Maureen Poissonnier
Alexandre Taconnet

Modélisations mathématiques

M3202 : Modélisations mathématiques

Sommaire

Etat d'avancement des projets **3**
Projet 1 : création des modèles de langage 3
Projet 2 : reconnaissance des auteurs 4

Description des algorithmes et performances obtenues **5**
Description 5
Performances obtenues 6

I. Etat d'avancement des projets

a. Projet 1 : création des modèles de langage

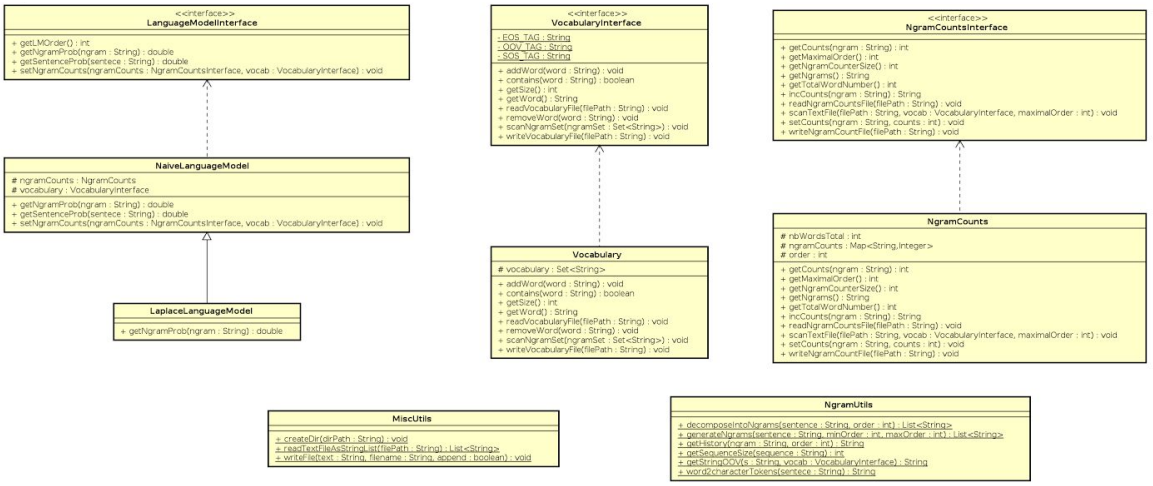

Diagramme de classe obtenu avec Astah

L'objectif de ce projet est de nous permettre de comprendre et d'implémenter une bibliothèque gérant des modèles de langages afin de calculer des probabilités pour une phrase donnée selon le modèle de langage utilisé.

Avant de construire notre propre modèle de langage, nous disposions d'une bibliothèque nous fournissant l'ensemble des méthodes nécessaires pour la création d'un vocabulaire, sa lecture, la création d'un n-gramme et le calcul des probabilités d'une phrase selon le modèle de langage choisi (LaplaceLanguageModel ou NaiveLanguageModel). Afin de bien comprendre le fonctionnement de chacune des classes, nous avons réalisé le diagramme de classe ci-dessus pour mieux nous y retrouver.
Aidés de la documentation, nous avons pu déduire que la classe NgramUtils contenait des méthodes utiles à la création d'un n-gramme selon une phrase donnée. La phrase peut être décomposée selon un n-gramme voulu de taille n. On peut également lire l'historique du n-gramme ou encore générer un n-gramme selon la phrase et une taille minimum et maximum pour le n-gramme généré. La classe NgramCounts permet quant à elle de construire un n-gramme à partir d'un corpus et d'un vocabulaire donné et de générer le fichier associé. A l'inverse, on peut aussi lire un fichier déjà créé contenant le n-gramme. Les classes NaiveLanguageModel et LaplaceLanguageModel sont les modèles de langages utilisés pour calculer les probabilités d'une phrase ou d'un n-gramme. En effet, il suffit d'associer un n-gramme et un vocabulaire au modèle de langage afin de calculer la probabilité d'un n-gramme ou d'une phrase. Enfin, la classe Vocabulary nous permet de créer un vocabulaire ou d'en lire un.

Après l'utilisation de la bibliothèque donnée, il nous a fallu créer notre propre bibliothèque. Etant donné que nous travaillions en binôme, nous nous sommes répartis les exercices à faire afin d'avancer au plus vite. Toutefois, les problèmes de compréhension sont vites apparus. En effet, sur le moment même, il a fallu bien comprendre le fonctionnement de chaque classe et ce que chaque méthode devait donner comme résultat. Nous avons donc cherché dans les dossiers pour y

découvrir des fichiers déjà créés. Nous en avons déduis que les méthodes que nous étions en train de créer devait permettre de créer les mêmes fichiers.

Ainsi, après cette déduction, la création des classes Vocabulary et NgramUtils n'ont pas été très complexes à créer. La classe NgramCounts a quant à elle été plus compliquée à créer, notamment les méthodes de scan, de lecture et d'écriture d'un fichier. En effet, nous avions pris le parti de développer l'algorithme de lecture et d'écriture d'un fichier entièrement ce qui a pris un peu de temps alors que nous disposions de méthodes pour lire et écrire dans un fichier fournies par la classe MiscUtils. Enfin, les classes NaiveModelLanguage et LaplaceModeleLanguage, et plus principalement les méthodes respectives permettant de calculer la probabilité d'un mot ont été plus difficiles à implémenter du fait de leur complexité. En effet, les formules décrites dans le cours étaient difficiles à comprendre. De facto, leurs implémentations ont également été compliquées.

Aussi, nous avons pu effectuer nos test avec JUnit à chaque fois que nous avions terminé la création d'une classe. Cela nous a permis de corriger nos erreurs et notamment de découvrir que lors de l'utilisation du modèle de Laplace pour calculer nos probabilités nous avions une erreur de calcul par rapport à l'utilisation que nous en avons fait au début du projet. Après diverses questions et analyse du code du côté de cette classe nous nous sommes aperçus que nous avions oublié de traiter le cas où le mot serait inconnu (balise <unk>) empêchant ainsi les probabilités d'être correctes.

A l'heure actuelle, ce projet est totalement terminé et fonctionnel. Malgré les quelques problèmes cités ci-dessus, nous avons su nous adapter et corriger nos erreurs.

b. Projet 2 : reconnaissance des auteurs

L'objectif de ce second projet était de construire un système de reconnaissance des auteurs de phrases dans un fichier en utilisant la bibliothèque précédemment créée. Pour cela, nous avons dû créer des modèles de langage pour chaque auteur pour ensuite construire un système de reconnaissance en utilisant les modèles de langages précédemment construits. Enfin, afin de pouvoir voir si notre système est valide, nous avons dû créer des fichiers d'hypothèses d'auteur, le but étant que chaque nouveau système construits améliore le précédent. Nous sommes ainsi partis d'un modèle mettant un auteur aléatoire sur chaque phrase à un modèle permettant de dire quel auteur a écrit cette phrase et dans le cas où aucun auteur présent dans la liste ne l'ai écrite, dire que c'est d'un auteur inconnu.

Grâce à la classe CreateLanguageModels, nous avons pu créer nos modèles de langages et plus principalement la création des bigrams de chaque auteur ainsi que le fichier de configuration nécessaire pour les classes AuthorRecognizer et UnknownAuthorRecognizer.

Nous avons ensuite mis ces classes en place. Les mains sont identiques puisqu'ils doivent effectuer la même chose. De, plus le constructeur est également le même. Si le main n'a pas été difficile à trouver, le constructeur à été plus difficile à mettre en place. En effet, nous avons bien réussi à charger le fichier de configuration ainsi que celui du vocabulaire mais le remplissage de la HashMap a été plus compliqué. En effet, si les ngrams changeait bien dans la boucle implémentée, celui mis dans la HashMap était quant à lui toujours le même, celui correspondant au premier. Après plusieurs essais infructueux, nous nous sommes rendus compte que l'erreur venait d'un oubli de

créer une nouvelle instance du ngram et du modèle de langage choisi. Une fois ce problème réglé, nous avons ainsi pu tester notre méthode recognizeAuthorSentence().

Cette dernière repose tout simplement sur la liste des auteurs. Nous parcourons ainsi pour chaque phrase passée en paramètre la probabilité correspondant à chaque auteur. Si la probabilité calculée est supérieure à celle définie, la probabilité change et l'auteur est mis à jour. Les différences entre les classes AuthorRecognizer et UnknownAuthorRecognizer résident dans le fait que dans le premier cas, la probabilité de départ et l'auteur sont fixées par rapport au premier auteur de la liste tandis que dans le second cas, la probabilité initiale est fixée à 0.0 tandis que l'auteur est fixé à "unknown".

Nous avons ainsi pu créer différentes classes AuthorRecognizer et UnknownAuthorRecognizer avec différents paramètres changeants pour en tester l'efficacité. Ces caractéristiques sont décrites ci-après.

II. Description des algorithmes et performances obtenues

a. Description

Dans le but de tester différents paramètres nous avons utilisé plusieurs variables :

- le modèle de langage utilisé (NaiveLanguageModel ou LaplaceLanguageModel)
- la taille maximale du ngram (1, 2 ou 3)
- le type de classe (prenant ou non en charge les auteurs inconnus)

Nous obtenons ainsi les classes et caractéristiques pouvant être résumée dans le tableau suivant :

NOM DE LA CLASSE	MODÈLE DE LANGAGE	TAILLE MAX DU NGRAM
AuthorRecognizer1	Naive	2
UnknownAuthorRecognizer1	Naive	2
AuthorRecognizer2	Laplace	2
UnknownAuthorRecognizer2	Laplace	2
AuthorRecognizer3	Laplace	1
UnknownAuthorRecognizer3	Laplace	1
AuthorRecognizer4	Laplace	3
UnknownAuthorRecognizer4	Laplace	3

b. Performances obtenues

NOM DE LA CLASSE	PERFORMANCES SMALL_CORPUS	PERFORMANCES BIG_CORPUS
AuthorRecognizer1	0.25	0.2204
UnknownAuthorRecognizer1	0.14	0.1238
AuthorRecognizer2	0.43	0.5084
UnknownAuthorRecognizer2	0.43	0.5068
AuthorRecognizer3	0.53	0.5768
UnknownAuthorRecognizer3	0.53	0.5764
AuthorRecognizer4	0.13	0.1442
UnknownAuthorRecognizer4	0.14	0.1238

Nous pouvons ainsi remarquer que les classes donnant les pires performances sont les UnknownAuthorRecognizer 1 et 4, c'est à dire celle comportant le modèle de langage Naive dans le cas de la première et celle comportant un bigrams de 3. Au contraire, les classes les plus prometteuses sont celles qui comportent un bigrams de taille max 1 et un modèle de langage de Laplace.

Néanmoins, il ne nous est pas encore possible de décider vraiment laquelle de ces deux classes est la meilleure. En effet, si sur des phrases où tous les auteurs sont connu, la classe AuthorRecognizer3 sera la meilleure car celle-ci aura une chance sur le nombre d'auteurs d'avoir la bonne réponse à contrario de la classe UnknownRecognizer3 qui elle mettra que l'auteur est inconnu.

Mais dans le cas où les phrases à tester peuvent être d'auteurs inconnus, il nous paraît probable que la classe UnknownRecognizer3 soit la meilleure dans la mesure où elle sera capable de détecter si un auteur est inconnu.