Twit'Oz: un prédicateur de texte

Projet LINFO1104

Christophe Crochet Damien Sprockeels Thomas Wirtgen Peter Van Roy

Avril 2023

Contexte

Les articles écrits par des bots en cet âge de l'intelligence artificielle ont attiré l'attention d'Elon Musk, fondateur et PDG de Tesla et SpaceX. En cette période où son temps est précieux pour gérer ses entreprises, il souhaite automatiser une partie de ses interactions sur les réseaux sociaux. Il a entendu parler de l'efficacité du langage de programmation Oz pour son utilisation de la récursion terminale et du threading efficace. Elon Musk souhaite donc vous engager pour développer une application graphique capable de prédire le prochain mot qu'il écrira sur Twitter en se basant sur un historique de tweets qu'il aura récupéré grâce à ses contacts chez Twitter. Votre mission, si vous l'acceptez, sera de créer un système automatisé pour générer ses tweets en utilisant des techniques d'intelligence artificielle et de programmation avancées. Le projet impliquera l'analyse de l'historique de tweets, la création d'un modèle de prédiction basé sur ces données et l'implémentation d'une interface utilisateur pour permettre à Elon Musk de visualiser et d'éditer les tweets générés. Ce projet offre une occasion unique de mettre en pratique des concepts d'IA et de programmation avancés pour répondre aux besoins de l'un des plus grands entrepreneurs technologiques de notre époque.

Données et objectifs

Ce projet se divise en 2 grandes parties:

- La lecture des données de manière parallèle
- La prédiction par interface graphique

Toute la documentation se trouve ici (ce qui était valable pour Oz 1.4, l'est en grande partie pour Oz 2.0) et dans ce livre de référence vous pouvez aussi trouver certains exemples qui vous faciliteront la vie.

Traitement des données

Vous avez 208 fichiers texte de chacun maximum 100 lignes contenant tous les tweets sur lesquels vous devrez vous baser. Des informations sur la lecture ligne par ligne d'un fichier texte peuvent être trouvées ici. Vous choisirez vous-même le nombre de threads (minimum 2) que vous voulez assigner à la lecture des fichiers tant que celles-ci se fait de manière parallèle. Il faut ensuite parser et sauvegarder les données lues. Nous vous conseillons d'utiliser au moins autant de threads de parsing qu'il y a de threads de lecture et un thread pour la sauvegarde des données comme dans la fig. 1. Nous vous conseillons d'utiliser un arbre pour stocker les N-grammes. Si vous décidez d'utiliser une autre structure, veillez à bien justifier votre choix dans le rapport.

Prédictions

Afin de prédire le mot suivant d'une phrase, vous devrez utiliser des N-grammes. Nous ne vous demandons pas un parsing très compliqué des mots pour extraire leur type syntaxique, les utiliser tels quels sera suffisant. Il faudra néanmoins réfléchir aux ruptures de phrases (points, virgules, hashtags etc.) et au majuscules. Soyez cependant attentifs car cette partie **devra** être récursive terminale. Le principe d'un N-gramme est le suivant: vous prédisez toujours le mot le plus fréquent. Celui d'un 1-gramme est que vous prédisez le mot le plus fréquent en sachant quel était le mot précédent. Par exemple si dans tous

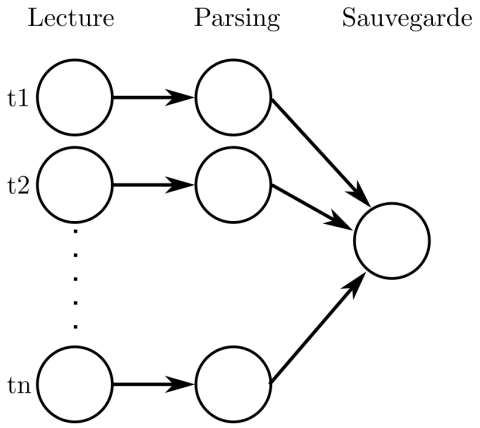

Figure 1: La structure conseillée du threading pour ce projet.

les tweets le mot “you” est majoritairement suivi de “are”, c’est ce dernier que vous devrez prédire si on input “you” dans la zone de texte. Un 2-gramme étend cette logique aux 2 mots précédents. Donc si on input “you are” le mot le plus fréquent pourrait être “fake” car c’est une partie de phrase souvent utilisée. Vous remarquerez donc que la capacité de mémoire nécessaire croit exponentiellement avec le N du N-gramme. Pour la version de base limitez-vous donc à maximum un 2-gramme en sachant que **nous attendons de vous au minimum un 2-gramme fonctionnel!** Pour la suite du projet, nous vous demanderons de développer des extensions qui nécessiteront, pour certaines d’entres elles, de modifier le code qui génère les 2-gramme. **Faites donc bien attention à conserver le code pour le 2-gramme qui sera considéré comme la version de base de votre projet.** Pour le 2-gramme vous devrez déjà limiter les choix de mots ou en extraire les racines pour limiter l’utilisation de la mémoire. A vous de voir comment vous voulez gérer ce cas-là.

Pour le côté graphique, une fenêtre basique avec une zone de texte et un bouton pour demander la prochaine prédiction vous est fournie, mais vous êtes libres d’améliorer cela à votre guise (votre note finale ne pourra qu’en être plus favorable, voir section *Extensions*).

Soumission

Ce travail se fera par groupe de 2, ni plus ni moins. La soumission s’effectuera sur INGInious. Plus d’informations concernant la soumission vous seront données sur la plateforme en temps voulu.

Si jamais vous ne trouvez pas de partenaire avec qui faire le projet, vous pouvez le faire seul à condition d’avoir une bonne raison (nombre impair d’étudiants, personne n’a répondu à vos messages pour trouver un binôme sur le forum). La date limite du projet est le lundi **1** mai à 23h55. Vous devrez être inscrit dans un groupe INGInious pour soumettre, même si vous êtes seul. Faites attention à cela.

Vous devez tous commencer par faire une solution de base que vous soumettrez dès que celles-ci répondra aux consignes. Ensuite, vous pourrez agrémenter votre solution en implémentant des extensions supplémentaires en bonus.

Votre solution doit être **entièrement déclarative**, i.e., aucune structure contenant des cellules n’est autorisée. De même, il existe des structures de données de la librairie standard Oz qui ne sont pas déclaratives. Veillez à ne pas les utiliser. Le partage entre groupe est évidemment interdit et sera considéré comme du plagiat. Vous pouvez cependant discuter de vos stratégies pour réaliser votre solution mais sans échanger de code.

Respectez scrupuleusement les spécifications données pour les fonctions, procédures, etc. Une partie de votre évaluation sera effectuée par des tests automatisés, faites donc attention **à ne pas modifier le nom des fonctions et procédures** qui vous sont fournies.

Nous ne divulguerons pas les tests. Vous êtes néanmoins encouragés à écrire vos propres tests afin de vous assurer que votre code fonctionne bien, peut-être même avant de commencer à coder.

Rapport

En plus de votre code, vous devrez soumettre un rapport de **maximum 2 pages** dans lequel vous expliquerez en détails les extensions que vous avez implémentées et pourquoi, ainsi que toute information que vous jugerez nécéssaire concernant votre implémentation, pour les extensions et les fonctionnalités de base. Certaines extensions nécéssiteront plus d'explications. Ne négliez donc pas l'écriture du rapport !

Troubleshooting

En cas de non-compréhension ou de doute sur les consignes, vous êtes invités à adresser toutes questions sur le forum Moodle. Vous êtes bien évidemment encouragés à répondre sur le forum Moodle si vous estimez avoir la réponse à une question d'un de vos compères.

En cas de questions techniques, adressez également vos questions sur Moodle. Si votre question nécessite absolument de montrer votre code, vous pouvez l'adresser par email à damien.sprockeels@uclouvain.be **ET** christophe.crochet@uclouvain.be **ET** thomas.wirtgen@uclouvain.be (notez bien l'utilisation des **ET**), MAIS, si vous voulez une réponse, celui-ci devra impérativement contenir:

- Une formulation claire de la question. Ne supposez pas que tous les détails du projet soient frais, introduisez du contexte.
- La liste des choses que vous avez tenté pour résoudre et/ou diagnostiquer votre projet:
  - Pensez en particulier à toute les expériences que vous pourriez vous-même entreprendre à l'aide de Mozart pour obtenir des réponses à vos questions.
  - Quelles recherches dans le cours et sur le net avez vous entreprises?
- Si votre problème concerne un comportement observé dans votre code, un exemple minimal fonctionnel qui démontre votre problème.

On rappelle que vous êtes plusieurs centaines et qu'il est impossible de tous vous aider dans le détail. À ce titre, les questions sont réservées aux véritables problèmes, pas à ceux que vous devriez être capables de régler vous-même avec un poil d'huile de coude (ça fait aussi partie de ce que le projet évalue).

Les questions du type "ça marche localement", "j'ai testé en local" ou "INGInious est cassé" ne sont (presque) pas autorisées. Dans la plupart des cas, l'erreur provient de votre code. Cependant, nous sommes conscients que les tests sur INGInious ne sont pas parfaits, des problèmes peuvent survenir. Ces questions sont tolérées **SEULEMENT** si vous rédigez votre question dans ce modèle :

```
#### Description du problème


#### Exemple minimal fonctionnel


#### Étapes pour reproduire le problème

1.
2.
3.

#### Quel est le résultat attendu ?

-

#### Quel est le résulat obtenu ?

-

#### Détails additionels / Capture d'écran

- ![Screenshot]()
-
```

Ce type de modèle est un exemple de bonne pratique et est largement utilisé dans la communauté des logiciels libres, notamment sur GitHub pour avertir les développeurs d'un bug potentiel dans leur code.

Fichiers & Fonctions Fournis

- Un `Makefile` permettant de **compiler** (`make`) et **exécuter** (`make run`) les fichiers Oz. Pour les utilisateurs de Windows, passez soit par le WSL ou bien par l'invité de commande (`cmd.exe`) pour utiliser le Makefile. Un document est à votre disposition sur Moodle pour configurer votre machine Windows pour utiliser les Makefile.

  Inutile d'essayer d'interpréter le code (feed {buffer,region}) dans emacs ou dans Visual Studio Code! L'OPI (Oz Programming Interface) refusera tout simplement de le compiler. La syntaxe des fichiers qui vous sont fournis n'est pas supportée par l'environnement de développement interactif. Le seul moyen de compiler et d'exécuter le code est d'utiliser respectivement le compilateur ozc et l'interpréteur ozengine. D'où l'utilisation d'un Makefile pour vous faciliter le travail.

- Un fichier `main.oz` qui contient le squelette du code pour que vous implémentiez votre propre solution. Vous êtes libres de créer d'autres fichiers `.oz` si vous le souhaitez, mais dans ce cas, il faudra modifier le Makefile en conséquence. Un Makefile qui ne fonctionne pas **recevra une pénalité**. Nous vous conseillons donc dans un premier temps d'implémenter toute votre solution dans ce fichier.

  `main.oz` est le point d'entrée de votre programme. Votre programme doit pouvoir être exécuté à partir d'une ligne de commande dans le terminal. Il lira les options qui lui sont passées sur la ligne de commande et agira en conséquence. Pour vous faciliter la tâche, et pour que vous ne perdiez pas votre temps sur cette partie, nous vous fournissons le code qui permet de récupérer le dossier de Tweets à parser. Nous vous donnons également un exemple de code qui liste les fichiers contenus dans le dossier. Le `Makefile` est également configuré de telle sorte que lorsque vous faites `make run`, le programme est lancé avec le dossier de Tweets qui est fourni dans l'archive.

  Comme expliqué précédement, vous êtes tenus de **ne pas modifier le nom des fonctions, procédures et identificateurs** déjà présents dans ce fichier. Cela permet ainsi à INGInious de vérifier si votre code satisfait aux exigences de base de ce projet. Il va sans dire que vous ne devez pas toucher à la manière dont le programme récupère le dossier de Tweets à partir de la ligne de commande.

  Dans ce fichier, plusieurs fonctions/procedures sont laissées vides. Vous devez donc les remplir en respectant leurs spécifications:

  - `Main`: Cette procedure est appelée par l'interpréteur Oz lorsqu'on exécute le programme. Elle ne prend aucun paramètre et ne retourne rien. Elle lance l'interface graphique QTk ainsi que des threads permettant de lire et parser les inputs utilisateurs.
  - `LaunchThreads`: Lance les N threads de lecture et de parsing qui liront et traiteront tous les fichiers. Les threads de parsing envoient leur resultat au port Port. Vous pouvez trouver plus de détails sur le fonctionnement des ports Dans le chapitre 5 du livre de théorie.
  - `Press`: Fonction qui lance la prédiction en tant que telle. Elle est appelée lorsqu'on appuie sur le bouton de prediction. Affiche la prédiction la plus probable du prochain mot selon les deux derniers mots entrés. Elle retourne également une liste contenant [1] la liste du/des mot(s) le(s) plus probable(s) [2] la probabilité/fréquence correspondant à ce(s) mot(s) le(s) plus probable(s). La valeur de retour doit prendre la forme:

```
<return_val>            := <most_probable_words> '|' <probability/frequence> '|' nil
<most_probable_words>   := <atom> '|' <most_probable_words>
                           | nil
<probability/frequence> := <int> | <float>
```

  Example pour l'input "I am":

  - [[cool swag nice] 0.7]
  - [[cool swag nice] 7]

- Un dossier "`tweets`" qui contient une banque de données de plusieurs phrases à utiliser pour entrainer votre modèle.

Extensions

Nous avons parlé jusqu'ici de la base à implémenter. Maintenant, parlons un peu des extensions possibles. Chaque extension a une difficulté qui lui est associée. Vous devez choisir un sous-ensemble de ces extensions pour un minimum de **5 étoiles**. Vous pouvez réaliser d'avantage d'extensions pour obtenir des points bonus, mais gardez bien en tête que vous devez avoir la base du projet fonctionnelle pour que les extensions soient évaluées. Il vaut donc mieux se concentrer sur la base qui est à soumettre avant de travailler sur les extensions.

Attention, veillez toujours à avoir une version fonctionnelle des fonctionnalités de base que nous vous demandons d'implémenter. En particulier, ajouter toutes les extensions sous forme de fonctions ou procédures qu'on peut appeler, éventuellement à la place des fonctions ou procédures pour l'implémentation de base. Dans votre soumission, faites attention à bien appeler les fonctions et procédures pour les fonctionnalités de base et expliquez bien comment utiliser les extensions.

Extension	Étoiles
Généralisation de la formule du N-gramme	**
Optimisation de la formule du N-gramme	***
Ajouter des bases de données custom	**
Garder un historique des inputs de l'utilisateur	**
Proposition automatique (tier 1)	*
Proposition automatique (tier 2)	**
Proposer plus d'un N-gramme	*
Proposer la correction de mots dans une phrase déjà existante	***
Améliorer l'interface graphique existante	*
Modifier le Makefile pour pouvoir spécifier les extensions	*

Généralisation de la formule du N-gramme (séparer)

Vous avez normalemement implémenté au moins le 2-gramme. Essayez de généraliser votre algorithme pour qu'il supporte n'importe quelle suite de N-gramme ($N > 0$). Cela doit au minimum marcher avec un 1-gramme et un 3-gramme.

Optimisation de la formule du N-gramme

Il est possible d'optimiser la formule du N-gramme pour qu'elle soit plus efficace. Cette extension devra être bien documentée et expliquée dans le rapport que vous allez nous rendre.

Ajouter des bases de données custom

Ajouter la possibilité à l'utilisateur, par l'interface graphique, d'uploader d'autres corpus de phrases pour que votre predicteur l'utilise aussi pour prédire la suite d'une phrase. Vous pouvez aussi lui permettre de sélectionner le dossier de phrases à charger pour prédire les mots.

Garder un historique des inputs de l'utilisateur

Ajouter une fonctionnalité sauvegardant les phrases entrées par l'utilisateur, y compris après avoir fermé et relancé le programme, afin de les utiliser pour les futures prédictions.

Proposition automatique

Au lieu de cliquer sur un bouton pour demander une proposition de mot, votre interface graphique proposera automatiquement la suite de la phrase **(tier 1)** sans cliquer sur le bouton.

Cette extension peut être améliorée pour s'adapter en fonction de l'input. Si par exemple la phrase entrée est "Je ne sais seulement ce que je s", on peut se baser sur la première lettre donnée pour proposer à l'utilisateur le mot le plus vraissemblable commencant par "s". (p.ex., "sais" ou "suis") **(tier 2)**. L'extension ne doit pas se limiter à une seule lettre, mais aussi à plusieurs (p.ex, si le début du mot rentré commence par "sus", votre programme pourrait proposer "suspecte" ou "susurre"). Cela nécéssitera peut-être une adaptation de votre algorithme pour le N-gramme. Pensez dans ce cas à conserver votre 2-gramme fonctionnel et à implémenter cette extension dans une autre fonction/procédure.

Proposer plus d'un N-gramme à l'utilisateur

La version de base prend le mot le plus probable pour ce qui est donné comme entrée. Étendez cette version pour proposer les X mots ($X > 1$) les plus probables, pour un N-gramme donné. Par exemple pour la phrase "Les penseurs insistent sur un", votre programme pourrait proposer la liste "détail", "mot", "disque", "ruban".

Proposer la correction de mots dans une phrase déjà existante

Vous appliquerez votre algorithme à une phrase existante et suggérerez d'autres mots que le prédicteur aurait choisis à la place des mots existants. Par exemple, si l'utilisateur saisit la phrase "Je mange une pomme et des croissants". Vous pouvez demander à votre programme de proposer un autre mot pour "pomme" afin qu'il suggère celui qu'il aurait choisi à la place de "pomme".

Partie créative: étendre l'interface graphique existante.

Vous pouvez rendre l'interface plus agréable pour l'utilisateur en changeant les couleurs, en ajoutant des images, etc. Soyez créatifs! L'étendue de vos changements influencera la note que vous recevrez pour cette extension.

Modifier le Makefile pour pouvoir spécifier les extensions

Vous pouvez modifier le makefile pour permettre d'entrer les extensions que l'on veut utiliser, et compiler les fichiers en fonction. Dans ce cas, veillez à bien expliquer comment ça marche et veillez à bien garder les targets de base afin qu'on puisse compiler votre projet sans extensions.