Utilisation des grands modèles de langages pour la detection et la correction des erreurs

Asia Auville

▶ **To cite this version:**

Asia Auville. Utilisation des grands modèles de langages pour la detection et la correction des erreurs. Informatique [cs]. 2024. hal-04877570

HAL Id: hal-04877570
https://inria.hal.science/hal-04877570v1

Submitted on 9 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Rapport de stage de fin d'études

Utilisation des grands modèles de langages pour la détection et la correction des erreurs

Encadrants:
Popov Mihail
Saillard Emmanuelle

Auville Asia

2024

1 Introduction

1.1 Mon parcours

Comme beaucoup d'élèves ingénieur·e·s, après l'obtention de mon Baccalauréat S je me suis dirigée vers une classe préparatoire. Je n'avais pas d'objectif de carrière précis en tête, je pensais vaguement à l'enseignement depuis quelques années mais je n'avais aucun projet fixe. J'y ai découvert une attirance pour l'informatique, et tout naturellement cela m'a conduite vers une école proposant un parcours avec une filière spécialisée dans le domaine. Mes stages de première et de deuxième année à l'ENSEIRB-MATMECA ont été très différents: la première année, j'ai assisté une doctorante dans ses recherches par le développement d'une interface web pour une expérimentation. La deuxième année, j'ai fait un stage de développement logiciel dans une Start-Up bordelaise. Ces deux expériences m'ont permis de nourrir ma réflexion sur ce que je voulais faire après mon diplôme. Le milieu de la recherche m'a de plus en plus séduite, j'ai donc décidé de rechercher un stage de fin d'études me permettant de mieux découvrir ce milieu, avec comme éventuel objectif de pouvoir effectuer un doctorat dans la continuité de mon stage.

J'ai eu l'opportunité d'être contactée par mon maître de stage, Mihail Popov, qui a été un de mes encadrants de TD en 2ème année à l'ENSEIRB-MATMECA. Après un entretien, nous avons décidé de travailler ensemble sur ce sujet, qui était en l'occurrence un travail préliminaire à un sujet de doctorat dont le financement était déjà obtenu. Ce sujet est à cheval entre l'intelligence artificielle, la science des données, et le Calcul Haute Performance. Ayant passé mon semestre 9 à l'étranger, je n'ai pas pu profiter des options de spécialisation proposées par l'école et j'étais un peu perdue sur le domaine dans lequel je voulais m'orienter. Ce sujet m'a intéressée et j'étais très enjouée à l'idée de pouvoir développer mes connaissances dans des domaines que je ne connaissais que très peu.

Ce stage a été, comme je l'avais espéré, la confirmation que je voulais travailler dans la recherche et que je voulais faire un doctorat. J'ai pu faire l'expérience du rythme de travail en thèse, et j'ai pu travailler assez sur le sujet pour être sûre qu'il me plaisait. Je suis consciente de la chance que j'ai eu de m'être fait proposer une offre de thèse sur le même sujet que mon stage et je suis très reconnaissante envers mes encadrants de m'avoir fait confiance. Je vais donc pouvoir poursuivre mon travail sur ce sujet pendant 3 ans, et exploiter moi même les résultats des recherches que j'ai effectuées pendant ces 6 mois.

1.2 Contexte

Mon stage s'est déroulé au centre Inria de l'université de Bordeaux. Inria est l'institut national de recherche en sciences et technologies du numérique. J'ai travaillé plus particulièrement dans l'équipe STORM, dont la mission est ciblée sur le calcul haute performance, notamment sur l'optimisation statique et les

méthodes d'exécution. L'équipe est composée d'une trentaine de personnes en comptant les nombreuses et nombreux doctorantes, doctorants et stagiaires, ce qui favorise un échange constant entre les différents membres de l'équipe venant de tous horizons. J'ai été encadrée durant tout mon stage par Mihail Popov et Emmanuelle Saillard, ainsi que par Eric Petit et Pablo de Oliveira lors de réunions hebdomadaires en visioconférence.

2 Grands axes du sujet

Ce sujet de stage s'inscrit dans un sujet de thèse en 3 ans, et le stage sert en particulier à poser les bases pour le travail de thèse qui suivra. Un des objectifs est donc de définir ce qui aidera à générer des résultats à long terme pour un projet de plus grande envergure.

Ce projet est né d'une observation dans le domaine du HPC (High Performance Computing, ou Calcul Haute Performance en français) : il existe un besoin d'outils de vérification capables de détecter des erreurs avec précision dans le code, et l'essor actuel des LLM (Large Language Models) ouvre la voie à un nouveau type de solution. Actuellement, des outils comme GitHub Copilot ou GPT-4 permettent de détecter et corriger des erreurs dans du code, mais ils sont très généraux et peu adaptés aux problèmes spécifiques du HPC. Par ailleurs, bien que certains travaux prometteurs aient montré qu'il est possible de détecter certaines erreurs à petite échelle (en utilisant des bases de données de 3000 programmes synthétiques générés à l'aide de scripts), ces méthodes restent limitées.

Les outils de vérification existants pour le HPC ne répondent pas pleinement à nos besoins pour plusieurs raisons. D'abord, ils sont souvent fastidieux à installer et configurer. De plus, aucun de ces outils n'est capable de détecter toutes les erreurs possibles pour un modèle comme MPI, et la plupart utilisent une analyse dynamique qui nécessite l'exécution des codes. Cette méthode d'analyse peut être très chronophage et dépend du jeu de données d'entrée, ce qui peut entraîner la non-détection de certaines erreurs.

Notre objectif est donc de créer un modèle à grande échelle, capable dans un premier temps de détecter certaines erreurs spécifiques dans du code HPC, puis à terme de les corriger. Pour ce faire, nous comptons utiliser des méthodes de *clustering* et de *natural language processing* (que je détaillerai plus loin) sur un ensemble de données représentant des corrections d'erreurs dans le code, ainsi que les messages de log associés en langage naturel. L'intuition est que ces messages de log, ou messages de commit, censés décrire les modifications lors d'un changement dans un programme, pourraient fournir une clé supplémentaire pour la classification automatique des données et la reconnaissance des formats d'erreurs. Nous devons donc réunir une grande quantité de données et créer un ensemble de données cohérent, tout en déterminant la meilleure manière de représenter les données récupérées, notamment grâce à des méthodes d'*embedding*.

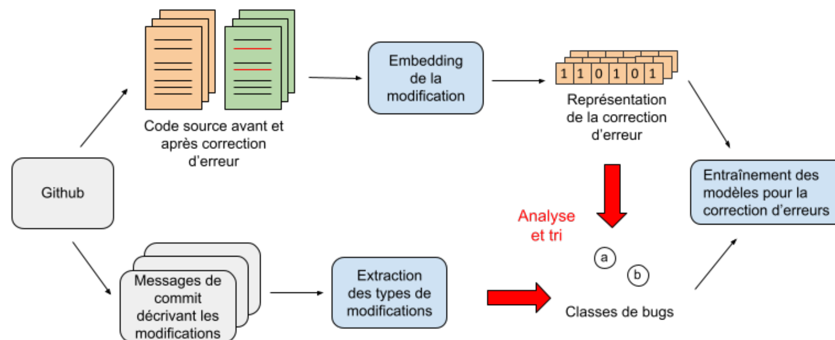

Figure 1: Objectif du projet à long terme

Mon stage s'inscrit alors dans ce contexte à long terme. Les travaux qui me sont confiés sont donc:

- Faire une étude de l'existant, afin de déterminer toutes les solutions envisageables pour la classification et la représentation des données.
- Récolter des données utilisables pour entraîner le modèle.
- A l'aide de différentes méthodes, filtrer, classifier et nettoyer les données récoltées.
- Représenter correctement les données, et juger si la représentation est capable d'identifier et de catégoriser les erreurs que nous souhaitons corriger.

La figure 1 représente visuellement les grandes étapes du projet qui seront à explorer et traiter.

3 Récoltes de données

3.1 Bases de données existantes

3.1.1 État des lieux

Mes premières recherches se sont concentrées sur la nécessité de disposer de données pour entraîner le modèle que nous cherchons à développer. Mon premier réflexe a été de vérifier si une ou plusieurs bases de données existaient déjà et pouvaient être exploitées, soit en l'état, soit avec quelques modifications.

Pour rappel, l'objectif est de constituer une collection de codes en C, idéalement du code MPI (ou Message Passing Interface, modèle de programmation parallèle à mémoire distribuée), avec des couples de données : nous avons besoin des versions de code avant et après une modification (qui corrige une erreur

ciblée), ainsi que du message de log laissé par le développeur décrivant cette modification. Ce message de log pourra par exemple être utilisé comme label pour notre modèle et servira à appliquer des scripts de *NLP (Natural Language Processing)* pour aider à la reconnaissance et à la catégorisation des erreurs.

Un travail a déjà été effectué lors d'un ancien stage sur une base de données de codes synthétiques (générés à l'aide d'algorithmes) et les résultats étaient prometteurs. La quantité de fichiers de codes que nous allons récupérer permettra de passer à l'échelle supérieure par rapport aux tests déjà effectués avec une base de données de codes synthétiques. Les textes associés (messages de commit, message d'issue, ou autre) nous permettront d'avoir une information supplémentaire pour caractériser le changement dans le code. Nous sommes également à la recherche de toutes autres données qui nous permettraient d'améliorer les performances de notre modèle et d'avoir des informations en plus sur nos modifications de code.

J'ai pu explorer principalement trois solutions de bases de données pour collecter les données d'entraînement de mon modèle.

- La première, la base de données *The stack - Github issues* [2], comprend 66 Go de données sur des *issues* GitHub, avec des conversations entre utilisateurs et des informations sur les *Pull Requests* associées. Son principal problème est que le code associé aux *issues* n'est pas toujours de bonne qualité, et il n'y a aucun filtre sur le langage de programmation utilisé. La granularité du changement de code considéré ici est au niveau de la Pull Request GitHub.
- La deuxième que j'ai explorée, *MPICodeCorpus* [8], est un corpus de code MPI récupérés depuis GitHub. Bien qu'il contienne des codes courts et de bonne qualité, les codes sont anonymisés et donc non traçables sur GitHub, ce qui rend la récupération des conversations textuelles entre utilisateurs impossible ou alors très compliquée. De plus, comme notre étude se focalise sur les modifications de code, par exemple à la hauteur d'un commit, il est important de pouvoir retracer les versions des codes, et c'est impossible ici à cause de leur anonymisation.
- Enfin, l'outil *EasyPap* [1] nous donne accès à des codes d'étudiants avec des labels de performances associés, mais les messages de commit écrits par les étudiants ne sont pas assez qualitatifs pour être exploitables. Les labels de performance sont cependant une piste intéressante pour pouvoir caractériser les données et possiblement quantifier et évaluer un changement dans le code.

La table 1 présente un récapitulatif des avantages et inconvénients des trois bases de données explorées.

EasyPAP	MPICodeCorpus	The Stack	
✓	x	x	labels
✓	✓✓	x	codes
x	x	✓✓	texte

Table 1: Etat des lieux des trois bases de données explorées. Une croix représente l'absence de données pour la base considérées, et un ou deux checks représentent la présence plus ou moins forte de ces données. Le texte ici est un ensemble de messages de log liés à une modification de code, et le label est une catégorisation du changement selon un critère défini.

On peut voir qu'aucune des bases de données ici n'est assez satisfaisante pour l'utiliser telle quelle, et donc nous avons du faire un choix sur la piste à suivre. Nous avons alors décidé d'ignorer la non-exploitabilité des codes de la première base de données (The stack - GitHub issues) et de se concentrer sur son grand corpus de messages d'*issues* GitHub afin de tester notre intuition sur la classification des données grâce au NLP.

3.1.2 Filtrage de The stack - GitHub Issues

A partir du dataset sélectionné, il a fallu faire un premier traitement sur les données avant de pouvoir les exploiter. L'objectif était d'extraire une série de textes correspondant à des *issues* sur GitHub. A partir de ces textes, nous voulions exécuter des algorithmes de *NLP* afin de faire du *clustering* pour essayer d'extraire des thèmes du corpus. La structure de base du dataset indiquée dans sa documentation sur le site huggingface est présentée ci dessous:

```
1       Dataset({
2       features: ['repo', 'issue_id', 'issue_number', 'pull_request', 'events',
        ↪  'text_size', 'content', 'usernames'],
3       num_rows: 30982955
4   })
5
```

Le processus de filtrage commence par le script `format_db.py`, qui lit les données du dépôt cloné de la base de données [2]. Il extrait un certain nombre de lignes du dataset, élimine les lignes qui correspondent à des *issues* sans *Pull Request* associée, conserve uniquement les colonnes pertinentes et enregistre le tout dans un seul dataframe au format parquet.

Ensuite, `get_pr_info.py` traite ce fichier. En utilisant un argument de ligne de commande, je parcours avec ce script les lignes de *data_filtered.parquet* et les enregistre dans *data_filtered_with_pr_info.parquet*, en filtrant les PR (*Pull Request*) non fusionnées, rebasées ou introuvables. De cette manière, je garde uniquement les PR exploitables pour chercher dans le code une différence avant/après

générée par celles-ci. Ce script ajoute également deux colonnes avec les hash des commits avant et après chaque PR sur la branche principale. Cette étape sert à récupérer les versions de "code avant" et "code après" la modification induits par la PR afin de faire la comparaison par la suite. Ce script peut être exécuté plusieurs fois sans créer de doublons, avec des logs générés à chaque exécution.

Le script `fix_code_extraction.py` utilise ensuite la base de données *data_filtered_with_pr_info.parquet* pour récupérer les fichiers concernés par les PR ayant des différences entre les deux commits. Enfin, `format_for_nlp.py` transforme *data_filtered_with_pr_info.parquet* en CSV, corrige les caractères illisibles et formate le fichier pour qu'il soit prêt pour le clustering. La figure 2 résume le processus de filtrage.

Je me suis rendue compte à cette étape que l'étude de cette base de données ne pourra pas être complète car elle ne nous permettra pas d'aller plus loin du point de vue des codes sources. En effet, cette base de données n'a aucune considération pour la qualité des programmes associés aux textes analysés. En essayant de filtrer à nouveau mes données pour ne garder que les *issues* associées à des *PR* concernant moins de 3 fichiers différents et écrits uniquement en langage C ou C++, on obtient un nombre bien trop faible de données exploitables (environ 0,25% des données de base). Par conséquent, j'ai pris la décision d'étudier cette base de données uniquement par rapport à ses messages d'*issues*, et de créer une autre base de données plus adaptée pour l'étude de codes.

La piste EasyPap n'a pas été abandonnée, mais elle a été mise de côté. Je n'ai pas eu le temps de l'exploiter dans le cadre du stage, mais dans une vision à plus long terme, c'est une base de données qu'il reste à étudier.

3.2 Minage de GitHub

Comme aucune des bases de données précédemment étudiées n'était entièrement adaptée à l'étude que je voulais en faire, j'ai décidé de réunir par moi même des données depuis GitHub sans partir d'une base de données existante. Cette fois ci, je voulais être sûre que les morceaux de codes que j'allais sélectionner allaient correspondre à mes besoins. J'ai utilisé le script *github-clone-all* [7], qui est le même outil que les auteurs d'une des bases de données [8] que j'avais explorée ont utilisé. Il permet très facilement de cloner sur ma machine une grande quantité de répertoires GitHub ayant des caractéristiques précises.

J'ai donc récupéré de GitHub environ 1000 répertoires avec un nombre d'étoiles arbitrairement supérieur à 30 (pour m'assurer d'une certaine qualité dans le code), et dont la mention "MPI" apparaissait dans le titre, le fichier README.md ou dans la description. A partir de ces répertoires, j'ai récupéré tous les fichiers en C ou en C++, puis j'ai vérifié si ceux çi contenaient la ligne suivante:

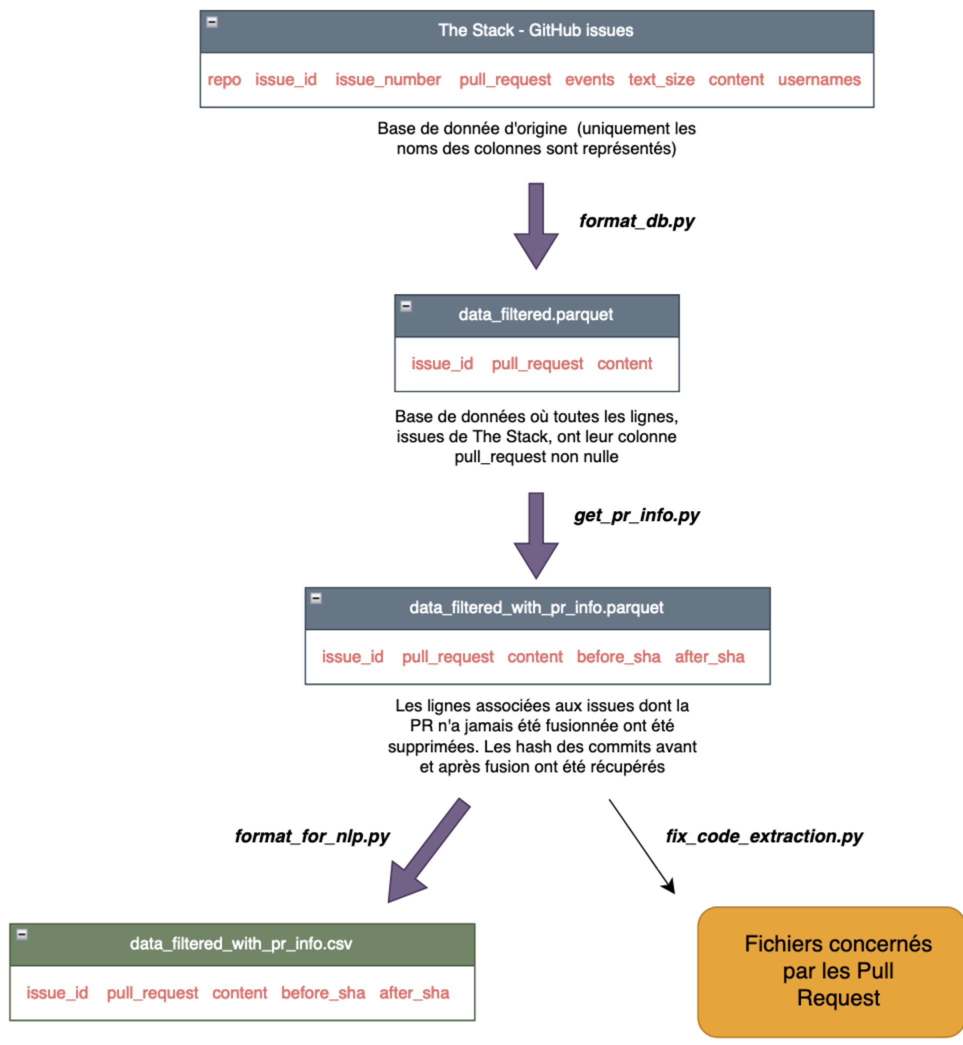

Figure 2: Processus de filtrage de la base de données *The Stack - GitHub Issues* pour la récupération des textes d'*issues* et des fichiers modifiés

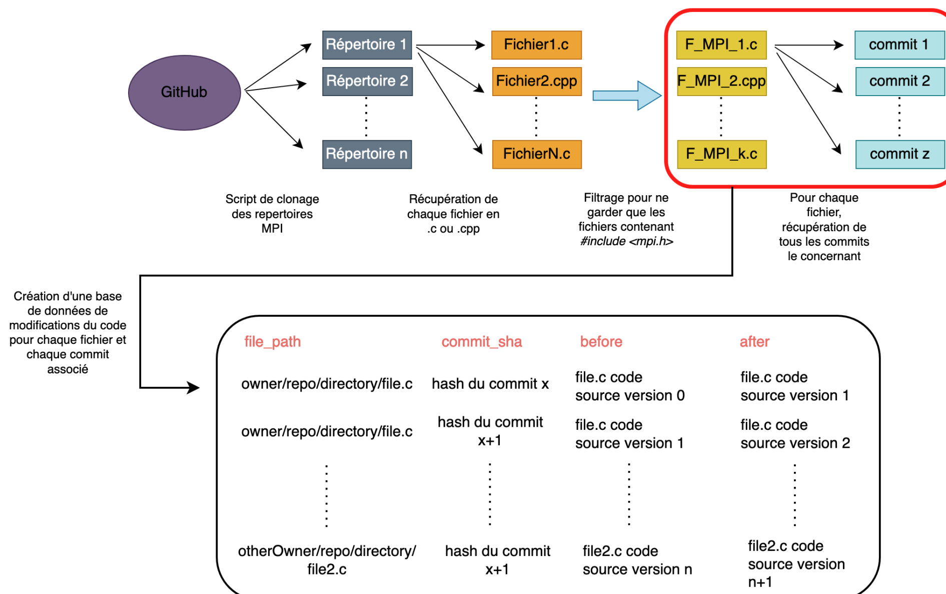

Figure 3: Processus de minage de GitHub et de création de la base de données de modifications de code MPI

```
1 #include <mpi.h>
```

J'ai déterminé que ce critère était en premier lieu suffisant pour récupérer les fichiers de code utilisant la librairie. Cependant, pour avoir une base de données plus raffinée, il faudrait faire d'autres tests plus poussés sur le contenu du code afin de voir si il convient vraiment.

Une fois la liste des fichiers récupérée, je vais, à l'aide d'une commande *git log* avec les options adaptées, récupérer la liste des hash de tous les commits qui introduisent une modification sur ce fichier. Ces hash vont me permettre d'accéder à toutes les versions des fichiers que j'ai sélectionnés, et de les enregistrer dans une base de données deux par deux, en mettant face à face les deux versions du même code avant et après chaque commit qui le concerne. Par la suite, je ferai référence à ces deux versions par rapport à un commit précis par "code avant" et "code après". La figure 3 détaille le processus en entier.

J'ai de cette manière pu récupérer dans un premier lieu environ 30 000 couples code avant/code après à partir d'environ 100 répertoires GitHub. Cette méthode est totalement reproductible avec d'autres critères de sélection des répertoires (par exemple moins d'étoiles requises).

Cette base de données, bien que fonctionnelle pour les premiers tests que je voulais effectuer, n'est pas optimisée: en effet, stocker en double presque chaque

fichier de code source (en tant que code avant et code après deux différents commits) semble être une mauvaise utilisation de la mémoire. De plus, il est possible que le fait de considérer dans certains cas un grand nombre de versions du même code implique un biais de sélection dans les données. Cette base de données ayant comme vocation d'être utilisée et analysée par des algorithmes de clustering, il est donc important de vérifier que l'on n'a pas introduit des patterns involontaires. Il y aurait donc encore un travail de vérification et de nettoyage à faire pour que cette base de données puisse être considérée comme un point de départ fiable pour des études ultérieures.

4 Méthodes d'embedding

Pour pouvoir exploiter des données, il faut en premier lieu trouver une manière de les représenter de manière plus compacte et uniforme, c'est à dire par des vecteurs. En effet, l'objectif principal est d'étudier si on arrive à trouver des liens entre des programmes MPI qui comportent les mêmes erreurs en les représentant sous forme de vecteurs et en les comparant. On espère trouver une méthode de représentation vectorielle des données qui saura capturer des motifs inhérents à certains types d'erreurs difficilement détectables avec des outils classiques. Nous allons donc chercher différentes méthodes de représentations vectorielles, que l'on appelle ici des *embeddings*. Un embedding est défini par une représentation vectorielle d'une donnée choisie. Dans cette partie, je détaillerai les différentes méthodes explorées afin de pouvoir justifier les méthodes choisies parmi la grande diversité déjà existante.

4.1 Embedding de langage naturel

L'embedding des textes est l'étape qui va nous permettre de transformer un corpus de textes en un ensemble de vecteurs, censés traduire par leurs dimensions les liens sémantiques existant entre les textes. Pour faire cela, j'ai récupéré un programme déjà écrit par mon encadrant pour un autre projet, et je l'ai retravaillé pour l'adapter à notre contexte. Je n'ai donc pas écrit directement le code moi même mais j'ai bien compris son fonctionnement et chacune de ses étapes.

Le principe global repose sur l'analyse de fréquences d'apparition des mots dans un texte, et sur le calcul de l'impact qu'ont ces mots sur chacun des textes en considérant le corpus tout entier. Pour être plus claire, je m'appuierai sur un exemple fictif;

Imaginons un corpus de 10 textes et choisissons un texte au hasard qu'on appelle "texte A". Nous cherchons à donner un poids à chaque mot du texte en fonction de l'importance de sa contribution au thème qui ressort de celui ci. Par exemple, des mots comme "le, la, les" sont susceptibles d'apparaître beaucoup de fois, et dans tous les textes. Nous voulons donc leur accorder un poids faible. Il se trouve que le mot "chat" apparaît dans le texte A, mais dans aucun des

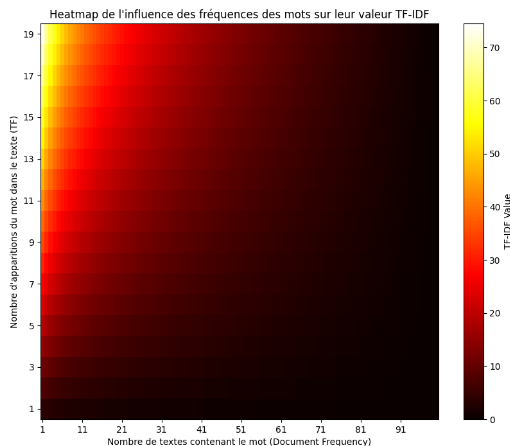

Figure 4: Heatmap représentant l'influence des fréquences des mots sur leur valeur TF-IDF, pour un nombre de textes dans le corpus fixé à 100. Plus la couleur est claire, plus la valeur TF-IDF du mot choisi pour son texte associé sera grande.

autres textes. Nous décidons donc qu'il est trop marginal pour être pertinent. Cependant, le mot "voiture" apparaît plusieurs fois dans le texte A, ainsi que dans quelques autres textes du corpus. Nous pouvons alors lui donner un caractère discriminant, et reporter sa forte contribution au texte dans la dimension associée de son vecteur.

Ce principe va être matérialisé par les techniques de TF-IDF (Term Frequency-Inverse Document Frequency) [12] qui servent à l'analyse de fréquence relative de mots dans un texte appartenant à un corpus. La figure 4 représente une *heatmap* montrant l'influence de la fréquence des mots sur leur valeur TF-IDF. Dans notre cas, nous allons supprimer du corpus tous les mots apparaissant dans trop peu de textes, malgré leur valeur TF-IDF élevée, car nous les considérons comme trop marginaux.

Comme nous n'allons pas utiliser des vecteurs aussi grands que le nombre de mots dans le corpus, nous allons utiliser la méthode NMF (*Non negative matrix factorization*) [11] sur la matrice des poids TF-IDF de chaque mot de chaque texte du corpus, et extraire un nombre de *topics* arbitraire et prédéfini. Chaque *topic* représente un ensemble de mots qui sont souvent utilisés et pertinents ensemble, et ils servent à extraire un ensemble de thèmes du corpus de textes.

Après tout ce traitement, nous obtenons pour chaque texte du corpus, un vecteur à n dimensions (n correspond au nombre de topics), représentant sa

contribution et son lien à chaque *topic* représenté sur chaque dimension.

Dans mon cas, j'ai exécuté cet algorithme sur mon corpus de texte récupéré depuis la base de données *The stack - GitHub issues*. Chaque texte est une concaténation du texte original de l'issue GitHub et de l'ensemble des discussions qui ont suivi. Cela m'a permis de récupérer une liste de vecteurs représentant de manière distribuée chaque texte sémantiquement par rapport aux autres.

4.2 Embedding de code C et C++

Pour créer des embeddings sur du code de programmation, les problématiques sont différentes. En effet, les liens entre les différents mots d'un texte écrit dans un langage de programmation sont différents de ceux écrits en langage naturel, et pour représenter correctement le sens profond de chaque programme en un seul vecteur, il est primordial de capturer les liens spécifiques qui sont formés. Mon travail dans cette partie consiste en grande partie en de la recherche et de la documentation sur ce qui existe. Je ferai donc un résumé de ce que j'ai pu apprendre en lisant des articles scientifiques, et de ce que j'ai retenu de pertinent pour ce projet.

4.2.1 Embedding de code source "pur"

Plusieurs méthodes ont été ici envisagées. La première est l'embedding à base de tokens. De la même manière que avec du langage naturel, un outil va segmenter le programme en tokens, dans les faits un token représentera un ou quelques mots, ou morceaux de mots. Chaque token va ensuite être encodé à l'aide d'un vocabulaire pré existant qui lie un token à une valeur vectorielle. Puis les vecteurs vont être assemblés de manière à n'en former qu'un et représenter le programme en entier.

En premier lieu, j'ai envisagé d'utiliser *CodeBERT* [3], un outil d'embedding très connu et réutilisé dans énormément de projets scientifiques. Le problème est qu'il n'a pas été entraîné pour fonctionner sur du code C, ce qui est bloquant pour nous. J'ai essayé de trouver des travaux de personnes tierces qui auraient adapté *CodeBERT* pour du langage C mais je n'ai rien trouvé de très concluant. J'ai donc laissé cette piste de côté.

J'ai également découvert que dans la plupart des articles que je lisais, l'utilisation de l'AST (Abstract Syntax Tree) semblait apporter un réel avantage par rapport à un embedding purement basé sur des tokens, dans le sens où l'AST permet de capter beaucoup plus de sens sémantique et de lien entre les différentes parties du code. Cette méthode nécessite donc de générer l'AST, et peut alors poser des problèmes lors de sa génération si le code comporte des erreurs, car l'AST nécessite un code sans erreur lexicale ou syntaxique. Passer par les AST soulève alors un problème: si nous sommes obligés d'exclure les codes qui ne génèrent pas d'AST correct, nous risquons d'introduire un biais dans le set de

données utilisé pour l'entraînement, et donc d'avoir un modèle incomplet qui aurait exclu certains types d'erreurs qui empêchent la génération de l'AST. Cependant, d'un autre coté l'utilisation d'AST est peut être profitable car ce genre d'erreur représente principalement des erreurs de syntaxe grossières, cela permettrait donc de faire un premier filtre sans effort des erreurs qu'on ne veut pas considérer. Les erreurs que nous cherchons sont compliquées et non détectables simplement par un compilateur, contrairement aux erreurs qui pourraient gêner la génération de l'AST. Utiliser l'AST semble alors réaliste dans ce contexte de recherche d'erreurs complexes.

J'ai cependant décidé de focaliser mes efforts sur une méthode d'embedding qui était compatible et entraînée tout particulièrement pour des codes longs, les *Jina Embeddings* [4]. Comme ma base de données comporte des programmes qui peuvent avoir un nombre de lignes très conséquent et que la plupart des méthodes d'embedding existantes ne peuvent se focaliser que sur des petits morceaux de texte ou de code, cet outil spécifique m'a paru adapté. C'était le meilleur moyen de pouvoir faire des premières expérimentations rapidement sur des représentations de code.

Avec cette méthode, j'ai alors généré environ 2500 couples d'embeddings des codes avant/après de la base de données que j'avais créée. Je n'ai pas encore pu traiter les 30 000 lignes existantes car la génération d'un embedding prend beaucoup de temps et la puissance de calcul dont je dispose ne m'a pas permis d'en générer plus. Cependant, nous avons déterminé que pour un premier test ce nombre de couples était suffisant.

4.2.2 Embedding de modifications de code

En parallèle de mon étude des méthodes d'embedding de code source, j'ai essayé de trouver des travaux sur des méthodes d'embedding qui capturent directement en un vecteur le changement entre deux versions de code, au lieu de générer deux vecteurs séparés. Cette méthode parait beaucoup plus fiable et logique: à partir de deux vecteurs séparés de deux versions d'un code, il reste encore un travail conséquent à faire pour en obtenir un seul qui représenterait correctement le changement entre les deux programmes. Il m'a donc paru pertinent de chercher un outil déjà développé sur cette tâche car celle ci est loin d'être triviale.

Parmi les outils que j'ai explorés, on peut citer *CCRep* [6], *CC2Vec* [5], *Patcherizer* [9], ainsi que les travaux de Yin et al. [13]. Bien que ces outils fonctionnent différemment, ils partagent tous l'objectif de générer un vecteur représentant le changement entre deux versions de code. Cependant, par manque de temps, je n'ai pas pu tester ces outils, car leur mise en place est non triviale. Certains auteurs n'ont pas publié le code utilisé dans leurs travaux, tandis que d'autres l'ont fait, mais avec une documentation souvent insuffisante.

Mon travail sur ces outils s'est donc principalement concentré sur leur identification et leur potentiel d'exploitabilité pour le projet. Par exemple, *CC2Vec* utilise les messages de commits laissés par les développeurs pour superviser

l'apprentissage du modèle générant la représentation vectorielle du patch. Cependant, dans notre cas, il est difficile de récupérer ces messages en quantité et en qualité suffisantes. *CCRep* utilise un modèle pré-entraîné (les auteurs mentionnent *CodeBERT*, mais précisent qu'un autre modèle pourrait convenir), mais le code fourni est peu documenté et semble peu accessible sans une étude approfondie, ce qui est chronophage.

La meilleure piste pour l'instant semble provenir des travaux de Yin et al., que je n'ai pas pu étudier par manque de temps, mais qui sera à explorer davantage dans des travaux futurs. Étudier ces différentes possibilités en parallèle, sans même les citer toutes, s'est révélé très chronophage et a souvent conduit à des impasses après quelques jours d'étude, me faisant réaliser que l'outil en question n'était pas exploitable. Par conséquent, même si je n'ai pas de résultats concrets sur cet aspect de mon stage, il représente une part importante de mon travail, consistant à explorer les différentes options pour assurer le bon déroulement de la suite du projet.

Au cours des dernières semaines de mon stage, j'ai alors exploré une approche plus simple pour représenter les modifications dans le code. Mon intuition était qu'une représentation d'un changement dans du code pouvait être exprimée par une fonction f telle que $f(\text{code avant}) = \text{code après}$. Étant donné un embedding x d'une version de code et un embedding x' de sa version modifiée, la représentation que l'on souhaite apprendre serait une fonction g telle que $g(x) = x'$.

Disposant déjà d'une base de données de couples (x, x'), j'ai tenté de calculer une fonction g pour chaque couple et de la représenter sous forme vectorielle. J'ai ainsi défini arbitrairement un vecteur d'embedding de modification de code y tel que $y = x' - x$. Ce vecteur représente donc ce que l'on ajoute (relativement) à un programme pour obtenir sa version modifiée. J'ai alors simplement généré ces vecteurs à l'aide de scripts Python et je les ai ajoutés à ma base de données.

5 Clustering et analyses des résultats

A l'aide des vecteurs obtenus lors des opérations détaillées dans les parties précédentes, j'ai pu générer des représentations qui allaient me permettre d'évaluer la qualité et la fiabilité de mes solutions. Pour rappel, l'objectif était de trouver une manière de représenter les données qui permettrait de les trier en plusieurs groupes représentant le type de modification appliquée au code dans chaque cas. J'ai voulu tester avec deux types d'informations liées aux commits accessibles sur GitHub: le texte associé à une issue laissé par le développeur et les conversations liées à celle ci, et le code source en lui même. Les deux bases de données détaillées en section 3 seront la base de mes expérimentations.

Les opérations précédentes nous ont permis de créer des ensembles de points des espaces à n ou m dimensions, où chaque point correspond respectivement à un vecteur d'embedding de texte de notre corpus ou a un vecteur calculé à partir

des embeddings du code source, et où la distance entre chaque point correspond à une distance vectorielle entre chacun.

Dans notre contexte, le clustering décrit l'opération consistant à extraire de ces nuages de points un certain nombre de groupes de vecteurs qui sont proches dans l'espace, donc qui ont des similarités entre eux. Le nombre optimal de clusters (groupes de vecteurs) est déterminé algorithmiquement par l'*elbow method* [10].

5.1 Clustering sur les textes des *issues* GitHub

Nous espérons ici extraire des groupes de textes avec un fort lien sémantique. Pour rappel, ici un texte dans notre base de données correspond à l'ensemble du texte et des messages envoyés dans une issue GitHub étudiée. Avec cette méthode, nous espérons séparer les *issues* par thème, afin de pouvoir extraire spécifiquement celles qui nous intéressent pour entraîner notre modèle ultérieurement, c'est à dire celles qui se concentrent sur la correction d'erreurs spécifiques. Cette expérience permet de jauger le potentiel de clustering sur un grand groupe de codes.

J'ai lancé un algorithme de clustering *Kmeans* sur mon ensemble de vecteurs générés par l'algorithme TF-IDF sur le corpus extrait et filtré d'environ 3000 textes d'*issues* GitHub. J'utilise la bibliothèque *Wordcloud* pour représenter visuellement les *topics* et les mots qui les constituent. Plus leur taille est grande sur l'image, plus leur contribution au *topic* est grande.

Après plusieurs tests pour réajuster les valeurs des paramètres, nous obtenons des résultats prometteurs. En effet, nous arrivons à extraire notamment un topic qui a l'air de représenter le "fix" d'erreurs dans du code, comme visualisable sur la figure 5.

Figure 5: Exemple de WordCloud généré à partir d'un corpus de 3000 textes

J'ai également généré un graphe pour avoir une représentation visuelle du clustering. Les vecteurs sont réduits à deux dimensions à l'aide d'une opération de PCA (*Principal Component Analysis*) afin de pouvoir les représenter sur un graphe en 2D en figure 6.

Figure 6: Représentation du clustering sur le corpus avec 4 topics et 3 clusters

Sur la figure 6, chaque point représente un vecteur de la base de données réduit en deux dimensions par un PCA, et les distances entre les points reproduisent au mieux les distances vectorielles entre les points originaux à n dimensions. Les topics (représentés par les WordClouds) sont générés en fonction des similarités entre les vecteurs, et les trois clusters représentés sur la figure par les trois couleurs différentes représentent les trois principaux groupes que l'on peut extraire du corpus, qui contiennent tous respectivement des vecteurs qui ont des contributions similaires à chacun des topics.

Ces résultats nous ont montré qu'il est possible d'extraire des thèmes des conversations issues des *issues* GitHub. Cependant, cette méthode présente de nombreuses limitations. Les résultats obtenus manquent de précision, et bien qu'un thème lié à la correction d'erreurs dans le code semble se détacher des autres, les données ainsi obtenues ne sont pas directement exploitables. Pour obtenir des résultats fiables, il serait nécessaire de croiser ce clustering avec d'autres types de filtrages basés sur différents paramètres, et de comparer les données entre elles.

De plus, il est important de rappeler que les textes des *issues* GitHub sont souvent imprécis et varient considérablement en fonction du développeur et des conventions de codage utilisées. Ce type de données semble trop hétérogène et subjectif pour être utilisé comme source principale de filtrage dans une base de données.

Je pense donc que les méthodes de filtrage basées sur le texte, comme celles-ci, peuvent être exploitées, mais à condition de les combiner avec d'autres types de données plus objectives et d'utiliser un corpus plus précis. Par exemple, on pourrait envisager de travailler avec un deuxième corpus, plus restreint mais créé manuellement, contenant des textes de haute qualité, décrivant précisément les erreurs rencontrées. Ce corpus pourrait alors être utilisé pour affiner un modèle initialement trop imprécis via des opérations de *fine-tuning*.

5.2 Clustering sur les embeddings de modifications dans le code source

Le deuxième test que je voulais effectuer se porte sur les embeddings de changement. Pour rappel, ici on définit un embedding de changement ou de modification de code par la soustraction des embeddings respectifs de deux versions d'un même code après et avant un commit donné. Le détail de la création de ces embeddings est précisé en partie 4.2.2.

5.2.1 Représentation du nuage de points

J'ai donc lancé un algorithme Kmeans sur l'ensemble des presque 2000 vecteurs calculés, et à nouveau utilisé l'elbow method pour déterminer le nombre idéal de clusters. Ici, on ne travaille pas avec des topics, on cherche simplement à visualiser la répartition des points dans l'espace. Pour une visualisation plus

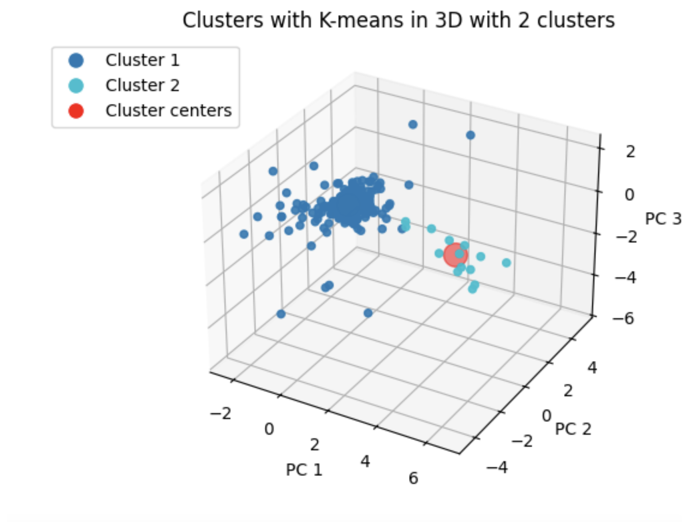

Figure 7: Clustering en 3D des vecteurs d'embedding de modification de code en fonction des 3 composantes principales des vecteurs. Le centre du premier cluster n'est pas visible car caché sous le nuage de points.

claire, j'ai fait un PCA en 3 dimensions cette fois ci et j'ai représenté le nuage de points dans un espace 3D.

```
# Apply K-means
kmeans = KMeans(n_clusters=nb_clusters)
kmeans.fit(vectors)

# Reduce dimensionality to 3D
pca = PCA(n_components=3)
reduced_vectors = pca.fit_transform(vectors)

# detect manually the outliers
detect_outliers(vectors, reduced_vectors, df)

# Plot clusters in 3D
fig = plot_clusters(reduced_vectors, kmeans, pca, nb_clusters)
```

J'ai lancé le programme plusieurs fois, la méthode Kmeans étant non déterministe on obtient des résultats légèrement différents à chaque exécution. En figure 7 est donné un des résultats obtenus le plus représentatif.

On observe que deux groupes principaux se détachent, un comportant une majorité des points et un plus petit, à l'écart. Ce n'est pas le résultat que je cherchais à obtenir, car deux clusters sont bien trop peu suffisants pour iden-

tifier des motifs dans les modifications de code. Ce résultat peut être dû à différents facteurs: la méthode d'embedding de base, *Jina*, n'est peut être pas adaptée pour capter le type d'information recherché. Il est aussi possible qu'un biais ait été introduit dans la base de donnée qui fait que tous les points sont autant rapprochés. Enfin, et je pense que c'est l'explication principale, le calcul (*embedding code avant - embedding code après*) me parait beaucoup trop simpliste pour pouvoir capturer, dans une représentation telle, des changements si complexes dans du code. Cela confirme qu'un travail important sur l'outil de représentation du changement est primordial pour trier nos données.

5.2.2 Étude manuelle des valeurs aberrantes

J'ai manuellement étudié quelques valeurs aberrantes qui se détachaient nettement du nuage de points. Mon objectif était de comprendre pourquoi ces vecteurs étaient si éloignés les uns des autres et de déterminer s'il existait un lien entre l'écart des vecteurs et l'écart sémantique de leurs codes sources correspondants.

Sur deux points que j'ai sélectionnés à la main, j'ai pu observer des exemples significatifs. Pour chaque point, j'ai récupéré les deux versions de code source associées au commit, que j'appellerai "code avant" et "code après".

- Le premier point concernait un couple de codes très différents l'un de l'autre. Le code avant commit était très sommaire, comprenant uniquement des imports de librairie et quelques définitions de constantes globales. En revanche, le code après commit était un fichier complet avec un nombre conséquent de lignes. L'écart de ce point par rapport aux autres est probablement dû à la grande différence entre les vecteurs d'embedding avant et après, qui ont été soustraits pour obtenir la valeur représentée. Cette différence a probablement induit des composantes aberrantes dans le vecteur de changement, le plaçant ainsi très loin des autres points.

- Le deuxième point étudié était également surprenant, mais pour une autre raison. Les deux versions de code étaient relativement courtes et similaires, la principale modification étant l'ajout d'une licence au début du programme avec un dessin en *ASCII art* (dessin réalisé à l'aide de caractères du clavier, comme des ponctuations ou des parenthèses). Il est probable que l'outil d'embedding *Jina* ne soit pas adapté pour interpréter correctement le sens des tokens générés par ce code source, ce qui a conduit à des valeurs aberrantes dans le vecteur de représentation.

Les résultats, sans être rigoureusement quantitatifs, donnent un début de piste sur le lien entre la représentation et le code source. L'étude des valeurs aberrantes pourrait permettre d'identifier les codes dont le sens est aberrant par rapport au reste du corpus.

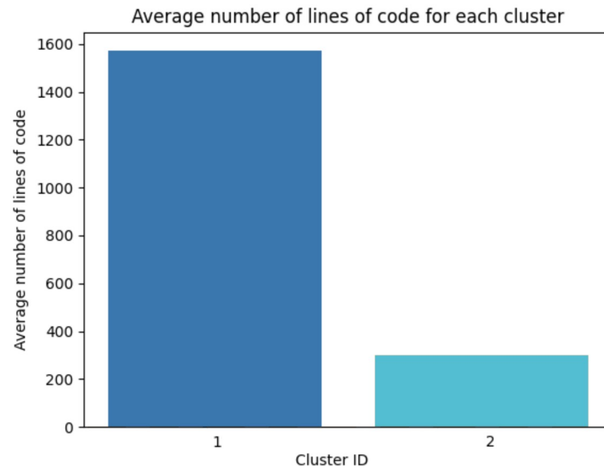

Figure 8: Nombre moyen de lignes dans chaque code en fonction du cluster auquel il appartient

5.2.3 Investigation des deux clusters

Afin de comprendre quel critère séparait les deux clusters, j'ai écrit un algorithme qui récupère le point le plus proche du centre de chaque cluster et qui sauvegarde dans des fichiers séparés les codes avant/après correspondant.

Après plusieurs exécutions de mon programme, je me suis vite rendue compte qu'une tendance semblait ressortir: Les vecteurs générés étaient réunis en clusters en fonction de la longueur du code de base. C'est assez contre intuitif car on pourrait se dire que la soustraction entre les deux vecteurs d'embedding des deux versions permettrait de retirer un biais lié à la taille des codes, mais c'est pourtant ce qui ressort de mes expérimentations. J'ai calculé pour chaque cluster le nombre de lignes moyen dans chacun des points qui le constitue, et le résultat présenté en figure 8 corrobore mon hypothèse.

Il semblerait alors que cette méthode n'est pas assez précise et efficace pour discriminer les points en fonction d'un sens profond dans le changement induit par le commit. L'impact de la taille des programmes utilisés est beaucoup trop grand par rapport à un éventuel impact d'un autre critère présent dans le code. Par conséquent, il est compliqué d'identifier des motifs avec cette méthode.

6 Conclusion

Durant tout ce stage, un protocole de tests a été mis en place pour identifier des données adaptées à l'entraînement d'un modèle de détection d'erreurs en MPI. Bien que de nombreuses recherches bibliographiques aient permis de trouver des outils potentiellement exploitables, la méthode actuelle de représentation

des données s'est révélée insuffisante pour identifier avec précision des erreurs complexes à partir des résultats de *clustering*.

Les messages de log laissés par les développeurs, se sont avérés difficilement discriminants en raison de leur qualité variable. Ils pourraient toutefois être exploités de manière plus efficace, par exemple en tant que labels supplémentaires pour le code, si l'on utilise un échantillon restreint et bien choisi, ou en les recoupant avec d'autres types de données plus objectives.

De plus, la méthode d'*embedding* des changements de code explorée n'a pas donné de résultats concluants. Cependant, cette expérience nous a permis d'identifier plusieurs pistes expliquant ces échecs, ce qui ouvre la voie à des améliorations futures. Il était prévisible qu'un calcul aussi simple ne pourrait pas entièrement capturer les motifs recherchés, et l'essai expérimental des outils existants tels que *CCRep* [6] ou *Patcherizer* [9] est la prochaine étape.

Pour avancer dans ce projet, il est essentiel de développer une méthode d'*embedding* plus précise et adaptée aux spécificités des erreurs en MPI. En parallèle, l'amélioration de la qualité des données d'entraînement, en particulier des messages de log, ainsi que leur recoupement avec d'autres sources, sera cruciale pour la réussite du modèle.

References

[1] Pierre-André Wacrenier Alice Lasserre Raymond Namyst. *EasyPAP: A graphical environment designed to facilitate learning of parallel programming*. URL: https://gforgeron.gitlab.io/easypap/.

[2] *bigcode/the-stack-github-issues · Datasets at Hugging Face*. Mar. 2024. URL: https://huggingface.co/datasets/bigcode/the-stack-github-issues.

[3] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *arXiv preprint arXiv:2002.08155* (Sept. 2020). URL: http://arxiv.org/abs/2002.08155.

[4] Michael Günther et al. "Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents". In: *arXiv preprint arXiv:2310.19923* (Feb. 2024). URL: http://arxiv.org/abs/2310.19923.

[5] Thong Hoang et al. "CC2Vec: Distributed Representations of Code Changes". In: *CoRR* abs/2003.05620 (2020). arXiv: 2003.05620. URL: https://arxiv.org/abs/2003.05620.

[6] Zhongxin Liu et al. *CCRep: Learning Code Change Representations via Pre-Trained Code Model and Query Back*. 2023. arXiv: 2302.03924 [cs.SE].

[7] Rhysd. *github-clone-all*. https://github.com/rhysd/github-clone-all.

[8] Nadav Schneider et al. "MPI-RICAL: Data-Driven MPI Distributed Parallelism Assistance with Transformers". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 2–10. DOI: 10.1145/3624062.3624063. URL: https://doi.org/10.1145/3624062.3624063.

[9] Xunzhu Tang et al. "Learning to Represent Patches". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. Lisbon, Portugal: ACM, 2024, pp. 396–397. DOI: 10.1145/3639478.3643521. URL: https://doi.org/10.1145/3639478.3643521.

[10] Wikipedia. *Elbow method (clustering) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Elbow_method_clustering&oldid=1210217728. 2024.

[11] Wikipedia. *Non-negative matrix factorization — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Non-negative_matrix_factorization&oldid=1216627798. 2024.

[12] Wikipedia. *TF-IDF — Wikipedia, The Free Encyclopedia*. http://fr.wikipedia.org/w/index.php?title=TF-IDF&oldid=214135550. 2024.

[13] Pengcheng Yin et al. "Learning to Represent Edits". In: *arXiv preprint arXiv:1810.13337* (Feb. 2019). Accessed: 2024-07-30. URL: http://arxiv.org/abs/1810.13337.