esprit d'analyse ; ainsi, vos nuages de mots et autres fichiers de comparaison en *Tf-Idf* me seront rendus avec commentaires et hypothèses étayés.

1.2 Le présent document :

En amont de tout effort d'évaluation & d'adaptation, sont rappelées puis détaillées les étapes de construction du SRAP en EN/FR à partir de la solution Kaldi :
- sont donc exposés des éléments d'installation et de mises en place mais aussi des conseils méthodologiques & les commandes commentées quant à la construction du système.

1.3 Le sujet du TD :

Trois parties vont composer ce TD dont l'objectif est d'obtenir la meilleure entrée possible pour les traitements NLP ultérieurs, i.e. un système de Reconnaissance de la Parole robuste et dédié à la langue française spontanée.

1.3.1 Partie 1 — l'installation de Kaldi en mono-machine :

Matériel = chaque étudiant amène sa machine et cherche à en apprendre davantage sur KALDI (cf. le cours)

`https://github.com/kaldi-asr/kaldi`
— s'interroger sur :
- Quelle installation ?
- Quels chemins modifier ?
- Sur quelles données travailler ? Quel(s) nettoyage(s) envisager ?

L'installation à proprement parler :
- se doter d'une machine vierge et installer un Linux (Ubuntu 16.04 OU Virtual box OU Boot Dual),

- mettre à jour le système car ce dernier ne l'est pas (sudo apt-get update et consorts),
À noter : en revanche, plutôt que d'utiliser un gestionnaire de packages comme *apt-get* dont la complexité est réelle, on installe aptitude donc *sudo apt-get install aptitude* & éventuellement on fait l'économie des éléments trop verbeux par l'ajout d'un *-y*.

- récupérer les outils de compilation qui seront nécessaires (sudo apt-get install build-essential),

ON PEUT MAINTENANT INSTALLER KALDI en :

- se créant un répertoire sources pour ne pas travailler à la racine du compte mais dans un répertoire IDOINE (vives les *mkdir*).

- récupérer Kaldi en conservant la branche la plus stable (upstream) : *git clone http ://github.com/kaldi-asr/kaldi.git kaldi –origin upstream.*

- travailler dans *tools* pour l'installation de tous les éléments.

- procéder à l'installation
=> cat INSTALL
=> exécution des commandes
=> *make*

À noter : POINTS À VOIR => attention aux cœurs selon les machines (j = nb de coeurs de la machine) donc make -j si toute la puissance de la machine convoquée sinon make -j 8 (on spécifie sur quoi ça tourne) !

- ON INSTALLE PAS MKL mais Atlas ; donc quand le terminal signalera qu'MKL n'y est pas, on s'en fiche un peu...
./configure –shared –mathlib=ATLAS

- avant de compiler, il faut nettoyer :
make clean -j (si rien, on tourne à 1 sinon on le précise pour aller plus vite car on utilise tout)

- À NE PAS OUBLIER : construire les dépendances (les dépendances seront dans les fichiers cachés de tous les répertoires - *.depend*)
make -j
»> run.pl de la recette anglaise (copie du RÉPERTOIRE) et on lance

Quelques itérations de la recette anglaise en lançant le *run.pl* puis on stoppe tout APRÈS L'ÉTAPE 15 *exit 0*...

1.3.2 Partie 2 — Entre Nuages de mots & *Tf-Idf* = s'approprier la langue des corpus :

- Choisir un vocabulaire dédié

- Calculer la partie *Idf*

- Calculer le *Tf-Idf*

- Comparer chaque élément à l'ensemble pour déterminer les similarités ou du moins proximité(s)

1.3.3 Partie 3 — Réfléchissons à partir de ces bribes sur une recette adaptée au FR :

Quel corpus ? => Article et curiosité convoquée `http://www.lrec-conf.org/proceedings/lrec2016/pdf/474_Paper.pdf`.
S'interroger sur :
QUEL VOLUME / QUELLE RÉPARTITION ?
QUELLE ÉVALUATION ?
QUELLES DIFFÉRENCES PAR RAPPORT AUX TRAVAUX SUR L'ANGLAIS (*stop-list*, nuages de mots & *Tf-Idf*) ?
QUID DE LA NORMALISATION ? À quoi s'attendre *vs* réalités ?

=> UTILISEZ LA RECETTE ANGLAISE MAINTENANT INSTALLÉE ET USUELLE POUR CONSTRUIRE UN SRAP DÉDIÉ KALDI EN FRANÇAIS avec analyses en amont & en aval.
=> FAIRE TOURNER LA RECETTE ANGLAISE AVEC LE CORPUS DECODA (DANS SA VERSION COMPLÈTE SUR MON SITE, RÉPERTOIRES DÉDIÉS) SUR QUELQUES PASSES POUR OBTENIR UNE VRAISEMBLANCE :

Que FAUDRAIT-IL PRÉVOIR :

- POUR UNE RECO DÉDIÉE ET PARFAITEMENT FONCTIONNELLE ?

- EN FONCTION D'UN CAS D'USAGE DÉDIÉ ? :
=> vous êtes un dirigeant de boite de transport en commun, quel projet *NLP* mettre en place quand on a DECODA à sa disposition et des envies d'outils d'IA selon vous (chatbot, *Entités Nommées* => quels cas d'usages, quels traitements en amont, quelles investigations à mener ?)...

2 *How to* — Kaldi :

Vous trouverez ci-après un récapitulatif des éléments à prendre en compte dans l'ensemble des briques KALDI. L'objectif est de mettre en relief les points forts et faibles des techniques utilisées tout en s'assurant de la complète lisibilité des chemins ainsi que d'une intégration pleinement réussie, selon des architectures différentes.

2.1 Les pré-requis — Rappels génériques :

Avant toute chose, quelques rappels nécessaires :

- il est préférable de travailler sous LINUX.
- Voici l'adresse où récupérer Kaldi : `https://github.com/kaldi-asr/kaldi`.
- POUR CHAQUE RECETTE : faire une copie récursive de tous les fichiers pour le répertoire de travail. Il est important de NE PAS TRAVAILLER AVEC LES ORIGINAUX DE RECETTES POUR REMONTER EN CAS D'ERREUR (même si pour l'anglais, en utilisant le corpus TEDLIUM2, il n'y a pas grand chose à changer).

- Prendre le temps d'installer la bibliothèque mathématique **Atlas** et plus particulièrement **Atlas-dev** pour disposer non seulement d'un outil capable de réaliser des calculs matriciels mais aussi des fichiers *includes* associés aux bibliothèques compilées.
- De même, si l'on souhaite travailler sous GPU pour améliorer les calculs et la vitesse de traitement (entre autres), il est indispensable d'installer **Cuda**, la bibliothèque d'utilisation des GPU, mise au point par Nvidia.

Une fois, le clone Kaldi effectué, il "suffit" de suivre les instructions données dans le mode d'emploi à la racine (*cat install*). Il faut ensuite descendre dans le répertoire *Tools* et suivre encore une fois les installations demandées. Après avoir lancé la compilation, il est impératif de prendre le temps **de vérifier dans le fichier source (*cd src*, avec la commande (*./configure – shared*) que tout est prêt à fonctionner et que tout est en place (Cuda entre autres, mais aussi OpenFST). Ne pas oublier évidemment de lancer le *make* et de vérifier que tout se déroule normalement.**

Ces vérifications étant effectuées, on peut revenir à la racine et commencer à explorer le répertoire **egs** afin de choisir un pipeline Kaldi à monter puis à tester. Il est **fortement conseillé d'effectuer une copie récursive** de la recette de travail à tester afin de ne pas biaiser la progression et de pouvoir travailler avec méthode. Le plus abordable, méthodologiquement parlant, est de débuter avec le Kaldi Anglais qui utilise le corpus TedLium. En choisissant la recette **s5_r2**, il est possible de vérifier la progression du travail et la **conformité à l'état de l'art en se référant au fichier *results***. Par ailleurs, rien n'interdit de fonder la première mise en œuvre sur cette recette avec TedLium2 et de constater les améliorations avec le corpus TedLium3, disponible depuis peu sur le site du LIUM (attention à l'adresse).

Dernière précision, si l'on souhaite "tester" ce Kaldi sur une machine seule, sans recourir à une baie de serveur, il faut utiliser le script *run.pl* qui répartit les jobs sur un seul noeud. Enfin, **il faut évidemment bien penser à mettre à jour les chemins dans le code** et rien n'interdit la lecture du répertoire *utils* pour trouver toutes les réponses aux questions.

3 Quelques préconisations d'usage — En marche :

3.1 Les commandes :

Sont répertoriées ici les différentes commandes et de nombreux commentaires les accompagnant pour élaborer et *monitorer* au mieux le système ainsi obtenu, évaluations comprises :
Est ajouté, en sus, des éléments pour suivre de plus près encore l'installation complète de Kaldi. **Ainsi, l'autonomie sur LE SRAP peut être TOTALE.**

3.2 Installation de Kaldi :

On considère partir d'une **machine vierge tournant sur Ubuntu 16.04** : un certain nombre de package sont à installer préalablement. On peut y ajouter une version de Java pour permettre au logiciel de segmentation de tourner. De la même manière, installer *G2PSequitur* pour disposer d'un système d'apprentissage des phonétisations constitue un passage obligé. Ce dernier outil est celui qui a servi à l'élaboration du dictionnaire et peut permettre, à la faveur d'une adaptation, d'augmenter le dictionnaire de phonétisations avec un apprentissage à partir de la version actuelle.

Les commandes à effectuer sont donc les suivantes :

```
c_lailler@modele:~$ sudo apt-get update
c_lailler@modele:~$ sudo apt-get install build-essential
c_lailler@modele:~$ sudo apt-get install -y openjdk-8-jdk
c_lailler@modele:~$ sudo apt-get install aptitude
c_lailler@modele:~$ sudo aptitude install libatlas-dev  libatlas-base-dev
c_lailler@modele:~$ sudo aptitude install python3-pip
c_lailler@modele:~$ sudo aptitude install swig
c_lailler@modele:~$ sudo pip3 install numpy
c_lailler@modele:~$ sudo pip3 install git+https://github.com/sequitur-g2p/
   ↪ sequitur-g2p@master
voire si besoin :
c_lailler@modele:~$ sudo pip3 install --upgrade pip
```

Il convient d'ajouter le logiciel XmlStartlet qui permet de récupérer facilement les informations dans les fichiers de transcription. Ce n'est pas obligatoire mais toujours utile.

```
c_lailler@modele:~$ sudo aptitude install xmlstarlet
```

Lors de l'apprentissage des modèles acoustiques, il est nécessaire d'utiliser des cartes GPU Nvidia. Par conséquent, la bibliothèque CUDA et le driver des cartes sont convoqués. Pour la version installée de Kaldi, il faut **la version 9 de cette bibliothèque**.
Ici, on ne procède pas à ce travail autour des cartes GPU. Toutefois, les *geeks* qui veulent s'amuser et qui un jour seront confrontés aux *DNN*...

```
c_lailler@modele:~$ sudo apt-get install linux-headers-$(uname -r)     # pour
   ↪  l'installation du driver
c_lailler@modele:~$ wget https://developer.nvidia.com/compute/cuda/9.2/Prod2
   ↪ /local_installers/cuda_9.2.148_396.37_linux
c_lailler@modele:~$ wget https://developer.nvidia.com/compute/cuda/9.2/Prod2
   ↪ /patches/1/cuda_9.2.148.1_linux
c_lailler@modele:~$ chmod +x cuda_9.2.148_396.37_linux cuda_9.2.148.1_linux
c_lailler@modele:~$ sudo sh cuda_9.2.148_396.37_linux
```

```
c_lailler@modele:~$ ls -l /usr/local/cuda
c_lailler@modele:~$ sudo sh cuda_9.2.148.1_linux
```

Lors de ce processus, il faut refuser l'installation du driver si la machine n'a pas encore de carte GPU associée, car cela va provoquer des erreurs.
Il ne reste plus qu'à installer Kaldi dans un répertoire et vérifier que l'on dispose de bien tous les *packages*. Pour l'installation, on a choisi le répertoire `$HOME/src`.

```
c_lailler@modele:~$ mkdir src
c_lailler@modele:~$ cd src
c_lailler@modele:~/src$ git clone https://github.com/kaldi-asr/kaldi.git
    ↪ kaldi --origin upstream
c_lailler@modele:~/src$ cd kaldi/
c_lailler@modele:~/src/kaldi$ cd tools/
c_lailler@modele:~/src/kaldi/tools$ extras/check_dependencies.sh # on n'
    ↪ oublie rien
c_lailler@modele:~/src/kaldi/tools$ sudo apt-get install zlib1g-dev automake
    ↪  autoconf unzip sox libtool subversion
```

On rajoute ce qui est demandé et on lance la compilation. Bref, on fait ce qu'il demande...

```
c_lailler@modele:~/src/kaldi/tools$ make -j
```

Pour certains scripts, l'utilisation de SRILM peut être nécessaire. Si l'on dispose déjà d'une version sur la machine, on crée le lien vers la version présente sinon le script vous indiquera ce qu'il faut faire et vous demandera de valider sa licence. **Très important donc !** Une solution efficace : venir sur mon instance, récupérer le fameux *tgz* pour ensuite l'envoyer (commande *scp - attention aux noms et à la destination* sur votre machine.)
Par ailleurs, on installe aussi PocoLM qui servira aussi pour les modèles de langage et qui n'a aucune restriction d'utilisation.

```
c_lailler@modele:~/src/kaldi/tools$ ./install_srilm.sh
Donc :
c_lailler@modele:~/src/kaldi/tools$ ln -s srilm-1.7.2.tgz srilm.tgz
```

À vérifier quant au nom et à déposer où il se doit évidemment. Ne pas oublier de relancer...

```
c_lailler@modele:~/src/kaldi/tools$ ./extras/install_pocolm.sh
```

La suite concerne la compilation de Kaldi proprement dite :

```
c_lailler@modele:~/src/kaldi/tools$ cd ../src/
c_lailler@modele:~/src/kaldi/src$ cat INSTALL   #c'est un fichier !
c_lailler@modele:~/src/kaldi/src$ ./configure --shared --mathlib=ATLAS (--
    ↪ use-cuda) ##cf. GPU
c_lailler@modele:~/src/kaldi/src$ make  clean
c_lailler@modele:~/src/kaldi/src$ make depend  -j
c_lailler@modele:~/src/kaldi/src$ make  -j
```

Le *make* peut être réalisé en deux fois ; suivre les instructions. Attention toutefois à la mémoire et aux CPU dans le dernier *make* : la compilation est gourmande...

Pour faire fonctionner cette recette sans toutefois aller dans des méandres par trop gourmands et inutiles (frontières meilleures, redécodage, réalignements surtout avec les *DNN*), on s'arrêtera après l'étape 15 en éditant le script idoine et en plaçant un *exit 0*...

3.3 La recette anglaise :

3.3.1 Préambule :

Les données pour l'anglais sont disponibles et gratuites, ce qui rend l'usage de la recette anglaise bien agréable ! Attention, le LIUM a changé d'adresse ; fort heureusement, la seconde adresse qui pointe vers le corpus et dont se sert KALDI est maintenue et effective. ATTENTION Les fichiers audio sont des sphères - c'est un format très usité dans les travaux académiques et qui induit, dans notre cas, l'utilisation du script dédié.

On exécute toujours le run.sh de la recette qui est plutôt bien commenté. Toutefois, avant exécution, PENSER à modifier les deux seuls éléments à modifier.

3.3.2 Les chemins & modifications :

Voici ci-après la recette expliquée pas à pas — il va sans dire que la langage de script utilisé (ou *shell*) est bash — :

export KALDI_ROOT=$HOME/src/kaldi
l'endroit où on a fait le git clone
ls $KALDI_ROOT/

Faire une copie locale de la recette.
r-> récursif
L -> faire une vrai copie en remplaçant les liens symboliques par les fichiers cibles ATTENTION

cp -rL $KALDI_ROOT/egs/tedlium/s5_r2/* .
#Tous les répertoires de s5_r2 sont donc ramenés.

éditer (en utilisant un éditeur idoine, *emacs* par exemple) cmd.sh suivant la configuration local (Slurm ou autre) ; ici on tourne en local avec une machine ayant **2 coeurs avec 1 thread chacun**.

LES 2 EXPORTS DOIVENT ÊTRE IDENTIQUES À CELA (NE PAS S'OCCUPER DES IF) :

```
export train_cmd="run.pl --max-jobs-run 2"

export decode_cmd="run.pl --max-jobs-run 1"
```

éditer path.sh pour mettre à jour KALDI_ROOT

L'EXPORT DOIT ÊTRE IDENTIQUE À CELA (attention au répertoire sélectionné/racine) :

export KALDI_ROOT=$HOME/src/kaldi
#export LC_ALL=C ==> définit le tri indépendamment de l'encodage... Toujours utile.

---

mkdir -p lesLogs/train
faire un répertoire à coté pour les logs

---

lancer l'apprentissage et les décodages internes.
nohup ./run.sh > lesLogs/train/run.1 2>&1 &

3.3.3 Examinons les résultats :

Pour regarder plus aisément dans les logs, la réalité du travail effectué, il est de bon ton de pratiquer un *grep -v* afin d'éliminer les lignes de téléchargement. Je vous laisse trouver le *pattern* le plus efficace pour aérer votre résultat.
Ensuite, n'hésitez pas à regarder les résultats en eux-même et **à consulter les *pra***. À noter : pour trouver ces fameux fichiers de comparaison, on peut bien sûr transformer un *sys* en *pra* mais rien ne vaut un petit tour dans l'arborescence, ne serait-ce que pour contempler le ***local/score.sh***. Vous pourrez ainsi constater que le *lattice* devient *ctm* et qu'il est possible de jouer avec les poids minimaux et maximaux des ML, ainsi qu'avec les pénalités.
À noter : # en cas d'erreur à une étape, corriger et relancer (là, exemple de pb à l'étape 6 avec le chemin de sph2pipe : nohup ./run.sh –stage 6 > lesLogs/train/run.6 2>&1 &)

ATTENTION, le fichier result.sh (en minuscules) est un script qui une fois lancé vous donne vos résultats à vous ! Plus pratique pour la comparaison...

score c'est le petit LM - small LM
rescore/score c'est le grand - large LM
Kaldi a un modèle avec plus de bigrammes, de trigrammes et de quadri donc une perplexité plus faible d'où le *rescore*.
regarder le fichier RESULT et les commandes internes pour comparer les résultats.

=> Qu'en déduisez-vous ? Quelles conclusions formulez-vous ?

3.3.4 Pour aller plus loin :

Ci-après sont ajoutés quelques commentaires sur le script *run.pl* qui, comme son nom l'indique, permet de lancer le système et les décodages :

- Au début du script, on peut lire :
Based mostly on the Switchboard recipe. The training database is TED-LIUM, # it consists of TED talks with cleaned automatic transcripts : ceci indique que les transcriptions sont relativement propres et renettoyées ensuite dans la recette pour les réseaux. **Je me permets donc de souligner l'importance d'avoir des data propres surtout pour dessiner des architectures en *deep*.**

- certaines options du scripts sont modifiables si besoin :
nj=35
decode_nj=30 # note : should not be >38 which is the number of speakers in the dev set
# after applying –seconds-per-spk-max 180. We decode with 4 threads, so
# this will be too many jobs if you're using run.pl.
stage=0

- on peut commenter et décommenter, et donc jouer sur les LM et appeler le local/ted_train_lm.sh en lieu et place de local/ted_download_lm.sh :
if [$stage -le 4]; then
# Download the pre-built LMs from kaldi-asr.org instead of building them
# locally.
local/ted_download_lm.sh
Uncomment this script to build the language models instead of
downloading them from kaldi-asr.org.
local/ted_train_lm.sh
fi

- POUR INFO : les lm sont compilés sous forme de FST :
if [$stage -le 5]; then local/format_lms.sh fi

- POUR INFO : on ajoute ici quelques proba de phonétisations à partir du *Train*, efficace s'il en est :
if [$stage -le 13]; then
steps/get_prons.sh –cmd "$train_cmd" data/train data/lang_nosp exp/tri2
utils/dict_dir_add_pronprobs.sh –max-normalize true
data/local/dict_nosp exp/tri2/pron_counts_nowb.txt
exp/tri2/sil_counts_nowb.txt
exp/tri2/pron_bigram_counts_nowb.txt data/local/dict fi

- POUR INFO : KALDI embarque un modèle dédié à l'adaptation aux locuteurs :
if [$stage -le 15]; then
steps/align_si.sh –nj $nj –cmd "$train_cmd"
data/train data/lang exp/tri2 exp/tri2_ali

steps/train_sat.sh –cmd "$train_cmd"
5000 100000 data/train data/lang exp/tri2_ali exp/

- POUR INFO : Attention ça paraît simple mais allez voir le sh en question (il s'agit effectivement d'un empilement complexe) :
The nnet3 TDNN recipe :
local/nnet3/run_tdnn.sh

3.3.5 Les nuages de mots :

CONSIGNES — À vous de jouer !

Vous allez construire, grâce au site `https://www.wordclouds.com`*, des nuages de mots sur les textes de référence et de décodage pour être ensuite à même de comparer, discuter, émettre des hypothèses sur la réussite du décodage et ses palliatifs à mettre en place. Pour cela :*

- Effectuer un nuage à partir de tous les *stm* de la REF,

- Effectuer un nuage à partir de tous les *ctm* du décodage — choisir le meilleur !

- Pensez à déterminer votre *stop-list* intelligemment, *i.e.* en la déterminant sur la langue courante par exemple,

- Utiliser la commande *cut* et un script pour compter les mots ; pensez à afficher le nombre d'occurrences avant le mot pour ne pas heurter l'outil qui fait le nuage...

- Évidemment les nuages de mots sont à sauvegarder ET à commenter !

3.3.6 Le *Tf-Idf* :

CONSIGNES — À vous de jouer !

Vous allez effectuer un Tf-Idf primaire en prenant les documents du train de façon à avoir suffisamment de matière. L'idée est non pas d'utiliser les nombreux outils disponibles (en Python beaucoup aujourd'hui ; ces derniers utilisant des matrices creuses notamment) mais d'oeuvrer avec quelques scripts en perl pour comprendre les mécanismes de cette distance et son intérêt quand on travaille sur la langue. Pour cela :

- Choisir un vocabulaire ni trop ni trop peu "garni" à partir des *stm* du *train* — habituellement, on considère ôter les 250 mots les plus courants pour ensuite ne conserver qu'une part honnête (plus ou moins 1000 mots) du restant,

- Calculer la partie *Idf* selon la collection de documents à disposition,

- Calculer la totalité du *Tf-Idf* normalisé (selon le mot le plus fréquent, c'est un bon point de référence),

- Grâce au script idoine de comparaison, récupérer chaque document avec son vecteur pour le comparer avec tous les vecteurs de ceux de la collection. Bien sûr, vous penserez à trier pour retrouver son jumeau puis tous les documents similaires... *(./Compare.perl 'récupération d'un grep' < fichier texte contenant tous les vecteurs pour chaque document de la collection | sort -k 2nr,2 puis qu'on veut trier sur le 2ème champ par ordre décroissant*)

- Évidemment vous sauverez quelques exemples "parlants" que vous discuterez allégrement !

3.4 La recette française :

3.4.1 Avant de se lancer :

Avant même de se lancer dans l'élaboration d'un système en français, quelques remarques liminaires et conseils quant aux traitements des données s'imposent.

Il convient de rappeler que KALDI est un outil libre qui se caractérise par une attention toute particulière portée sur les modèles acoustiques et le traitement de l'audio : les problématiques d'alignements des phonèmes sont ainsi au coeur du système. **Le script *run.sh* le montre bien dans la mesure où un apprentissage de plus en plus fin est mis en oeuvre avec des HMM d'abord monogaussiennes avant d'autoriser les multigaussiennes et le passage à un alignement des phonèmes en contexte. Kaldi peut donc être résumé (rapidement) en un système d'apprentissage des modèles acoustiques doté d'un décodeur performant, l'ensemble étant fondé sur des FST. Petit bémol : avec le si peu de corpus dont nous disposons, nous n'irons pas jusqu'au *TDNN* donc les résultats ne seront pas époustouflants. L'objectif est de se confronter aux dures réalités des adaptations de scripts pour faire tourner le système...**

3.4.1.1 Les audios :

Pour le système anglais, les choses sont relativement simples dans la mesure où le corpus audio est fourni sous le format *sphère*. En revanche, pour construire un système en français, il sera nécessaire de constituer un corpus de fichiers son utilisables de bonne qualité, idéalement en canal séparé...
Afin de rendre ces données "appréhendables" par les technologies Kaldi, une manipulation de conversion peut être nécessaire.

3.4.1.2 Pour alimenter le ML — notes de travail :

Outre les analyses recommandées pour mieux circonscrire les usages et comprendre la cible & les éléments clés et coeur de cible, un effort de transcription manuelle doit souvent être entrepris. Certes, ce travail est laborieux et coûteux dans la mesure où un transcripteur aguerri ne parvient à transcrire "que" 1 heure de parole pour 7 heures de travail mais il demeure indispensable. L'avantage de Kaldi est qu'il n'est pas forcément trop gourmand. Un corpus cible de 25 heures correctement transcrit avec le logiciel dédié *Transcriber* et un guide de transcription construit pour les besoins détectés (pour gérer les disfluences, les Entités Nommées idoines, les souffles, les respirations ainsi que l'overlap et les bruits parasites) peut pallier de nombreux manques et donc autoriser réellement une adaptation.

3.4.2 Construire un système français — *How to ?* :

Deux solutions peuvent être convoquées pour élaborer un système en français, une fois le système anglais maîtrisé. Il me semble qu'il est intéressant, d'un point de vue méthodologique, d'avoir recours aux deux. TOUTEFOIS, AU SEIN DE CE TD, NOUS AURONS "SEULEMENT" RECOURS À LA SECONDE :

La première solution est de prendre une recette anglaise dédiée téléphone, construite sur *Switchboard*, et d'adapter/adopter d'autres corpus : Il faut donc concentrer ses efforts, non sur les briques techno mais sur la constitution des corpus de données.

- prendre la recette *swbd/5c* & ADAPTER les corpus, notamment leur format. Toutefois, ceci peut se révéler fort efficace car les réseaux et les tailles sont un peu plus adaptés dans cette recette.

La seconde solution est de CONSERVER la recette déjà travaillée et de procéder à quelques changements :

3.4.3 La recette adaptée :

CONSIGNES — À vous de jouer !
Vous allez construire un Kaldi pour du Français en prenant la recette anglaise (la "fameuse" s5_r2) en modifiant ce qui concerne les fichiers son, la préparation des données, le dico, la construction du ML... Bref, ***il s'agit non seulement de suivre l'arborescence du TedLium (et donc de renommer ce qui doit l'être) mais aussi de respecter les appels du run.sh voire d'adapter au téléphone (changer la fréquence d'échantillonnage dans conf/mfcc.conf) ou de changer les seuil pour le pruning du ML.*** *L'idée n'est pas d'être optimal mais de parvenir à passer tous les stage avec succès ! Pour cela :*

- Revoir la totalité du Répertoire *db* pour que le système retrouve ses petits (quand les *sphere* deviennent *wave*),

- Installer la recette Anglaise en procédant aux changements qui s'imposent et en n'oubliant de pas rééditer les fichiers *path.sh* & *cmd.sh*,

- Le corpus est très petit, il est donc prudent d'adapter les seuils (dans *run.sh* : *nj=6* & *decode_nj=2*)

- La version française utilise *PocoLm* du Python2 ; il est donc primordial de procéder à l'installation de la version idoine : *sudo -H pip2 install 'numpy>1.15.0,<1.17.0'*,

- Modifier dans le répertoire *local/*, les fichiers suivants : *prepare_data.sh*, *prepare_dict.sh*, *ted_train_lm.sh.* (que l'on peut renommer car on travaille sur un autre corpus), *format_lms.sh*

- on travaillera avec un modèle **trigramme**,

- Évidemment vous modifierez le *run.sh* en conséquence (attention auX *download*)...

Si l'on reprend avecle résultat des commandes diff :

— *prepare_ data.sh* :

- nul besoin d'une conversion puisque nous avons déjà des *wave* :

```
<   cat $dir/spk2utt | awk -v set=$set -v pwd=$PWD '{ printf("%s␣
    ↪ sph2pipe␣-f␣wav␣-p␣%s/db/TEDLIUM_release2/%s/sph/%s.sph␣|\n", $1,
    ↪  pwd, set, $1); }' > $dir/wav.scp
---
 cat $dir/spk2utt | awk -v set=$set -v pwd=$PWD '{ printf("%s␣%s/db/
    ↪ DECODA/%s/wav/%s.wav␣\n", $1, pwd, set, $1); }' > $dir/wav.scp
```

- il faut supprimer le *padding* à la fin du fichier en commentant la partie dédiée.

— *prepare_ dict.sh* :

- il faut revoir le nom du dictionnaire et la gestion des *fillers*

```
 22c22
<   grep -v SIL | sort > $dir/nonsilence_phones.txt
---
 egrep -v 'SIL|\+b\+|\+par\+|\+i\+|\+mus\+' | sort > $dir/
    ↪ nonsilence_phones.txt
24c24
< ( echo SIL; echo NSN ) > $dir/silence_phones.txt
---
( echo SIL; echo NSN;echo '+b+' ;echo '+par+'; echo '+i+' ;echo '+mus+'
   ↪ ) > $dir/silence_phones.txt
```

— dans *ted_ train_ lm.sh* renommé en *decoda_ train_ lm.sh* :

```
38c38
< num_dev_sentences=10000
---
num_dev_sentences=5000
47c47
< bypass_metaparam_optim_opt="--bypass-metaparameter-optimization
   ↪ =0.854,0.0722,0.5808,0.338,0.166,0.015,0.999,0.6228,0.340,0.172,0.999,0.788
   ↪ "
---
#bypass_metaparam_optim_opt="--bypass-metaparameter-optimization
   ↪ =0.854,0.0722,0.5808,0.338,0.166,0.015,0.999,0.6228,0.340,0.172,0.999,0.788
   ↪
62c62
<   gunzip -c db/TEDLIUM_release2/LM/*.en.gz | sed 's/ <\/s>//g' | local
   ↪ /join_suffix.py | gzip -c  > ${dir}/data/text/train.txt.gz
---
 #gunzip -c db/TEDLIUM_release2/LM/*.en.gz | sed 's/ <\/s>//g' | local/
    ↪ join_suffix.py | gzip -c  > ${dir}/data/text/train.txt.gz
79c79
<   awk '{print $1}' db/TEDLIUM_release2/TEDLIUM.152k.dic | sed 's
   ↪ :([0-9])::g' | sort | uniq > ${dir}/data/wordlist
---
 awk '{print $1}' db/DECODA/decoda.dict | sed 's:([0-9])::g' | sort |
    ↪ uniq > ${dir}/data/wordlist
```

```
82c82
< order=4
---
order=3
91c91
<    min_counts='train=2 ted=1'
---
 min_counts=2
112c112
<    size=10000000
---
 size=170000
129c129
<    size=2000000
---
 size=150000
```

- il faut enfin penser à adapter à l'ordre du ML dans *format_ lms.sh*

- bien sûr, les *TedLium* disparaîtront au profit de *DECODA*...