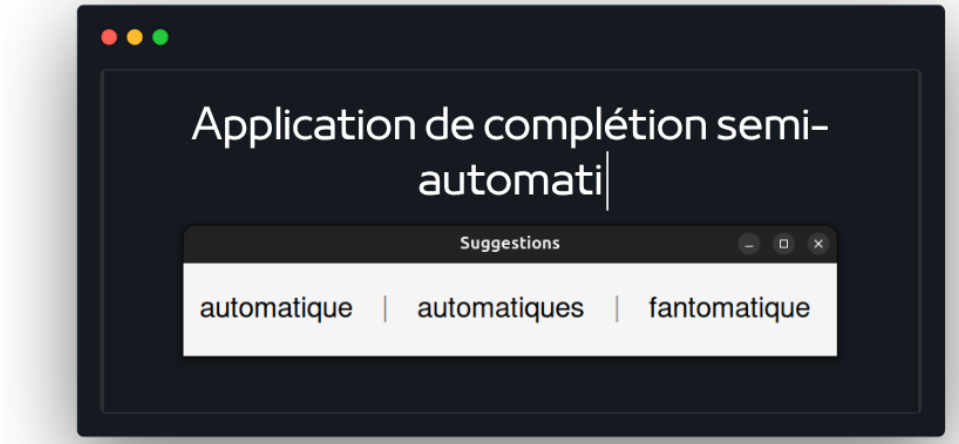
Université Marie & Louis Pasteur

Projet d'Initiation à la recherche
L2 - CMI

Complétion semi-automatique

BENALI Samia
BOITEUX Elouan



2024/2025

Remerciements

Nous souhaiterions remercier dans un premier temps Pierre-Cyril HEAM de nous avoir permis d'effectuer un projet semestriel sur la complétion semi-automatique et de nous avoir encadré tout au long du semestre.

Nous souhaiterions également remercier chaleureusement Jean-Michel HUFFLEN de nous avoir enseigné le langage LaTeX pour la rédaction de nos rapports ainsi que tous les professeurs que nous avons rencontrés et qui nous ont permis d'acquérir nos connaissances actuelles sur les langages utilisés.

Nos remerciements vont également à toutes les personnes avec lesquelles nous avons pu discuter et partager autour des projets de recherche et tout spécialement les étudiants du CMI Informatique pour leur gentillesse et leur aide précieuse.

Table des matières

Table des figures

Introduction

Dans le cadre de notre projet de recherche du CMI informatique de l'Université de Marie & Louis Pasteur de Besançon, encadré par Monsieur Héam, nous avons travaillé sur la complétion semi-automatique.

Ce projet nous a permis de découvrir ce qu'était la complétion automatique et la complétion semi-automatique et de comprendre sur quoi repose ces deux notions. La complétion automatique est un processus par lequel un système va prédire et compléter une entrée en fonction de certaines données et contextes. Tandis que, la complétion semi-automatique est une assistance permettant au système de proposer des options tout en laissant à l'utilisateur la décision finale. La complétion semi-automatique peut être utilisée dans de nombreuses applications : une saisie sur clavier, une complétion de codes, une recherche sur un moteur de recherche, une assistance virtuelle etc.

Ce rapport va nous permettre, dans un premier temps, d'étudier les différentes approches qui existent ainsi que les différents algorithmes de distance. Ensuite, nous parlerons des chaînes de Markov pour la gestion d'historique et enfin vous retrouverez l'application que nous avons créé permettant une complétion semi-automatique.

Chapitre 1

Les différentes approches utilisées

1.1 Introduction

Il existe de nombreuses façons de proposer des suggestions automatiques ou semi-automatiques dans les outils numériques. Certaines méthodes sont simples et rapides, d'autres plus complexes et plus précises. Ce chapitre présente quatre grandes approches : les modèles s'appuyant sur des règles, ceux utilisant des statistiques, ceux qui s'appuient sur l'intelligence artificielle, et enfin les modèles de deep learning.

Chacune de ces approches a ses avantages et ses inconvénients, selon ce que l'on cherche à faire. Nous allons voir comment ces méthodes fonctionnent et dans quels cas elles sont efficaces ou non.

1.2 Modèles s'appuyant sur des règles

Pour proposer des suggestions, ce modèle utilisent des algorithmes simples qui s'appuient sur des règles préprogrammées telles que la correspondance de préfixes ou une séquence donnée en amont. Ces algorithmes vont être gérer principalement grâce à des dictionnaires statiques ou des listes. Cette implémentation est plutôt rapide et simple à mettre en place, elle est cependant très peu flexible et empêche donc une utilisation complexe.

1.3 Modèles s'appuyant sur des statistiques

Pour proposer des suggestions, ce modèle utilisent des statistiques fournies grâce aux données d'un historique. Cela permet de prédire des séquences comme avec le modèle de Markov (expliqué dans le chapitre 3) ou la méthode TF-IDF *(term frequency-inverse document frequency)*.
Cette implémentation permet d'obtenir des résultats rapidement. On a cependant aucune compréhension sémantique donc les suggestions ne conviennent que rarement au contexte.

1.4 Modèles s'appuyant sur l'intelligence artificielle

Pour proposer des suggestions, ce modèle utilise des algorithmes qui s'appuient sur l'intelligence artificielle et les réseaux neuronaux. Avec l'apprentissage supervisé et non supervisé, ces modèles apprennent des motifs complexes à partir des données. Ils peuvent inclure des algorithmes comme les forêts aléatoires ou les régressions pour fournir des prédictions plus contextuelles. Cette implémentation permet d'être efficace face à des problèmes très complexes et de répondre à des demandes rares. Cependant, ce genre d'implémentation nécéssite énormément de temps de calcul et de ressources.

1.5 Modèles s'appuyant sur le deep learning

Pour proposer des suggestions, ce modèle utilise des algorithmes qui s'appuient sur l'amélioration en temps réel. C'est à l'utilisateur de faire des choix et ces mêmes choix sont mémorisés pour une utilisation personnalisée et plus précise. Cette implémentation est donc très adaptable et permet des réponses précises avec un sens sémantique. Cependant, cette implémentation est complexe et très longue à mettre en place puisque les choix de l'utilisateur sont nécessaires et les réponses dépendront en grande partie des données collectées.

Chapitre 2

Les algorithmes de calcul de distance

2.1 Introduction

Un algorithme de calcul de distance permet de mesurer la similarité et/ou la différence entre deux objets tels que du texte, des vecteurs, des chaînes de caractères. On utilise ces algorithmes de calcul principalement pour la correction d'orthographe, le traitement de texte, les alignements de séquences ADN ou encore pour la recherche d'informations.

2.2 Distance d'édition ou de Levenshtein

L'algorithme de calcul de distance d'édition permet de rechercher le nombre minimal d'opérations élémentaires, tels que les insertions, les suppressions et les substitutions, qui seront nécessaires à la transformation d'une chaîne de caractères en une autre. L'insertion permet d'ajouter un caractère, la suppression d'en enlever, et la substitution de remplacer un caractère par un autre. La figure 2.1 montre un exemple d'illustration de ces opérations.

La compexité de l'algorithme est en $\mathcal{O}(n \times m)$ où n est la longueur de la première chaîne et m la longueur de la seconde. Sa complexité dans l'espace est équivalente. Dans certaines implémentations avancées, la complexité dans l'espace peut être réduite à $\mathcal{O}(\min(n, m))$.

Il s'agit d'une implémentation simple et intuitive qu'il est facile de mettre en place. Cependant, elle ne prend pas en compte les permutations et ne peut pas résoudre des erreurs plus complexes que l'insertion, la suppression et la substitution.

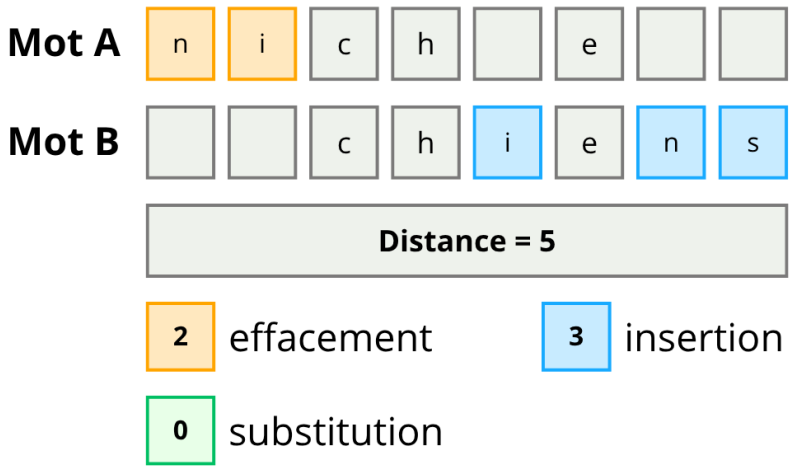


FIGURE 2.1 – Exemple distance de Levenshtein

La formule du calcul de la distance de Levenshtein est la suivante :

$$\text{lev}(a,b) = \begin{cases} \max(|a|,|b|) & \text{si } \min(|a|,|b|) = 0, \\ \text{lev}(a-1,b-1) & \text{si } a[0] = b[0], \\ 1 + \min \begin{cases} \text{lev}(a-1,b) \\ \text{lev}(a,b-1) \\ \text{lev}(a-1,b-1) \end{cases} & \text{sinon.} \end{cases}$$

On cherche le nombre minimal d'opérations pour transformer la chaîne `a` en chaine `b`.

2.3 Distance de Damerau-Levenshtein

L'algorithme de calcul de distance de Damerau-Levenshtein permet de faire la même chose que l'algorithme de calcul de distance de Levenshtein en y ajoutant l'opération de transposition de deux caractères adjacents.

Les complexités en temps et dans l'espace de l'algorithme restent les mêmes que la distance de Levenshtein. Cependant, même dans les implémentations avancées, la complexité dans l'espace ne peut pas être réduite.

Il s'agit d'une implémentation qui reste simple et intuitive et qui, de plus, a une meilleure gestion des erreurs de type (« ab » et « ba »). Elle est toute fois plus coûteuse que Levenshtein.

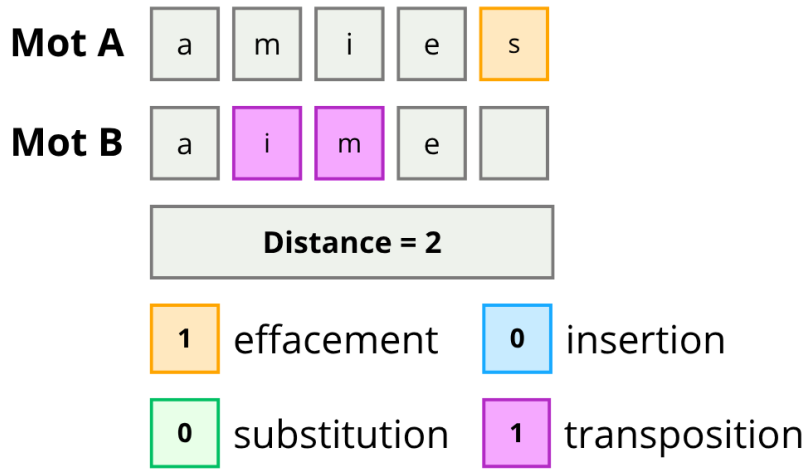


FIGURE 2.2 – Exemple distance de Damerau-Levenshtein

2.4 Distance de Hamming

L'algorithme de calcul de distance de Hamming va rechercher, dans deux chaînes de même longueur, le nombre de positions qui vont différer. La figure 2.3 montre un exemple de ce calcul de distance.

Les complexités en temps et dans l'espace de l'algorithme sont en $\mathcal{O}(n)$ où n est la longueur des deux chaînes.

C'est un algorithme facile à implémenter et très rapide. Cependant, il ne fonctionne que sur des chaînes de même longueur ce qui peut vite s'avérer restrictif.

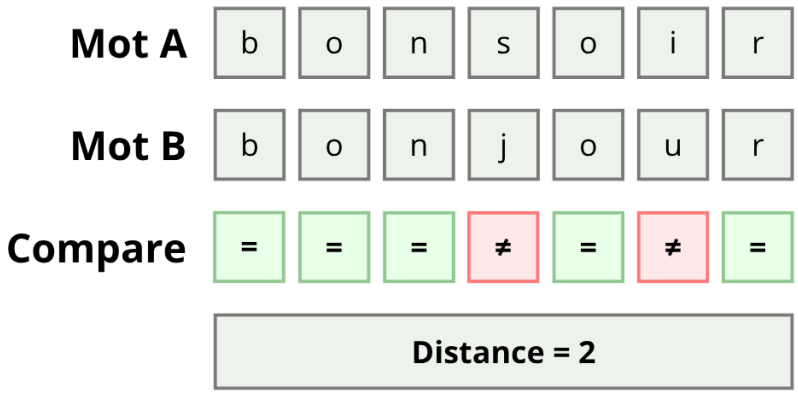


FIGURE 2.3 – Exemple distance de Hamming

Chapitre 3

Chaine de Markov

3.1 Définition

Une chaîne de Markov est un modèle mathématique qui représente un sytème où la probabilité de passer d'un état à un autre dépend uniquement de l'état actuel. Les états sont les différents éléments du système. Les transitions sont les différentes probabilités de passer d'un état à un autre.

Une matrice de transition est une table qui permet de regrouper les probabilités de transition entre tous les états.

3.2 Application

On utilise les chaînes de Markov pour modéliser des processus aléatoires, pour analyser des séquences données, des prédictions ou encore des historiques de navigation. Nous allons nous concentrer sur les historiques de navigation sur le web ainsi que l'historique des actions de l'utilisateur.

Imaginons qu'un utilisateur navigue entre trois pages web A, B et C.
Alors une chaîne de Markov pourrait ressembler à :

Depuis	**Vers**	**Probabilité**
A	B	0.6
A	C	0.4
B	A	0.3
B	C	0.7
C	A	0.5
C	B	0.5

Table 3.1 – Probabilités de transition entre les états

La matrice de transition serait donc :

$$M = \begin{bmatrix} 0 & 0.6 & 0.4 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0.5 & 0 \end{bmatrix}$$

L'objectif serait d'utiliser l'historique de navigation ou l'historique des actions de l'utilisateur pour modéliser les transitions probables entre les différentes étapes. Pour cela, on doit premièrement collecter les données des historiques. Ensuite, on compte les transitions observées entre les états pour calculer les probabilités de transition. Enfin, on construit la matrice de transition.

3.3 Exemple

Prenons l'exemple de l'historique suivant :

$$A \Longrightarrow C \Longrightarrow B \Longrightarrow A \Longrightarrow B \Longrightarrow C$$

La première chose à faire est d'observer le nombre de transitions :

Depuis	**Vers**	**Nombre d'observations**
A	C	1
C	B	1
B	A	1
A	B	1
B	C	1

TABLE 3.2 – Transitions observées et leur fréquence

Ensuite, il faut calculer les probabilités de transition.
Pour ce faire, on considère la probabilité de transition entre deux états X et Y. Il s'agit de compter le nombre de fois où la transition de X vers Y a été observée, puis de diviser ce nombre par le total des transitions partant de l'état X.

Transitions depuis A

- Transitions observées : $A \to C$, $A \to B$
- Nombre total de transitions depuis A : 2
- Probabilités :
  - $P(A \to C) = 0.5$
  - $P(A \to B) = 0.5$

Transitions depuis B

- Transitions observées : $B \to A$, $B \to C$
- Nombre total de transitions depuis B : 2
- Probabilités :
  - $P(B \to A) = 0.5$
  - $P(B \to C) = 0.5$

Transitions depuis C

— Transition observée : $C \to B$
— Nombre total de transitions depuis C : 1
— Probabilité :
  — $P(C \to B) = 1$

Matrice de transition

On obtient donc une matrice de transition.

$$M = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix}$$

3.4 Conclusion

Les chaînes de Markov permettent de prédire l'étape suivante en se basant sur les actions précédentes. Cela va permettre, par la suite, de personnaliser les suggestions pour chaque utilisateur.

De plus, ce système est facile à mettre en place et à implémenter. Cependant, une chaîne de Markov ne prend pas en compte l'historique complet ce qui mène à un problème de mémoire. Il faut, de plus, un très grand nombre de données et un historique plus grand pour avoir des données fiables.

Chapitre 4

Notre outil de complétion semi-automatique

4.1 Choix pour l'implémentation

Tout d'abord, nous avons dû choisir un langage de programmation pour implémenter ce projet semestriel. Il nous fallait un langage permettant de minimiser le temps de calcul de la distance entre les mots avec lesquels nous travaillons. De plus, nous souhaitions utiliser cette occasion pour découvrir un nouveau langage. `Rust` s'est donc révélé plus adapté que les autres langages, puisqu'il offre une excellente vitesse de calcul et limite donc considérablement la latence.

Nous devions ensuite mettre en place le projet. Nous avons décidé d'à la fois recréer une interface de suggestions de mots comme sur les téléphones portables mais sur ordinateurs, et utilisable depuis n'importe quelles applications. En effet, créer une application qui suggère des mots mais qui n'est pas utilisable dans d'autres applications n'est pas d'un grand intérêt. Nous voulions que notre application soit utilisable sur `Ubuntu`.

4.2 Les étapes de la réalisation

4.2.1 Keylogger & mouselogger

Récupération des évènements clavier

Dans un premier temps, nous voulions pouvoir récupérer tous les évènements clavier de l'utilisateur quelle que soit l'application sur laquelle il se trouvait. La solution retenue pour ce projet a été de développer un keylogger. Un keylogger (ou enregistreur de frappe) est un programme qui s'exécute en arrière-plan sur l'ordinateur et récupère les frappes de l'utilisateur sur son clavier. Après la détection de chaque touche pressée, notre keylogger traduit l'identifiant reçu en caractère grâce au fichier **/usr/include/linux/input-event-codes.h** que l'on retrouve dans les fichiers de l'`OS`. Ce fichier associe une touche (lettre I par exemple) en un identifiant (23 pour I). La figure 4.1 montre un extrait de ce fichier. On utilise donc l'identifiant, que l'on traduit à nouveau, et cela permet ainsi de construire le mot au fur et à mesure.

```
#define KEY_I                   23
#define KEY_O                   24
#define KEY_P                   25
#define KEY_LEFTBRACE           26
#define KEY_RIGHTBRACE          27
#define KEY_ENTER               28
#define KEY_LEFTCTRL            29
#define KEY_A                   30
#define KEY_S                   31
#define KEY_D                   32
```

FIGURE 4.1 – Extrait du fichier input-event-codes.h

Gestion du mot en cours de saisie

Une fois ajouté à notre programme principal, nous pouvons désormais récupérer les frappes de l'utilisateur. Grâce à cela, nous pouvons désormais concaténer les lettres et afficher dans un terminal le mot en cours de saisie. Pour une utilisation plus fluide, nous avons ajouté quelques fonctionnalités supplémentaires. Notre programme gère également certaines touches spéciales comme les flèches directionnelles. Lorsque la flèche gauche ou droite est détectée, le keylogger met à jour la position du curseur dans le mot en cours de construction. Ainsi, lorsqu'une nouvelle lettre est tapée après un déplacement avec les flèches, elle est insérée à la bonne position dans la chaîne, et non simplement ajoutée à la fin. Ce comportement permet à notre programme de reconstituer précisément les mots saisis, y compris avec des corrections ou insertions en milieu de mot. De plus, s'il clique sur la touche retour arrière, il peut supprimer la lettre à gauche du curseur sans que notre programme perde l'information du mot en cours de saisie. Enfin, si l'utilisateur clique sur n'importe quelle autre touche qu'une lettre (accentuée ou non), le retour arrière ou les flèches de navigation alors, on annule les suggestions de mots.

Détection des clics souris

Nous avons décidé d'ajouter une détection de clics de souris afin d'annuler la proposition de suggestions lorsque l'utilisateur n'a plus le curseur de texte sur le mot qu'il est en train d'écrire. Cette implémentation a été simple et rapide puisque nous reprenons le même principe que le keylogger.

Prise en charge de plusieurs périphériques

Une fois cette première partie terminée, nous avons constaté que notre programme ne détectait qu'un seul clavier et qu'une seule souris. Si un utilisateur utilise un ordinateur portable et y branche un clavier externe, notre détection de saisie ne fonctionnait plus.

Gestion de l'arrêt du programme

La principale difficulté a été de détecter tous les claviers et souris puis de les faire fonctionner en même temps. On a décidé d'utiliser des threads pour lire les fichiers des claviers et des souris en concurrence. Un thread est une séquence d'instructions qui peut être exécutée indépendamment au sein d'un programme. Les threads permettent une exécution simultanée et un fonctionnement multitâche au sein d'une même application. La figure 4.2 montre un schéma de la lecture des fichiers en concurrence grâce aux threads.

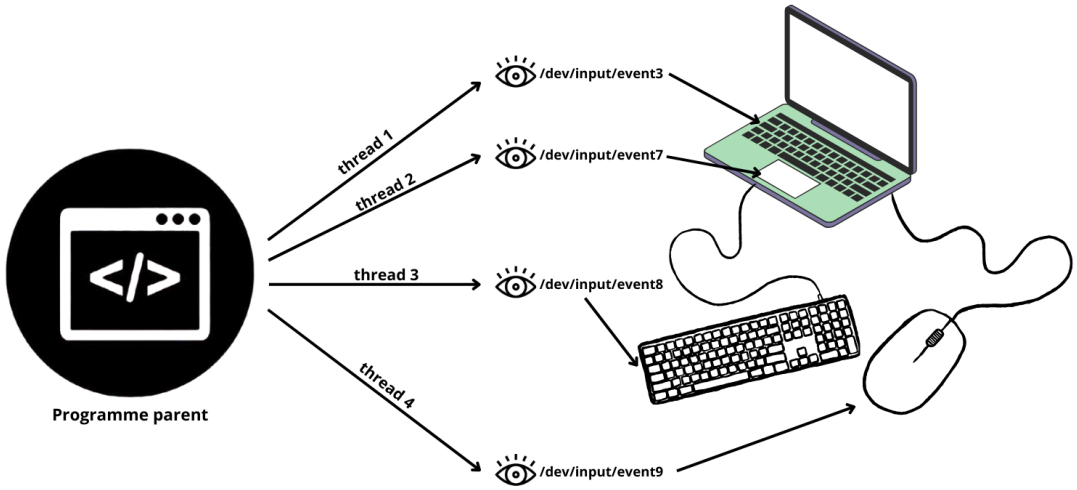


FIGURE 4.2 – Schéma descriptif des lectures fichiers

Cependant, il restait un problème : arrêter proprement notre programme. En effet, lorsque l'on ferme l'interface graphique, les threads sont encore en train de lire les fichiers des claviers et des souris et ils ne se ferment donc pas. Pour pallier à ce problème, nous avons fait en sorte qu'un signal soit envoyé à notre code `Rust` lorsque l'interface graphique `Python` est fermée. Lorsque le signal est reçu, ce dernier l'envoie aux threads qui sont gérés de façon asynchrones et qui ferment ainsi la lecture du fichier. Afin que les signaux permettent d'arrêter la lecture des fichiers de périphériques, nous avons dû modifier la structure de notre code. On obtient ainsi une façon plus propre de fermer le programme.

4.2.2 Ecrire et remplacer un mot

Pour cette deuxième étape, nous devions faire en sorte de pouvoir écrire un mot dans n'importe quelle application, et cela de façon automatique via notre code. Pour ce faire, nous avons mis en place un clavier virtuel pour simuler les frappes de l'utilisateur et ainsi écrire le mot demandé. Notre code fonctionne ainsi : On crée une fonction permettant de traduire un caractère en l'évènement clavier correspondant. Ensuite, on crée une fonction permettant de cliquer sur un de ces caractères sur le clavier virtuel. Enfin, on répète cette opération avec chaque lettre du mot pour l'écrire en entier.

Lorsque cela a été mis en place, nous nous sommes rendu compte que les lettres précédemment écrites par l'utilisateur n'étaient pas effacées. Pour pallier à cela, nous avons simplement utilisé la fonction permettant de cliquer sur une touche du clavier virtuel que l'on a lancé **n** fois sur la touche retour arrière (où **n** est la longueur du mot écrit par l'utilisateur). Comme tout est automatique, le remplacement du mot est instantané, non visible par l'utilisateur et la réécriture reste fluide et naturelle.

4.2.3 Algorithme de suggestions

Maintenant que nous pouvons détecter les frappes de l'utilisateur ainsi qu'écrire et/ou remplacer un mot, nous devons écrire l'algorithme qui nous permettra de faire 3 suggestions à l'utilisateur.

Choix du dictionnaire et premier algorithme

Nous avons décidé de nous baser sur un dictionnaire de plus de 140 000 mots trouvé dans [NP19] qui contient des informations utiles que nous exploiterons par la suite. Dans un premier temps, nous avons choisi de développer l'algorithme de distance de Levenshtein (présenté dans la partie 2.2) car il n'a besoin ni d'entraînement ni d'historique pour fonctionner. C'est également le plus simple à mettre en place. Nous avons essayé d'optimiser l'algorithme afin d'avoir le moins d'attente possible entre le moment où l'utilisateur tape sur une touche du clavier et le moment où les suggestions s'affichent.

En testant notre algorithme, nous avons pris peur puisque, même après notre optimisation, il prenait trop de temps (environ 1 à 2 secondes). Après quelques recherches, nous avons vu qu'il existait différents profils de compilation tels que : « debug » et « release ». Durant le développement, nous avons travaillé sur le profil de « debug » qui permet une compilation plus rapide mais qui n'est pas optimisée et donc plus lent que le profil « release ». Avec le profil « release » l'algorithme s'est déroulé de façon instantanée et était donc utilisable pour notre application.

Ajout du filtrage par préfixe

Ensuite, nous avons ajouté une fonction qui permet de déterminer si la chaîne écrite par l'utilisateur est le préfixe du mot du dictionnaire que nous sommes en train de traiter. Nous l'utiliserons par la suite pour suggérer les mots les plus cohérents avec la chaîne écrite par l'utilisateur, et non pas pour corriger la chaîne écrite.

Combinaison des critères

Maintenant que nous disposons de deux outils permettant de suggérer un mot plutôt qu'un autre, nous les avons combinés pour construire notre algorithme de suggestions. Au départ, nous avions choisi de prioriser la distance de Levenshtein, puis d'utiliser le préfixe en cas d'égalité entre deux mots, comme le montre la figure 4.3. Cependant, nous avons constaté que les résultats étaient bien meilleurs lorsque l'on inversait cette logique (voir figure 4.4). D'abord filtrer par préfixe, puis départager par distance de Levenshtein. En effet, en privilégiant le préfixe, on obtient des mots qui commencent par la chaîne écrite et qui peuvent être bien plus longs que le mot tapé. Si l'utilisateur fait une faute de frappe (par exemple en inversant deux lettres), la chaîne erronée ne sera probablement pas un préfixe commun avec l'un des différents mots du dictionnaire. Dans ce cas, le filtre par préfixe échoue, et c'est à ce moment que le tri par la distance de Levenshtein prend le relais afin de proposer les suggestions les plus proches.

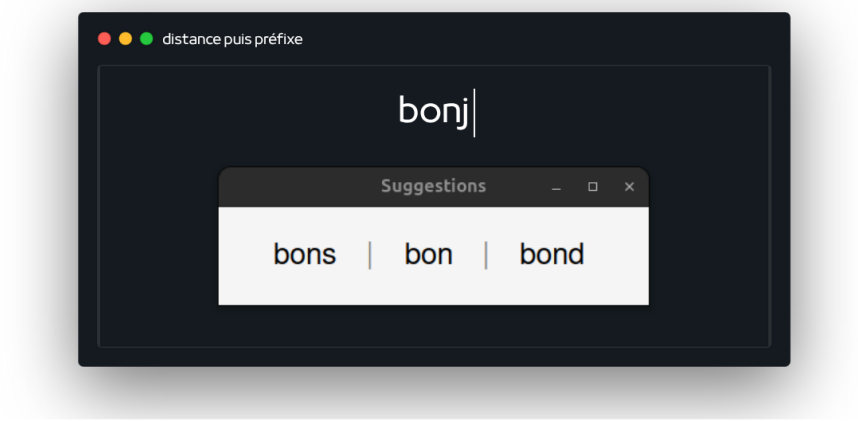


FIGURE 4.3 – Tri par distance en premier

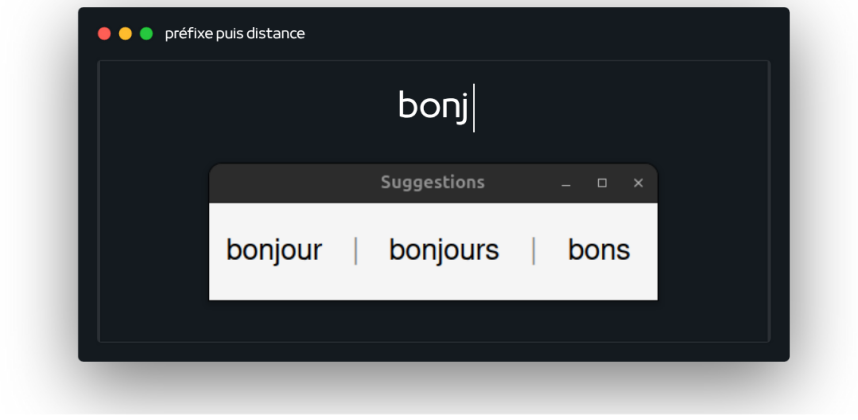


FIGURE 4.4 – Tri par préfixation en premier

Ajout du critère de fréquence d'usage

Cependant, un autre problème est apparu : beaucoup de mots ont la même distance de Levenshtein. Pour les départager, nous avons alors intégré un critère supplémentaire : la fréquence d'usage des mots. Pour cela, nous avons choisi d'utiliser la fréquence d'apparition des mots dans les livres, parmi les différentes options possibles telles que la fréquence d'apparition dans les films ou dans les médias. Nous avons développé un programme en `Python` permettant de convertir les fichiers au format `.tsv` en fichier au format `.csv`, un format que nous maîtrisons mieux. Ce programme nettoie également les données en supprimant les doublons, les mots contenant des espaces, ainsi que les colonnes inutiles. Notre but étant de conserver uniquement le mot et sa fréquence. Ce programme retire donc toutes les autres informations telles que la classe grammaticale, l'infinitif ou encore le nombre de lettres. Nous utiliserons cette fréquence comme critère final de tri dans notre algorithme de suggestions.

Exemple illustratif du fonctionnement de l'algorithme

Prenons l'exemple du mot « bonjour ». Imaginons que, pour l'instant, l'utilisateur a écrit le mot « bonj », alors les suggestions faites grâce au tri des préfixes seront des mots tels que « bonjour », « bonjours » et « bons » comme on le voit dans la figure 4.4. Maintenant imaginons que nous faisons une faute : « bpnj », on remarque qu'aucun mot commence par ce préfixe, on trie alors selon la distance de Levenshtein qui va nous permettre de corriger la faute de frappe. Enfin, le tri selon la fréquence permet de gérer le cas d'une égalité face au tri des préfixes et de la distance : avec les mots « bonjour » et « bonsoir » nous observons que nous avons le même préfixe « bon » et la même distance. Cependant, c'est le mot « bonjour » qui aura la plus grande priorité puisqu'il est beaucoup plus présent dans les livres que le mot « bonsoir ».

La figure 4.5 présente un récapitulatif de notre algorithme de suggestion sous la forme d'un diagramme UML.

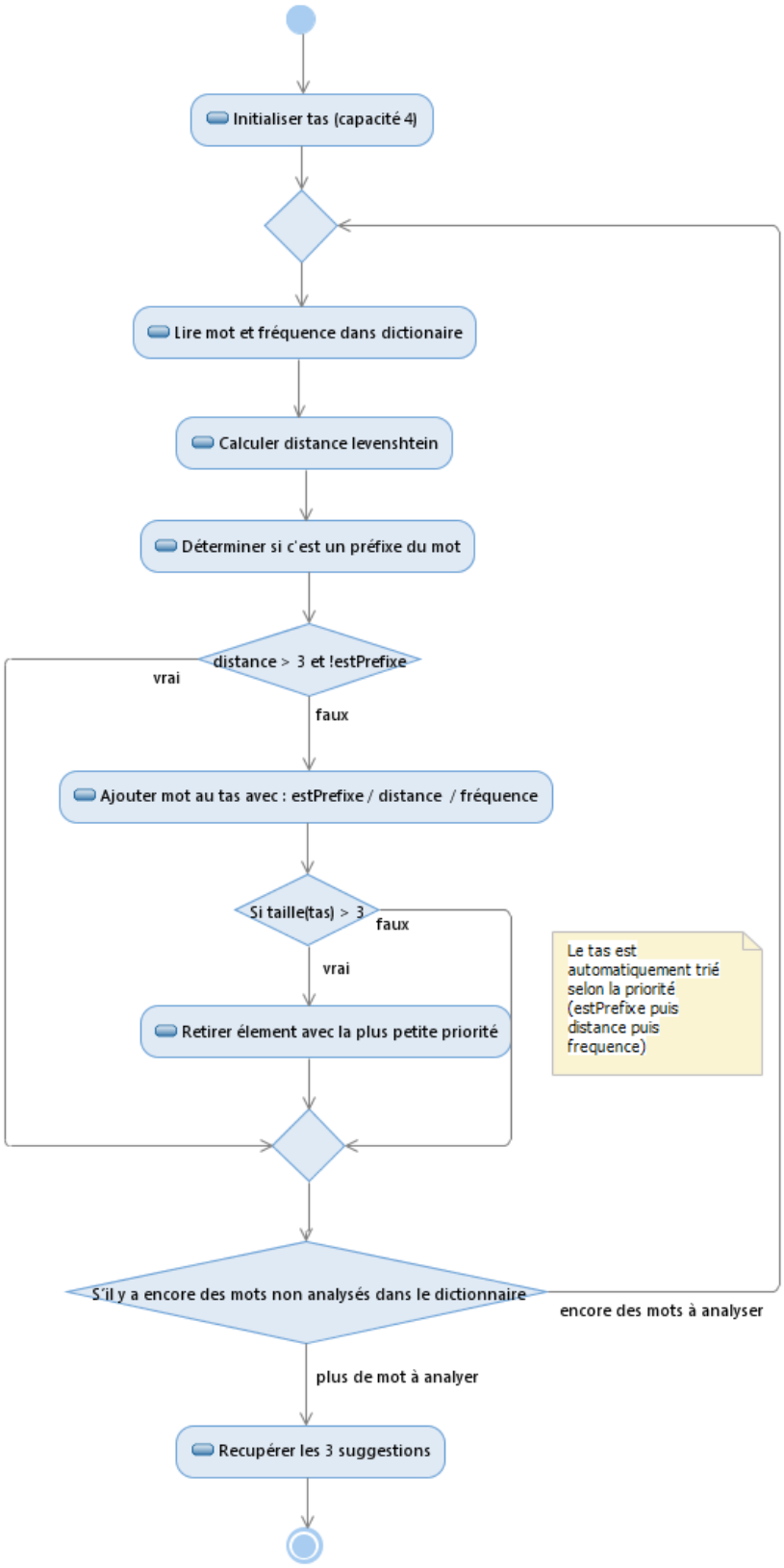


FIGURE 4.5 – Schéma UML de l'algorithme de suggestion

4.2.4 Interface graphique

Maintenant que l'algorithme est fonctionnel, nous devons afficher à l'écran les suggestions proposées grâce à une interface graphique. Nous voulions dans un premier temps faire cette interface en `Rust` avec `GTK`, la boîte à outils de `Rust` mais nous n'avons malheureusement pas réussi à faire en sorte de garder la fenêtre au premier plan lorsque l'utilisateur est sur une autre application. Cette approche devient donc inutilisable car l'application disparait à chaque fois que l'utilisateur clique sur une fenêtre Il ne peut ainsi pas voir les différentes suggestions.

On a donc décidé de faire l'interface en `Python` avec Tkinter car nous l'avions déjà utilisée pour d'autres projets et nous savions comment garder la fenêtre au premier plan. Pour ce faire, notre code `Rust` lance un programme `Python` et lui envoie les 3 mots suggérés via l'entrée standard du code `Python`. Notre code `Python` les affiche dans l'interface. Si l'utilisateur clique sur l'un des trois mots, `Python` envoie l'information au code `Rust` qui va ensuite l'écrire dans l'application de l'utilisateur grâce à notre clavier virtuel.

La plus grosse difficulté de l'interface graphique a été la communication de `Rust` et `Python` mais également de faire en sorte que la page reste au premier plan.

La figure 4.6 illustre l'interface finale de notre application, développée en `Python` à l'aide de la bibliothèque `Tkinter`.

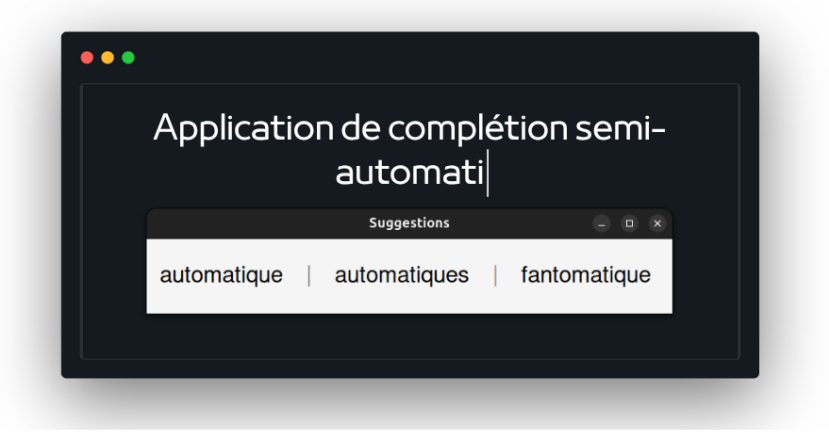


FIGURE 4.6 – Illustration de l'application

4.2.5 Installeur de l'application

Une fois que nous avons tout mis en place et que l'application a été fonctionnelle, nous avons décidé d'ajouter un installeur d'application qui permet de télécharger toutes les dépendances, d'ajouter l'application dans un dossier contenu dans le `PATH`. Le `PATH` est une variable d'environnement utilisée par le système d'exploitation pour localiser les fichiers exécutables utilisable depuis n'importe quel emplacement dans le système.

Notre installeur crée aussi un fichier `.desktop` pour permettre le lancement l'appli-

cation depuis le menu d'application d'`Ubuntu` (voir figure 4.7). Il applique également les droits nécessaires au bon fonctionnement de l'application car certaines commandes nécessitent les droits de l'utilisateur `root`, par exemple la lecture des fichiers des périphériques.

Nous avons géré l'installeur grâce à un Makefile, un fichier permettant d'automatiser un ensemble d'actions telles que la génération de fichiers. Nous avons créé une documentation technique pour installer notre application. Vous pouvez la retrouver à l'annexe de la page 25.

Grâce à toute cette installation et à une icône dans la barre des tâches, l'application est plus facile d'accès.

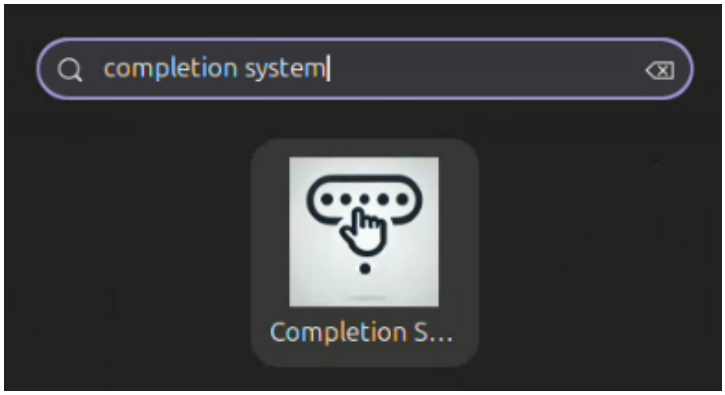


FIGURE 4.7 – Installeur de l'application

4.3 Résultats

Nous avons donc implémenté une application utilisable sur `Ubuntu` permettant une complétion semi-automatique. Cette application peut se lancer depuis un terminal mais également depuis le gestionnaire d'application d'`Ubuntu`.

Une fois l'application lancée, une fenêtre apparait en premier plan et restera toujours en premier plan tant que l'utilisateur ne ferme pas l'application. Si l'utilisateur commence à écrire un mot, trois suggestions seront affichées sur l'interface graphique. Ces suggestions sont calculées grâce à l'algorithme de Levenstein complété avec la gestion des préfixes des mots ainsi que la fréquence d'utilisation des mots proposés. Si l'utilisateur ne veut pas l'un des trois mots proposés, il peut continuer à taper son mot. S'il clique sur l'un des mots, celui-ci sera écrit à la place du mot qu'il était en train d'écrire.

Chapitre 5

Informations complémentaires

5.1 Points à améliorer

Pour améliorer notre application, nous pourrions faire en sorte qu'elle soit compatible sur d'autres distributions Linux et sur le système d'exploitation MacOS. Nous pourrions également changer d'algorithme pour améliorer les suggestions selon l'historique de l'utilisateur par exemple. Enfin, nous aurions également aimé que la fenêtre se place sous le curseur de la souris pour avoir les suggestions toujours sous les yeux mais nous n'avons pas réussi à trouver une solution permettant de récupérer les coordonnées du curseur.

5.2 Outils utilisés

Nous avons eu l'occasion d'utiliser de nombreux outils pour la communication au sein de notre binôme ainsi que pour la réalisation globale du projet semestriel. Nous avons communiqué sur Discord et utilisé le versionnage proposé par Git pour garder une trace des différentes modifications. Nous avons également partagé l'avancée des projets sur GitHub. Nous avons utilisé `Rust`, `Python` et `Tkinter` pour l'implémentation et l'interface graphique. Enfin le rapport a été rédigé sur LaTeX.

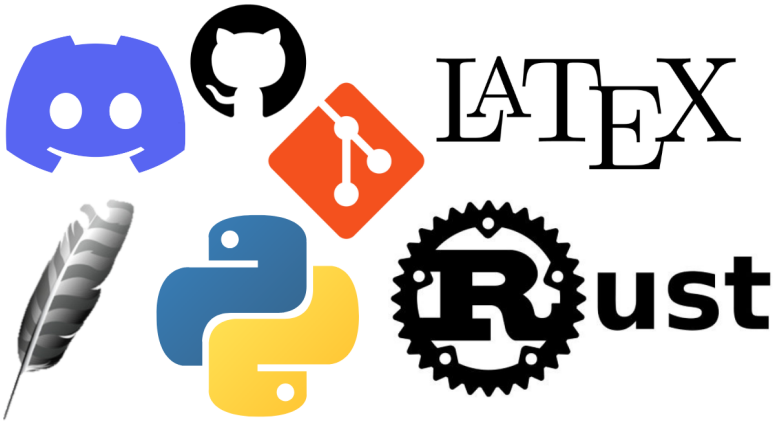


FIGURE 5.1 – Logos des outils utilisés

Conclusion

Pour conclure, ce projet semestriel nous a permis de mettre en pratique des compétences techniques variées, allant de la découverte du monde de la recherche et la mise en place de recherche d'informations, à la mise en application du sujet de nos recherches grâce à `Rust` et `Python`.

La mise en place d'une application après la compréhension du sujet nous ont permis de renforcer nos compétences techniques et nous ont également offert une précieuse expérience en gestion de projet et de collaboration au sein d'un binôme.

Ce projet a donc été une véritable opportunité pour mettre en pratique les compétences que nous avons acquises durant nos deux premières années en CMI informatique, de développer de nouvelles compétences et de découvrir un domaine que nous n'avions jamais abordé auparavant.

Bibliographie

[Bri23] Brilliant.org. Markov chains, 2023. https://brilliant.org/wiki/markov-chains/.

[Dev24] PyO3 Developers. Pyo3 - rust bindings for python, 2024. https://pyo3.rs/v0.24.2/.

[Gee21] GeeksForGeeks. Damerau-levenshtein distance, 2021. https://www.geeksforgeeks.org/damerau-levenshtein-distance/.

[GN24a] Eddie Gerbais-Nief. Apprendre rust - chapitre 2, 2024. https://www.youtube.com/watch?v=wgjw5lGv-EI.

[GN24b] Eddie Gerbais-Nief. Apprendre rust - chapitre 3, 2024. https://www.youtube.com/watch?v=3kBk3sjREOM.

[GN24c] Eddie Gerbais-Nief. Apprendre rust - introduction, 2024. https://www.youtube.com/watch?v=mZasv3__A9k.

[KN24] Steve Klabnik and Carol Nichols. The rust programming language, 2024. https://doc.rust-lang.org/book/.

[lin23] linuxembedded.fr. Une introduction à uinput, 2023. https://linuxembedded.fr/2023/04/une-introduction-a-uinput.

[NP19] Boris New and Christophe Pallier. Lexique 3.83. http://www.lexique.org/, 2019. Lexical database for French, version 3.83.

[Ove10] Stack Overflow. How can i translate linux keycodes from /dev/input/event to ascii in perl ?, 2010. https://stackoverflow.com/questions/2547616/how-can-i-translate-linux-keycodes-from-dev-input-event-to-ascii-in-perl.

[Sch21] Data School. Auto-complete with deep learning, 2021. https://www.youtube.com/watch?v=NEaUSP4YerM.

[Sin20] Gurdeep Singh. Keylogger, 2020. Code source https://github.com/gsingh93/keylogger/tree/master.

[ucc24a] The udev crate contributors. udev — rust documentation, 2024. https://docs.rs/udev/latest/udev/.

[ucc24b] The uinput crate contributors. uinput — rust documentation, 2024. https://docs.rs/uinput/latest/uinput/.

[Wik24] Wikipedia. Distance de levenshtein, 2024. https://fr.wikipedia.org/wiki/Distance_de_Levenshtein.

[wJS20] StatQuest with Josh Starmer. Markov chains - clearly explained, 2020. https://www.youtube.com/watch?v=uvYTGEZQTEs.

Annexe 1 : Documentation technique

Description générale

Ce projet implémente un outil de complétion semi-automatique de texte, utilisable dans toutes les applications sur un système Ubuntu.

Installation

Prérequis

— Système **Ubuntu** (testé sur Ubuntu 22.04.2 LTS)
— Le raccourci clavier **Alt + Tab** doit être actif (natif sur Ubuntu) pour changer d'application facilement

Étapes d'installation

```
cd completion-system
make install
```

Le `Makefile` s'occupe de :
— Installer les dépendances nécessaires
— Copier le binaire compilé dans `/usr/local/bin/` (inclus dans le PATH)
— Donner les bons droits d'exécution à tous les fichiers nécessaires
— Créer le fichier .desktop pour le menu d'applications

Lancement

Deux méthodes sont possibles :
— **Depuis le terminal** :

```
completion-system
```

— **Depuis le gestionnaire d'applications Ubuntu** : Ouvrir le menu et rechercher `Completion System`, puis appuyer sur Entrée.

Remarque : sur certains ordinateurs, il est possible que le lancement via le gestionnaire d'applications ne fonctionne pas. Dans ce cas, le terminal reste pleinement fonctionnel.

Une fois lancé, le programme tourne en arrière-plan et fonctionne dans **toutes les applications**.

Désinstallation

```
cd completion-system
make uninstall
```

Binaire autonome

Le fichier binaire `completion-system`, déjà compilé et fourni dans le dossier `bin/`, est entièrement autonome après installation.

— Il est copié dans `/usr/local/bin`, ce qui permet de le lancer depuis n'importe où.
— Il ne dépend plus des fichiers sources ni du scripts Python.

Structure des fichiers

L'arborescence du projet est la suivante :

```
.
|- data/
|       |- dico\_freq.csv
|- src/
|       |- gui.py
|       |- keylogger.rs
|       |- main.rs
|       |- mouselogger.rs
|       |- offset.rs
|       |- python_gui.rs
|       |- suggestions.rs
|       |- virtual_input.rs
|- tools/
|       |- format_dico.py
|       |- Lexique383.tsv
|- Cargo.toml
|- completion-system.png
|- Makefile
|- README.md
```

— `src/` : contient tous les fichiers source Rust ainsi que le script Python pour l'interface graphique (`gui.py`).
— `data/dico_freq.csv` : dictionnaire final utilisé pour la complétion.
— `tools/Lexique383.tsv` : base de données brute téléchargée.
— `tools/format_dico.py` : script Python utilisé pour convertir le fichier `.tsv` en `.csv`, il permet de plus d'enlever les informations inutiles.
— `Cargo.toml` : fichier de configuration qui indique les dépendances Rust et leurs versions.
— `completion-system.png` : icône affichée dans le menu d'applications Ubuntu.

Résumé

Rapport de notre projet semestriel de deuxième année de licence en Cursus Master Ingénierie Informatique à l'Université de Marie & Louis Pasteur, portant sur la complétion semi-automatique, les méthodes qui existent, les algorithmes possibles pour la mise en place de la complétion semi-automatique et les améliorations possibles grâce aux chaînes de Markov. Nous avons ajouté notre implémentation permettant de proposer des suggestions de corrections des mots sur n'importe quelles applications sous Linux. Durant ce projet, plusieurs thèmes ont été abordés : l'utilisation de Git, de `Python` et de `Rust`, les bases informatiques et mathématiques de la complétion semi-automatique.
Dans ce rapport, nous étudions les différentes approches qui existent ainsi que les différents algorithmes de distance. Ensuite, nous parlons des chaînes de Markov pour la gestion d'historique et enfin vous retrouverez l'application que nous avons créé permettant une complétion semi-automatique.

Mots-clés

Rust - Python - Tkinter - Complétion semi-automatique - Levenshtein - Markov - Résolution de problème - Implémentation -Recherche - Application

Abstract

Report on our semester-long project for the second year of the Cursus Master Ingénierie Informatique degree program at the Université de Marie & Louis Pasteur, focusing on semi-automatic completion, existing methods, possible algorithms for implementing semi-automatic completion and possible improvements using Markov chains. We have also added our own implementation, enabling us to offer word correction suggestions on any Linux application. During this project, several topics were covered : the use of Git, `Python` and `Rust`, the computational and mathematical foundations of semi-automatic completion.
In this report, we study the different approaches that exist, as well as the different distance algorithms. Then we'll talk about Markov chains for history management, and finally you'll find the application we've created for semi-automatic completion.

Key-Words

Rust - Python - Tkinter - Semi-automatic completion - Levenshtein - Markov - Problem solving - Implementation - Research - Application