Rapport de projet L3S5 : TNM3A03A : Linguistique de Corpus

Ousseynou GUEYE

December 19, 2017

CONTENTS

CHAPTER 1

DÉFINITION DES OBJECTIFS

1.1 Scénario

Une maison d'édition reçoit des centaines de manuscrits par semaine. Sur suggestion, elle fait appel à un programmeur en herbe pour créer un petit programme qui permettrait de faire un premier tri.

Spécialisée dans la littérature ancienne, elle dispose d'un corpus de référence, composé de textes de V. Hugo, A. Dumas, Montaigne, et Homère. Un jour, elle reçoit un ensemble de texte de notre professeur, Mr Paroubek.

1.2 Matériaux

Nous disposons donc de l'ensemble des cours de L3 de Mr Paroubek sur l'année 2016-17[1], ainsi que d'un corpus de texte littéraire de grands auteurs[2].

1.3 Cahier de charge

Il s'agit pour nous de créer un programme en langage python[3] capable, après normalisation des fichiers, d'effectuer des analyses statistiques de base sur un corpus (deux en fait), d'en faire l'étiquetage morpho-syntaxique, et d'exporter le résultat sous forme d'un fichier xml.

[1] disponible sur *www.perso.limsi.fr/pap/inalco*
[2] Les textes sont disponibles sur *www.gutemberg.org*
[3] Python3

Mais aussi, nous voulons comparer les deux corpus pour voir si le corpus reçu passe le test statistique.

À priori, nous nous attendons à une différence nette entre un corpus littéraire et un corpus de diapositives destinées à des cours d'informatique.

CHAPTER 2

ORGANISATION DES DOSSIERS ET DU CODE

2.1 Approche

Le défi était plutôt conceptuel et organisationnel que purement technique. Non que nous nous considérions comme des programmeurs ou des utilisateurs de python3 confirmés, néanmoins si le professeur estime que des étudiants, trois après leur initiation, sont capable de produire un programme, il est raisonnable de penser que quelques années d'expérience ne font pas de mal.

Cela ne diminue en rien la quantité de travail fournie, au contraire. Comme le dit le proverbe «à qui beaucoup fut donné, beaucoup sera demandé. »

Dans un premier temps, nous avons programmé un ensemble de petites fonctions, souvent ne faisant qu'une seule chose. Cette phase a duré environ trois semaines. Ce temps fut comme un brainstorming, et nous a aussi servi à réfléchir au processus global.

Après cela, nous avons commencé à faire interagir ces fonctions les unes les autres, en les adaptant. Ce fut la phase la plus demandante, car il y a avait toujours un écart entre ce que l'on pensait avoir fait, et ce que nous avions effectivement fait. Nous y reviendrons dans notre chapitre 5.

Puis cette dernière semaine fut consacrée au nettoyage du code, sa documentation, ainsi que l'écriture de ce rapport.

2.2 Organisation des dossiers

AVANT EXÉCUTION DU PROGRAMME

- **PROJECT_ROOT/**
  - code_source/
    - modules/
  - corpus_litterature/
  - corpus_professeur/
  - from_outside_treetagger/
  - morphalou/
  - rapport/

APRÈS EXÉCUTION DU PROGRAMME

- **PROJECT_ROOT/**
  - code_source/
    - modules/
  - corpus_litterature/
    - **statistiques/**
    - **xml/**
    - **pdf/**
  - corpus_professeur/
    - **statistiques/**
    - **xml/**
  - from_outside_treetagger/
  - morphalou/
  - rapport/
  - **resultats_xml/**

Les noms nous paraissent assez explicites pour ne pas être tous détaillés. Néanmoins, certains dossiers nécessitent une petite explication.

- **from_outside_treetagger/** : au début de la programmation, l'accès aux éléments de morphalou était difficile. Pour ne pas perdre de temps, nous avions décidé d'utiliser aussi tree_tagger.

  Celui-ci ayant besoin de fichiers particuliers pour fonctionner, nous les avons regroupés sous ce dossier.

- **corpus$_x$/statistiques/** : Contient les statistiques de base, ainsi que les distributions de mots de chaque fichier pris séparément, et du corpus en entier.

- **corpus$_x$/xml/** : Contient l'arbre xml concernant le corpus.

- **resultats_xml** : S'appuyant sur les deux dossiers précédemment cités, ce dossier contient toutes les données que nous avons pu recueillir.

2.3 Présentation des modules

La partie programmée est regroupée dans le dossier *code_source*.

code_source/

- main.py
- settings.py
- modules/
  - big_process.py
  - ponctuation_texte.py
  - tagging.py
  - statistiques.py
  - writing_in_files.py
  - others.py

Cette fois-ci, chaque fichier mérite que l'on vous dise ce qu'il contient.

De même, nous aimerions préciser la logique de notre mouvement (flow). C'est en la suivant que j'ai pu, de manière systématique naviguer à travers le code[1].

En plus des messages souvent clairs de l'interprète de commande, je pars toujours du niveau 0 vers le niveau 2.

Au début de chaque fichier.py du module se trouve la **liste des fonctions** qu'il contient.

Niveau 0 : Si ce n'est pour contrôler la présentation, le niveau zéro fut très peu sollicité en cas de bug.

- **main.py** : Contient l'interface. C'est le fichier qui englobe le tout.
- **settings.py** : Pour faciliter la *gestion des liens*, nous avons regroupé les liens vers nos dossiers dans ce fichier. Il ne contient aucune fonction, mais juste des variables.

Niveau 1 : Ce niveau est toujours sollicité, car c'est lui qui coordonne le programme, mais aussi donne les paramètres aux différentes fonctions du niveau 2. C'est le niveau le plus délicat à manipuler.

- **modules/big_process.py** : Ne contient que des *fonctions composées d'autres fonctions* des sous-modules. C'est le seul fichier du dossier modules/ qu'appelle le main. On peut dire qu'il sert d'interface entre le main.py et les autres modules.

Niveau 2 : contient toutes les fonctions élémentaires. Quand un problème apparait, il y a de fortes chances qu'un des fichiers de ce niveau doive être vérifier après le niveau 1.

[1] qui devenait de plus en plus gros.

- **modules/ponctuation_texte.py** : contient toutes les fonctions aidant à la *normalisation des textes*, surtout en ce qui concerne la *ponctuation*, notre plus gros problème durant ces deux mois.
- **modules/statistiques.py** : contient toutes les fonctions permettant de réaliser les *distributions* (lettres ou mots).
- **modules/tagging.py** : contient les fonctions relatives au *tagging*, et aussi l'*interaction avec morphalou.*
- **modules/writing_in_files.py** : contient les fonctions nous permettant d'*interagir avec les fichiers* (.txt ou .xml).
- **modules/others.py** : contient toutes les autres fonctions, celle d'*importation*, les *recherches*, les *créations de dossier*, ainsi que l'*exécution de scripts bash.*

2.4 Processus

Ce que nous dirons pour un élément (corpus, texte, ... , mot) est valable pour tous ses éléments-frères.

- **-** [**x**] : NIVEAU CORPUS
    1. [**x**] : Création dossiers *statistiques* et *xml*.
    2. [**x**] : Récupération liste des fichiers.
    3. [**x**] : Agrégation du contenu des fichiers pour en faire une seule string.
    4. [**x**] : Traitement de la string.
    5. [**x**] : Calcul des statistiques de base + distribution des lettres et mots.
    6. [**x**] : Création des fichiers *distribution* et *statistiques* du corpus.
    7. [**x**] : Écriture des résultats dans les fichiers créés en 6.
- **-** [**x**] : NIVEAU TEXTE
    1. [**x**] : Récupération et traitement du contenu du fichier.
    2. [**x**] : Calcul des statistiques de base + distribution des lettres et mots.
    3. [**x**] : Création des fichiers *distribution* et *statistiques* du texte.
    4. [**x**] : Écriture des résultats.
- **-** [**x**] : NIVEAU PHRASE et MOT
    1. [**x**] : Tagging selon treetagger.
       Pour chaque mot :
    2. [**x**] : Récupérer les données (treetagger, fichier statistique et morphalou) pour chaque mot.
    3. [**x**] : Mise sous forme xml.
    4. [**x**] : Création du fichier *rendu_de_*...xml.
    5. [**x**] : Écriture des résultats dans fichier 4.
- **-** [**x**] : À LA FIN
    1. [**x**] : Création du dossier *xml* à la base du dossier.
    2. [**x**] : Ouverture et copie des fichiers xml précédemment créés.
    3. [**x**] : Création fichier *dtd*.
    4. [**x**] : Tentative de validation.

CHAPTER 3

DIFFICULTÉS RENCONTRÉES

3.1 Liste des problèmes

Voici la liste des problèmes rencontrés et des solutions que nous avons trouvés. Pensant que de vous donner la date[1] ne serait pas utile, on vous les présente néanmoins par ordre chronologique du plus ancien au plus récent.

Num	Problème ou questionnement	Solution ou réponse
1	Pendant le processus de normalisation, *doit-on tout mettre en minuscule*, prenant le risque de perdre certaines informations (comme une indication de nom propre) ?	Nous avons d'abord penché pour la création d'un objet "texte" avec plusieurs éléments dont une liste avec les majuscules, et une liste sans. Cependant, dans notre travail-ci, les majuscules n'étaient pas pertinente, elles ont été enlevées.
2	*Les dictionnaires sont non classés.* Vu que nous devions travailler avec eux, c'était particulièrement inconfortable pour nos tests. On voulait pouvoir suivre les mêmes x éléments au courant des modifications.	Dans un premier temps, nous avons pensé à **utiliser des OrderedDict du module collections**. Néanmoins, plus d'un mois plus tard, on s'est rendu que l'utilisation des dictionnaires rendrait l'exécution du programme trop lourde, nous y reviendrons. Cela n'empêche que nous avons pu les manipuler .
3	Comment choisir les *noms de fonctions* pour qu'ils soient le plus explicites possible ?	On a choisi comme convention d'écrire `'action'+'format_sortie'+'format_entrée'`.

[1] que nous avons toujours consignée

Num	Problème ou questionnement	Solution ou réponse
4	*Qu'est-ce qu'une phrase* ?	Après réflexion, on a décidé de définir comme **phrase ce qui se situe entre deux points (`'.'`), points d'interrogation (`'?'`) ou points d'exclamation (`'!'`)**. Au delà du choix, cela a attiré notre attention sur la difficulté à définir des objets utilisés au quotidien.
5	*Comment importer nos textes* ?	Dans un premier temps, nous avons défini une fonction pour les importer sous forme de liste de ligne, ou de liste de mots. Mais cela devait presque aléatoire. Donc on a décidé d'**importer nos textes en une longue string** grâce à `'_'.join()` puis de partir de là.
6	Plutôt un questionnement : pourquoi le professeur utilise-t-il `f = codecs.open(...)` au lieu de `with codecs.open(...) as f` qui semble plus commode ?	Avec nos `try ... except ... else`, nous nous sommes rendus compte qu'il y a au moins un avantage : on peut certes mettre un `with codecs.open(...) as f` dans un bloc `try`, mais cela devient plus difficile de tester nos différentes sous-étapes.
7	Je ne m'attendais qu'à du français, ou au pire de l'anglais, mais notre distribution de mots a affiché des mots très bizarres et à forte occurence.	Rendu à ce point de notre travail (plus d'un mois), la crainte de devoir tout reprendre n'était pas réjouissante. En réalité, il n'y avait rien de bizarre, c'étaient des mots grecs abondamment utilisés par Montaigne. Mais cet épisode nous a **convaincu de l'utilité de toujours regarder les distributions** avant d'avancer.
8	Malgré l'utilisation de la classe `\w` du module `re`, je me suis retrouvé avec des 'mots' de la forme `_abcd_`.	Cela m'a conduit a **voir comment est définie cette classe** dans python3. Finalement, pour n'avoir que les caractères alphabétiques, j'ai utilisé `[^\W_]`.
9	*Comment lire la documentation* et accéder à Morphalou ?	J'ai lu une blague quelque part où l'on demandait à un programmeur pourquoi il recréait tout. Sa réponse fut «parce qu'aucun programmeur n'aime lire la doc, ni écrire une doc. » Sans aller jusque là, je dirai que ce fut très difficile, mais formateur de devoir trouver son chemin dans une dtd aussi longue.

10	Réflexions quant à **l'utilisation des dictionnaires**.	Au départ, nous avions écrit le programme de telle sorte que toutes les données soient stockés dans un dictionnaire à écrire plus tard dans un fichier. Cependant, vu la quantité de données, nous avons pensé que cette approche était risquée pour plusieurs raisons : sollicitation excessive de la mémoire, perte de toute nos données en cas de crash. Finalement, nous avons opté pour une solution qui nous semblait plus simple : **celle d'écrire au fur et à mesure nos données dans leur fichier**.

CHAPTER 4

RÉSULTATS

4.1 Fichier xml final

Le résultat du programme est inscrit dans le fichier xml qui se présente ainsi :

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE banque_donnees SYSTEM "synthese_xml_total.dtd">

<banque_donnees>
   <corpus type_corpus='corpus_*'>
      <text text_id='c.t.'>
         <sentence sentence_id='c.t.s.'>
            <word word_id='c.t.s.w.' word_form='xxx'>
               <morphology>
                 <lemme>       <![CDATA[ x ]]>      </lemme>
                 <POS_treetagger>  <![CDATA[ x ]]> </POS_treetagger>
                 <POS_morphalou>  <![CDATA[ x ]]> </POS_morphalou>
                 <genre>  <![CDATA[ x ]]> </genre>
                 <nombre>  <![CDATA[ x ]]> </nombre>
               </morphology>
               <statistics>
                 <nb_apparition_text>  <![CDATA[ x ]]> </nb_apparition_text>
                 <nb_apparition_corpus>  <![CDATA[ x ]]> </nb_apparition_corpus>
                 <frequence_in_text>  <![CDATA[ x ]]> </frequence_in_text>
                 <frequence_in_corpus>  <![CDATA[ x ]]> </frequence_in_corpus>
               </statistics>
            </word>
         </sentence>
      </text>
   </corpus>
</banque_donnees>
```

Chaque élément xml (du corpus au mot) peut être multiplié autant de fois que nécessaire.

4.2 Temps d'exécution

Nous avons testé le programme avec un nombre de phrases à traiter différent[1].

Nombre de phrases par fichier	Temps d'exécution approximatif
1 phrase	3 minutes
10 phrases	20 minutes
15 phrases	30 minutes
50 phrases	1 heure 30 minutes

4.3 Réponse à notre hypothèse initiale

Comme dit au point 1.3, il nous fallait voir si le corpus de Mr Paroubek passait notre premier texte.

Contrairement à ce que l'on pensait, au niveau du nombre de mots par phrase[2], la différence n'est pas significative :

Corpus	Nombre de mots total	Nombre de mots par phrase	Différence
Corpus littérature	2.570.695	22.80	+ 8 %
Corpus Mr Paroubek	23.163	21.07	- 8 %

Ainsi, basé sur ce critère, le corpus de Mr Paroubek passe le premier test.

[1]Si le nombre de phrases à traiter excède le nombre de phrases contenues dans le fichier, la variable s'ajuste automatiquement au nombre de phrases du fichier

[2]qui était notre principal outil de séparation

CHAPTER 5

LIMITES ET OUVERTURES

Vous me permettrez dans cette dernière partie d'être un peu plus personnel.

5.1 Limites

Si ce fut très instructif, nous sommes aussi conscient des limites de ce programme.

Parmi elles :

- Nous aurions aimé le tester sur tous les fichiers de nos deux corpus. Mais comme dit dans l'interface, cela prendrait au moins 24 à 48 heures. Au moment où le programme fut 'terminé', il ne nous restait pas ce temps-là.

- Nous avons du pour enlever manuellement toutes les informations des fichiers de *gutemberg.org*. Nous sommes conscient que cela ne devrait pas être le cas, d'où le fait de le mentionner explicitement, par respect pour leur travail.

- Nous n'avons toujours pas régler le problème, et de la ponctuation, et de la définition de mots, phrases. Nous avons fait un choix pour pouvoir avancer, mais la question mérite réflexion.

- Enfin, nous aurions aimé affiner le programme, le rendre plus interactif, proposer plus d'options... y travailler encore plus.

5.2 Ouverture et réflexions diverses

Malgré ce que nous avons dit plus haut, ce fut une expérience très enrichissante, et pas juste au niveau académique.

Si j'ai souvent programmé pour le plaisir, pour la première fois, j'avais un délai, un rapport à rendre et donc un peu plus de 'bonne' pression.

Parmi les points les plus marquants :

- Nous avons consacré beaucoup d'heures (plusieurs dizaines voire une centaine) à réfléchir, essayer, coder, mais chaque petit pas fut une grande récompense.

- Ce fut un exercice d'humilité et de lucidité.

  D'humilité, car par définition, à chaque lancement, on s'attend à voir le programme ne pas marcher. Au départ, on s'agace car l'égo en prend un coup, puis on trouve cela normal, et on finit même par bien le prendre, et en faire un défi de plus à relever[1].

  De lucidité, car on est obligé de bien s'exprimer. Le programme ne fait pas ce que l'*on veut qu'il fasse*, mais plutôt ce qu'*on lui dit de faire*. Il ne sert donc à rien d'invoquer le ciel[2] ou de s'énerver, mais plutôt il nous faut réfléchir à ce que l'on a donné comme instruction, et non ce que l'on voulait dire.

- Enfin, et nous terminerons par ce point, le meilleur conseil que nous ayons reçu depuis fort longtemps nous a été donné par Mr Paroubek **«Testez votre programme sur un corpus.»** Nous ne sommes pas prêts de l'oublier.

  Nous étions à mi-chemin lorsque ce conseil nous a été donné. Si dans la théorie, toutes nos fonctions faisaient leur travail, et s'appliquaient bien aux phrases-test que nous avions créées, il n'y en a pas une qui ait résisté au test sur corpus.

  Pendant deux jours, nous avons retouché, voire recodé toutes les fonctions, mais mieux vaut cela que de se rendre au dernier moment que rien ne marche, après y avoir consacré beaucoup de temps et d'énergie.

  Ainsi, dans l'informatique comme dans la vie, si la théorie a sa place, rien ne vaut l'expérience pratique.

***** FIN *****

[1] sauf si c'est le jour de la présentation. ☺

[2] sauf pour avoir l'inspiration.