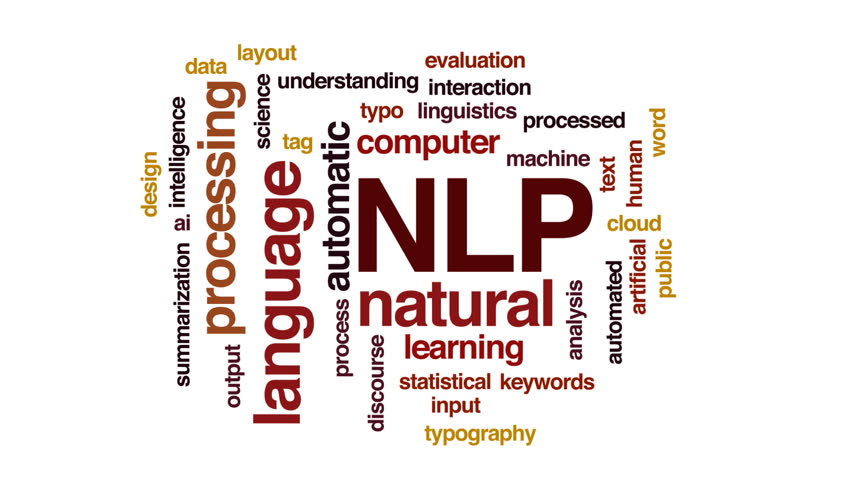
RAPPORT DE PROJET EN TAL

Sujet 1 : Détection de la langue d’un texte



**Emeline CARUANA, Adèle RICHARD**

Juin 2019

L3 MIASHS/LINGUISTIQUE INFORMATIQUE

INTRODUCTION

Le projet que nous avons décidé de traiter est le sujet de Détection de langue dans un corpus. Ce type d’algorithme sont souvent utilisés surtout lors de reconnaissance de langue pour la traduction automatique.

Le sujet se divise en trois parties : le pré-process, la recherche du modèle et l’évaluation. Le but du pré-process était de récupérer différents corpus de plusieurs langues pour créer les données comparables pour l’algorithme. La recherche du modèle peut être considérée comme la création de l’algorithme avec les différentes fonctions permettant la comparaison des données des langues et les textes à évaluer. Et enfin, l’évaluation, la partie la plus importante du projet, consiste à mesurer les modèles entre eux. C’est-à-dire de comparer les données des langues du pré-process et celles des textes à évaluer et de donner un résultat sur la langue de ces derniers.

Nous avons choisi de séparer le projet en au moins deux parties : une partie qui gère les différents textes et une autre partie qui gère tous les calculs liés aux vecteurs de langues. Pour expliquer rapidement le déroulement de ces parties : la partie qui gère les textes s’occupe de récupérer les bi-grammes et leur fréquence d’apparition et la partie qui gère les calculs transforme ces fréquences en vecteurs pour ensuite faire les calculs de similarités.

PARTIE THÉORIQUE

Le but de ce sujet est de créer un algorithme qui après comparaison des données permet de donner la langue d’un certain texte. L’algorithme fonctionne de cette façon : il récupère des textes d’une certaine langues et les différents mots leur appartenant. Les mots sont ensuite découpés de façon à avoir des groupes d’une à deux lettres pour créer des uni-grammes et des bi-grammes. La fréquence d’apparition de ces uni-grammes et bi-grammes permet de créer un vecteur pour la langue donnée. C’est ce vecteur qui nous nous permet de comparer les langues grâce à un calcul de similarité cosinus.

La partie à automatiser est donc la récupération de tous les fichiers textes composant nos corpus de langue, le découpage de ces textes en bi-grammes ainsi que la création des vecteurs. Dans notre projet, cette partie se fait une seule fois au lancement du programme sur tous les dossiers de corpus des langues utilisées pour l’entraînement uniquement. Pour la réaliser, nous avons eu à créer les fonctions de lecture de fichier, de découpage en n-grammes et de création en vecteurs que nous expliqueront plus en détail dans la partie informatique.

Cette automatisation a comme avantage le fait que tout se fait au moment de l’exécution du programme. Il n’y a pas besoin de rentrer de lignes spéciales pour faire toutes les étapes liées à l’entraînement du programme. Le plus grand inconvénient à cette automatisation est le temps que le programme prend pour faire toutes les étapes. Cela ne prend que quelques secondes mais ce n’est quand même pas instantané. Le temps d’exécution augmente en fonction du nombre de textes d’entraînement que nous avons. Au départ, nous n’avions que 3 langues (anglais, français, espagnol) mais plus on ajoutait de textes pour ces 3 langues ou plus on ajoutait de langues, plus le temps d’exécution devenait plus long. De plus, même si on dit que c’est une partie automatique, le code que nous avons écrit pour l’exécution est très long à cause de tous les tests et de toutes les variables que nous avons créées. Nous expliquerons plus en détails ce que nous avons écrit dans la partie informatique du rapport.

La partie qui nous paraissait la plus théorique de ce projet était la transformation des fréquences d’apparition des uni-grammes et des bi-grammes en vecteurs. Nous avons eu du mal à visualiser cette transformation et avons dû beaucoup nous renseigner sur le sujet. Nous avons trouvé des cours sur les vecteurs, le produit scalaire et la similarité cosinus. Cependant, tous ces cours ne portaient que sur des vecteurs de dimensions 2, c’est-à dire avec deux coordonnées a et b telles que le vecteur u = (a,b). Dans notre projet, les vecteurs ont beaucoup plus de coordonnées que les vecteurs à deux dimensions car les coordonnées des vecteurs sont les fréquences d’apparitions des bi-grammes. Nous avons tout de même réussi à trouver les formules mathématiques utiles à notre projet et à définir correctement les vecteurs de nos langues.

Comme tous les cours que nous trouvions étaient sur des vecteurs à deux dimensions, il nous paraissait logique de transformer vecteurs de langues en vecteurs à deux dimensions. Cependant, après avoir fait de nombreux calculs, nous avons remarqué que la langue qui était donnée n’était pas tout le temps la bonne. C’est après tout cela que nous nous sommes rendues compte que le but n’est pas de faire des calculs au hasard mais de faire des calculs en fonctions de l’uni-gramme ou du bi-gramme. Par exemple, on sait qu’en anglais les lettres “w” et “k” apparaissent très souvent, plus souvent qu’en français. C’est cette différence d’utilisation qui nous permet de dire que le français et l’anglais sont deux langues différentes. Donc lors des calculs de produit scalaire de deux vecteurs de langue, il fallait faire attention à multiplier les coordonnées correspondant au même uni-gramme ou bi-gramme. Nous avons choisi de faire un calcul de similarité cosinus plutôt que la distance euclidienne car cela nous semblait beaucoup simple à écrire. De plus, l’une de nous ayant suivi un partout mathématique, les vecteurs et le cosinus sont des éléments très importants de ce parcours. Le fait d’avoir des connaissances, mêmes lointaines, nous a rassurées pendant la création du projet.

PARTIE EXPÉRIENCES ET RÉSULTATS

Comme précisé dans l’introduction, le pré-process est composé de toutes les étapes avant la création du projet. Nous avons donc décidé de commencer par le choix des langues, nous avons donc choisi 3 langues au départ : l’anglais, le français et l’espagnol. Les différents corpus de ces langues proviennent de différentes sources pour essayer d’avoir le plus de distributions possibles. Nous avons donc des textes littéraires, des transcriptions de discours politiques ou scientifiques, des sous-titres de films et des textes de cours de langues étrangères. Le plus simple a été de trouver des textes français mais ceux-ci étant plutôt courts, nous avons dû en chercher d’autres plus longs. Idem pour les deux autres langues. Plus nous avancions dans le projet plus nous voulions rajouter des langues. Donc nous avons cherché des corpus en allemand, italien et néerlandais. Les corpus en néerlandais ont été les plus compliqués à trouver car il existe peu de cours de cette langue mais nous en avions assez pour notre projet.

Avant de pouvoir nous occuper des vecteurs, nous avons travaillé sur la récupération des fichiers, de leur contenus et de leur découpage en uni-grammes et en bi-grammes. Nous avons toujours fait des tests sur nos fonctions avant de passer aux étapes suivantes. Nos fonctions pour faire cette étape sont décrites dans la Partie Informatique du rapport. À part évaluer nos fonctions en utilisant les textes déjà récupérées, nous n’avons pas fait d’expérience surprenante. Le plus intéressant lors de l’évaluation est de voir si nos calculs de vecteurs sont corrects et que le programme nous donne la langue exacte d’un texte.

Pour évaluer nos formules de calcul sur les vecteurs, nous avons choisi de faire ces calculs sur les dictionnaires de l’anglais, du français et de l’espagnol. Le calcul fait est celui de la similarité cosinus. Ce calcul nous donne un résultat compris entre -1 et 1. Si la similarité cosinus est égale à 0 alors les deux vecteurs sont orthogonaux, c’est-à-dire indépendants. Si elle est égale à -1 alors les vecteurs sont totalement opposés et donc ce sont deux langues différentes. Si elle est égale à 1 alors les vecteurs sont identiques. Les valeurs intermédiaires illustrent le degré de similarité entre les vecteurs. Pour la première version de nos calculs, nous avions comme résultats :

* 1.0 pour la similarité entre le dictionnaire anglais et le dictionnaire anglais
* 0.6 à peu près la similarité entre le dictionnaire anglais et le dictionnaire français
* -0.5 à peu près pour la similarité entre le dictionnaire anglais et le dictionnaire espagnol
* 0.2 à peu près pour la similarité entre le dictionnaire français et le dictionnaire espagnol

Nous pensions que ces résultats étaient corrects, nous avons alors fait cette même expérience sur différents textes en français et en anglais. C’est à ce moment là que nous avons constaté que nos calculs étaient faux car nous obtenions très peu de résultats corrects pour les textes français. Le programme nous disait que ces textes étaient soient en anglais soit en espagnol.

Après avoir compris d’où venait nos erreurs, nous avons modifié nos fonctions et nous avons testé au fur et à mesure si elles fonctionnaient. Nous avons, comme précédemment, commencé par comparer les dictionnaires de nos langues (français, anglais, espagnol) ce qui nous a donné les résultats ci-dessous :

* 1.0 pour la similarité entre le dictionnaire anglais et le dictionnaire anglais
* 0.924 à peu près la similarité entre le dictionnaire anglais et le dictionnaire français
* 0.913 à peu près pour la similarité entre le dictionnaire anglais et le dictionnaire espagnol
* 0.934 à peu près pour la similarité entre le dictionnaire français et le dictionnaire espagnol

Ces résultats étaient largement plus satisfaisants que les précédents car les langues que nous avons choisies ont une racine en commun et ne sont pas si différentes les unes des autres. Nous avons ensuite fait les calculs sur les textes utilisés pour l’évaluation et les résultats étaient eux aussi corrects. Le programme nous donnait la bonne langue pour chaque texte.

Nous avons alors décidé d’ajouter plus de langues à notre programme pour qu’il soit plus riche en données. Nous avons inséré des corpus en italien, allemand et néerlandais. Et nous avons recommencé toutes les étapes de l’évaluation pour vérifier que tout fonctionnait avec ces additions. Tout s’est déroulé correctement et nous avons obtenus de bons résultats à nouveau. Nous avions alors enfin un bon programme qui donne les bonnes réponses.

DONNÉES : SIMILARITÉ COSINUS DE NOS LANGUES

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| LANGUES | Anglais | Français | Espagnol | Allemand | Italien | Néerlandais |
| Anglais | 1.0 | 0.9239163499116421 | 0.9130887171207092 | 0.8954609634777082 | 0.9329411281987745 | 0.8906221438282617 |
| Français | 0.9239163499116421 | 1.0 | 0.9342602924459298 | 0.9051496774770925 | 0.9267155119745895 | 0.8831032820361886 |
| Espagnol | 0.9130887171207092 | 0.9342602924459298 | 1.0 | 0.8513587831264476 | 0.9619221172894445 | 0.8651693261354502 |
| Allemand | 0.8954609634777082 | 0.9051496774770925 | 0.8513587831264476 | 1.0 | 0.8393077419766279 | 0.9417620104470994 |
| Italien | 0.9329411281987745 | 0.9267155119745895 | 0.9619221172894445 | 0.8393077419766279 | 1.0 | 0.8451395410560172 |
| Néerlandais | 0.8906221438282617 | 0.8831032820361886 | 0.8651693261354502 | 0.9417620104470994 | 0.8451395410560172 | 1.0 |

PARTIE INFORMATIQUE

Organisation et présentation des classes

Notre programme s’organise principalement autour de quatres fichiers python : ReadFiles.py, Token.py, Vecteur.py et Final.py.

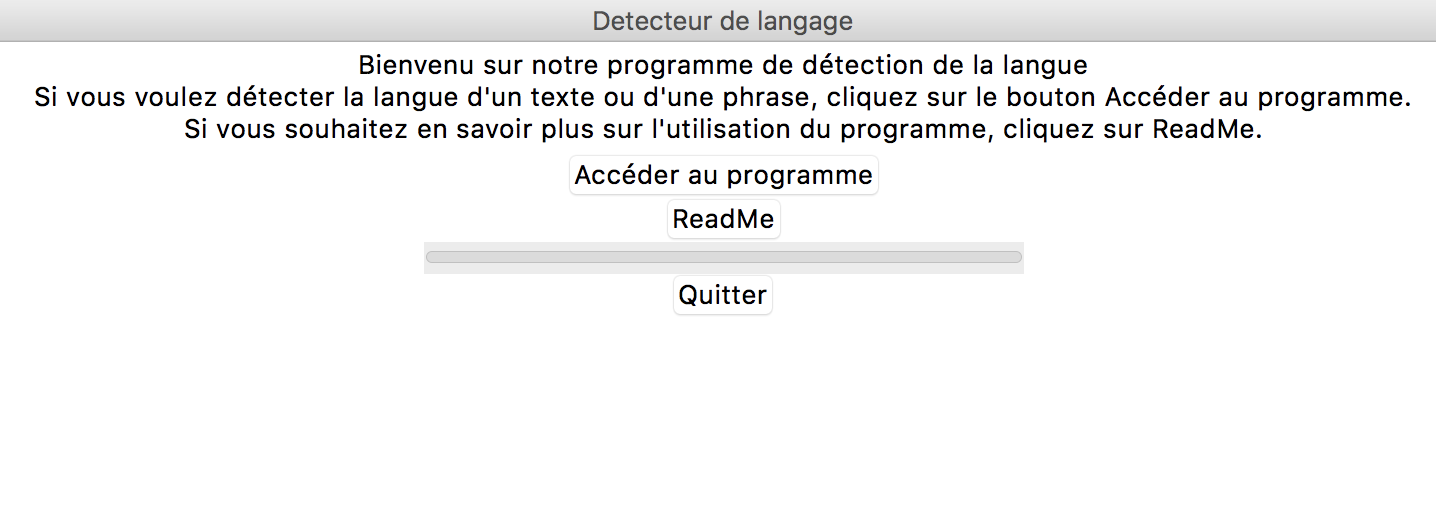
Le premier, **ReadFiles.py** a été programmé de manière à ce qu’il s’occupe principalement de la lecture des fichiers textes qui sont dans notre dossier **DataSet**. Pour cette classe, nous avons pris la liberté d’utiliser le module **glob** qui nous retourne instantanément une liste de tous les fichiers qui sont dans le dossier souhaité (Celui-ci doit être dans DataSet/Train). Par exemple, si on demande à notre fichier ReadFiles.py de chercher tous les fichiers .txt dans le sous-dossier Eng, cela nous donnera une liste [‘text1.txt’, ‘text2.tx’, etc] . Le plus intéressant avec ce module, c’est qu’il parcourt rapidement chaque texte et que cela nous fait économiser du temps d'exécution du programme.

Le deuxième fichier, **Token.py** sert à la création des dictionnaires des langues. Ainsi, chaque fonction dans cette classe a son propre rôle bien définit. La fonction lireF permet de lire un fichier et d’en retourner seulement les lettres d’un texte, sans toutes les “nuisances” qu’il contenait telles que les chiffres ou les symboles de ponctuation. LireP fait la même chose mais avec une phrase. Les fonction **occ\_uni(fichier)** et **occ\_bi(fichier)** vont chercher les uni-grammes et les bi-grammes respectivement sur chaque String retournée par **lireF(fichier)** ou **lireP(string)** et nous retourne un dictionnaire. Ces fonctions en terme de complexité ne sont pas les plus optimales mais nous semblaient être les plus intuitives et donc les plus simples à écrire. Ce dictionnaire est composé des uni-grammes et bi-grammes comme clé et du nombre d’apparition dans le texte comme valeur. Il est ensuite traité par la fonction **ecrireDico(fichier)** qui va donc écrire le contenu de ce dictionnaire au format JSON dans un fichier texte. Le fait d’avoir utilisé le JSON nous vient seulement du fait, qu’il est plus rapide et facile à traiter lors de la lecture de celui-ci. Ce processus de lecture de dictionnaire se met en place si un dictionnaire du même nom que celui que nous essayons de créer existe. Si il en existe un qui possède le même nom, alors une lecture de celui-ci va se faire dans **lireDico(fichier)** et un nouveau dictionnaire va être créée. Ce dictionnaire va être utilisé dans **ecrireDico(fichier)** pour incrémenter le nombre d'occurrence des lettres qui sont en commun entre le nouveau et l’ancien dictionnaire. Ainsi, le dictionnaire de la langue est seulement réécrit avec les valeurs incrémentées ou non. Pour éviter les répétitions ou l’utilisation de données provenant d’un autre dictionnaire, nous avons écrit la fonction **effacerOcc()**. Cette fonction efface juste le dictionnaire occ qui est une variable de classe à chacun de ses appels. Les fonctions **completeDico(fichier1,fichier2)** et **supprZeroDico(fichier)** modifient le dictionnaire rentré en argument. La première complète un dictionnaire en ajoutant les uni-grammes et les bi-grammes présents dans un autre dictionnaire mais pas dans le premier. Ces uni-grammes et bi-grammes ont alors une valeur égale à 0. La deuxième fonction supprime tous les éléments du dictionnaire rentré en argument ayant une valeur égale à 0. Ces fonctions vont surtout nous servir pour les calculs vectoriels et donc pour le prochain fichier.

Le troisième, **Vecteur.py** est composé de toutes les fonctions de calcul et de créations des vecteurs. La fonction **recupVect(fichier)** ne récupère que les valeurs du dictionnaire écrit dns le fichier pour les mettre dans une liste. Cette fonction est utilisée dans une autre fonction appelée **normeVect(fichier)** qui fait le calcul de norme du dictionnaire, ce qui nous donne la norme du vecteur du fichier et qui nous est utile pour le calcul de similarité cosinus. Cette autre fonction, nommée **simCosinus(fichier)**, exécute la formule os(l) = (a\*b)/(||a||\*||b||) avec a et b deux vecteurs, ||a|| et ||b|| les normes des vecteurs respectivement. Ce qui revient à faire le produite scalaire divisé par le produit des normes, car le produit scalaire est égal à a\*b = cos(a-b)\*||a||\*||b||. Cette fonction va nous donner un résultat compris entre -1 et 1, comme expliqué dans la partie expériences et résultats. Pour faire le produit scalaire dans la fonction **simCosinus(fichier)**, nous avons créé une fonction **produitScal(fichier1,fichier2)**. La formule du produit scalaire peut être écrite de différentes manières, celle que nous avons choisi d’utiliser pour notre fonction est : a\*b = (x,y)\*(x’,y’) = x\*x’ + y\*y’ . Cette fonction récupère donc les dictionnaires de deux fichiers et va multiplier les valeurs des deux dictionnaires ayant le même uni-gramme ou bi-gramme. Par exemple : imaginons que les dictionnaires du français et de l’anglais ont un unique bi-gramme qui est “my”. Le dictionnaire anglais va avoir une valeur égale à 3500 pour ce bi-gramme alors que le dictionnaire français a une valeur égale à 1000. Le produit scalaire du dictionnaire français et du dictionnaire anglais est donc 3500\*1000 = 3 500 000. Donc cette fonction fait ce calcul à chaque fois et on fait l’addition de tous ces calculs pour avoir le résultat final du produit scalaire.

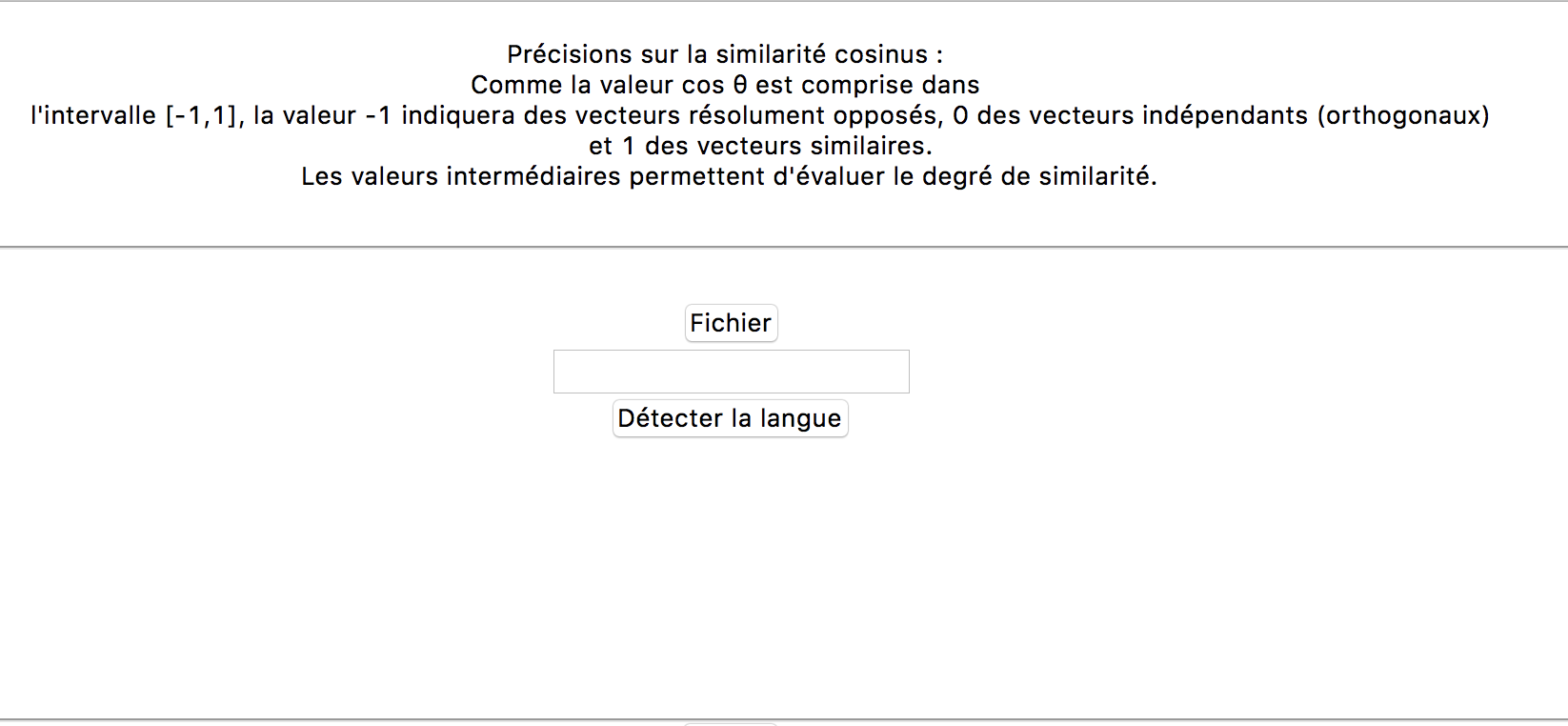
Et enfin le quatrième fichier, **Final.py**, sert à gérer l’interface graphique, les actions possibles de l’utilisateur ainsi que tous les tests liés au résultat que donne le programme. Cette classe est la plus importante dans l'exécution et l'enchaînement des actions. Nous avons, à l’ouverture du programme, un processus qui se lance en arrière plan et qui vérifie si les dictionnaires des langues, définies dans un tableau, existes. Si oui, il les supprimes d’abord, sinon, la première fenêtre de notre interface s’ouvre. Nous supprimons à chaque nouvelle exécution du programme les dictionnaires existant pour éviter de fausser les résultats. Toute l’interface graphique a été faite avec le module Tkinter.

Fenêtre 1

: 

Depuis celle-ci, si l’utilisateur clique sur le bouton “**Acceder au programme**”, la méthode **startProg()** va s’exécuter et démarrer la création des dictionnaires des différentes langues de notre corpus. Cette opération prenant un petit temps, nous avons donc pensé à ajouter une barre de progression qui permet de donner un indication visuelle sur l’avancé du processus et qui ajoute un petit point esthétique à notre projet. Après cela, une deuxième fenêtre s’ouvre.

Fenêtre 2:



Ici, si l’utilisateur clique sur “**Fichier**”, la fonction **openFile()** va s’exécuter et va sauvegarder le chemin du fichier que l’utilisateur aura choisis. Le bouton “**Détecter la langue**” est relié à la fonction **checkString()**, qui va vérifier que toutes les conditions sont remplies correctement avant de pouvoir lancer soit **choiceText(fichier)** si l’utilisateur a choisi un texte, soit **choicePh(string)** si il a écrit une phrase. La fonction **choiceText(fichier)** va appeler la méthode **lireF(fichier)** de la classe **Token.py** pour pouvoir créer un dictionnaire de fichier demandé. La fonction **choicePh(string)** va quant à elle lancer la méthode **lirP(string)** de **Token.py** avec comme paramètre la phrase écrite dans la zone de texte. Une fois l’une ou l’autre des actions effectuée, la dernière fonction **result()** est exécutée et les calcules vectoriels sont effectués avant d’afficher le résultat trouvé dans la fenêtre. Dans cette dernière fonction, on crée d’abord un dictionnaire pour le fichier, ou la phrase, dont le programme doit donner la langue. On doit y faire plusieurs tests sur les valeurs des différentes similarités cosinus entre les langues de nos corpus et le fichier, ou phrase, à évaluer. Le nombre de test augmente beaucoup pour l’ajout d’une seule langue dans nos corpus.

Problèmes rencontrés

Nous avons eu affaire à quelques difficultés pendant notre période de programmation du projet :

* **Problème d’encodage des textes** :

Nous avons pris pas mal de temps à essayer d’empêcher l’ apparition de caractères inhabituels dans nos dictionnaires. Nous avons tout d’abord essayé de changer l’encodage des fichiers textes lors de la lecture de ceux-ci. Mais cette méthode n’a pas tout résolu et nous avions toujours des caractères étranges. En faisant des recherches nous avons réussi à trouver une solution simple et efficace en terme de longueur de code. Nous avons utilisé une regex avec la fonction re.sub(), qui permet d’échanger un certain caractère avec un autre ou avec la chaîne vide “”. Cette fonction nous a permis de ne garder que les caractères essentiels à notre programme.

* **Problème d’écriture des dictionnaires** :

Nous avons également dû nous renseigner beaucoup sur l’écriture et la lecture des fichiers. Malgré les cours que nous trouvions à ce propos, nous n’étions pas capable d’obtenir un dictionnaire qui représentait bien toutes les occurrences des lettres que l’on rencontre dans les langues. Encore une fois, suite à de nombreuses recherches faites sur internet, nous avons trouvé que le langage JSON est plus simple pour écrire et lire en terme de code. De plus, cela nous sauve un peu de temps d’exécution du programme.

* **Problème de création des vecteurs** :

Comme il a été précisé plus tôt dans le rapport, nous avons beaucoup de mal au départ à comprendre comment créer les vecteurs. Il est difficile d’imaginer un vecteurs à plus de trois dimensions car les cours de mathématiques s’arrêtent généralement à cette dimension. Cela a été un frein pendant un moment jusqu’à rencontrer à nouveau M. Segonne. Ses explications nous ont permis de mieux nous représenter les vecteurs et de mieux aborder cette partie du projet. Nous avons aussi demandé conseil à un autre groupe qui faisait le même sujet pour avoir leur avis sur comment gérer cette partie.

* **Problèmes de calculs** :

Le plus compliqué dans les calculs a été de choisir la meilleure version des formules. Toutes les formules pour un calcul ne se ressemblent pas obligatoirement et certaines étaient plus compliqués que d’autres. Au départ, nous étions lancées dans des calculs difficiles à écrire et donc nous avions souvent des erreurs. Du moment où nous en avons pris conscience, nous avons tout changé et tout simplifié. Ce qui nous a permis d’avoir des résultats pertinents.

* **Problème d’affichage des fenêtre et de la barre de chargement** :

Au début de notre création d’interface graphique, nous avions fait un fichier, séparé de tous les autres .py, qui ne comportait que du code lié au module tkinter. Mais quand nous avons essayé de l’intégrer à nos fonctions, nous avons rencontré des problèmes d’importation du module et cela devenait compliqué de refaire tout notre code de l’interface graphique. Nous sommes donc partis sur l’utilisation d’une seule classe, Final.py qui regroupe l’interface des deux fenêtres et les méthodes liées au actions des boutons. Pour le côté un peu plus esthétique et interactif, nous avons rajouté une barre de chargement. Elle ne fonctionnait pas avant de fusionner nos deux fichiers, mais une fois la fusion effectuée, l’implémentation de celle-ci était relativement facile.

* **Problème d’affichage du résultat** :

Nous n’avons pas réussi à faire en sorte que le résultat qui s’affiche en bas de la fenêtre disparaisse entre deux détections de texte. C’est tout à fait faisable mais le temps nous a pris de cours sur les 2 derniers jours.

CONCLUSION

Lors de ce projet nous avons pu découvrir ce qu’est vraiment le traitement automatique du langage et en quoi il est différent de l’informatique pur. Ce fut une première étape très importante pour nos études et elle fut satisfaisante car nous avons eu un certain engouement envers celui-ci. Le fait de le faire en python et pas en JAVA nous a permis d’apprendre ou de mieux approfondir ce que nous connaissions de ce langage de programmation. Celui-ci est très facile d'utilisation et possède de vrai atouts, notamment avec la gestion et la création des dictionnaires. La gestion des fichier est aussi intéressante grâce aux différentes façons et modules qu’a ce langage.

Faire ce projet nous a aussi permis de renforcer notre capacité à travailler en groupe et à nous organiser. Nous avons été confrontées à de nombreux défis, que ce soit au niveau du code ou des calculs des vecteurs ou encore de la création de l’interface graphique. Il est des plus satisfaisant de voir que notre programme fonctionne, avec seulement un problème d’affichage de résultat du texte comme cité plus haut.

Nous aurions aimé faire plus de langues, mais du temps a été perdu sur le code et les différents bugs. En revanche, les textes pour le Néerlandais sont présents dans le dossier Train, mais pas implémentés dans la vérification de la similarité cosinus.

Si nous devions revenir sur ce projet, nous aurions fait une interface graphique plus propre au niveau du code, avec un \_\_init\_\_ ou une meilleure implémentation. Il aurait également été intéressant de trouver une manière de récupérer les textes plus efficacement que la méthode actuelle. Nous avions pensé à faire une fonction qui récupérerait les noms des fichiers des différentes langues et qui les ajouterait au fur et à mesure dans un tableau. Cette méthode aurait pu remplacer notre tableau contenant les différents langages, entré à la main.

La façon dont nous affichons les résultats finaux n’est pas non plus optimale, nous aurions sûrement dû créer une fonction de vérification et utiliser celle-ci dans les différentes conditions.

Si nous voulions aussi rendre le projet plus précis, nous aurions pu augmenter les n-grammes et ne pas s’arrêter aux bi-grammes.

ANNEXE 1 : MANUEL UTILISATEUR

Le but de ce projet est d'avoir un programme capable de détecter la langue d'un texte.

## Organisation du projet

Notre projet s'organise simplement :

**La racine**

|\_\_\_\_\_\_\_\_\_\_ **DataSet**

| |\_\_\_\_\_\_\_ **Evaluation** -> Fichiers texts qui nous permettent de tester notre programmes après l'avoir entraîné

| |\_\_\_\_\_\_\_ **Test** -> Fichiers texts qui nous permettent de tester notre programme en démonstration

| |\_\_\_\_\_\_\_ **Train** -> Fichiers texts qui entraînent notre programmes à construire les dictionnaires des langues

|

|\_\_\_\_\_\_\_\_\_\_ **LangDetection**

| |\_\_\_\_\_\_\_\_\_\_\_ Final.py

| |\_\_\_\_\_\_\_\_\_\_\_ ReadFiles.py

| |\_\_\_\_\_\_\_\_\_\_\_ Token.py

| |\_\_\_\_\_\_\_\_\_\_\_ Vect.py

|

|\_\_\_\_ Main.py

|\_\_\_\_ README.md

Si vous souhaitez ajouter des langues à détecter par le programme il vous faudra ajouter vos corpus dans un dossier portant les trois premières lettres du langage dans Train. De plus, il est recommandé de changer le fichier Final.py du code pour ajouter celle-ci.

## Faire tourner le programme

Dans un terminal il suffit d'entrer la commande `**python3 Main.py**`

Vous verrez une fenêtre apparaître dans laquelle vous pourrez choisir entre lire ce README ou bien ouvrir une autre fenêtre sur laquelle vous avez la détection de langue.

### Fenêtre "Détecteur de langue"

Dans cette fenêtre vous pourrez :

### 1. Cliquer sur le bouton `fichier`

Celui-ci permet d'aller cherche un fichier avec une extension .txt dans votre ordinateur.

Une fois le fichier récupéré, cliquez sur `Détecter la langue` et le résultat s'affiche en bas de la fenêtre.

### 2. Ecrire du texte dans la zone dédiée

Cette zone de texte est utilisée par le programme pour identifier la langue de ce que vous avez tapé.

Une fois le ou les mots entrés, cliquez sur `Détecter la langue` et le résultat s'affiche en bas de la fenêtre.

### Précautions d'utilisation

Vous ne devez pas choisir un texte et entrer des mots dans la zone de texte, choisissez seulement un des deux moyens de détection.

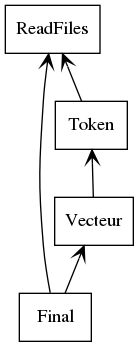
Vous ne pouvez pas entrer de numéros dans la zone de texte.

Vous ne pouvez pas lire de fichier dans un autre format que .txt

## Fermer le programme

Si vous souhaitez fermer le programme, vous pouvez appuyer à tout moment sur le bouton `Quitter`

ANNEXE 2 : MODÉLISATION DU PROJET



SOURCES

Sitographie :

1. Recherche de corpus :

* <http://fr.bilinguis.com/>
* <http://www.livresallemands.com/Divers/livresallemandsgratuits.html>
* <https://allemand.org/lecture/>
* <https://lingua.com/fr/italien/lecture/>
* <https://sherlock-holm.es/stories/pdf/a4/1-sided/stud.pdf>
* Transcriptions de discours politiques de Donald Trump, Barack Obama, Angela Merkel
* Transcriptions de discours scientifiques
* Sous-titres provenant du site <http://www.opensubtitles.org/>

1. Aide à la création du programme :

* <https://hal.archives-ouvertes.fr/hal-00874280/document>
* <http://www.statelem.com/similarite_cosinus.php>
* <https://medium.com/@paritosh_30025/natural-language-processing-text-data-vectorization-af2520529cf7>
* <https://fr.wikipedia.org/wiki/Produit_scalaire>
* <https://fr.wikipedia.org/wiki/Similarit%C3%A9_cosinus>
* <https://www.youtube.com/watch?v=d5NnUpwZjUM>
* <https://www.youtube.com/watch?v=2lq6rHomDUE&t=15s>