Institut National des Langues et Civilisations Orientales

Département Textes, Informatique, Multilinguisme

Projet du cours Programmation itérative/récursive

MASTER

TRAITEMENT AUTOMATIQUE DES LANGUES

Parcours :

Ingénierie Multilingue

par

Jianying Liu

Année universitaire 2020/2021

TABLE DES MATIÈRES

Première partie

Présentation du projet

1. Introduction

1.1 Présentation générale du projet

Dans ce projet, nous essayons de réaliser une analyse automatique de texte à partir d'un corpus de recettes formatées en XML. L'idée principale est de considérer les recettes comme des algorithmes, modéliser la manière de calcul de la complexité en temps et en espace, pour finalement pouvoir les comparer avec le niveau de difficulté défini dans le XML.

Pendant notre réalisation, nous utilisons extrêmement la librairie Spacy pour analyser automatiquement le corpus, afin de pouvoir par la suite extraire les ingrédients, les opérations, ainsi que les relations éventuelles entre ces deux composants importants de recette. En utilisant une méthode symbolique, nous profitons de la liste d'ingrédients fournie dans le fichier pour établir le corpus d'entrainement et d'évaluation pour exercer la méthode statique CRF dans la tâche d'extraction des ingrédients.

Ensuite, afin d'évider le plus possible l'erreur, nous décidons d'utiliser les ingrédients annotés par méthode symbolique pour extraire les opérations liées, et finalement les utiliser pour le calcul de complexité. Cette phase nous demande de lire et étudier le contenu du corpus, afin d'avoir une idée sur la structure des phrases présentées ainsi que les mots utilisés souvent dans ce genre de corpus, qui nous permet d'extraire un lexique des opérations et les patrons à chercher pour lier les opérations avec les ingrédients. À partir de ce constat, nous arrivons finalement à construire notre pipeline de détection des ingrédients et des opérations.

Finalement pour calculer la complexité de chaque recette, nous modélisons des règles à suivre, en distribuant à chaque ingrédient et opération détecté des attributs différents, tels que la dénombrabilité de l'ingrédient, la relation entre son nombre et le nombre de récipient ainsi que le nombre d'opération atomique, l'unité spatiotemporel de chaque opération, etc. Ces règles nous permettent de généraliser finalement deux fonctions linéaires de complexité en temps et en espace. Nous arrivons finalement à étudier la corrélation entre ces complexités calculées et le niveau de difficulté défini par humain pour chaque recette.

1.2 État de l'art

La nourriture est un aspect vital dans notre vie, ce sujet est incontournable pour tout le monde et donc intéresse fortement le public. De nos jours, considérant l'essor des appareils électroniques et la technologie d'intelligence artificielle, le besoin sur l'aide de machine sur l'alimentation se développe aussi, d'où vient l'intérêt de recherche. Pour permettre à la machine de nous fournir des suggestions

sur l'alimentation, il faut qu'il puisse analyser automatiquement les informations alimentaires. Ayant ce but final d'application, beaucoup de recherches ont été réalisés sur ce thème, y compris l'analyse automatique des recettes de cuisines. Dans le domaine de vison par ordinateur (Computer Vision), Luis Herranz *et al.* (2018) ont créé un framework pour reconnaître automatiquement les aliments et effectuer ensuite une analyse des recettes, ils ont présenté aussi les nouvelles tendances dans ce domaine et montrés que les connaissances externes pourraient beaucoup aider la machine pour cette tâche.

S'agissant du traitement automatique de langues, l'analyse des recettes textuels de cuisine fait une tâche concrète dans le domaine de fouilles de textes. La campagne d'évaluation internationale *Computer Cooking Contest* se concentre sur le cas concret des recettes cuisines, et le défi DEFT2013 a aussi porté sur cette thématique. Parmi les 4 tâches proposées dans le défi, le 4e, qui est une tâche de l'extraction d'information, demande d'extraire les ingrédients d'une recette, ce qui est aussi une des tâches dans notre projet.

Parmi les participants, Luca Dini *et al.* (2013) ont utiliser une approche hybride pour la détection des ingrédients : ayant déjà une liste des ingrédients, ils effectuent un traitement symbolique avec regex sur les ingrédients listés dans un premier temps, et ensuite réalise un traitement statistique sur le résultat du premier stade leur permet d'enrichir les résultats obtenus.

Deuxième partie

Réalisation du projet

1. Introduction

Afin de réaliser notre tache de reconnaissance automatique des ingrédients et des opérations pour finalement calculer la complexité, nous avons diviser la tâche en 5 partie : 1) détection des ingrédients dans la liste des ingrédients fournis; 2) à partir de ce résultat, entraîner un algorithme CRF pour la détection des ingrédients; 3) basant sur les ingrédient trouvés dans la recette et des caractéristiques linguistiques, détecter les opérations et la relation éventuellement existe; 4) modéliser la manière pour calculer la complexité et le réaliser en annotant finalement le fichier xml; 5) calculer la corrélation entre la complexité obtenu et le niveau de difficulté jugé par l'humain.

2. Détection des ingrédients

2.1. Détection à partir de la liste des ingrédients

2.1.1. Départ de l'idée

La manière la plus simple et la plus efficace pour détecter les ingrédients est d'avoir une liste de tous les ingrédients possibles ainsi que toutes ses formes possibles pour ensuite chercher dans le texte les formes listées dans le lexique. Comme ce qui est mentionnés dans les articles de recherche, ces ressources externes peuvent améliorer fortement le résultat.

Néanmoins, malheureusement, dans notre cas, nous n'avons pas directement une telle liste. Par conséquent, il nous faut tout d'abord essayer de créer cette liste.

Par contre, dans notre corpus de recette, à côté de la recette textuel, nous disposons d'une liste des ingrédients spécifiques pour cette recette, accompagnés éventuellement d'une quantité dont il a besoin. En plus, la quantité dont l'on a besoin pour le calcul de complexité est souvent préciser dans cette partie au lieu d'indiquer/répéter dans le texte de recette. De ce fait, nous décidons de profiter de cette liste pour ces informations, et essayer d'extraire les ingrédients à partir de ce texte beaucoup plus structuré que celui de « préparation ».

2.1.2. Études de structures

La figure suivante montre un exemple de cette liste.

```
<ingredients>
  <p>Quelques belles feuilles de laitue lavées et essorées</p>
  <p>500 g de tomates cerises</p>
  <p>4 belles tranches de saumon fumé </p>
  <p>1 grand Carré Frais</p>
  <p>1 pot d'oeufs de lompe</p>
  <p>1 poire jaune bien tendre</p>
  <p>2 cuillères à soupe de crème fraîche épaisse</p>
  <p>sel, poivre du moulin</p>
  <p>1 jus de citron</p>
  <p>aneth fraîche mixée </p>
</ingredients>
```

Même si cette partie a l'air d'être beaucoup plus structurée que le texte recette, il a quand même beaucoup de variantes. Après avoir parcourir une partie de corpus, nous constatons principalement les structures linguistiques suivant dans la liste :

1) Chiffres/nombre général (comme « quelques ») + unité de quantité + « de » + ingrédient ;
   e.g. : 1 pot d'œufs, quelques feuilles de laitue
2) Chiffres + ingrédient ;
   e.g. : 1 pomme
3) Ingrédient_1 + séparateur + ingrédient_2 + séparateur … (le séparateur peut être « , » ou « + » ou « et ») ;
   e.g. : sel, poivre
4) Ingrédient (le nom tout seul, sans d'autres informations) ;
   e.g. : aneth fraîche mixée
5) Ingrédient + explication de son état ou de son but entre parenthèses;
6) Ingrédient + « ou » + ingrédient (pour indiquer un choix possible entre deux ingrédients);
   e.g. : 3 pommes de terre ou patate douce
7) Unités d'ingrédients (sans les chiffres précisant la quantité) + ingrédient;
   e.g. : feuilles de laitue
8) Ligne indiquant le but d'usage des ingrédients suivant, qui ne contient pas d'ingrédient lui-même;
   e.g. : Pour faire la pâte
9) Ligne indiquant les ustensiles spécifiques et ne contient pas d'ingrédient.
   e.g. : 1 fourchette pour écraser
   …

Parmi tous les cas principaux, les deux premiers sont les plus fréquents. Autrement dit, la nombre indiquant la quantité et l'unité de quantité sont fortement présents dans cette partie et ils peuvent nous servir comme des délimiteurs de frontière des ingrédients. De plus, pour calculer par la suite la complexité, les quantités et l'unité peuvent nous servir pour définir le nombre d'opération et le nombre de récipients.

2.1.3. Transformation de la question en détection des unités

Comparé aux formes des ingrédients, celles des unités d'ingrédient sont beaucoup moins nombreux, donc il est possible pour nous de créer manuellement une liste des unités à partir d'un corpus petit, et puis l'utiliser dans un plus grand corpus. Nous avons créé cette liste basant sur les 100 premières recettes. De cette manière, nous pouvons transformer notre problème de détection des ingrédients à une tâche plus faisable : détections des unités.

Après ce stade de définition des unités, nous avons utilisé la librairie Spacy pour les détecter dans le corpus. Spacy est un outils TAL qui annote automatiquement le texte entré et qui nous permet de parser le texte ou matcher des patrons à partir de ces annotations. Une fonctionnalité spécifique de Spacy nous permet de détecter les entités à partir des patterns donnés. Nous utilisons donc le lexique des unités prédéfinis pour construire cet ensemble de patrons. Basant sur cette lexique, Spacy va chercher les entités en ignorant la casse des lettres, ou même lancer la recherche sur les lemmes de tokens. Nous disons donc à l'outils de chercher le lemme pour le premier token et la forme pour le reste dans une unité prédéfinie. Les quantités pourraient au pluriel ou au singulier donc on a besoin du lemme de 1er token, qui est souvent la tête de ce chunk. Mais pour une unité de plusieurs tokens, comme « cuillère à café », la partie reste sont plutôt fixé donc nous n'avons pas besoin d'inclure ses variantes.

2.1.4. Pipeline pour obtenir une liste des ingrédients d'une recette

Pour chaque texte entre la balise « p », nous effectuons un processus de traitement suivant :

1) Supprimer les contenus entre parenthèses et la partie après « ou » pour évider les formes problématiques;
2) Éliminer le texte commençant par « pour », puisqu'il est fortement possible qu'aucun ingrédient est inclus dedans;
3) Séparer le texte en des parties s'il contient des séparateurs « , » « et » et « + » ;
4) Entrer séparément les portions obtenues dans Spacy, pour chercher les unités;
5) S'il en a trouvé, nous prenons la partie après l'unité comme notre ingrédient, et le chiffre avant l'unité comme la quantité de l'ingrédient;
6) S'il n'a rien trouvé, nous choisissons la partie après le dernier chiffre (puisqu'on pourrait avoir des cas comme « 4 ou 5 pommes ») comme l'ingrédient, et le 1er chiffres comme la quantité;
7) Si ni l'unité ni un chiffre est présent, nous prenons toute la portion comme notre ingrédient.

Ensuite, puisque les ingrédients définit ici contient probablement des compléments, qui n'apparait plus dans le texte suivant, nous décidons d'élargir notre liste en choisissant seulement le 1er token. De cette manière, nous

obtenons une liste de tous les ingrédients qui devraient apparaître dans le texte de recette.

2.1.5. Utilisation de listes générées dans l'annotations

Après d'avoir pour chaque recette sa propre liste d'ingrédients, nous avons deux choix possibles :

1) Englober tous ces ingrédients dans une liste globale, et puis, quand nous annotons chaque recette, il peut utiliser les formes présentes dans d'autres recettes;
2) Utiliser directement cette liste spécifique à la recette pour l'annotation.

Puisque la liste des ingrédients vient d'une annotation automatique basant sur des règles simples, il est fortement possible qu'il y ait des cas non inclut et il contienne des bruits ou il manque certains ingrédients. Tous ces deux choix pourraient rencontrer des problèmes à cause de ces fautes. Le premier donne une priorité au rappel mais les bruits augmentent aussi. Le deuxième assure la précision du résultat mais il a plus de chance de rater certains ingrédients.
Nous avons testé les deux choix et constater que le l'augmentation de bruits est beaucoup plus importante que son amélioration apportée, puisqu'une faute faite dans une recette va ensuite causer des fautes sur toutes les autres recettes. Par conséquent, nous décidons d'annoter chaque recette avec sa propre liste générée. Nous transformons le texte annoté en une liste de tokens, contenant aussi le numéro de phrase et son étiquette de POS, pour servir à l'étape suivante : l'entrainement et l'évaluation d'un classifieur CRF pour détecter les entités nommées dans le texte.
Néanmoins, pour le calcul de complexité, nous utilisons le texte annoté ici. Premièrement, les ingrédients annotés ici sont presque 100% des « vrai » ingrédients, et pour les tâches suivantes nous donnons une priorité à la précision. Deuxièmement, notre recherche des opérations sera basée sur le résultat de parsing de Spacy, donc il est pratique de continuer l'opération sans l'interrompre. De plus, nous avons effectué des nettoyages d'espace avant d'entrer le texte dans spacy, donc ce sera difficile de garder une tokenisation identique avec deux lancements différents.

2.1.6. Problèmes rencontrés dans ce stade

Même si nous avons fait tous nos efforts pour englober le plus possibles des cas différents dans notre pipeline d'annotation, la variété infinie de la langue humaine nous emmène toujours des cas nouveaux, de plus, Spay peut aussi faire des erreurs pendant l'annotation, par exemple, le modèle « fr_core_news_lg » n'arrive pas toujours à détecter les chiffres. Ces deux inconvénients pourront ensuite détruire les relations dépendantes entre les règles définies et rompre notre programme. Nous avons dû introduire un traitement d'erreurs dans notre programme pour sauter ces fichiers troublants quand il est bloqué à cause de ces problèmes.

2.2. Détection des ingrédients : méthode statistique

L'annotation des ingrédients est une tâche concrète de reconnaissance des entités nommées. Pour ce type de tâche, le modèle statistique CRF (*conditional random fields* en anglais, champs aléatoires conditionnels en français) a souvent une performance satisfaisante, puisqu'il prend en compte le contexte autour de token à annoter. Nous utilisons donc ce modèle comme classifieur et les fichiers de sortie de la phase précédente comme le corpus d'entraînement et le standard pour l'évaluation.

2.2.1. Fichiers entrés pour l'entrainement

Dans notre projet, nous choisissons scikit-learn pour implémenter ce modèle, et nous fournissons à CRF seulement 3 genres d'informations : la forme de tokens, l'étiquette POS et le numéro de phrases. Toutes les recettes annotées avant sont concaténées dans un seul fichier. La figure suivante montre un morceau de ce fichier :

```
sent_id word    pos tag
1   Casser  VERB    O
1   le  DET O
1   chocolat    NOUN    B-INGRED
1   et  CCONJ   O
1   faire   AUX O
1   fondre  VERB    O
1   doucement   ADV O
1   avec    ADP O
1   la  DET O
1   margarine   NOUN    B-INGRED
1   .   PUNCT   O
2   Ajouter VERB    O
2   du  DET O
2   sucre   NOUN    O
2   (   PUNCT   O
2   selon   ADP O
2   les DET O
```

2.2.2. Features et paramètres choisis

Dans notre programme, nous regroupons les tokens dans différents phrases grâce à son sent_id. Quant à l'extraction des features, nous choisissons les informations sur sa forme (majuscule, minuscule, etc.) et son POS tag, ainsi que les informations de ses voisins. Pour les paramètres de CRF, nous utilisons ceux fournis dans l'exemple sur sa page de documentation.

2.2.3. Évaluation de sa sortie

Dans notre premier essai, nous donnons au classifieur un nombre total de 500 recettes et tester sur un autre sous-corpus de 100 recettes, en considérant les erreurs possibles existés dans notre corpus d'entrainement, le résultat de ce test est assez satisfaisant. Voici le rapport de cette évaluation :

	precision	recall	f1-score	support
B-INGRED	0.60	0.89	0.71	1051
I-INGRED	0.66	0.66	0.66	327
micro avg	0.61	0.84	0.70	1378
macro avg	0.63	0.78	0.69	1378
weighted avg	0.61	0.84	0.70	1378

Nous constatons qu'il a un rappel beaucoup plus élevé que la précision, cela pourrait provienne des bruits dans notre fichier entrée, et le classifieur les a appris comme des entités à annoter. Mais quand le fichier entré élimine correctement les tokens dans le même cas, ce trait faussement appris conduit à un faut positif de la part de classifieur.

Après cet essai, nous nous demandons si l'augmentation de la taille de corpus d'entrainement peut améliorer le résultat. Donc nous avons entraîné notre modèle avec un corpus d'une taille de 5000 recettes et testé le classifieur sur un autre 1000 recettes. Voici le résultat de cet essai :

	precision	recall	f1-score	support
B-INGRED	0.50	0.65	0.56	16236
I-INGRED	0.50	0.85	0.63	3710
micro avg	0.50	0.69	0.58	19946
macro avg	0.50	0.75	0.60	19946
weighted avg	0.50	0.69	0.58	19946

Le résultat a beaucoup plus baissé! Cette baisse de performance est étonnante à premier vue mais logique en réalité, prenant compte du fait que l'on n'a pas un corpus d'entraînement parfait. Le classifieur pourrait surapprendre sur ces fautes qui apparaissent beaucoup plus dans un corpus plus grand.

De ce fait, nous avons testé le classifieur sur plusieurs tailles et trouve qu'il se stabilise autour d'une f-mesure de 0,7 sans beaucoup d'écart entre la précision et le rappel. Cette meilleure performance vient de l'entraiment sur une taille de 50 000 phrases, qui est même plus petite que la taille de corpus pour le premier essai, qui contient environs 70000 phrases. Voici l'évaluation sur cette meilleure performance :

	precision	recall	f1-score	support
I-INGRED	0.68	0.59	0.63	377
micro avg	0.73	0.70	0.71	1456
macro avg	0.71	0.66	0.69	1456
weighted avg	0.73	0.70	0.71	1456

2.2.4. Problème rencontré dans ce stade

La mise en œuvre de CRF dans *scikit-learn* utilise un module indépendant

sklearn_crfsuite, qui demande une version ancienne que celle la plus récente -- 0,24. Et l'ancienne version ne soutient pas la fonctionnalité de train-test-split, il nous faut la réaliser par nous-mêmes. Et nous avons simplifié sa réalisation en exécutant les essais sur plusieurs fichiers, mais ce point pourrait être amélioré par rendre l'entrée aléatoire.

3. Détection des opérations

Dans ce 2e temps, nous essayons d'extraire les opérations dans la recette et les lier à ses ingrédients opérés. Cette relation nous aide à compter le nombre des opérations atomiques par la suite.

3.1. Détection des opérations

La détection des opérations dans une recette est beaucoup plus facile que celle pour les ingrédients. Puisque notre corpus se caractérise par une suite d'instructions, les actions à faire sont presque toujours les verbes au mode impératif ou sous forme infinitif. Nous pouvons annoter ces deux genres de verbe pour la majorité des cas.

Néanmoins, en constatant la sortie de ce traitement, nous trouvons qu'il existe encore une fois des exceptions. Pour le cas comme « laisser reposer » ou des verbes qui ne soit pas un mouvement comme « permettre », nous ne devons pas compter 2 fois l'opération ou même il ne faut pas les compter comme opération. En conséquence, nous définissons une petite liste des verbes à ignorer pendant l'annotation des opérations.

Troisièmement, dans notre cas spécifique, nous ne devons non plus compter les verbes après « pour » qui ne soit pas un vrai mouvement mais un objectif à réaliser. Et les verbes après « ou » sont aussi un deuxième choix pour une opération donc il faut les sauter afin de ne pas doubler le nombre d'opération.

3.2. Détection des relations

Pour lier les opérations avec les ingrédients, il nous faut d'abord avoir des ingrédients annotés dans le corpus. Comme ce que l'on mentionne avant, nos ingrédients sont annotés par Spacy en utilisant « pattern search », qui est définit à partir de l'extraction des ingrédients dans la liste précédant le texte recette.

En considérant la spécificité de recette, c'est-à-dire que les ingrédients sont très probablement les compléments d'objet direct ou les compléments d'objet indirecte du verbe de l'opération, nous établissons la relation entre ces deux entités à la base de cette relation dépendante syntaxique. Néanmoins, au lieu de spécifier les types de relation à chercher, nous décidons de chercher l'ingrédient annoté parmi tous les enfants du verbe. Puisque nous ne pouvons pas assurer que l'ingrédient est toujours annoté dans l'arbre de dépendance comme l'objet du verbe et ayant une relation directe avec ce verbe. Néanmoins, c'est presque 100% le cas qu'il soit un enfant de ce verbe, puisque les verbes

sont toujours l'entête de son groupe verbal. Par conséquent, notre choix élargit le domaine à chercher et rendre une plus grande possibilité pour trouver l'ingrédient possiblement existant. Un inconvénient causé par ce choix est que l'on pourrait trouver plusieurs ingrédients pour un seul verbe. Mais cette relation un-à-plusieurs n'est pas fortement une faute, puisqu'il est fortement possible que l'on ait des expressions comme « éplucher les pommes et les tomates ». De ce fait, nous supposons que ce genre de correspondance est toujours correctement annoté et cela révèle le fait que l'opération est exécutée sur tous ses ingrédients liés. Cette supposition aura une influence sur notre calcul de complexité en temps.

4. Modélisation du calcul de complexité

4.1. Complexité en temps

4.1.1. Explication de l'idée théorique sur les algorithmes informatiques

Théoriquement, pour un algorithme, la complexité en temps est $O(1)$ si le nombre d'exécution des instructions est inchangeable avec n'importe quelle entrée fournie. C'est le cas des fonctions qui n'ont pas de boucle. Ensuite, on multiplie le constant 1 avec un n pour chaque ajout d'un niveau de boucle. Autrement dit, une fonction ayant une boucle aura une complexité de $O(n)$, idem pour une fonction ayant plusieurs boucles qui succèdent l'une après l'autre, mais la complexité multipliée à $O(n^2)$ si nous avons 2 boucles dont une est imbriquée dans l'autre. Sa manière de multiplication pourrait changer si l'entrée a un effet exponentiel sur le nombre d'exécutions de boucle, comme pour les cas de $O(\log n)$.

4.1.2. Application du principe théorique dans notre tache

Ayant compris les principes de calcul de la complexité, nous pouvons revenir à nos moutons – modélisation la manière pour calculer la complexité en temps d'une recette. Puisque pour certaines opérations, comme « éplucher une pomme de terre », le nombre de cette actions effectuée est dépend du nombre de pommes de terre, autrement dit, l'action de l'épluchage se répète selon le nombre de pommes de terre à traiter et nous avons ici une boucle, donc nous avons ici une complexité linéaire $O(n)$, n soit ici le nombre de pomme de terre. Pour la plupart des ingrédients, les instructions sont séquentielles, et il existe rarement des opérations contenues dans une autre opération en considérant la vie réelle.

Néanmoins, à noter que $O(n^2)$ est également possible. Par exemple, nous pouvons rencontrer des recettes disant « répéter les étapes suivantes pour traiter toutes les pommes de terre » et parmi ces étapes à refaire il y a une instruction contient une opération répétitive au niveau atomique, nous aurons une boucle imbriquée dans l'autre. Mais cette situation est théoriquement possible mais réellement rare, de plus, le repérage automatique des opérations concernées dans ce cas est très difficile à réaliser, nous simplifions le traitement

de « répéter » en lui donnant une plus grande valeur pour son unité de temps et lui définit comme dépendant du nombre de l'ingrédient.

4.1.3. Modélisation concrète du calcul

Pour calculer le nombre d'opération atomique, il nous faut d'abord séparer les opérations en deux types : l'un qui va changer selon le nombre des ingrédients, comme le cas de « éplucher », l'autre qui est indépendant de la quantité des ingrédients, c'est le cas comme « parsemer du sel » ou « cuire les pommes ». Étant donné de cette réalité, nous définissons pour les opérations fréquentes dans la recette (repérées manuellement basant sur le constat de corpus) l'attribut de dépendance sur l'ingrédient.

De plus, nous imaginons l'existence d'un cas où nous ne devons pas multiplier le temps malgré l'existence de cette dépendance de l'opération au sens général : l'ingrédient est fourni de manière indénombrable. Afin de traiter cette différence, il nous faut aussi fournir à nos ingrédients trouvés un attribut de dénombrabilité. Le tableau suivant illustre nos règles pour cette définition à partir des informations repérées de la liste des ingrédients.

Quantité de l'ingrédient (le chiffre précis avant l'ingrédient) trouvée ?	Unité de quantité trouvée ?	Dénombrabilité	Exemple (D pour dénombrable et I pour indénombrable
Oui et ce n'est pas une fraction	Oui	Dénombrable	1 sachet de patate → D 2/3 bol de farine → I
Oui	Oui, mais ce sont des mesures comme g, ml, etc.	Indénombrable	250 g d'eau
Non	Oui	Indénombrable	Feuilles de laitue → I
Oui	Non	Dénombrable	3 pommes → D
Non	Non	Indénombrable	Sel → I

Néanmoins, dans la pratique réelle nous pouvons fortement tomber dans une situation hors de ces définitions, telle que le verbe trouvé ne présente pas dans notre liste limitée des opérations, ou malgré la dépendance de l'opération sur l'ingrédient lié, notre système n'a rien trouvé pour lui. De ce fait, nous sommes obligées de définir un ensemble de règles à suivre, qui essaie de prendre en compte tous les cas possibles. À noter que ce système de règles a simplifié

certains cas concrets pour que le calcul suivant soit faisable.
Nous expliquons ici l'ensemble de ces règles par le tableau suivant :

Règles pour les opérations : (X : nombre de multiplication de la recette)

Existence de l'opération dans le lexique prédéfini	Trouver des ingrédients relatifs?	l'opération influencé par l'ingrédient?	Nombre d'opérations (Si le nombre de l'ingrédient lié est n, cn=> dénombrable; uc =>indénombrable)	Temps réel dans la recette pour cette opération (Si le nombre de l'ingrédient lié est n)
Dans lexique	Oui	Oui	Un ingrédient trouvé : n*X pour cn, 1 pour uc	Temps_moyen * n * X pour cn
			Plusieurs trouvé : (somme des n de cn) *X + somme de 1 de uc	Temps_moyen * X pour uc
		Non	1	Temps_moyen
	Non	Oui	1	Temps_moyen * X
		Non		Temps_moyen
Hors lexique	Oui/Non		1	1

Nous introduisons également deux autres idées dans ce tableau. Tout d'abord le variable **X**, qui est le nombre de fois de tous les ingrédients pour une certaine recette. Par exemple, la recette présente la manière pour faire une seule omelette et nous voulons en réalité cuire 2 omelettes, théoriquement il nous faut une fois de plus d'ingrédients que la quantité mentionnée dans la recette, donc nous aurons ici X=2 pour calculer la complexité et le « temps réel » estimé. Le **n** en minuscule est la quantité d'un seul ingrédient indiquée dans la recette. Cette variable est en fait fixée pour une certaine recette précise. Par exemple, si nous avons une recette de curry qui indique 3 pommes de terre et 1 oignon dans sa liste d'ingrédients, le n sera fixé à 3 pour la pomme de terre et 1 pour l'oignon pendant notre calcul.
À part le nombre des opérations atomiques, la dernière colonne essaie de calculer le « vrai » temps dont chaque opération mentionnée dans la recette aura besoin. Ce calcul est basé sur un « temps moyen » prédéfini également dans notre liste des opérations. L'unité de cette durée donnée est fictive sans de vraie unité en temps, mais nous supposons une approximation d'une minute pour chaque unité pendant la définition. La figure suivante montre comment nous avons défini ces valeurs pour chaque type d'opération.

lemma	influence_ingred	espace	temps_moyenne
préparer	0	0	20
mélanger	0	1	5
découper	1	1	1
enlever	0	0	1
reposer	0	1	30
griller	1	1	5
rôtir	1	1	5
revenir	0	1	1
couper	1	1	1
ajouter	0	0	1

(Pour les colonnes « influence_ingred » et « espace », nous utilisons ici une valeur booléenne, donc 0 signifie False et 1 signifie True)

Évidemment, dans la réalité la durée de chaque opération est également déterminée par l'ingrédient sur lequel l'on pratique l'opération. Néanmoins, ici nous ignorons cette influence pour simplifier de nouveau la réalisation de tâche.

4.1.4. Démarche de calcul pour la complexité en temps

Ayant ces valeurs définies, nous pouvons finalement arriver à calculer la complexité en temps de pas en pas.

Si le nombre d'ingrédient est **n**, le nombre de fois de la quantité à préparer par rapport à celle dans la recette est **X** :

Le nombre total de tous les ingrédients seront :

$$N_{(ingreds)} = \sum_{i=1}^{j} n_i X + 1 * k$$

Dans cette formule, ***j*** est le nombre des ingrédients dénombrables dans la recette et ***k*** est le nombre d'ingrédients indénombrable.

Dans ce calcul, nous fixons le nombre des ingrédients indénombrables à 1, puisque nous pensons que le changement de la quantité du plat à préparer n'influence pas le nombre des ingrédients indénombrables. D'un côté, pour certains ingrédients comme « sel », le changement de quantité est presque négligeable ; d'un autre côté, puisqu'il est indénombrable, c'est difficile de les mesurer séparément, donc, dans notre tableau avant, nous ne prenons pas en compte l'effet du changement de ***X*** sur le nombre d'opérations atomiques si l'ingrédient est indénombrable. Et notre choix ici correspond au choix précédemment fait.

Ensuite, pour le nombre total des opérations atomiques, nous aurons :

$$N_{(opérations)} = \sum_{i=1}^{a} m_i$$

Ici, $\boldsymbol{m_i}$ est le nombre d'opérations atomiques de chaque mention de l'opération, comme ce qui est prédéfini avant dans le tableau, et $\boldsymbol{a}$ est le nombre des opérations trouvées dans le texte.

Ensuite, le coefficient de linéarité sera :

$$c = N_{(opérations)}/N_{(ingreds)} \quad (X = 1)$$

Puisque pour une recette précises, $\boldsymbol{m_i}$, $\boldsymbol{a}$, $\boldsymbol{j}$, $\boldsymbol{k}$ et $\boldsymbol{n_i}$ seront tous fixé à une valeur précise, et nous choisissons de calculer le coefficient en utilisant une unité de la recette donc $\boldsymbol{X}$ prendra la valeur 1[1], de cette manière, le coefficient sera un constant.

Finalement nous avons la fonction de la complexité en temps pour la recette :

$$f(X) = c * N_{(ingreds)}$$

Puisque toutes les autres variables seront fixées à une valeur précise sauf le $\boldsymbol{X}$ dans $\boldsymbol{N_{(ingreds)}}$, le $\boldsymbol{f(X)}$ sera certainement linéaire, et notre complexité en temps des recettes sera O(X), qui correspond à notre hypothèse de départ.

En plus, grâce à notre définition précédente sur la durée de chaque opération, nous pouvons maintenant calculer le temps moyen de l'opération atomique pour la recette annotée :

$$Temps\_avg\_OPBase = \sum_{i=1}^{o} temps\_réel_i \Big/ N_{(opérations)}$$

Ici, $\boldsymbol{o}$ signifie le nombre des mentions des opérations trouvés dans le texte.

Et nous arrivons finalement au calcul de temps estimé d'une certaine recette précise :

$$Temps_{estimé} = f(X) * Temps\_avg\_OPBase$$

4.1.5. Un dernier mot sur le coefficient

Selon notre définition des nombres des opérations et des ingrédients, il est possible pour tous les deux d'avoir une forme simplifié comme $\boldsymbol{aX + b}$, par conséquent, le réel coefficient sera sous forme $C(X) = \frac{aX+b}{cX+d}$ comme une fonction homographique, et elle change en même temps avec $\boldsymbol{X}$.
Pour ce type de fonction, $\Delta C(X)$ change radicalement près de $X = -d/c$ même s'il se stabilise vers la valeur a/c quand on va jusqu'à l'infini, nous ne

[1] Ce choix a en effet simplifié un peu le calcul de coefficient puisque notre manière pour définir les nombres des ingrédients et des opérations changera tout le temps le ratio de ces deux chiffres.

pouvons pas assurer que notre X=1 est dans ce domaine stable, est il est très probablement qu'il se trouve dans ce domaine instable. Donc le résultat de temps estimé n'est pas très fiable si X n'est pas 1.

4.2. Complexité en espace

4.2.1. Explication de l'idée théorique sur les algorithmes informatiques

L'espace pris par un programme est déterminé par les variables qu'il prend comme l'entrée et les variables locales définissent à l'intérieur du programme. La complexité en espace nous permet donc de mesurer le changement de l'espace nécessaire pour l'exécution d'un certain programme. Et cette mesure est déterminée par la récursivité et l'utilisation de pile par le programme, ainsi que la dimension des tableaux définis. Pour un programme ayant un tableau d'une seule dimension, la complexité en espace est $O(n)$. Si aucune récursivité ou tableau dans le programme, la complexité sera toujours de $O(1)$.

4.2.2. Application du principe théorique dans notre tache

Pour appliquer ces idées théoriques dans le calcul sur les recettes. Nous traitons séparément les ingrédients utilisés et les opérations exécutées. S'agissant des ingrédients, ce sont des entrées de notre « algo » recette, et nous traitons les ingrédients dénombrables comme les tableaux donc ils prennent n récipients (n est le nombre de l'ingrédient traité). Et nous divisons les opérations en 2 types pour la modélisation : l'un qui prend 1 récipient pour l'exécution, il a l'air d'être la déclaration d'une variable locale à l'intérieur du programme; l'autre qui ne prend pas de récipient. À noter que nous simplifions notre tâche en ignorant la taille différente de chaque récipient.
Les deux tableaux suivants résument nos choix de modélisation pour ce calcul.

Opérations :

Existence dans lexique prédéfinie	Prendre de récipient?	Nombre de récipients
Dans lexique	Oui	1
	Non	0
Hors lexique		0

Ingrédients :

	Nombre de récipients
Dénombrable	n * X (si n est un int); 1*X (n est une fraction ou décimale)
Indénombrable	1

(X : nombre de multiplication de la recette; n : quantité de chaque ingrédient)
Ensuite, une simple addition de ces deux résultats pour la fonction de Complexité en espace :

$$f(X) = \sum_{i=1}^{c} n_i X + \sum_{i=1}^{b} X + u + o$$

Ici, ***u*** est le nombre des ingrédients indénombrables, ***o*** et le nombre des opérations qui prend un récipient, ***c*** est le nombre des ingrédients dénombrables ayant un entier comme sa quantité, ***b*** est le nombre des ingrédients aussi dénombrables mais utilisant une décimale pour la quantité. Nous redéfinissons sa quantité à 1 puisque l'on pense les récipients ne sont pas divisibles.

Troisième partie

Analyse de la corrélation et du programme

1. Calcul de corrélation

1.1. Réduction et équilibrage du corpus à étudier

Le corpus fourni contient des informations sur le niveau de difficulté pour chaque recette. La difficulté de recette est classifiée en 4 niveaux : *Très facile, Facile, Moyennement difficile* et *Difficile*. Néanmoins, après une analyse sur le nombre des recettes de chaque niveau, nous trouvons que le corpus est vraiment déséquilibré [2], et notre premier répertoire de fichier-sortie ne contienne même pas de recette du niveau « Difficile », donc un simple choix aléatoire de sous-corpus pourrait causé des problèmes après. De plus, puisque nous utilisons un pipeline de Spacy qui est plus juste mais un peu longue, cela prend du temps pour annoter tout le corpus.

Étant donnée de cette situation, nous décidons de parser d'abord tous les fichiers xml et les diviser en 4 classes selon leur niveau. Ensuite, nous utilisons le nombre du niveau qui apparaît le moindre comme le nombre de fichiers à annoter pour chaque niveau, pour utiliser finalement ces fichiers annotés pendent le calcul de corrélation.

S'agissant du calcul de corrélation entre le niveau et la complexité en temps et en espace. Nous donnons à nos fonctions de complexité une valeur X égale à 1 pour calculer le nombre d'opérations et le nombre de récipients nécessaires, ensuite calculer la moyenne de ces résultats pour chaque catégorie. La complexité en deux est une simple addition de ces deux moyennes.

Nous avons aussi pensé d'étudier simplement le coefficient, dans le but de généraliser le résultat de l'étude. Néanmoins, comme ce qui est mentionné avant, le coefficient réel de notre fonction de complexité en temps change tout le temps donc cette valeur n'est pas un bon choix si nous voulons généraliser le cas.

1.2. Analyse du résultat obtenu

Voici le résultat obtenu après le calcul :

Niveau	Très facile	Facile	Moyennement difficile	Difficile
Nombre de fichiers	72	76	74	77
Complexité en temps	39.08	36.49	65.44	55.25
Complexité en espace	48.39	85.33	39.41	46.81
Complexité en deux	87.47	121.82	104.85	102.06

[2] Après avoir retiré 6100 fichiers pour créer des sous-corpus pour les étapes précédentes, le corpus reste contient 8647 fichier au niveau « Très facile », 7123 au niveau « Facile », 1116 au niveau « moyennement difficile » et 84 au niveau « Difficile ».

Le résultat obtenu se révèle une corrélation entre les niveaux et la complexité en temps. Si nous compare le résultat en regroupant les « très facile » avec « facile », « moyennement difficile » avec « difficile », nous pouvons trouver qu'il existe une corrélation : les recettes jugées comme difficile ont tendance d'avoir plus d'opérations atomiques à faire. Mais à l'intérieur de chacun de ces deux grandes classes, nous ne trouve pas telle corrélation positive. C'est peut-être parce que la distinction entre « très facile » et « facile », ainsi que celle entre « moyennement difficile » et « difficile », sont un peu délicats et subjectifs, et chacun suit ses propres critères, d'où vient le problème.

2. Analyse de complexité en temps d'une fonction

Nous choisissons ici le code « ComplexCalculator » pour analyser sa complexité en temps[3].

Nous commençons par la première fonction. « calcul_nb_ingreds » contient un seul boucle donc sa complexité est O(n), le branchement ici n'a pas d'influence sur le nombre d'exécutions de la boucle donc la complexité reste inchangeable; la fonction « calcul_nb_opers » et « calcul_avg_temps_opb » contient un boucle for donc leur complexités sont aussi O(n). « O_espace_f » compose 2 boucles, mais ces deux se succèdent, donc la complexité est toujours linéaire soit O(n). Tout le reste sauf la dernière fonction « get_O_temps » a une complexité de $O(1)$. La dernière fonction est une fonction globale qui englobe les fonctions précédentes pour pouvoir réaliser le calcul avec un seul lancement. De ce fait, même s'il a l'air d'avoir une complexité de $O(1)$, il nous faut fouiller dans les fonctions appelées par elle pour obtenir la vraie complexité en temps. Puisqu'il n'existe pas d'imbrication à son intérieur, et toutes les fonctions qu'il appelées portent au plus une complexité en temps de O(n), nous pouvons en déduire que cette fonction a aussi une complexité de O(n).

[3] Le code de ce script est ajouté dans l'annexe.

Quatrième partie

Conclusion

L'analyse automatique des recettes de cuisine est un sujet intéressant et attirant l'attention du monde scientifique, et de nombreuses compétitions ont été organisé sur ce sujet. Notre corpus de travail vient de l'un de ces défis : le défi DEFT2013. Inspirées par la démarche suivie par l'un de ses participants, nous pratiquons dans ce travail une méthode symbolique pour extraire automatiquement une première liste des ingrédients et annoter le corpus à partir de ce résultat, qui est ensuite utilisé pour entraîner le classifieur CRF. Le résultat de ce classifieur est assez satisfaisant, signifiant que la première annotation de ce corpus est assez fiable pour les tâches suivantes. Nous modélisons ensuite la manière de calculer la complexité en temps et en espace de ces recettes. À partir des formules obtenues, nous effectuons une analyse de corrélation entre ces complexités calculées et les niveaux de difficulté définis par humain. Le résultat révèle qu'une corrélation positive existe entre ces niveaux et la complexité en temps si nous traitons les 4 niveaux comme 2 niveaux principaux. Néanmoins, cette corrélation n'existe plus si nous les regardons comme 4 niveaux bien séparés.

Notre travail a encore des points à améliorer.

Premièrement, nous avons défini par nous-mêmes des lexiques pour réaliser la tâche, donc ses limites sont assez évidentes dans les tâches suivantes. Ayant des ressources externes beaucoup plus complètes, la performance de notre système sera fortement améliorée.

Deuxièmement, Nous avons simplifié certains points pendant la modélisation du calcul de complexité, mais est-ce que la manière choisie pour ce calcul est logique et correcte ? Maintenant, le choix de coefficient pour la complexité en temps pourrait d'être un peu simpliste, nous pouvons continuer à étudier sur ce point. En revanche, selon le constat sur la corrélation, la complexité en temps a l'air bien fait son travail pour le cas de X=1, puisqu'il a une tendance de donner plus moins de nombres des opérations pour les plus simples recettes. Mais ce n'est pas le cas pour la complexité en espace. Ce problème pourrait venu des fautes faites pendant la détection des ingrédients et des opérations. Mais il est aussi possible que notre choix de modélisation n'est pas la bonne piste à suivre.

Finalement, certains traitements dans le script pourraient être amélioré dans le futur, tels que la création des corpus d'entrainement et de test pour le classifieur CRF.

Bibliographie

Herranz, L., Min, W., & Jiang, S. (2018). Food recognition and recipe analysis: integrating visual content, context and external knowledge. arXiv preprint arXiv:1801.07239.

Dini, L., Bittar, A., & Ruhlmann, M. (2013). Approches hybrides pour l'analyse de recettes de cuisine DEFT, TALN-RECITAL 2013. Actes du neuvième DÉfi Fouille de Textes, 52.

Susan Li (2018). Named Entity Recognition and Classification with Scikit-Learn https://www.kdnuggets.com/2018/10/named-entity-recognition-classification-scikit-learn.html (consulté le 5 mai 2021)
https://github.com/susanli2016/NLP-with-Python/blob/master/NER_sklearn.ipynb

Documentation de Spacy. https://spacy.io/usage/rule-based-matching (consulté le 5 mai 2021)

Annexe

```python
from sympy import symbols

class ComplexCalculator:

    def __init__(self, dico_ingreds, dico_opers):
        self.ingreds = dico_ingreds.values()
        self.opers = dico_opers.values()
        self.X = symbols("X")


    def calcul_nb_ingreds(self):
        """
        calculer le nombre total des ingrédient de la recette, avec variable X
        """
        X = self.X
        result = []
        for ingred in self.ingreds:
            if ingred['denombrable'] == "True":
                n = ingred['quantite'] + "*X"
            else:
                n = "1"
            result.append(n)
        self.nbre_ingreds = str(eval("+".join(result)))
        return self.nbre_ingreds

    def calcul_nb_opers(self):
        X = self.X
        result = []
        for oper in self.opers:
            result.append(oper['nb_oper'])
        self.nbre_opers = str(eval("+".join(result)))
        return self.nbre_opers

    def calcul_coef(self):
        X = 1
        coef = "(" + self.nbre_opers + ")/(" + self.nbre_ingreds + ")"
        self.coef = eval(coef)
        return self.coef

    def O_temps_f(self):
        X = self.X
        O_temps = f"{self.coef}*({self.nbre_ingreds})"
        self.O_temps = str(eval(O_temps))
        return "f(X) = " + self.O_temps
```

```
    def calcul_avg_temps_opb(self):
        X = 1
        result = []
        for oper in self.opers:
            result.append(oper['temps'])
        self.temps_estime = eval("+".join(result))
        nb_opb = eval(self.nbre_opers)
        self.avg_t_oper = self.temps_estime / nb_opb
        return self.avg_t_oper

    def temps_estime_X(self,X):
        result = eval(f"{self.O_temps}*{self.avg_t_oper}")
        return result

    def O_espace_f(self):
        X = self.X
        result = []
        for oper in self.opers:
            result.append(oper['recipient'])
        for ingred in self.ingreds:
            result.append(ingred['recipient'])
        self.O_espace = str(eval("+".join(result)))
        return "f(X) = " + self.O_espace

    def get_O_temps(self):
        self.calcul_nb_ingreds()
        self.calcul_nb_opers()
        self.calcul_coef()
        return self.O_temps_f()
```