







Université du Québec  
à Rimouski

# **Vers une nouvelle approche basée sur l'apprentissage profond pour la classification des changements du code source par activités de maintenance**

**Par Marouane Benjillali**

Mémoire présenté dans le cadre du programme de maîtrise en informatique de l'UQAC  
offert par extension à l'UQAR en vue de l'obtention du grade de maître ès science (M. Sc.)

Québec, Canada

© Marouane Benjillali, 2023



**Composition du jury :**

**Yacine Benhamed, président du jury, Université du Québec à Rimouski**

**Ismail Khriss, directeur de recherche, Université du Québec à Rimouski**

**Hamid Mcheick, professeur, Université du Québec à Chicoutimi**

**Wahab Hamou-Lhadj, examinateur externe, Université de Concordia**

Dépôt initial le 11 mars 2023

Dépôt final le 12 juillet 2023



UNIVERSITÉ DU QUÉBEC À RIMOUSKI  
Service de la bibliothèque

Avertissement

La diffusion de ce mémoire ou de cette thèse se fait dans le respect des droits de son auteur, qui a signé le formulaire « *Autorisation de reproduire et de diffuser un rapport, un mémoire ou une thèse* ». En signant ce formulaire, l'auteur concède à l'Université du Québec à Rimouski une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de son travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, l'auteur autorise l'Université du Québec à Rimouski à reproduire, diffuser, prêter, distribuer ou vendre des copies de son travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de la part de l'auteur à ses droits moraux ni à ses droits de propriété intellectuelle. Sauf entente contraire, l'auteur conserve la liberté de diffuser et de commercialiser ou non ce travail dont il possède un exemplaire.





## **REMERCIEMENTS**

Je tiens à remercier mon directeur de recherche, M. Ismaïl Khriss qui m'a donné l'occasion de travailler sur ce projet, et surtout d'explorer un domaine comme l'intelligence artificielle dont l'avenir est très prometteur. Je tiens à le remercier aussi pour sa disponibilité, son encadrement ainsi que ses encouragements.

Je remercie aussi les étudiants de la faculté des sciences et techniques d'Errachidia au Maroc, pour leur contribution dans le processus d'étiquetage du jeu de données.

Mes remerciements vont également aux membres de jury pour le temps qu'ils ont accordé à la lecture de mon mémoire et pour leurs commentaires pertinents.

Et bien évidemment, je ne peux terminer sans remercier énormément mes parents et mes sœurs pour leur soutien inconditionnel.



## RÉSUMÉ

Le domaine du développement logiciel possède une vraie mine d'information qui est sous forme d'historique de changements appliqués aux logiciels pendant leur cycle de vie. En effet, cet historique dont une partie importante est publiquement accessible à partir des systèmes de contrôle de versions fait l'objet d'exploration et d'analyse scientifique à travers le domaine du forage des référentiels de logiciels (MSR pour Mining Software Repositories en anglais) dont le but est d'améliorer plusieurs aspects rencontrés par les parties prenantes pendant le développement d'un logiciel. Dans ce travail, nous nous sommes intéressés à la détermination des types d'activité de maintenance qui sont présents dans une modification du code source. Plusieurs études se sont intéressées à ce sujet, et l'ont traité en exploitant les informations fournies par un programmeur comme le message décrivant les changements effectués ainsi que le code modifié qui est sous forme d'ajout et suppression de lignes de code. Cependant, la majorité d'entre elles considèrent qu'un changement comprend un seul type d'activité de maintenance, ce qui n'est pas toujours vrai en pratique. Ensuite, dans leurs exploitations des données textuelles, ces études se limitent au message alors que ce dernier comprend souvent seulement une description du code modifié et non la raison du changement. Et puis, dans leurs approches, elles se limitent à étudier des projets utilisant le même langage de programmation. À travers cette étude, nous répondons à ces enjeux en proposant un modèle de classification par activités de maintenance basé sur des modèles en apprentissage profond, qui seront également responsables de l'extraction de caractéristiques, que ce soit à partir d'une information textuelle (le message et la proposition de changement) ou du code modifié, indépendamment de son langage de programmation. Nous proposons également un nouveau jeu de données pour cette tâche afin de répondre à un autre enjeu qui est la rareté des jeux de données disponibles. Ce jeu de données tient compte du fait qu'un changement peut appartenir à plusieurs classes de changements. L'architecture de notre modèle est composée d'un modèle préentraîné permettant la génération des représentations distribuées des données textuelles, en plus d'un classificateur sous forme d'un réseau de neurones qui prendra en entrée la sortie du modèle préentraîné en plus des caractéristiques qui concernent le code modifié. Notre approche, dont l'entraînement est basé sur un apprentissage par transfert, a donné des résultats encourageants non seulement sur notre jeu de données, mais aussi en ce qui concerne le support des jeux de données des travaux reliés.

Mots-clés : Activités de maintenance, systèmes de contrôle de version, forage des référentiels de logiciels, apprentissage profond, apprentissage par transfert, représentation distribuée, classification.



## ABSTRACT

Software development has a wealth of information in the form of a history of changes applied to software during its life cycle. Indeed, a part of this history, publicly accessible from version control systems, is the subject of exploration and scientific analysis through mining software repositories (MSR). MSR aims to facilitate and improve several aspects stakeholders encounter during software development. In this work, we are interested in determining the types of maintenance activity present in modifying the source code. Several studies have been interested in this subject and have dealt with it by exploiting the information provided by a programmer, like a message describing the changes made and the modified code in the form of added and removed lines of code. However, most consider that a change includes only one type of maintenance activity, which is not always accurate in practice. Also, in using textual data, these studies limit themselves to the message, which often includes only a description of the modified code and not the reason for the change. Additionally, their approaches limit themselves to studying projects that use the same programming language. Through this study, we respond to these challenges by proposing a classification model by maintenance activities based on deep learning models. It will also be responsible for feature extraction, whether from textual information (message and issue description) or modified code, regardless of its programming language. We also provide a new dataset for this task to address another issue: the scarcity of available datasets. This dataset takes into account the fact that a change can belong to several classes of changes. The architecture of our model is composed of a pre-trained model allowing the generation of distributed representations of textual data, in addition to a classifier in the form of a neural network. This network inputs are the output of the pre-trained model and the characteristics related to the modified code. Our approach, whose training is based on transfer learning, has given encouraging results not only on our dataset but also on the support of related work datasets.

*Keywords:* Maintenance activities, version control systems, software repository mining, deep learning, transfer learning, distributed representation, classification.

## TABLE DES MATIÈRES

REMERCIEMENTS.....	vii
RÉSUMÉ .....	ix
ABSTRACT .....	xi
TABLE DES MATIÈRES .....	xii
LISTE DES TABLEAUX.....	xv
LISTE DES FIGURES.....	xvi
LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES .....	xix
LISTE DES SYMBOLES .....	xxii
INTRODUCTION GÉNÉRALE.....	1
CHAPITRE 1 NOTIONS DE BASE.....	7
1.1 LA GESTION DES CHANGEMENTS DES SYSTÈMES LOGICIELS .....	7
1.1.1 Les catégories de changements .....	7
1.1.2 Processus des changements .....	10
1.1.3 Les systèmes de contrôle de versions.....	11
1.1.4 L'intégration continue .....	12
1.2 L'APPRENTISSAGE AUTOMATIQUE.....	14
1.2.1 Les tâches résolues .....	14
1.2.2 Principes généraux .....	15
1.3 L'APPRENTISSAGE PROFOND.....	21
1.3.1 Principes généraux .....	22
1.3.2 Les réseaux de neurones .....	23
1.3.3 La rétropropagation de l'erreur .....	26
1.3.4 La régularisation .....	26
1.3.5 L'analyse de données séquentielles .....	28
1.4 SYNTHÈSE.....	34
CHAPITRE 2 LE TRAITEMENT AUTOMATIQUE DES LANGUES .....	36
2.1 CONCEPTS GÉNÉRAUX .....	36

2.2	LA REPRÉSENTATION LOCALISTE.....	37
2.3	LA REPRÉSENTATION DISTRIBUÉE : LES PLONGEMENTS .....	38
	2.3.1 Factorisation matricielle globale.....	39
	2.3.2 Fenêtres contextuelles locales.....	40
	2.3.3 Modèles de régression log bilinéaire globale .....	41
2.4	LES PLONGEMENTS DE PHRASES .....	42
2.5	LE MODÈLE USE.....	46
2.6	SYNTHÈSE .....	50
CHAPITRE 3 ÉTAT DE L'ART.....		52
3.1	CONTEXTE DE LA TÂCHE.....	52
3.2	TRAVAUX RELIÉS .....	56
	3.2.1 La classification par activité de maintenance .....	56
	3.2.2 La représentation des changements de code.....	66
3.3	SYNTHÈSE.....	74
CHAPITRE 4 L'ÉTUDE PROPOSÉE .....		76
4.1	PRÉSENTATION DE L'APPROCHE.....	76
4.2	CONSTITUTION DU JEU DE DONNÉES .....	83
	4.2.1 Le choix des projets .....	83
	4.2.2 Extraction des commits .....	84
	4.2.3 La phase d'étiquetage.....	86
4.3	GESTION DES CLASSES ASYMÉTRIQUES.....	90
4.4	CHOIX DES MÉTRIQUES.....	92
4.5	TECHNOLOGIES UTILISÉES.....	93
4.6	LA VALIDATION DE L'APPROCHE.....	94
	4.6.1 Les résultats .....	94
	4.6.2 Discussion et proposition de travaux futurs .....	107
4.7	SYNTHÈSE .....	11818
CONCLUSION GÉNÉRALE .....		11919
ANNEXE A – Structure des deux projets utilisés dans notre jeu de données.....		12121
ANNEXE B.....		12323

ANNEXE C.....	12525
ANNEXE D.....	12727
ANNEXE E – Le code du programme.....	12929
RÉFÉRENCES BIBLIOGRAPHIQUES .....	13434



## LISTE DES TABLEAUX

Tableau 3.1 - Un aperçu sur les études s'intéressant à la classification par activité de maintenance avec les résultats obtenus .....	55
Tableau 3.2 - Le meilleur classificateur (moyenne des projets) dans chaque jeu de données .....	59
Tableau 3.3 - Les règles définies par Sarwar et al. (2020) pour l'étiquetage du jeu de données .....	63
Tableau 4.1 - Les règles que nous avons définies pour certains cas de changements .....	87
Tableau 4.2 - La répartition du jeu de donnée sur les trois ensembles .....	90
Tableau 4.3 - Les meilleurs résultats pour chaque valeur de nombre de couches entraînées pour le DAN et Transformer .....	96
Tableau 4.4 - Les différentes valeurs de L1 et L2 testées .....	98
Tableau 4.5 - Les résultats du modèle 2 (meilleur modèle) sur les données de test .....	106
Tableau 4.6 - Une comparaison entre notre modèle et celui de Sarwar et al. (2020) par rapport au jeu de données de Mauczka et al. (2015).....	106

## LISTE DES FIGURES

Figure 1.1 - Répartition des efforts de maintenance.....	9
Figure 1.2 - Les métriques d'évaluation utilisées dans le contexte multiétiquettes.....	18
Figure 1.3 - Illustration de la hiérarchie d'un modèle d'apprentissage profond.....	23
Figure 1.4 - Représentation d'un réseau de neurones à 2 couches.....	25
Figure 1.5 - Capacité d'un modèle. ....	27
Figure 1.6 - Illustration du principe d'un RNN mappant une séquence d'entrée x à une séquence de sortie y. ....	29
Figure 1.7 - Illustration du fonctionnement des cellules RNN standard, LSTM et GRU.....	31
Figure 1.8 - Architecture de la partie encodeur du transformateur. ....	33
Figure 2.1 - Les probabilités de cooccurrence (et le ratio) des mots « ice » et « steam » dans corpus de 6 milliards de mots.....	41
Figure 2.2 - Architecture du modèle InferSent. ....	44
Figure 2.3 - Performance du modèle USE sur différentes tâches .....	49
Figure 2.4 - Performance du modèle USE sur la tâche SST en variant la taille de l'ensemble d'entraînement.....	50
Figure 3.1 - Les caractéristiques extraites à partir des changements de code.....	65
Figure 3.2 - Un exemple de l'arbre de la syntaxe abstraite d'une fonction en Python. ....	67
Figure 3.3 - Processus de génération du vecteur de plongement des changements dans un fichier. ....	69
Figure 3.4 - L'architecture du HAN .....	73
Figure 4.1 - Une représentation de l'architecture utilisée pour tester nos 4 modèles. ....	78
Figure 4.2 - Processus de l'approche suivie.....	79

Figure 4.3 - Un exemple du fichier généré pour chaque commit.....	85
Figure 4.4 - Aperçu de l'application développée pour effectuer l'étiquetage .....	89
Figure 4.5 - Combinaisons de valeurs testées pour les hyperparamètres .....	95
Figure 4.6 - Résultats d'entraînement et de validation du modèle 1 .....	97
Figure 4.7 - Résultats d'entraînement et de validation du modèle 1 avec pénalité de norme $L_1$ (0.02) et $L_2$ (0.1) .....	99
Figure 4.8 - Résultats d'entraînement et de validation du modèle 1 après ajout du titre de la proposition de changement.....	101
Figure 4.9 - Résultats d'entraînement et de validation du meilleur modèle 1 obtenu.....	102
Figure 4.10 - Résultats d'entraînement et de validation du modèle 2 .....	104
Figure 4.11 - Résultats d'entraînement et de validation du modèle 3 et 4 .....	105
Figure 4.12 - Le commit « A » .....	109
Figure 4.13 - La proposition de changement implémentée par le commit « A » .....	110
Figure 4.14 - Le commit « B ».....	111
Figure 4.15 – La suite du commit « B » .....	112
Figure 4.16 – La proposition de changement implémentée par le commit « B » .....	113
Figure 4.17 - Le commit « C ».....	114
Figure 4.18 - Le commit « D » .....	115
Figure 4.19 – La suite du commit « D ».....	116
Figure A.1 - Principaux dossiers et fichiers du projet « Create-React-App » avec leur rôle.....	121
Figure A.2 - Principaux dossiers et fichiers du projet « Flask » avec leur rôle .....	122
Figure E.1 - La configuration des scripts lancés pour exécuter les entraînements et tests.....	129
Figure E.2 - La préparation des données .....	129
Figure E.3 - La fonction de création du modèle 1.....	130
Figure E.4 - La fonction de création du modèle 2.....	130

Figure E.5 - La fonction de création du modèle 3.....	131
Figure E.6 - La fonction de création du modèle 4.....	131
Figure E.7 - La construction des dictionnaires pour le modèle CC2Vec .....	132
Figure E.8 - La partie responsable de l'entraînement et test du modèle .....	133

## LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

<b>AST</b>	Abstract Syntax Tree
<b>AUC</b>	Area Under the Curve
<b>avNNeT</b>	Averaged Neural Networks
<b>BCE</b>	Binary Cross Entropy
<b>Bi-LSTM</b>	Bidirectional Long Short-Term Memory
<b>CNN</b>	Convolutional Neural Network
<b>DAN</b>	Deep Averaging Network
<b>DNN</b>	Deep Neural Network
<b>ESM</b>	Entreprise Systems Management
<b>FN</b>	Faux Négatif
<b>FP</b>	Faux Positif
<b>GBM</b>	Gradient Boosting Machine
<b>GLM</b>	Generalized Linear Model
<b>GRU</b>	Gated Recurrent Unit
<b>HAN</b>	Hierarchical Attention Network
<b>HTML</b>	HyperText Markup Language
<b>IA</b>	Intelligence Artificielle
<b>IC</b>	Intégration Continue
<b>LDA</b>	Linear Discriminant Analysis
<b>LLDA</b>	Labelled Latent Dirichlet Allocation.
<b>LOC</b>	Line Of Code
<b>LSTM</b>	Long Short-Term Memory

<b>LSA</b>	Latent Semantic Analysis
<b>LSI</b>	Latent Semantic Indexation
<b>MDA</b>	Mixture Discriminant Analysis
<b>MLALAL</b>	Multi-label Active Learning with Auxiliary Learner
<b>MTL</b>	Multi-Task Learning
<b>NN</b>	Neural Network
<b>NT</b>	Neural Tensor Network
<b>OMT</b>	Ordre de Modification Technique
<b>PCA</b>	Principal Component Analysis
<b>ReLU</b>	Rectified Linear Unit
<b>RF</b>	Random Forest
<b>RFE</b>	Recursive Feature Elimination
<b>RIPPER</b>	Repeated Incremental Pruning to Produce Error Reduction
<b>RNN</b>	Recurrent Neural Network
<b>ROC</b>	Receiver Operating Characteristic
<b>SCV</b>	Système de Contrôle de version
<b>SCCV</b>	Système Centralisé de Contrôle de Version
<b>SCM</b>	Software Configuration Management
<b>SDCV</b>	Système Distribué de Contrôle de Version
<b>SIF</b>	Smooth Inverse Frequency
<b>SMO</b>	Sequential Minimal Optimization
<b>SNLI</b>	Stanford Natural Language Inference
<b>SVM</b>	Support Vector Machine

<b>SVD</b>	Singular Value Decomposition
<b>TALN</b>	Traitement Automatique des Langues Naturelles
<b>TF-IDF</b>	Term Frequency–Inverse Document Frequency
<b>URL</b>	Uniform Resource Locator
<b>USE</b>	Universal Sentence Encoder
<b>VN</b>	Vrai Négatif
<b>VP</b>	Vrai Positif
<b>WSGI</b>	Web Server Gateway Interface
<b>XGBoost</b>	eXtreme Gradient Boosting
<b>xgbTree</b>	eXtreme Gradient Boosting Tree

## LISTE DES SYMBOLES

$\Delta$  Différence symétrique.

$\subseteq$  Inclusion.

$\cap$  Intersection.

$\cup$  Union.



Symbole de début ou fin.



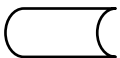
Symbole d'une action ou une tâche qui doit être effectuée.



Il indique la direction dans laquelle l'organigramme doit être lu.



Symbole de base de données.



Symbole de données stockées.



Symbole de fusion.



## INTRODUCTION GÉNÉRALE

### MOTIVATION

Dans un projet de développement logiciel, le code source est considéré comme la pièce maîtresse qu'il faut protéger, améliorer et garantir que son fonctionnement se fasse de manière adéquate. Pour aboutir à ceci, le code source sera sujet à des changements continus tout au long du cycle de vie du projet. En effet, le résultat de ces modifications de code permet, en parallèle, d'accumuler un nombre important de données sous forme d'historique de changements qui sont gérés par un système de contrôle de version. Et c'est bien cet historique qui sera exploité dans différentes études afin d'en extraire des connaissances et d'améliorer différents aspects du processus de développement logiciel, par exemple : l'amélioration des systèmes de recommandation de code (Nguyen et al. 2016), la génération de la documentation (Buse & Weimer 2010), la détection des clones (Kamiya et al. 2002), la détection des changements enchevêtrés (Kirinuki et al. 2014), l'identification des anomalies durant le processus de développement (Levin & Yehudai 2017), etc.

Dans cette étude, nous nous sommes intéressés aux types d'activités de maintenance présentes dans une modification du code source. Plus précisément, il s'agit de déterminer si cette modification représente une correction d'un bogue, un ajout d'un test, une suppression d'un code mort, un ajout d'une nouvelle fonctionnalité, une optimisation d'une fonction...etc. Ces cas de changements sont généralement regroupés en trois classes (perfective, corrective et adaptative), et l'un ou plusieurs d'entre eux peuvent être présents dans une même modification même s'il est normalement recommandé d'archiver chaque cas de changement séparément (Herzig & Zeller 2013). En effet, l'idée sera d'exploiter les

différents types d'informations liées à ce changement comme le titre de la tâche à faire, la description des modifications apportées et le code modifié lui-même.

À l'instar de plusieurs secteurs d'activités ces dernières années, le développement logiciel a connu aussi une transition vers l'utilisation de l'intelligence artificielle (IA) pour extraire des connaissances à partir de l'historique des changements de code. L'impact de cette transition se voit clairement tout au long du processus de développement logiciel à travers une amélioration au niveau de la productivité, la qualité et la rapidité de livraison. D'une manière générale, l'IA repose dans sa résolution des tâches sur le remplacement des moteurs de règles dont la logique est définie par l'être humain, avec des techniques d'apprentissage automatique ou d'apprentissage profond qui permettent de découvrir des aspects qui sont difficiles à percevoir par l'humain, surtout quand il s'agit d'une tâche complexe.

Dans notre contexte, l'adoption d'une approche basée sur l'IA pour la classification par activité de maintenance n'aura pas seulement un impact sur le processus de gestion des changements de code, mais aussi sur le déroulement de l'ensemble d'activités qui constituent le processus du développement logiciel. En effet, avec l'adoption récemment d'une culture comme DevOps<sup>1</sup>, les tâches de planification, d'implémentation, de test et de mise en production se font désormais conjointement, contrairement à ce qui se faisait auparavant où un logiciel n'est livré que lorsqu'il est achevé. Ceci mène à penser que toutes les parties prenantes (les développeurs, les testeurs, les chefs de projet, etc.) doivent être au courant d'une manière permanente des modifications qui ont eu lieu. En effet, l'identification des types activités de maintenance dans un changement de code permet, premièrement, de détecter les anomalies durant le processus de développement en vérifiant si les modifications qui ont été apportées à un système reflètent parfaitement ce qui a été prévu lors de la planification. Deuxièmement, pour la génération automatique de la documentation d'une nouvelle version d'un système (en anglais : *release*), faire référence uniquement aux composants modifiés du code source n'est pas suffisant, il est souvent nécessaire de préciser

---

<sup>1</sup> <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-devops/>

le type de modification dont il s'agissait. Troisièmement, la détection des activités de maintenance peut être également utile aux chefs de projets (ou gestionnaires), d'une part, pour leur permettre d'avoir une idée sur le profil de chaque développeur à travers l'analyse de leur pratique de modification du code (Levin & Yehudai 2016) ; et, d'autre part, pour une meilleure optimisation des coûts lors de l'allocation des ressources et d'estimation de la dette technique. En effet, par exemple à un moment où des changements correctifs deviennent de plus en plus fréquents, une décision pourrait être prise pour consacrer plus de temps à la planification de tests avancés qu'au développement de nouvelles fonctionnalités.

## **OBJECTIFS ET CONTRIBUTIONS**

Plusieurs études se sont intéressées à ce sujet en explorant à chaque fois de nouvelles informations (Mariano et al. 2019, Hönel et al. 2020) concernant les changements, et en testant aussi différentes méthodes de classification suivant l'évolution technologique, une tendance qu'on peut constater clairement dans les études les plus récentes (Sarwar et al. 2020, Ghadhab et al. 2021) avec l'utilisation des algorithmes d'apprentissage profond. En effet, ces derniers ont été exploités non seulement pour effectuer la classification, mais aussi pour le traitement d'une donnée non numérique (un texte, par exemple) en générant sa représentation distribuée sous forme d'un vecteur.

À travers ce travail, nous faisons les hypothèses suivantes : (1) notre jeu de données comportera des changements possiblement enchevêtrés vu qu'il s'agit d'une pratique fréquente chez les développeurs ; (2) l'utilisation d'une nouvelle information, non exploitée jusqu'à présent, qui est le titre de la proposition de changement aura un impact sur la prédiction des activités de maintenance ; et (3) que la transformation du code modifié (indépendamment du langage de programmation) en une représentation distribuée nous permettra d'explorer de nouveaux aspects et caractéristiques concernant les changements de code, et donc il semblerait intéressant de la tester dans le contexte de classification par activités de maintenance.

Notre contribution à travers ce projet de recherche consiste à proposer une approche de classification multiétiquettes basée sur l'apprentissage profond, et plus concrètement en faisant de l'apprentissage par transfert pour l'extraction des représentations distribuées, que ça soit à partir des données en langage naturel en utilisant le modèle pré-entraîné USE (Cer et al. 2018), ou bien à partir du code modifié en utilisant le modèle CC2Vec (Hoang et al. 2020). En plus de cela, nous exploiterons aussi quelques caractéristiques quantitatives qui concernent les lignes de codes et fichiers modifiés. Notons que notre utilisation de CC2Vec est dans le but d'évaluer sa performance étant donné qu'il représente actuellement l'état de l'art en termes de transformation du code modifié en un vecteur de plongement qui est indépendant du langage de programmation. Ce dernier point représente une contrainte dans plusieurs études (Levin & Yehudai 2016, Levin & Yehudai 2017, Ghadhab et al. 2021) qui se sont limitées à l'étude d'historique de projets développés avec le même langage de programmation à cause de leurs utilisations de certains outils qui ne supportent dans la majorité de cas que le langage Java. L'autre contribution de ce travail consiste dans la construction d'un nouveau jeu de données et le développement d'une application pour faciliter le processus d'étiquetage. Les résultats de notre approche ont donné des résultats encourageants non seulement sur notre jeu de données, mais aussi en ce qui concerne le support des jeux de données des travaux reliés.

## **ORGANISATION DU MÉMOIRE**

Concernant l'organisation du présent mémoire, il est présenté à travers quatre chapitres. Dans le premier chapitre, nous aborderons d'abord des notions concernant la gestion des changements des systèmes logiciels ; plus précisément : les différents types de changements, le processus de changement et les systèmes de contrôle de versions. Ensuite, comme nous utilisons l'IA, nous présenterons les concepts de base et techniques de cette discipline qui nous seront utiles dans l'implémentation de notre approche. Puis dans le deuxième chapitre, nous mettrons l'accent sur les concepts de traitement automatique du

langage naturel tout en montrant l'efficacité des modèles de langues, et ce, à travers une présentation du modèle pré-entraîné USE (Cer et al. 2018). Dans le troisième chapitre, nous présenterons d'abord l'état de l'art concernant la classification par activités de maintenance en explorant la manière avec laquelle la problématique a été traitée par les études précédentes, les résultats obtenus et les limites des approches utilisées. Ensuite, motivés par l'une de ces limites, nous explorerons s'il existe une méthode qui permet d'extraire la représentation distribuée des changements de code indépendamment du langage de programmation utilisé. Dans le quatrième chapitre, nous présenterons l'approche proposée en décrivant la méthodologie suivie en termes d'architecture des modèles, ainsi que la construction du jeu de données et de son étiquetage, puis finalement, nous rapporterons les résultats obtenus puis nous discuterons des pistes à explorer dans les travaux futurs.



# **CHAPITRE 1**

## **NOTIONS DE BASE**

L'approche que nous adoptons pour effectuer la classification se base essentiellement sur l'utilisation de l'apprentissage automatique et plus particulièrement de l'apprentissage profond. De ce fait, dans ce chapitre, nous allons commencer par présenter des notions relatives au cadre dans lequel se situe notre sujet qui concerne la gestion des changements des systèmes logiciels. Ensuite, nous introduirons les concepts de base et certaines techniques utilisées dans l'apprentissage automatique et en apprentissage profond qui nous seront utiles lors de l'implémentation de notre approche.

### **1.1 LA GESTION DES CHANGEMENTS DES SYSTÈMES LOGICIELS**

#### **1.1.1 Les catégories de changements**

À partir de 1974, Lehman et al. (1985, 1997) ont développé une série de lois en ce qui concerne l'évolution logicielle. Les lois, au nombre de huit, décrivent un équilibre entre les forces motrices de nouveaux développements et les forces qui ralentissent les progrès. La première loi nous concerne particulièrement puisqu'elle parle des changements. En effet, cette loi, au nom de changement continu, stipule qu'un système doit être continuellement adapté ou il devient progressivement moins satisfaisant. Selon Pressman & Maxim (2020), l'origine des changements est multiple :

- De nouvelles conditions d'affaires ou de marché dictent des changements dans les exigences des produits ou les règles d'affaires.

- De nouveaux besoins des parties prenantes exigent la modification des données produites par le système d'information, des fonctionnalités délivrées par des produits ou des services délivrés par un système informatique.
- La réorganisation ou la croissance ou la réduction des entreprises entraîne des changements dans les priorités du projet ou dans la structure des équipes d'ingénierie logicielle.
- Les contraintes budgétaires ou de planification provoquent une redéfinition du système ou du produit.

C'est pourquoi, tant qu'un logiciel est utilisé, il nécessite une maintenance continue pour suivre ce rythme. Le logiciel doit donc être capable d'évoluer en subissant des changements tout au long de son cycle de vie. D'ailleurs, ces changements entrent dans le cadre de ce qu'on appelle la maintenance logicielle qui regroupe toute modification apportée à un logiciel après sa mise en œuvre. Tout changement de code appartient à l'une des quatre catégories suivantes (Pressman & Maxim 2020) :

- **Corrective** : Elle est une modification réactive qui corrige les bogues, les failles du logiciel et se présente souvent sous forme de petites mises à jour rapides. Cette catégorie de changement provient généralement des rapports de bogues. Bien qu'elle soit le type de modification qu'on lie le plus souvent à la maintenance, elle ne représente que 21% de l'ensemble des changements (voir la figure 1.1).
- **Adaptative** : Elle est une modification réactive qui permet de garder le logiciel utilisable dans un environnement d'utilisateur final changeant. Cette catégorie de changement, représentant 25% de l'ensemble des changements, touche souvent l'infrastructure du logiciel et se caractérise comme étant des modifications de grande taille touchant souvent plusieurs lignes de code.
- **Perfective** : Elle est une modification proactive pour fournir de nouvelles fonctionnalités, une meilleure structure de code ou une documentation améliorée.



Notons ici que 50% des travaux de maintenance sont attribués à cette catégorie de changement.

- **Préventive** : Elle est une modification proactive qui permet de détecter des défauts latents avant qu'ils ne se transforment en défauts opérationnels. Cette catégorie de changement représente 4% de l'ensemble des changements.

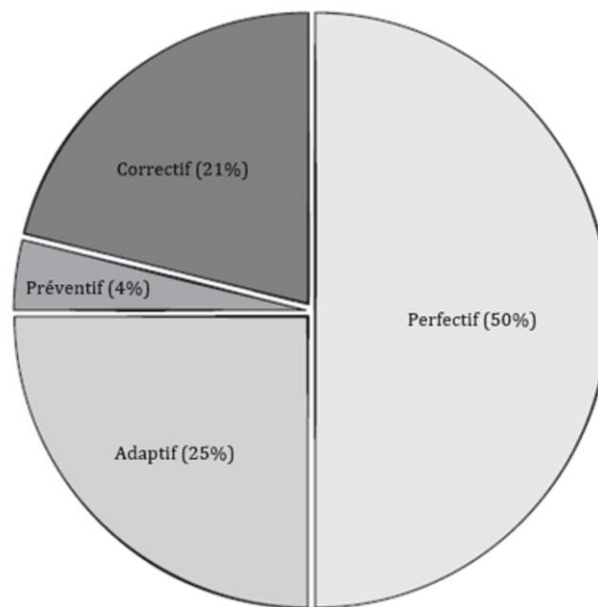


Figure 1.1 – Répartition des efforts de maintenance.

Source: *Software Engineering: A Practitioner's Approach*. (Pressman & Maxim 2020)

### 1.1.2 Processus des changements

Quand il s'agit d'un projet complexe, les changements de code incontrôlés conduisent rapidement au chaos. Pour de tels projets, l'assurance d'un bon contrôle doit combiner l'intervention humaine et l'utilisation d'outils automatisés. Dans le génie logiciel, la gestion de la configuration logicielle (SCM – *Software Configuration Management*, en anglais) est un ensemble d'activités appliquées tout au long du processus de développement d'un logiciel permettant l'identification, le contrôle, l'audit et la publication de toute modification qui se produit pendant le développement du logiciel et après sa mise à disposition du client (Pressman & Maxim 2020). Le processus de changement adopté par la SCM définit donc une série de tâches à accomplir avant de valider une modification. Ce processus a pour objectif d'identifier tous les éléments qui définissent collectivement la configuration logicielle, de gérer les modifications d'un ou plusieurs de ces éléments, de faciliter la construction de différentes versions d'une application, et de s'assurer que la qualité du logiciel est maintenue au fil du temps. Comme décrit par Pressman & Maxim (2020), ce processus commence par une demande de modification qui sera évaluée en analysant les effets indésirables qu'elle pourra engendrer, son coût et son impact global sur d'autres parties du logiciel. Une fois approuvé, un ordre de modification technique (OMT) sera présenté à la personne qui prendra en charge le développement de cette modification. L'OMT décrit le changement à apporter, les contraintes à respecter, et les critères d'examen et d'audit. Les prochaines étapes consisteront à récupérer le code, lui apporter des modifications puis de l'archiver tout en respectant les critères préalablement définis dans l'OMT.

La récupération et l'archivage du code se font au moyen d'un système de contrôle de versions que nous décrirons en détail dans la section 1.1.3. Nous parlerons également dans la section 1.1.4 d'une approche qui permet d'accélérer ce processus de changement qu'on vient de présenter, tout en garantissant son bon fonctionnement.

### 1.1.3 Les systèmes de contrôle de versions

Les systèmes de contrôle de version (SCV) sont des outils qui aident une équipe de développement à gérer les modifications apportées au code source et de maintenir leurs traçabilités. L'historique complet de ces modifications est sauvegardé sous forme d'unité qu'on appelle un « commit ». Ce dernier comporte non seulement le changement de code effectué, qui est sous forme d'ajout/suppression des lignes de code (ou des fichiers), mais il inclut également des informations concernant son auteur, sa date et sa description, etc. Parmi les fonctionnalités d'un SCV, le retour vers une ancienne version (action appelée « *revert* ») plus stable du code quand un nouveau changement engendre une régression.

Il existe deux catégories d'architecture de ces systèmes (Anuupadhyay 2021). La première architecture, dite centralisée (SCCV), utilise un serveur central pour stocker tous les fichiers puis chaque opération est effectuée directement sur ce dépôt central. Même s'il semble assez pratique de maintenir un référentiel unique, celui-ci présente des inconvénients majeurs comme le risque de perdre toutes les données quand ce référentiel est endommagé. La deuxième catégorie d'architecture, dite distribuée (SDCV), dans laquelle chaque contributeur dispose d'une copie du dépôt principal, y compris tous les fichiers et métadonnées. La mise à jour du code se fait sur le dépôt local sans aucune interférence avec le dépôt principal. Ensuite, la récupération de la dernière version du code à partir du dépôt principal se fait avec une opération appelée « *pull* » tandis que l'archivage des changements dans le dépôt principal se fait en créant d'abord un commit puis en effectuant une opération appelée « *push* ». D'ailleurs, ces dernières années, ce sont les outils basés sur une architecture distribuée (comme GitHub<sup>2</sup> et GitLab<sup>3</sup>) qui sont les plus recherchés par les équipes de développement (Google Trends 2023).

---

<sup>2</sup> <https://github.com/about>

<sup>3</sup> <https://about.gitlab.com>

En utilisant les SCV, qu'ils soient centralisés ou distribués, il est nécessaire de suivre certaines règles et bonnes pratiques afin de maintenir un historique de changements de code reflétant le vrai déroulement des tâches de développement. Parmi ces pratiques, le choix d'un message bien descriptif des changements de code effectués, et ce, lors de la création du commit. D'ailleurs, ce message doit contenir non seulement une description des changements effectués, mais aussi la raison pour laquelle ces changements ont été faits à la base. Une autre règle à respecter également est l'atomicité des commits, c.-à-d. que chaque commit doit avoir un seul objectif et doit le mettre en œuvre complètement. En effet, l'historique des changements de code dans les SCV devient biaisé si certains commits contiennent des modifications qui concernent plusieurs tâches n'ayant aucun lien en commun (Herzig & Zeller 2013), ou également si des modifications qui concernent une tâche spécifique sont réparties à travers plusieurs commits (Arima et al. 2018).

#### **1.1.4 L'intégration continue**

Il y a encore quelques années, l'intégration d'un logiciel se faisait une fois sa réalisation terminée. On s'apercevait que l'intégration de toutes les fonctionnalités à la fois est plus compliquée qu'il n'y paraissait, et on se retrouvait alors face à plusieurs imprévus. Alors qu'au niveau de la gestion des équipes, les ressources de développement et celles de l'exploitation travaillaient en silo. Cette séparation avait créé un environnement marqué par une mauvaise communication, un mauvais alignement et des retards dans la livraison des produits et services.

L'avènement des méthodes agiles comme Scrum (Schwaber & Sutherland 2011) et le besoin d'étendre les principes agiles au-delà des limites du code à l'ensemble des services fournis, une nouvelle culture, appelée DevOps une combinaison de deux mots (1) Développement et (2) Opération (Exploitation), s'est développée ces dernières années avec l'aide des experts de l'ESM (*Enterprise Systems Management*). En effet, ces derniers ont apporté les principales pratiques ESM telles que la gestion de configuration, le

visionnement automatisé, la surveillance des systèmes et l'approche de la chaîne d'outils. L'objectif principal étant de promouvoir un esprit de collaboration entre les équipes de développement et d'exploitation. La culture DevOps repose sur l'application de cinq pratiques essentielles (Pressman & Maxim 2020) :

- **Le développement continu** : la décomposition des livrables logiciels à travers des sprints. Une fois ces incréments développés, ils sont livrés aux membres de l'assurance qualité de l'équipe de développement.
- **Les tests continus** : l'utilisation d'outils de test automatisés pour aider les membres de l'équipe à tester leurs changements de code afin de s'assurer qu'ils sont exempts de bogues avant de les archiver.
- **L'intégration continue** : l'intégration des changements de code dans la branche principale du dépôt en les testant le plus rapidement et le plus souvent possible, en vue de les déployer par la suite dans un environnement de test ou de production. En suivant cette démarche, il sera beaucoup plus facile de recevoir et d'intégrer en continu les rétroactions des utilisateurs.
- **Le déploiement continu** : en complément du processus de la livraison continue qui automatise la publication d'une version prête à passer en production dans un référentiel de code, le déploiement continu permet de livrer automatiquement chaque modification apportée au logiciel directement aux utilisateurs finaux.
- **Le contrôle continu** : la surveillance des performances du logiciel dans l'environnement de production par l'équipe opérationnelle, en cherchant de manière proactive les problèmes éventuels avant que les utilisateurs ne les rencontrent.

## 1.2 L'APPRENTISSAGE AUTOMATIQUE

L'apprentissage automatique est un champ d'études de l'IA qui permet de donner aux machines la capacité d'apprendre à partir de données, et de traiter des problèmes trop difficiles à résoudre avec des programmes fixes qui sont développés par des êtres humains (Goodfellow et al. 2016). Formellement, cet apprentissage a été décrit comme suit : « On dit qu'un programme informatique apprend de l'expérience  $E$  par rapport à une certaine classe de tâches  $T$  et une mesure de performance  $P$ , si son exécution des tâches dans  $T$ , mesurée par  $P$ , s'améliore avec l'expérience  $E$  » (Mitchell 1997).

### 1.2.1 Les tâches résolues

Les tâches  $T$  d'apprentissage automatique sont généralement décrites d'une façon qui permet de montrer au système d'apprentissage comment la machine devrait traiter un exemple (Goodfellow et al. 2016). D'ailleurs, un exemple est une collection de caractéristiques qui ont été mesurées quantitativement à partir d'un objet ou un événement à traiter, et il est généralement représenté en tant que vecteur  $x \in \mathbb{R}^n$  où chaque entrée  $x_i$  du vecteur est une des ses caractéristiques (Goodfellow et al. 2016). Parmi les tâches les plus courantes, on trouve la classification, la régression, le partitionnement des données, la détection des anomalies ou encore la transcription.

La tâche qui nous intéresse est la classification, dans laquelle le programme informatique doit spécifier auxquelles des  $k$  catégories appartient une entrée. Pour résoudre cette tâche, il est généralement demandé à l'algorithme d'apprentissage de produire une fonction  $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$ . Lorsque  $y = f(x)$ , le modèle affecte une entrée décrite par le vecteur  $x$  à une catégorie identifiée par le code numérique  $y$ . Il existe d'autres variantes de la tâche de classification, par exemple, celle où  $f$  produit une distribution de probabilité sur les classes (Goodfellow et al. 2016).

### 1.2.2 Principes généraux

On distingue deux grands types d'apprentissage automatique, la différence principale entre les deux types réside dans le jeu de données utilisé. Le premier type, appelé apprentissage non supervisé, consiste à observer plusieurs exemples d'un vecteur aléatoire  $x$  puis d'essayer d'apprendre implicitement ou explicitement soit la distribution de probabilité  $p(x)$ , soit certaines propriétés intéressantes de cette distribution. Comme il peut être également utilisé pour simplifier la représentation des données en faisant, par exemple, une réduction de dimensionnalité (Goodfellow et al. 2016). Quant au deuxième type qui est l'apprentissage supervisé, il consiste à observer plusieurs exemples et les valeurs qui lui sont associées, dans le but d'apprendre à prédire  $y$  à partir de  $x$ , généralement en estimant  $p(y | x)$  (Goodfellow et al. 2016). Le terme apprentissage supervisé provient du point de vue de la cible ou l'étiquette  $y$  (*label*, en anglais) étant fourni au système d'apprentissage automatique.

D'autres types d'apprentissage sont possibles (Goodfellow et al. 2016), comme l'apprentissage semi-supervisé où certains exemples comprennent un objectif de supervision, mais d'autres ne l'ont pas. On trouve également des algorithmes d'apprentissage par renforcement qui interagissent avec un environnement, à travers une boucle de rétroaction entre le système d'apprentissage et ses expériences.

Pour évaluer l'efficacité d'un modèle, on divise souvent le jeu de données (*Dataset*, en anglais) en trois ensembles : un ensemble d'entraînement sur lequel le modèle fait son apprentissage, un ensemble de validation dans lequel se fait un réglage de certains hyperparamètres du modèle, et un ensemble de tests non vu auparavant par le modèle et qui permet de garantir la généralisation de sa performance. Cette évaluation peut se faire soit à travers quelques mesures quantitatives, soit en mesurant le taux d'erreur qui est la proportion d'exemples pour lesquels le modèle produit une sortie incorrecte, ce taux est appelé erreur d'apprentissage durant la phase d'entraînement et erreur de test ou de généralisation durant la phase de test (Goodfellow et al. 2016).

En parlant toujours de la tâche de classification, qui est une méthode d'apprentissage supervisé, il faut savoir que les prédictions concernant une classe A, par exemple, appartiendront à l'une de ces quatre catégories :

- **Vrai positif (VP)** : les exemples correctement attribués à la classe A.
- **Vrai négatif (VN)** : les exemples non attribués à la classe A, et qui n'appartiennent pas à celle-ci.
- **Faux positif (FP)** : les exemples attribués à la classe A, et qui n'appartiennent pas à celle-ci.
- **Faux négatif (FN)** : les exemples attribués à une autre classe, mais qui appartient en réalité à la classe A.

À travers ces catégories, on peut procéder au calcul de certaines métriques comme la précision (1) qui représente le nombre d'exemples attribués à une classe et qui appartiennent effectivement à cette classe, le rappel (*recall*, en anglais) (2) qui est le nombre d'exemples appartenant à une classe qui sont effectivement classés dans celle-ci, et le score F1 (3) qui combine les deux mesures, car elles ont tendance à être inversement liées, et puis aussi parce qu'il est beaucoup plus pratique de comparer plusieurs modèles avec une seule mesure de performance.

$$Précision = \frac{VP}{VP + FP} \quad (1)$$

$$Rappel = \frac{VP}{VP + FN} \quad (2)$$

$$Score F1 = 2 \times \frac{précision \times rappel}{précision + rappel} \quad (3)$$



L'un des inconvénients du score F1, c'est qu'il ne tient pas compte des VN (Powers 2015). De ce fait, certaines études (Levin & Yehudai 2017, Hönel et al. 2020) préfèrent utiliser deux métriques qui sont l'exactitude (*accuracy*, en anglais) (4) qui est le taux de prédictions correctes pour une classe sur l'ensemble de données, et la métrique Kappa de Cohen (Cohen 1960) (5) qui mesure le degré de concordance entre deux évaluateurs ( $p_o$ ) par rapport au hasard ( $p_e$ ). Elle est utilisée principalement pour pallier le problème d'asymétrie entre les classes (Levin & Yehudai 2017), un problème qui n'est pas couvert par l'exactitude.

$$Exactitude = \frac{VP + VN}{VP + FP + VN + FN} \quad (4)$$

$$Kappa = \frac{p_o - p_e}{1 - p_e} \quad (5)$$

Où :  $p_o$  : la proportion de concordance (exactitude du modèle).  
 $p_e$  : la proportion de concordance due au hasard.

Toutes ces métriques qu'on vient de présenter sont relatives à une seule classe, mais elles sont suffisantes pour évaluer une tâche de classification binaire ou en classes multiples, et ce, en procédant de la même manière pour le reste des classes puis en calculant, par exemple, la moyenne pour chaque métrique. Cependant, l'évaluation dans le cadre d'une classification multiétiquettes nous oblige à analyser les prédictions un peu différemment, car chaque exemple peut être associé à plusieurs classes simultanément, et par conséquent, la prédiction peut être totalement correcte, partiellement correcte ou totalement incorrecte. Et donc, comme aucune des mesures d'évaluation qu'on vient de présenter ne permet de capturer une telle notion (Sorower 2010), une adaptation de quelques métriques à la classification multiétiquettes a été proposée. Comme présenté à la figure 1.2, les métriques sont généralement classées en deux groupes (Zhang & Zhou 2013) : les métriques basées sur

les étiquettes, et les métriques basées sur les exemples qui sont spécifiques au contexte multiétiquettes.

Plus précisément, l'approche basée sur les exemples fonctionne en évaluant les prédictions sur chaque exemple séparément, puis en renvoyant la moyenne sur l'ensemble des exemples. De l'autre côté, l'approche basée sur les étiquettes fonctionne en évaluant les prédictions qui concerne chaque classe séparément, puis en renvoyant la moyenne macro ou micro de l'ensemble des classes (Sorower 2010).

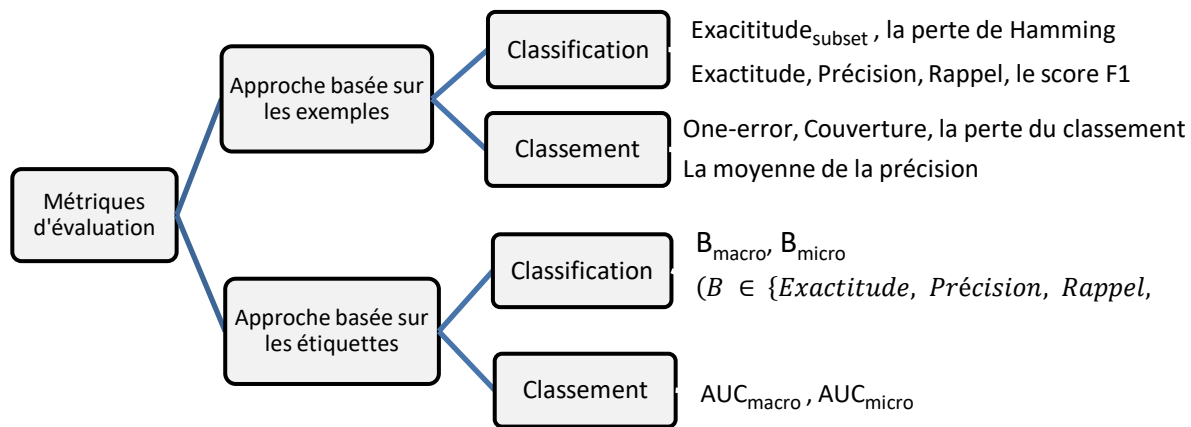


Figure 1.2 – Les métriques d'évaluation utilisées dans le contexte multiétiquettes.

Source: *A Review on Multi-Label Learning Algorithms.* (Zhang & Zhou 2013)

Comme on peut constater à travers la figure 1.2, pour ce qui est de la tâche de classification, certaines métriques seront calculées différemment selon l'approche adoptée, on trouve donc :

➤ **L'approche basée sur les exemples :**

Soit  $D$  un jeu de données composé de  $|D|$  exemples dont chacun peut avoir plus d'une étiquette  $(X_i, Y_i)$ ,  $i = 1 \dots |D|$ ,  $Y_i \subseteq L$ , où  $L$  est l'ensemble des classes,  $h$  est un classificateur

multi-étiquettes et  $Z_i = h(x_i)$  est l'ensemble des prédictions de  $h$  pour un exemple  $X_i$ . Les six métriques de classification dans cette approche sont définies comme suit (Tsoumakas & Vlahavas 2007) :

- L'exactitude du sous-ensemble (*subset accuracy*, en anglais) (6) : Il s'agit d'une mesure d'évaluation très stricte, car elle exige que l'ensemble d'étiquettes prédites pour un exemple corresponde exactement à l'ensemble d'étiquettes réelles. Ici,  $I$  est une fonction indicatrice, où  $I(true) = 1$  et  $I(false) = 0$ .

$$Exactitude_{subset}(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} I(Z_i = Y_i) \quad (6)$$

- La perte de Hamming (7) : elle donne une idée sur à quel point  $Z_i$  est proche de  $Y_i$ . Ici,  $\Delta$  calcule la différence symétrique<sup>6</sup> entre deux ensembles.

$$Perte\ de\ Hamming(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{|Y_i \Delta Z_i|}{|L|} \quad (7)$$

Ensuite nous avons la nouvelle façon de calculer les métriques précédemment présentées dans les formules (1, 2, 3 et 4) :

$$Exactitude(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{|Y_i \cap Z_i|}{|Y_i \cup Z_i|} \quad (8)$$

---

<sup>6</sup> La différence symétrique de A et de B, notée «  $A \Delta B$  » est l'ensemble des éléments qui appartiennent soit à A, soit à B, mais pas aux deux à la fois. Elle peut être représentée par l'équation suivante :  $A \Delta B = (A \cup B) / (A \cap B)$

$$Précision(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{|Y_i \cap Z_i|}{|Z_i|} \quad (9)$$

$$Rappel(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{|Y_i \cap Z_i|}{|Y_i|} \quad (10)$$

$$Score F1(h, D) = \frac{1}{|D|} \sum_{i=1}^{|D|} \frac{2 |Y_i \cap Z_i|}{|Z_i| + |Y_i|} \quad (11)$$

➤ **L'approche basée sur les étiquettes :**

Ici, les métriques possèdent les mêmes formules qu'on avait présentées pour l'évaluation d'une seule classe, mais puisque qu'on possède ici plusieurs classes, le calcul de ces métriques se fait à travers deux moyennes appelées : moyenne-macro et moyenne-micro. D'ailleurs, cette approche de calcul est la même utilisée dans la classification à classes multiples.

Comme décrit par Tsoumakas & Vlahavas (2007), considérons une métrique d'évaluation  $B$  qui est calculée en fonction du nombre de VP, VN, FP, et FN. Soit  $|L|$  le nombre de classes, et  $VP_\lambda$ ,  $VN_\lambda$ ,  $FP_\lambda$  et  $FN_\lambda$  le résultat des prédictions pour une classe  $\lambda$ . Les deux moyennes de  $B$  sont calculées comme suit (Yang 1999) :

$$B_{macro} = \frac{1}{|L|} \sum_{\lambda=1}^{|L|} B(VP_\lambda, FP_\lambda, VN_\lambda, FN_\lambda) \quad (12)$$

$$B_{micro} = B \left( \sum_{\lambda=1}^{|\mathcal{L}|} VP_{\lambda}, \sum_{\lambda=1}^{|\mathcal{L}|} FP_{\lambda}, \sum_{\lambda=1}^{|\mathcal{L}|} VN_{\lambda}, \sum_{\lambda=1}^{|\mathcal{L}|} FN_{\lambda}, \right) \quad (13)$$

Tout simplement, la moyenne-macro fait une moyenne simple des scores de chaque classe, et la moyenne-micro regroupe d'abord l'ensemble des prédictions par catégories (VP, VN, FP et FN) puis procède au calcul de la métrique.

Les différences entre les deux peuvent être nombreuses (Manning 2008) : la moyenne-macro donne un poids égal à chaque classe, tandis que la moyenne-micro donne un poids égal à chaque prédiction de classification pour un exemple. D'autre part, les résultats d'une moyenne-micro donne une idée claire sur l'efficacité du modèle sur les classes les plus représentées dans l'ensemble de test, tandis que la moyenne-macro la donne sur les classes les moins représentées.

### 1.3 L'APPRENTISSAGE PROFOND

L'apprentissage profond (*deep learning*, en anglais) est un type d'IA qui s'appuie essentiellement sur un réseau de neurones. On retrouve son origine dans des approches relativement anciennes, mais son exposition actuelle est due à l'amélioration de l'infrastructure informatique (matériel et logiciel) et à l'augmentation croissante du nombre de données numériques (Goodfellow et al. 2016).

### 1.3.1 Principes généraux

Vu que l'on s'inspire du fonctionnement du cerveau humain pour mener les ordinateurs à se comporter intelligemment, on fait souvent face à une grande difficulté à articuler de manière formelle les connaissances humaines subjective et intuitive (Goodfellow et al. 2016). La représentation des données joue un rôle essentiel dans la performance des algorithmes d'apprentissage automatique, surtout que pour de nombreuses tâches, il est difficile de savoir quelles caractéristiques à extraire. Une solution à ce problème consiste à découvrir non seulement le mapping de la représentation à la sortie, mais aussi la représentation elle-même à travers ce qu'on appelle l'apprentissage de la représentation (Goodfellow et al. 2016). L'enjeu majeur de ce dernier est que beaucoup de facteurs de variation souvent invisibles influencent chaque donnée que nous sommes capables d'observer. Ces facteurs peuvent être considérés comme des concepts ou abstractions qui aident à comprendre la riche variabilité des données. D'où l'apparition de l'apprentissage profond qui permet de résoudre ce problème en introduisant des représentations qui sont exprimées sur le plan d'autres représentations plus simples sous une forme hiérarchique (Goodfellow et al. 2016).

Comme on peut constater à travers la figure 1.3, cette hiérarchie permet un apprentissage des abstractions complexes à un niveau donné en fonction des abstractions relativement simples formulées dans le niveau précédent de la hiérarchie (Goodfellow et al. 2016). L'entrée est présentée à la couche visible, ainsi nommée, car elle contient les variables que nous sommes capables d'observer. Ensuite une série de couches cachées extrait des caractéristiques de plus en plus abstraites au fur et à mesure, jusqu'à l'arrivée à la couche de sortie où le modèle fait ses prédictions (Goodfellow et al. 2016). Dans la section suivante, nous expliquerons théoriquement ce qui se passe au sein de cette hiérarchie.

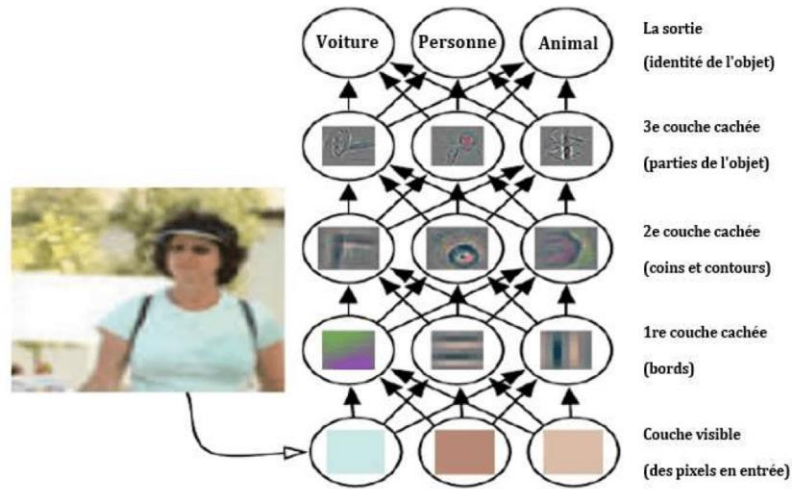


Figure 1.3 – Illustration de la hiérarchie d’un modèle d’apprentissage profond.

Source : *Deep Learning*, (Goodfellow et al. 2016)

### 1.3.2 Les réseaux de neurones

Le modèle de réseau de neurones basique peut être décrit comme une série de transformations fonctionnelles où tout d’abord  $M$  combinaisons linéaires des variables d’entrée  $x_1, \dots, x_D$  seront construites sous la forme (Bishop & Nasrabadi 2006) :

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad (14)$$

Où  $j = 1, \dots, M$  et l’exposant (1) indique que les paramètres correspondants à la première couche du réseau. Les paramètres  $w_{ji}^{(1)}$  sont appelés poids (*weights*, en anglais), et les paramètres  $w_{j0}^{(1)}$  des biais (*biases*, en anglais). Sur chacun des  $a_j$  résultant, une fonction

d'activation non linéaire différentiable  $h(\cdot)$  sera appliquée (Bishop & Nasrabadi 2006), pour donner :

$$z_j = h(a_j) \quad (15)$$

Les fonctions non linéaires  $h(\cdot)$  souvent utilisées sont les deux fonctions sigmoïde ou tanh (Bishop & Nasrabadi 2006). Comme le montre la figure 1.4, ces valeurs  $z_j$  représentant les sorties de la couche cachée sont à nouveau combinées linéairement et transmis à la couche de sortie, sous forme :

$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)} \quad (16)$$

Où  $k = 1, \dots, K$  et  $K$  est le nombre total de neurones en sortie. Cette transformation correspond à la deuxième couche du réseau, et là encore les  $w_{k0}^{(2)}$  sont les paramètres de biais. Finalement à ce stade, le choix de la fonction d'activation suit les mêmes considérations que pour les modèles linéaires (Bishop & Nasrabadi 2006). Ainsi, pour les problèmes de régression standard, la fonction d'activation est celle d'identité de sorte que  $y_k = a_k$ . Pour les problèmes de classification binaire ou multiétiquettes, c'est la fonction sigmoïde (17) qui est appliquée. Sinon quand il s'agit d'une classification en classes multiples, c'est la fonction Softmax qui est utilisée (Bishop & Nasrabadi 2006).

$$y_k = \sigma(a_k) , \quad \text{où : } \sigma(a) = \frac{1}{1 + e^{-a}} \quad (17)$$

En combinant finalement toutes ces étapes, la fonction du réseau globale, dans le cas d'utilisation de la fonction sigmoïde, prend la forme suivante (Bishop & Nasrabadi 2006) :



$$y_k(x, w) = \sigma \left( \sum_{j=1}^M w_{kj}^{(2)} h \left( \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \quad (18)$$

Cette fonction où tous les paramètres de poids et de biais sont regroupés dans un vecteur  $W$ . Ainsi, le modèle du réseau de neurones est simplement une fonction non linéaire d'un ensemble d'entrées  $\{x_i\}$  à un ensemble de sorties  $\{y_k\}$  contrôlées par un vecteur  $W$  de paramètres ajustables (Bishop & Nasrabadi 2006). Cette fonction peut être représentée sous forme d'un réseau comme le montre la figure 1.4. Dans ce réseau, les variables d'entrée cachée et de sortie sont représentées par des nœuds, les paramètres des poids sont représentés par des liens entre les nœuds, et les paramètres des biais  $z_0$  et  $x_0$  sont indiqués par un lien provenant d'une entrée supplémentaire et de variables cachées. Les flèches indiquent la direction du flux d'informations dans le réseau lors d'une propagation vers l'avant (Bishop & Nasrabadi 2006).

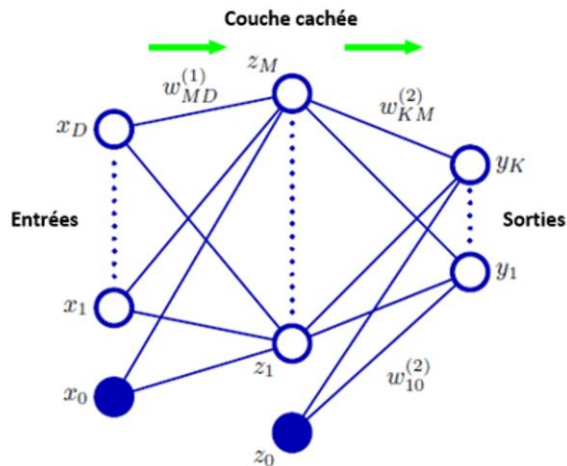


Figure 1.4 – Représentation d'un réseau de neurones à 2 couches.  
*Source : Pattern Recognition and Machine Learning. (Bishop & Nasrabadi 2006)*

### 1.3.3 La rétropropagation de l'erreur

La mise en place d'un modèle performant nécessite la détermination des meilleurs biais et poids applicables à chacune des connexions interneuronales. Ce processus s'effectue à travers ce qu'on appelle la rétropropagation (*backpropagation*, en anglais). Plus précisément, la rétropropagation permet de calculer le gradient de l'erreur pour chaque neurone, de la dernière couche vers la première. Cela permet de corriger les erreurs selon l'importance, c.-à-d. que les poids qui engendrent une erreur importante se verront modifiés d'une manière plus significative que ceux qui en engendrent moins (Goodfellow et al. 2016). Le but étant de minimiser au maximum la fonction de coût en convergeant de manière itérative vers une configuration optimale des poids et des biais.

### 1.3.4 La régularisation

Le grand défi de tout problème d'apprentissage automatique est de garantir une généralisation du modèle, c.-à-d. que le modèle obtenu doit être performant non seulement sur les données sur lesquelles il a été entraîné, mais également sur des données non observées par lui auparavant. Pour y parvenir, il faut minimiser l'erreur d'entraînement et réduire au maximum l'écart entre elle et l'erreur de test (Goodfellow et al. 2016). En ignorant ces deux facteurs, on risque de se retrouver soit avec ce qu'on appelle un sous-apprentissage (*underfitting*, en anglais) qui est l'incapacité du modèle à obtenir une valeur suffisamment faible pour l'erreur d'apprentissage, soit on se retrouve avec un surapprentissage (*overfitting*, en anglais) qui est un ajustement trop étroit ou exact à l'ensemble d'apprentissage, on l'aperçoit facilement avec une augmentation de l'erreur de test et une diminution de l'erreur d'apprentissage. Comme le montre bien la figure 1.5, l'objectif est donc de trouver le juste milieu entre ces deux notions afin d'avoir une capacité optimale du modèle (Goodfellow et al. 2016).

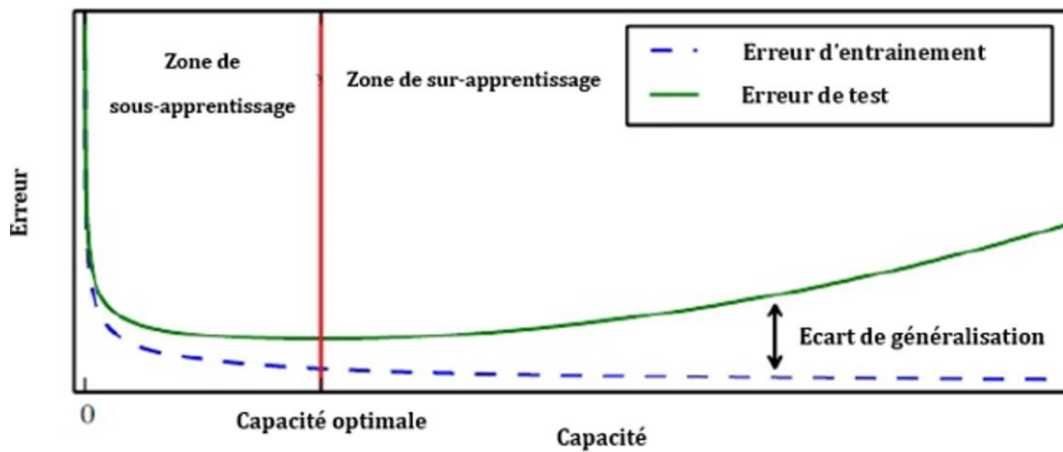


Figure 1.5 – Capacité d’un modèle.  
 Source : *Deep Learning*. (Goodfellow et al. 2016).

Ce problème de surapprentissage qui est fréquemment rencontré est la raison pour laquelle la technique de régularisation est utilisée. Cette dernière est définie comme étant toute modification qu’on apporte à un algorithme d’apprentissage dans le but de réduire l’erreur de généralisation, mais pas l’erreur d’apprentissage (Goodfellow et al. 2016). De nombreuses approches de régularisation reposent sur la limitation de la capacité des modèles en ajoutant une pénalité de normes des paramètres  $\Omega(\theta)$  à la fonction de coût. Parmi ces techniques les plus courantes, on trouve la pénalité  $L_2$  (19) appelé aussi méthode de dégradation des poids (*Weight decay*, en anglais) qui fait en sorte d’amener les poids  $\omega$  du modèle vers des valeurs proches de zéro, et la pénalité  $L_1$  (20) qui peut carrément les éliminer en leur attribuant la valeur zéro.

$$\Omega(\theta) = \frac{1}{2} \|\omega\|_2^2 \quad (19)$$

$$\Omega(\theta) = \|\omega\|_1 \quad (20)$$

On dénote donc la fonction de coût régularisée par (Goodfellow et al. 2016) :

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta) \quad (21)$$

Où  $\alpha \in [0, \infty)$  est un hyperparamètre qui contribue également sur la valeur de la pénalité  $\Omega(\theta)$ , et donc mettre  $\alpha$  à 0 n'entraînera aucune régularisation. À noter que pour les réseaux de neurones, on choisit typiquement d'utiliser une pénalité de normes des paramètres  $\Omega$  qui ne pénalise que les poids de la transformation affine à chaque couche et laisse les biais non régularisés (Goodfellow et al. 2016).

En outre, il existe d'autres techniques de régularisation comme l'abandon (*Dropout*, en anglais) qui élimine aléatoirement quelques neurones durant la phase l'entraînement, et l'augmentation des données qui consiste à ajouter des copies légèrement modifiées des données existantes.

### 1.3.5 L'analyse de données séquentielles

Pour passer des réseaux multicouches aux réseaux récurrents, il faut tirer parti de l'une des premières idées découvertes en apprentissage automatique et les modèles statistiques des années 1980 : le partage des paramètres entre différentes parties d'un modèle (Goodfellow et al. 2016). En effet, ce concept permet à un modèle de traiter des exemples de longueurs différentes et de généraliser entre eux. Un tel partage est particulièrement important lorsqu'une information spécifique peut apparaître à plusieurs positions dans une séquence (Goodfellow et al. 2016), et ce sont bien les réseaux de neurones récurrents ou RNN (Rumelhart et al. 1986) qui permettent de répondre au mieux à ce besoin.

Dans une architecture d'un RNN mappant une entrée d'une séquence de  $x$  valeurs à une séquence de sortie de  $o$  valeurs (voir figure 1.6), on constate à gauche sur le schéma que

le RNN et sa perte  $L$  sont représentés à travers des connexions récurrentes. Puis à droite, il s'agit du même graphe, mais qui est étalé dans le temps (Goodfellow et al. 2016), et où chaque cellule est associée à une instance temporelle particulière ( $h^{(t-1)}, h^{(t)}, h^{(t+1)}, \dots$ ) appelé état ou vecteur caché. Le calcul de ce dernier ( $h^{(t)}$ , par exemple) se fait en fonction la matrice de poids  $U$  provenant de l'entrée  $x^{(t)}$  et de la matrice de poids  $W$  provenant de l'état caché précédent  $h^{(t-1)}$ , puis il génère la prédiction  $o^{(t)}$  en utilisant la matrice de poids  $V$ .

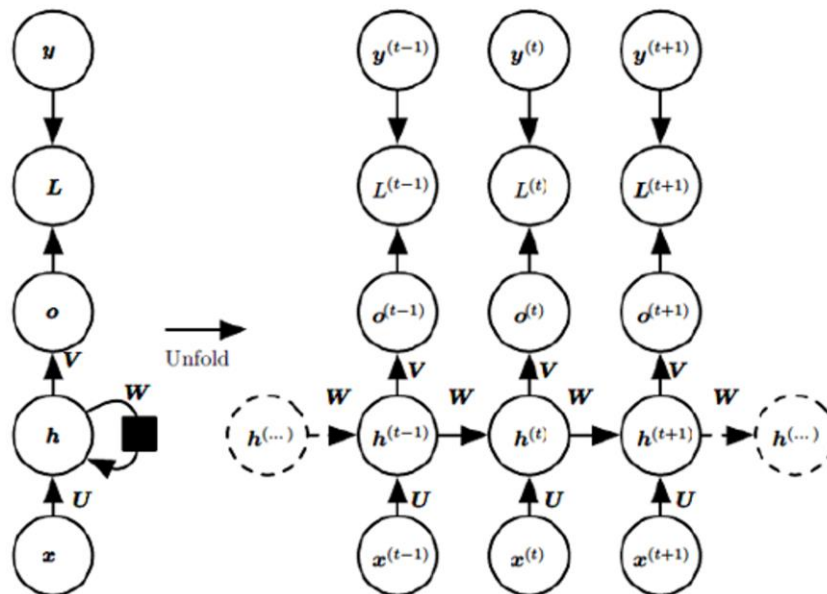


Figure 1.6 – Illustration du principe d'un RNN mappant une séquence d'entrée  $x$  à une séquence de sortie  $y$ . Source : *Deep Learning*. (Goodfellow et al. 2016).

Les cellules d'un RNN peuvent calculer les valeurs des états cachés de différentes manières, et ce, selon l'un de ces trois types d'architectures (voir figure 1.7) (Afshine & Shervine 2019, Ankit 2022, Uzila 2022) :

- **RNN standard** : Ici, chacune cellule reçoit de manière séquentielle une entrée  $x_t$ , tout en se souvenant de la séquence passée à travers l'état caché  $h_{t-1}$ . L'état caché est mis à jour en  $h_t$  et est transmis à la cellule suivante, ou bien utilisé pour générer une prédiction. Ce type de cellule possède deux inconvénients : le temps de calcul qui est long, et le problème de la disparition et l'explosion du gradient quand les séquences traitées sont très longues.
  
- **LSTM (Long Short-Term Memory)** : Pour pallier l'incapacité des cellules standard à maintenir l'information pour des séquences longues, un nouveau type de cellule a été créé avec un état de cellule  $C_t$ . Pour conserver sélectivement les informations, un LSTM dispose également de trois portes :
  - **Porte d'oubli** : Elle utilise  $h_{t-1}$  et  $x_t$ , et produit un vecteur  $f_t$  composé de nombres entre 0 et 1 indiquant l'information à rejeter de l'état de la cellule  $C_{t-1}$ .
  - **Porte d'entrée** : Similaire à la porte d'oubli, mais avec une sortie  $i_t$  utilisée pour extraire de l'entrée courante l'information à stocker dans l'état de la cellule.
  - **Porte de sortie** : Similaire à la porte d'oubli, mais avec une sortie  $o_t$  utilisée avec l'état de cellule mis à jour  $C_t$  pour définir le nouvel état caché  $h_t$ .
  
- **GRU (Gated Recurrent Unit)** : Similaire à LSTM, mais avec moins de complexité. Cette cellule fusionne l'état de la cellule et l'état caché, et combine les portes d'oubli et d'entrée en une porte de mise à jour. Plus précisément, elle comporte deux portes :
  - **Porte de réinitialisation** : Elle permet de décider quelles informations devraient être ignorées dans  $h_{t-1}$ .
  - **Porte de mise à jour** : Elle permet de déterminer les informations à conserver et celles à oublier afin de définir le nouvel état caché  $h_t$ .

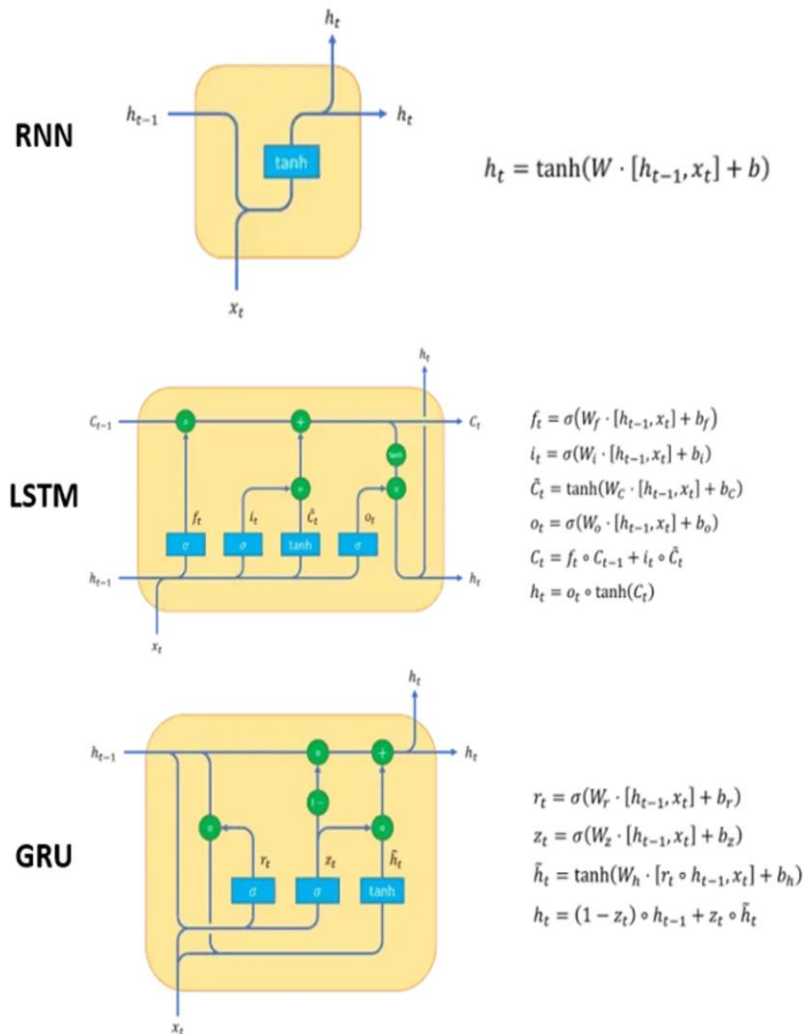


Figure 1.7 – Illustration du fonctionnement des cellules RNN standard, LSTM et GRU.  
 Source : *Beautifully Illustrated: NLP Models from RNN to Transformer* (Uzila 2022)

Malgré les améliorations que les LSTM et GRU proposent par rapport aux cellules RNN standards, ils n'ont pas pu régler définitivement le problème avec les longues séquences, et ce, du fait que la gestion de la mémoire se base uniquement sur la relation entre chaque deux cellules consécutives, ce qui implique que certaines informations risquent tout de même d'être oubliées au long terme (Uzila 2022). C'est pour cette raison que le mécanisme d'attention (Bahdanau et al. 2014) a été inventé,

celui-ci permet au modèle d'apprendre à partir de certaines parties (non seulement celle antérieure) de l'entrée pour calculer une sortie particulière, et ce, en attribuant des coefficients à ces parties en fonction de leur importance.

Même si la gestion de la mémoire a été améliorée avec le mécanisme d'attention, le temps de calcul ne l'a pas été. Et ce, à cause de l'impossibilité à tirer profit de la parallélisation vu que les RNN traitent les parties d'une entrée de manière séquentielle (Vaswani et al. 2017). Pour remédier à ce problème, une nouvelle structure appelée « Transformateur » (Vaswani et al. 2017) a été proposée, dans laquelle une architecture encodeur-décodeur basée sur des couches d'attentions a été utilisée. La figure 1.8 montre la partie encodeur du transformateur qui se compose des éléments suivants (Vaswani et al. 2017) :

- **Une couche de plongement** : Elle permet de transformer l'entrée textuelle en un vecteur de nombres réels que la machine pourra traiter. Ici, chaque mot, selon sa signification, est lié à une valeur particulière.
- **L'encodage positionnel** : Un mot peut avoir plusieurs significations dans différentes phrases. Les RNN prennent en compte l'ordre des mots en analysant séquentiellement une phrase mot par mot, ce qui n'est pas le cas pour le transformateur qui a éliminé cette récurrence. Du coup, il s'agit tout simplement de vecteurs qui permettent au modèle d'avoir une idée sur la position du mot dans une phrase.
- **L'autoattention multitêtes** : Ce module utilise le mécanisme d'attention pour déterminer la pertinence d'un mot particulier par rapport aux autres mots de la phrase. Il est dit multitêtes dans le sens où différentes combinaisons de trois composants qui sont la clé (key  $K$ ), la valeur (value  $V$ ) et la requête (query  $Q$ ), peuvent être divisées en partie parallèles permettant ainsi au



modèle de tenir compte de différentes perspectives concernant les mots d'une phrase.

- **Réseau de neurones à propagation avant** : Il permet de générer un vecteur de dimension « 512 »<sup>7</sup> pour chaque entrée individuelle. Il se compose de deux transformations linéaires avec une activation ReLU entre les deux. Tout comme dans le module d'autoattention multitêtes, une connexion résiduelle suivie d'une normalisation est effectuée.

Finalement, la sortie de l'encodeur peut être transmise à un autre encodeur, au décodeur comme c'est le cas avec une tâche de traduction, ou bien utilisée comme entrée pour toutes autres tâches d'apprentissage automatique.

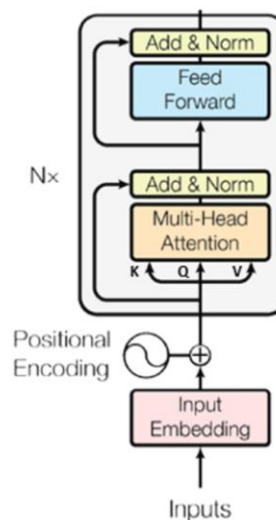


Figure 1.8 – Architecture de la partie encodeur du transformateur

Source : *Attention is all you need.* (Vaswani et al. 2017)

<sup>7</sup> Comme rapporté par Vaswani et al. (2017), le choix de « 512 » a pour but de faciliter les connexions résiduelles appliquées entre chaque deux sous-couches du transformateur. De ce fait, toutes les sous-couches ainsi que les couches de plongement génèrent un vecteur de dimension « 512 ».

## 1.4 SYNTHÈSE

À travers ce chapitre, nous nous sommes situés dans le contexte de notre sujet en présentant le processus des changements de code, les différents types d'activité de maintenance avec leurs caractéristiques, et les SCV qui représentent notre principale source de données. Ensuite, comme nous sommes menés à exploiter entre autres des données textuelles dans notre étude (le message du commit, par exemple), nous avons vu comment l'architecture des transformateurs est devenue la meilleure solution aujourd'hui pour traiter des données séquentielles, remplaçant ainsi les variantes de RNN comme le LSTM et le GRU. Dans le même cadre, le prochain chapitre abordera des principes du traitement automatique des langues naturelles qui est d'ailleurs l'une des branches de l'IA.



## **CHAPITRE 2**

### **LE TRAITEMENT AUTOMATIQUE DES LANGUES**

Les données textuelles sont l'une des principales informations à exploiter dans un commit pour effectuer la classification par activités de maintenance. Ceci nous mène à explorer comment une machine peut extraire des connaissances à partir de ce type de données qui est séquentiel. De ce fait, nous discuterons dans ce chapitre d'abord les principaux concepts du traitement automatique des langues naturelles (TALN), et tout particulièrement les modèles de langues, la représentation des mots et des phrases. Nous allons voir comment nous sommes passées d'une représentation localiste à une représentation distribuée avec la notion des plongements. Finalement, nous aborderons l'utilité des modèles pré-entraînés, tel que le modèle USE (Cer et al. 2018) utilisé dans notre approche, tout en explorant leur capacité de généralisation sur différents domaines d'applications.

#### **2.1 CONCEPTS GÉNÉRAUX**

Comme rapporté par IBM Cloud Education<sup>8</sup>, le langage humain est rempli d'ambiguïtés qui rendent difficile à la machine de déterminer avec précision la signification d'un texte (ou des données vocales). Les homonymes, les homophones, le sarcasme, les idiomes, les métaphores, les différentes variations possibles pour structurer une phrase : ce ne sont que quelques exemples des défis de ce langage que l'humain prend des années à apprendre. Le domaine du TALN, également connu sous le nom de linguistique

---

<sup>8</sup> <https://www.ibm.com/cloud/learn/natural-language-processing>

computationnelle, regroupe un ensemble de techniques utilisées pour l'analyse automatique et la représentation du langage humain (Cambria & White 2014).

Le TALN peut être divisé en deux grands sous-domaines (Otter et al. 2020) : les domaines fondamentaux et d'application, même s'il est difficile parfois de distinguer clairement auquel des deux appartient un problème. Les domaines fondamentaux traitent des thèmes tels que la modélisation des langues, mettant l'accent sur la détermination des associations entre les mots ; le traitement morphologique, traitant la segmentation des composants significatifs des mots afin de déterminer les morphèmes à partir desquels les mots sont construits ; le traitement syntaxique, construisant des diagrammes de phrases dont chaque partie occupe une fonction et dont l'ensemble confère une structure grammaticale à la phrase ; et le traitement sémantique, analysant le sens des mots et des phrases en mettant l'accent sur le contexte de leur utilisation. Quant au domaine d'application, ils impliquent des sujets tels que l'extraction d'informations (par exemple : les entités nommées), la traduction de texte, la génération des résumés de textes, la réponse automatique aux questions, et la classification et le regroupement des textes. Souvent, un domaine d'application doit gérer avec succès un ou plusieurs domaines fondamentaux pour résoudre des problèmes pratiques.

## 2.2 LA REPRÉSENTATION LOCALISTE

En TALN traditionnel, les mots sont considérés comme des symboles discrets. Ces symboles des mots sont représentés par des vecteurs one-hot (multi-hot pour représenter les mots présents dans une phrase) dont la dimension est égale à la taille du vocabulaire (Manning 2022). Suivant ce principe, en faisant la recherche suivante sur le Web : « *Seattle motel* », on est censé avoir également des résultats qui concernent des documents faisant référence à « *Seattle hotel* ». Mais vu que les deux vecteurs one-hot représentant les deux mots « *motel* = [0 0 1 0 0 0] » et « *hotel* = [0 0 0 0 1 0] » sont orthogonaux, il n'y a pas moyen de capturer la notion naturelle de similarité pour ce genre de représentation. Une solution

envisageable pour ce problème est de tirer profit de WordNet (Miller 1995), qui est un thésaurus contenant des listes d'ensembles de synonymes et d'hyperonymes, pour capturer les similitudes entre les mots. Sauf que même cette solution reste très limitée du fait que WordNet est dans l'incapacité de capturer le sens de certains mots dont la signification change selon le contexte, l'impossibilité à déterminer la similarité exacte (par exemple : « *proficient* » est considéré comme synonyme de « *good* »), et nécessite une intervention humaine pour adapter à chaque fois cette ressource (Manning 2022). Finalement, la meilleure solution trouvée pour ce problème est d'apprendre à coder la similarité dans les vecteurs eux-mêmes. C'est ce que nous expliquerons dans la section suivante.

### 2.3 LA REPRÉSENTATION DISTRIBUÉE : LES PLONGEMENTS

L'une des idées les plus réussies du TALN statistique moderne est la représentation (sémantique) distribuée. Celle-ci se base sur le fait que la signification d'un mot est donnée par les mots qui apparaissent fréquemment à proximité (Manning 2022).

Lorsqu'un mot  $w$  apparaît dans un texte, son contexte est l'ensemble des mots qui apparaissent à proximité (dans une fenêtre de taille fixe), et donc, l'idée serait d'utiliser de nombreux contextes de  $w$  pour construire une représentation de  $w$  (Manning 2022). Plus précisément, il faut construire un vecteur dense pour chaque mot, choisi de manière à ce qu'il soit similaire aux vecteurs de mots qui apparaissent dans des contextes similaires, en mesurant la similarité en tant que produit scalaire vectoriel (Manning 2022). Ces vecteurs de mots sont également appelés plongements (*embeddings*, en anglais) de mots ou représentations neuronales de mots, et comme nous verrons dans les sous-sections suivantes, on distingue trois différentes approches pour les construire.

### 2.3.1 Factorisation matricielle globale

Un exemple de cette approche est l'analyse sémantique latente (LSA, *Latent Semantic Analysis*) ou indexation sémantique latente (LSI, *Latent Semantic Indexation*). La LSA (Deerwester et al. 1990) permet d'établir des relations entre un ensemble de documents et les termes qu'ils contiennent, en construisant des concepts liés aux documents et aux termes. Elle se base sur la décomposition en valeurs singulières (SVD, *Singular Value Decomposition*) d'une matrice qui est un outil important de factorisation des matrices (Jurafsky & Martin 2009).

Pour extraire les plongements avec cette technique, il va falloir d'abord parcourir un ensemble massif de données et en accumulant la fréquence de la cooccurrence de mots dans une matrice  $X$ . Ensuite, il faut effectuer une décomposition en valeurs singulières sur  $X$  pour obtenir une décomposition  $USV^T$ , puis finalement, il suffit d'utiliser les lignes de  $U$  comme plongements de tous les mots du dictionnaire (Manning 2022).

En ce qui concerne la matrice  $X$ , deux choix sont envisageables (Manning 2022) : une matrice Mot-Document ou une matrice de cooccurrence basée sur la fenêtre. Le premier choix part du principe que les mots qui sont liés apparaîtront souvent dans les mêmes documents (par exemple : « banques », « obligations » et « actions »), et donc, la construction de la matrice se fait en itérant sur des milliards de documents et pour chaque fois que le mot  $i$  apparaît dans le document  $j$ , l'entrée  $X_{ij}$  sera incrémentée par un. Pour le deuxième choix, la matrice  $X$  stocke les cooccurrences de mots devenant ainsi une matrice d'affinité. Pour tous les mots du corpus, il va falloir compter le nombre de fois que chaque mot apparaît à l'intérieur d'une fenêtre d'une taille particulière autour du mot d'intérêt.

Certes, cette approche nous donne des vecteurs de mots qui sont plus que suffisants pour coder les informations sémantiques et syntaxiques (principalement utilisées pour saisir la similitude de mots). Cependant, elle comporte beaucoup d'inconvénients liés à la matrice (Manning 2022), comme le coût de calcul d'une matrice  $m \times n$  est  $O(mn^2)$ , sa dimension change fréquemment avec l'ajout de nouveaux mots, et sa grande dimension qui est en

général de ( $10^{16} \times 10^{16}$ ) en plus d'être extrêmement éparses puisque la plupart des mots ne coexistent pas.

### 2.3.2 Fenêtres contextuelles locales

Cette approche se base sur une très importante hypothèse en linguistique qui est la similarité distributionnelle, c.-à-d. que les mots similaires ont un contexte similaire (Manning 2022). Ici, les vecteurs des mots représentent tout simplement les paramètres d'un modèle. Ce dernier est entraîné sur un certain objectif, et donc à chaque itération, les paramètres du modèle qui ont causé l'erreur seront pénalisés suivant le concept de la rétropropagation.

L'exemple le plus connu dans cette approche est Word2vec (Mikolov et al. 2013b). Ce dernier se base sur un grand corpus de texte où chaque mot d'un vocabulaire fixe est représenté par un vecteur. Comme décrit par Manning (2022), l'idée est de parcourir chaque position  $t$  dans le texte qui a un mot central  $c$  et un contexte de mots  $o$ , et d'utiliser la similarité des vecteurs de mots pour  $c$  et  $o$  pour calculer la probabilité de  $o$  étant donné  $c$  (ou vice versa), puis de continuer à ajuster les vecteurs de mots pour maximiser cette probabilité. Le cadre Word2vec comprend deux méthodes d'apprentissage (Manning 2022) : un échantillonnage négatif qui définit un objectif en échantillonnant des exemples négatifs, et Softmax hiérarchique qui définit un objectif à l'aide d'une structure arborescente pour calculer les probabilités pour tout le vocabulaire. Word2Vec vient également sous forme de deux algorithmes :

- **Skip-gram** : il vise à prédire la distribution (probabilité) des mots de contexte à partir d'un mot central.
- **Sac de mots continu (CBOW, Continuous Bag Of Words)** : il fait le contraire et prédit un mot central à partir du contexte environnant en termes de vecteurs de mots.



Cette approche d'extraction des vecteurs des mots se distingue par rapport à la factorisation matricielle globale par sa performance en termes de généralisation sur d'autres tâches, sa capacité à capturer des patrons complexes au-delà de la similarité entre les mots, et à ne pas donner d'importance disproportionnée aux mots fréquents (Manning 2022). En contrepartie, cette approche éprouve quelques limites comme son utilisation inefficace des statistiques, et son évolution (*scaling*, en anglais) qui varie en fonction de la taille du corpus (Manning 2022).

### 2.3.3 Modèles de régression log bilinéaire globale

Pour faire face aux limites des deux approches précédentes, Pennington et al. (2014) ont proposé le modèle GloVe qui part du principe que les ratios de probabilités de cooccurrence peuvent encoder les composants de sens. Comme le montre la figure 2.1 représentant les probabilités de cooccurrence des mots « *ice* » et « *steam* » avec des mots du vocabulaire, la probabilité de cooccurrence du mot « *ice* » avec les deux mots « *solid* » et « *water* » est plus grande que celle le liant avec les deux mots « *gas* » et « *fashion* », et la même chose pour le mot « *steam* » qui est plus fréquemment lié aux deux mots « *gas* » et « *water* ». La dernière ligne du tableau montre qu'à travers la ratio des deux probabilités, il est possible de distinguer les mots ayant un lien direct avec l'un des deux mots étudiés quand le ratio est  $\gg 1$  ou  $\ll 1$ . Et puis pour les mots n'ayant aucun rapport avec les mots étudiés, la valeur du ratio est  $\approx 1$ .

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k ice)$	$1.9 \times 10^{-4}$	$6.6 \times 10^{-5}$	$3.0 \times 10^{-3}$	$1.7 \times 10^{-5}$
$P(k steam)$	$2.2 \times 10^{-5}$	$7.8 \times 10^{-4}$	$2.2 \times 10^{-3}$	$1.8 \times 10^{-5}$
$P(k ice)/P(k steam)$	8.9	$8.5 \times 10^{-2}$	1.36	0.96

Figure 2.1 – Les probabilités de cooccurrence (et le ratio) des mots « *ice* » et « *steam* » dans corpus de 6 milliards de mots. Source : *Glove: Global vectors for word representation* (Pennington et al. 2014)

Pour mettre en pratique ce principe, il fallait donc trouver un moyen pour saisir les ratios de probabilités de cooccurrence comme des composants linéaires de sens dans un espace vectoriel de mots (Manning 2022). La solution proposée avec GloVe est d’adopter la notion de modèle log bilinéaire  $w_i \cdot w_j = \ln P(i | j)$  où  $i$  et  $j$  font références à deux mots. Cette idée a permis d’avoir un modèle se caractérisant par un entraînement rapide, une capacité d’évolution jusqu’à d’énormes corpus et une bonne performance même avec des petits corpus et petits vecteurs (Manning 2022).

## 2.4 LES PLONGEMENTS DE PHRASES

Une méthode simple pour créer une représentation distribuée d’une phrase consiste à utiliser un modèle de plongement de mots pour générer séparément une représentation distribuée de chacun des mots constituant cette phrase, et ensuite de faire la moyenne de toutes les représentations résultantes (Wieting et al. 2016). L’exemple le plus sophistiqué de ces méthodes est le SIF (*Smooth Inverse Frequency*, en anglais) (Arora et al. 2017) qui se distingue par rapport à la méthode simple par l’attribution d’un poids à chacun des vecteurs de mots avant d’effectuer la moyenne. Pour choisir les poids, le SIF se base sur le principe que les mots les plus importants dans une phrase ont tendance à être les moins fréquemment utilisés. En prenant la phrase suivante : « *les nouvelles ont l’air rassurantes* » et en comptant les mots dans Wikipédia, qui est l’une des sources exploitées par cette technique, le mot « *rassurante* » qui apporte le plus de signification à la phrase apparaîtra moins souvent que les mots « *ont* » ou « *les* ». C’est en se basant sur ce concept que le vecteur de plongement de la phrase est construit en premier suivant la formule (22), puis suivi d’une réduction de dimensionnalité de ce vecteur en utilisant l’analyse en composante principale (PCA – *Principal Component Analysis*, Hotelling 1933) ou la décomposition en valeurs singulières (SVD – *Singular Value Decomposition*).

$$V_s \leftarrow \frac{1}{|S|} \sum_{w \in S} \frac{a}{a + p(w)} \times V_w \quad (22)$$

Où :

- $\{V_w : w \in V\}$  : les vecteurs de plongement des mots
- $S$  : une phrase.
- $a$  : un paramètre à ajuster.
- $\{p(w) : w \in V\}$  : la fréquence relative de l'occurrence d'un mot dans un large corpus.
- $V_S$  : le vecteur de plongement de la phrase.

Bien qu'elle soit valide dans certains cas, l'approche basée sur les plongements de mots pour générer la représentation distribuée d'une phrase est considérée comme une méthode simpliste qui ne permet pas de capturer des informations liées à l'ordre des mots dans une phrase et ne prend pas en considération la relation entre ces mots (Frreira 2020, Aponyi 2021). Par exemple, si on prend ces deux phrases : « *Il achète un sac de café* » et « *On se voit au café* », en se basant sur cette méthode, la représentation du mot « *café* » serait la même dans les vecteurs des deux phrases même si leur signification est complètement différente, on parle ici de polysémie. C'est à cause de ce genre de limite que l'utilisation du concept du plongement des phrases a commencé à prendre de l'ampleur. Ce concept se base principalement sur des tâches qui soit prédisent une phrase en sortie au lieu d'un mot, soit traitent la similarité entre deux phrases. Parmi les modèles les plus utilisés à ce sujet, on trouve :

- **InferSent** (Conneau et al. 2017) : l'entraînement de ce modèle s'est fait sur la tâche d'inférence du langage naturel, plus précisément sur le jeu de données SNLI pour « *Stanford Natural Language Inference* » (Bowman et al. 2015) qui se compose de 570 000 paires de phrases en anglais étiquetées manuellement avec l'une des trois classes : contradiction, neutre ou semblable. Comme le montre la figure 2.2, ce modèle vient sous forme d'un réseau siamois, et repose sur une architecture basée sur des Bi-LSTM pour encoder les deux phrases d'entrées. Ensuite, la relation entre les deux

vecteurs résultants ( $u$  et  $v$ ) sera déterminée en appliquant la concaténation, le produit matriciel de Hadamard, et la différence élémentaire absolue. Finalement, le vecteur résultant sera transmis à un classificateur sous forme de couches entièrement connectées.

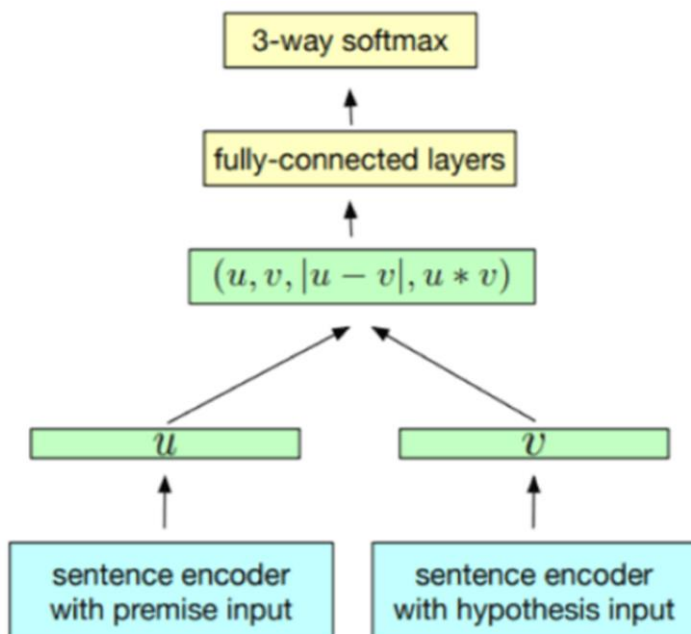


Figure 2.2 – Architecture du modèle InferSent.

Source : *Supervised learning of universal sentence representations from natural language inference data.* (Conneau et al. 2017)

- **Sbert** (Reimers & Gurevych 2019) : tout comme InferSent, ce modèle utilise un réseau siamois avec Bert comme encodeur suivi d'une couche de pooling, au lieu d'un Bi-LSTM. La relation entre les deux plongements générés sera extraite en utilisant la concaténation et la différence élémentaire absolue, puis le résultat sera transmis à un classificateur de 3 couches faisant référence à la tâche d'inférence du langage naturel. Pour l'entraînement du modèle, deux jeux de données ont été

combinés et exploités : SNLI (Bowman et al. 2015) et Multi-Genre NLI (Williams et al. 2018). Ce qui distingue Sbert par rapport au modèle Bert, qui permet lui également de générer une représentation distribuée d'une phrase, c'est que ce dernier comporte des limites quand il s'agit des tâches nécessitant une solide compréhension des phrases, comme c'est le cas pour la recherche sémantique et le regroupement. Et donc, Sbert permet à travers cette approche de construire des plongements de phrases tout en s'assurant que les phrases similaires seront proches dans l'espace vectoriel (Reimers & Gurevych 2019).

➤ **Universal Sentences Encoder – USE** (Cer et al. 2018) : ce modèle diffère des modèles de plongement de mots par son entraînement qui repose sur des tâches qui nécessitent de modéliser la signification de séquences de mots plutôt que de mots individuels. USE propose deux variantes pour son encodeur : « Transformer » basée sur l'architecture du transformateur (Vaswani et al. 2017) et « DAN » basée sur l'architecture de Iyyer et al. (2015). Quant à son entraînement, il est dit multitâche (MTL – *Multi-task learning*, en anglais) c.-à-d. que sa fonction de coût est calculée simultanément en fonction des tâches suivantes :

- « *Modified Skip-thought* » : la prédiction de la phrase qui précède et celle qui suit une phrase donnée.
- « *Conversational Input-Response Prediction* » : la prédiction de la meilleure réponse à un message dans une conversation.
- « *Natural Language Inference* » : elle prend en entrée deux phrases et prédit la relation entre elles à travers ces trois classes : contradiction, neutre ou semblable.

Pour les deux premières tâches qui reposent sur un apprentissage non supervisé, les données d'entraînement ont été extraites à partir du Web (Wikipédia, les actualités, les forums de discussion). Pour la troisième tâche, c'est le jeu de données SNLI (Bowman et al. 2015) qui a été utilisé.

Nous avons choisi d'utiliser le modèle pré-entraîné USE pour une raison évidente puisque son architecture est plus adaptée à notre contexte de classification par activités de maintenance. En effet, contrairement aux modèles InferSent et Sbert dont leur architecture est sous forme d'un réseau siamois prenant deux entrées. Sans oublier le fait que comme il a été rapporté que le modèle Sbert n'est pas adéquat pour une utilisation dans le cadre de l'apprentissage par transfert (Reimers & Gurevych 2019), contrairement au modèle USE qui, par son entraînement multitâche, offre déjà une marge importante de généralisation avant même de procéder à son réentraînement. C'est pourquoi, nous présenterons le modèle USE plus en détail dans la section suivante.

## **2.5 LE MODÈLE USE**

Il s'agit de l'un des modèles les plus récents dans le cadre de la génération des plongements des phrases. L'un de ses points forts réside dans son entraînement multitâche qui fait en sorte à ce que la fonction de coût du modèle soit calculée simultanément en fonction de trois tâches. Cet entraînement qui optimise les poids du composant principal de l'architecture de USE qui est l'encodeur. Ce dernier prend comme entrée une phrase et génère sa représentation distribuée sur un vecteur de taille 512. Il existe deux variantes de cet encodeur, le choix de l'une ou de l'autre dépend des compromis entre la précision et les ressources de calcul disponibles (Cer et al. 2018). Ces deux variantes d'encodeur sont (Cer et al. 2018) :

- **Transformer** : basée principalement sur l'architecture du transformateur (Vaswani et al. 2017), son architecture se compose de 6 couches de transformateurs. Chaque couche a un module d'autoattention (*self-attention*, en anglais) suivi d'un réseau de neurones à propagation avant. Le processus d'autoattention prend en considération l'ordre des mots et le contexte environnant avant de générer la représentation de chacun des mots constituant une phrase. Ensuite, en additionnant ces représentations générées et en divisant par la racine carrée de la longueur de la phrase, on obtient le plongement de la phrase. Cette variante d'encodeur se caractérise par une meilleure précision sur différentes tâches, mais elle nécessite beaucoup de ressources de calcul en raison de son architecture. Sa complexité temporelle est quadratique en fonction de la longueur de la phrase.
- **DAN (Deep Averaging Network)** : basée sur l'architecture de Iyyer et al. (2015), cette variante d'encodeur commence par la génération du vecteur résultant de la moyenne des vecteurs de plongements de chaque mot et ceux des bigrammes. Ensuite, ce vecteur résultant sera l'entrée d'un réseau de neurone profond de 4 couches qui, à sa sortie, va générer le vecteur de plongement de la phrase. Les plongements des mots et ceux des bi-grammes de la première étape sont appris pendant l'entraînement du modèle. Cette variante d'encodeur offre une précision légèrement réduite par rapport à la variante Transformer, mais se caractérise par sa rapidité d'exécution. Sa complexité temporelle est linéaire en fonction de la longueur de la phrase.

Pour voir la généralisation du modèle USE, deux expériences ont été faites (Cer et al. 2018) : la première avait comme but de tester la performance du modèle sur des tâches autres que celles sur lesquelles il a été entraîné. Pour cela, les jeux de données des tâches suivantes ont été utilisés : « MR » : analyse des sentiments des avis sur un film en donnant un score sur

une échelle de cinq étoiles (Pang & Lee 2005), « CR » : l'analyse des sentiments à partir des avis clients (Hu & Liu 2004), « SUBJ » : l'analyse de la subjectivité des résumés et avis concernant un film (Pang & Lee 2004), « MPQA » : l'analyse de la polarité d'opinions à partir des informations d'actualité (Wiebe et al. 2005), « TREC » : la classification détaillée des questions provenant de TREC (Li & Roth 2002), « SST » : l'analyse des sentiments pour la classification des phrases par polarité : positive ou négative (Socher et al. 2013) et « STS Benchmark » : l'analyse de la similarité sémantique entre des paires de phrases (Cer et al. 2017). Quant à la deuxième expérience, elle traite l'enjeu majeur derrière l'utilisation de l'apprentissage par transfert qui est la disposition d'un jeu de données limité pour une tâche donnée. L'idée était de voir l'impact de la variation de la taille du jeu de données sur les résultats de la tâche « SST ».

Les deux expériences ne se sont pas limitées à l'utilisation du modèle pré-entraîné USE uniquement (les modèles « A », voir figures 2.3 et 2.4), d'autres modèles et différentes architectures ont été testés également afin de comparer les résultats. À cet effet, une architecture basée sur les plongements des mots en utilisant le modèle pré-entraîné « word2vec skip-gram » (Mikolov et al. 2013a) a été expérimentée (les modèles « B »). Les plongements générés (*w<sub>2v</sub> w.e.*) par ce dernier sont transmis comme entrée soit à un réseau de neurones convolutif (CNN), soit à un DAN. Une autre architecture similaire à cette dernière, mais qui ne repose pas sur l'apprentissage par transfert a été testée. Dans cette architecture, les plongements de mots (*l<sub>rn</sub> w.e.*) sont d'abord initialisés avec des valeurs aléatoires puis appris durant l'entraînement de la tâche étudiée (les modèles « C »). Ensuite, nous avons une autre architecture où une concaténation des deux vecteurs résultants des modèles pré-entraînés de plongements de mots (*w<sub>2v</sub> w.e.*) et de phrases, qui est testée (les modèles « D »). Et pour finir, une architecture similaire à cette dernière a été testée également, mais avec une concaténation du vecteur résultant des modèles de plongement des phrases, avec le vecteur résultant de l'architecture dont les plongements de mots sont initialisés aléatoirement (*l<sub>rn</sub> w.e.*) (les modèles « E »).



Le résultat des deux expériences, comme le rapportent les figures 2.3 et 2.4, avait montré que les modèles faisant un apprentissage par transfert et qui sont basés sur la variante Transformer du modèle USE donnent généralement de meilleures performances que celles des modèles basés sur le DAN (Cer et al. 2018). D'autre part, les modèles USE mènent à de meilleurs résultats que ceux obtenus avec les modèles basés sur les plongements de mots, tandis que ceux utilisant les deux à la fois aboutissent aux meilleurs résultats sur la plupart des tâches (Cer et al. 2018). Pour la deuxième expérience, il a été remarqué que même pour des quantités de jeu de données limitées, le réentraînement des modèles de USE basés sur le transformateur a pu atteindre des résultats qui ne sont pas très loin de ceux qui sont obtenus quand le jeu de données est assez grand. Quant au reste des modèles, ce n'est qu'en augmentant la taille du jeu de donnée à des valeurs importantes que les modèles commencent à retourner de bons résultats (Cer et al. 2018).

	Model	MR	CR	SUBJ	MPQA	TREC	SST	STS Bench (dev / test)
<i>Sentence &amp; Word Embedding Transfer Learning</i>								
D	USE_D+DAN (w2v w.e.)	77.11	81.71	93.12	87.01	94.72	82.14	–
	USE_D+CNN (w2v w.e.)	78.20	82.04	93.24	85.87	97.67	85.29	–
	USE_T+DAN (w2v w.e.)	81.32	86.66	93.90	88.14	95.51	86.62	–
	USE_T+CNN (w2v w.e.)	81.18	87.45	93.58	87.32	98.07	86.69	–
<i>Sentence Embedding Transfer Learning</i>								
A	USE_D	74.45	80.97	92.65	85.38	91.19	77.62	0.763 / 0.719 (r)
	USE_T	81.44	87.43	93.87	86.98	92.51	85.38	0.814 / 0.782 (r)
E	USE_D+DAN (lm w.e.)	77.57	81.93	92.91	85.97	95.86	83.41	–
	USE_D+CNN (lm w.e.)	78.49	81.49	92.99	85.53	97.71	85.27	–
	USE_T+DAN (lm w.e.)	81.36	86.08	93.66	87.14	96.60	86.24	–
	USE_T+CNN (lm w.e.)	81.59	86.45	93.36	86.85	97.44	87.21	–
<i>Word Embedding Transfer Learning</i>								
B	DAN (w2v w.e.)	74.75	75.24	90.80	81.25	85.69	80.24	–
	CNN (w2v w.e.)	75.10	80.18	90.84	81.38	97.32	83.74	–
<i>Baselines with No Transfer Learning</i>								
C	DAN (lm w.e.)	75.97	76.91	89.49	80.93	93.88	81.52	–
	CNN (lm w.e.)	76.39	79.39	91.18	82.20	95.82	84.90	–

Figure 2.3 – Performance du modèle USE sur différentes tâches.

Source : « Universal sentence encoder ». (Cer et al. 2018)

Model	SST 1k	SST 2k	SST 4k	SST 8k	SST 16k	SST 32k	SST 67.3k	
<i>Sentence &amp; Word Embedding Transfer Learning</i>								
D	USE_D+DNN (w2v w.e.)	78.65	78.68	79.07	81.69	81.14	81.47	82.14
	USE_D+CNN (w2v w.e.)	77.79	79.19	79.75	82.32	82.70	83.56	85.29
	USE_T+DNN (w2v w.e.)	85.24	84.75	85.05	86.48	86.44	86.38	86.62
	USE_T+CNN (w2v w.e.)	84.44	84.16	84.77	85.70	85.22	86.38	86.69
<i>Sentence Embedding Transfer Learning</i>								
A	USE_D	77.47	76.38	77.39	79.02	78.38	77.79	77.62
	USE_T	84.85	84.25	85.18	85.63	85.83	85.59	85.38
E	USE_D+DNN (lrm w.e.)	75.90	78.68	79.01	82.31	82.31	82.14	83.41
	USE_D+CNN (lrm w.e.)	77.28	77.74	79.84	81.83	82.64	84.24	85.27
	USE_T+DNN (lrm w.e.)	84.51	84.87	84.55	85.96	85.62	85.86	86.24
	USE_T+CNN (lrm w.e.)	82.66	83.73	84.23	85.74	86.06	86.97	87.21
<i>Word Embedding Transfer Learning</i>								
B	DNN (w2v w.e.)	66.34	69.67	73.03	77.42	78.29	79.81	80.24
	CNN (w2v w.e.)	68.10	71.80	74.91	78.86	80.83	81.98	83.74
<i>Baselines with No Transfer Learning</i>								
C	DNN (lrm w.e.)	66.87	71.23	73.70	77.85	78.07	80.15	81.52
	CNN (lrm w.e.)	67.98	71.81	74.90	79.14	81.04	82.72	84.90

Figure 2.4 – Performance du modèle USE sur la tâche SST en variant la taille de l'ensemble d'entraînement. *Source* : « *Universal sentence encoder* ». (Cer et al. 2018)

## 2.6 SYNTHÈSE

Ce chapitre nous a permis de voir l'évolution des modèles utilisés en TALN. Les expérimentations faisant un apprentissage par transfert avec le modèle pré-entraîné USE nous permettent de conclure que son utilisation pour exploiter les données textuelles dans un commit pourrait mener également à de bons résultats.

Dans le chapitre suivant qui est consacré à l'état de l'art, nous présenterons les travaux s'intéressant à la tâche de classification par activité de maintenance, puis nous parlerons également des approches s'intéressant à la génération de la représentation distribuée du code modifié dans un commit.



## **CHAPITRE 3**

### **ÉTAT DE L'ART**

Avec l'accessibilité et la disponibilité d'un historique très riche à partir des SCV, plusieurs travaux se sont intéressés à l'étude des changements de code en exploitant les informations du commit qu'ils trouvent utiles pour affronter la problématique étudiée. De ce fait, nous découvrirons dans ce chapitre quelles informations ont été exploitées le plus souvent par les études effectuant la classification par activité de maintenance, les différentes démarches suivies et les résultats obtenus. À notre connaissance, aucune étude n'a proposé une démarche dans laquelle le code modifié est exploité sous forme d'une représentation distribuée. Donc, nous présenterons également en détail dans ce chapitre, un modèle considéré, à ce jour, comme le plus performant dans la génération de ce type de représentation, et ce, en vue de le tester dans notre approche et voir s'il contribuera dans l'amélioration des résultats.

#### **3.1 CONTEXTE DE LA TÂCHE**

Avec l'utilisation fréquente des SCV durant ces 15 dernières années et surtout avec l'augmentation importante des commits qui sont publiquement accessibles à partir des projets à code source ouvert, de plus en plus d'études se sont intéressées à la classification des commits. Selon l'objectif de la tâche étudiée, les études exploitent les informations du commit qui semblent bénéfiques à la résolution du problème. Parmi les sujets abordés, on trouve la classification : atomique ou enchevêtré (Kirinuki et al. 2014), stable ou bogué (Huo et al. 2016, Nayrolles & Hamou-Lhadj 2018, Hoang et al. 2019) , par activité de réusinage

du code (Matsuda et al. 2015, AlOmar et al. 2021), et par activité de maintenance qui est l'objet de notre travail.

Avant de procéder à la classification, le premier enjeu consiste dans le choix des exemples qui seront exploités, c.-à-d., des critères à mettre en place pour sélectionner les projets à partir desquels les commits seront extraits. Ces critères peuvent être par exemple : le type du projet (privée ou à code source ouvert), le domaine d'application du projet, le langage de programmation utilisé, la taille du projet, la phase du cycle de développement étudiée, etc. Les choix faits au niveau de ses critères n'auront pas seulement un impact sur la variété du jeu de données, et par conséquent sur la généralisation des résultats obtenus, mais aussi sur les méthodes à utiliser durant la phase de prétraitement, comme c'est le cas pour le choix du langage de programmation.

Une fois les projets sélectionnés, trois types de données peuvent être exploités à partir de ces commits. Le premier type est en langage naturel comme le message (ou description) du commit, le titre d'une proposition de changement et les noms des fichiers modifiés. Le deuxième type de données est en langage de programmation et concerne les changements de code qui sont sous forme d'ajouts/suppressions de lignes de code (LOC) dans un ou plusieurs fichiers. Le troisième type, appelé métadonnée, est sous forme de valeur numérique et concerne des informations quantitatives comme le nombre de fichiers modifiés et le nombre de LOC ajoutées/supprimées. L'enjeu principal de cette phase est de trouver une représentation des deux premiers types de données de telle sorte que toutes les caractéristiques importantes qui les constituent soient prises en considération lors de la classification. Pour les données en langage naturel, les premières études ont choisi d'encoder le message du commit sous forme d'un vecteur multi-hot dont les éléments sont les mots les plus fréquemment utilisés dans chacun des types d'activité de maintenance ; puis dernièrement, on constate que les études utilisent des modèles de langues pré-entraînés soit en générant directement les représentations distribuées de ces données textuelles (Ghadhab et al. 2021), soit en réentraînant d'abord le modèle sur la tâche étudiée (Sarwar et al. 2020).

Pour ce qui est des changements de code, toutes les études qui ont exploité ce type de données se sont limitées lors de la phase du choix des projets à un seul langage de programmation. Cette limitation est due à la différence dans la syntaxe entre les langages, et par conséquent, à la nécessité de traiter chaque langage différemment et de trouver l'outil correspondant à chacun d'eux pour représenter le code modifié. Cependant, dernièrement des études se sont intéressées à cette problématique, et ont proposé des modèles qui génèrent la représentation distribuée du code (Feng et al. 2020), ou celles des changements de code (Hoang et al. 2020) indépendamment du langage de programmation.

Une fois les informations concernant un commit sont toutes représentées sous forme d'un vecteur de caractéristiques, la dernière étape consiste à choisir un classificateur qui va prendre en entrée ce vecteur et va déterminer quel(s) type(s) d'activité de maintenance sont inclus dans ce commit. D'ailleurs, certaines études ont considéré cette classification à classes multiples, tandis que d'autres (Gharbi et al. 2019, Sarwar et al. 2020) l'ont considéré multiétiquettes, du fait que les commits incluant plus d'un type d'activité de maintenance surviennent fréquemment en pratique (Herzig & Zeller 2013, Nguyen et al. 2013).

Le tableau 3.1 ci-dessous représente une liste des travaux faits dans ce sujet. Dans la section suivante, nous aborderons plus en détail les approches suivies par ces travaux.

Tableau 3.1 – Un aperçu sur les travaux s’intéressant à la classification par activité de maintenance avec les résultats obtenus.

Travail	Informations du commit exploitées	Méthode de classification	Jeu de données	Résultat
(Mockus & Votta 2000)	Message	Moteur de règles	~ 132 684 modifications	Précision ≈ 60%
(Amor et al. 2006)	Message	Un classificateur bayésien naïf	~ 500 commits (C uniquement)	Exactitude = 70%
(Hindle et al. 2008)	Message, auteur du commit et modules modifiés	SMO <sup>9</sup>	2000 commits	Exactitude = 53.27% Score F1 = 51%
(Levin & Yehudai 2016)	Message et code modifié	GLM <sup>10</sup>	2.5 M commits <sup>11</sup> (Java uniquement)	Score R <sup>2</sup> = 74.36%
(Levin & Yehudai 2017)	Message et code modifié	Forêts aléatoires	1151 commits (Java uniquement)	Exactitude = 76% Kappa = 63%
(Mariano et al. 2019)	Message, code modifié et métadonnées	XGBoost <sup>12</sup>	Le même que celui de Levin & Yehudai (2017)	Exactitude = 77.32% Kappa = 64.61%
(Mondal et al. 2019)	Message	LLDA <sup>13</sup>	1350 commits	Précision = 33% Score F1 <sub>micro</sub> = 45%
(Gharbi et al. 2019)	Message	Régression logistique multinomiale	29 604 commits	Score F1 = 45.79% Perte de Hamming = 0.05
(Hönel et al. 2020)	Message, code modifié et métadonnées	LogitBoost	Le même que celui de Levin & Yehudai (2017)	Exactitude = 89% Kappa = 82%
(Sarwar et al. 2020)	Message	Réseau de neurones profond	2037 commits	Score F1 <sub>micro</sub> = 87% Perte de Hamming = 0.11
(Ghadhab et al. 2021)	Message et code modifié	Réseau de neurones profond	1739 commits (Java uniquement)	Exactitude = 79.66% Score F1 <sub>macro</sub> = 79.6%
(Sristy & Schneider 2021)	Message	Ils ont proposé leur propre classificateur	5000 commits	Score F1 <sub>micro</sub> = 69%

<sup>9</sup> *Sequential Minimal Optimization*, par Platt (1998).

<sup>10</sup> *Generalized linear models*, par Nelder & Wedderburn (1972).

<sup>11</sup> Les commits sont étiquetés automatiquement par des jetons prédéfinis pour chaque classe.

<sup>12</sup> *eXtreme Gradient Boosting*, par Chen et al. (2015).

<sup>13</sup> *Labelled Latent Dirichlet Allocation*, par Ramage et al. (2009).

## 3.2 TRAVAUX RELIÉS

### 3.2.1 La classification par activité de maintenance

Mockus & Votta (2000) étaient les premiers à faire une classification basée sur le message du commit, et ils ont obtenu une précision de ~60% avec jeu de données d'environ 132 684 modifications archivées appartenant à un seul projet. Dans leur approche, ils se sont basés sur l'analyse de la fréquence d'occurrence des mots et sur des techniques de normalisation du texte pour déterminer les mots fréquemment utilisés dans chaque type d'activité de maintenance. Par exemple, si au moins l'un des mots « *fix*, *bug*, *error*, *fixup* ou *fail* » est présent dans le message, le commit est classé comme correctif, et si les mots de deux types maintenances sont présents, le commit sera classé avec le type ayant le plus de mots. Et en cas d'égalité, la priorité est donnée à la classe perfective puis corrective. Suite à cette classification, un nombre important de commits perfectifs ont été remarqués, et puis en analysant les relations entre le type d'activité de maintenance, la taille du commit et le temps nécessaire pour le réaliser (ou la difficulté des changements), il a été remarqué que les commits correctifs se caractérisent par l'ajout et la suppression de quelques lignes de code et que les commits perfectifs se particularisent par la suppression de plusieurs lignes de code. L'autre remarque qui a été faite également c'est que les commits correctifs nécessitent plus de temps pour les implémenter à cause de leur difficulté, la même chose pour ceux adaptatifs, mais juste dans le cas où les changements sont d'une taille importante.

Dans une autre étude différente de celle de Mockus & Votta (2000), Amor et al. (2006) ont suivi une méthodologie qui exploite aussi le message du commit, mais qui ne prend pas en considération l'occurrence de certains mots dans telle ou telle activité de maintenance. Pour cela, ils ont utilisé un classificateur bayésien naïf (Friedman et al. 1997). Ce dernier est basé sur le théorème de Bayes et part du principe de l'existence d'une forte indépendance entre les caractéristiques. Les auteurs ont appliqué leur approche d'une manière itérative sur 33335 commits du projet FreeBSD (un système d'exploitation UNIX) qui est développé en langage C, en commençant par classer manuellement 300 commits choisis



aléatoirement, la première itération de classification a rapporté une probabilité moyenne de 50% pour l'activité de maintenance la plus probable pour chacun des commits. Après deux itérations dans lesquels les 200 commits ayant des probabilités de classification basse (moins de 40%) ont été classifiés manuellement, la probabilité moyenne est passée à environ 65% et l'exactitude du modèle était de 70% pour un échantillon de commits choisis aléatoirement.

Hindle et al. (2008) se sont intéressés plus à l'analyse des commits de grande taille, et ont proposé une approche qui combine entre l'utilisation du message, l'auteur du commit et les modules modifiés dans le projet pour entraîner des algorithmes d'apprentissage automatique. Ils ont commencé par sélectionner 9 projets<sup>14</sup> à code source ouvert qui sont largement utilisés, couvrent différents domaines d'application et qui sont implémentés avec une variété de langages de programmation (C, C++, Java et PHP). Ensuite, ils ont classé manuellement environ 2000 commits selon une version étendue des trois classes (perfective, corrective et adaptative) définies par Swanson (1976) en ajoutant deux autres classes qui sont : « implémentation » qui concerne l'ajout d'une nouvelle fonctionnalité, et « non fonctionnel » qui concerne tout changement lié à la licence du système ou au SCV utilisé. Ensuite pour chaque projet, sept jeux de données ont été construits à partir de chacune des trois informations exploitées et de leurs combinaisons. La première information est la distribution des mots, c.-à-d., la fréquence d'occurrence de chaque mot dans le message. La deuxième est l'auteur du commit. Le choix de cette caractéristique vient de l'hypothèse que certains auteurs pourraient être plus susceptibles de créer certains types de commits. La troisième information est le type des modules et fichiers et vient sous forme de deux ensembles de caractéristiques. Le premier ensemble concerne le nombre de fichiers modifiés par module, c.-à-d., que pour un projet ayant N dossiers, N caractéristiques seront créées. Le deuxième ensemble comporte cinq caractéristiques qui concernent le nombre de fichiers modifiés en fonction de leur rôle (STBDO (Holt et al.)) : code source (*Source code*), test (*Test*), gestion des configurations (*Build*), documentation (*Documentation*), ou autres (*Other*). Finalement, chaque jeu de donnée sera entraîné avec plusieurs classificateurs

---

<sup>14</sup> *Boost, Egroupware, Enlightenment, Evolution, Firebird, MySQL, PostgreSQL, Samba et Spring Framework*

d'apprentissage automatique : **J48** : une implémentation optimisée de l'arbre de décision C4.5 (Quinlan 2014) sur l'outil WEKA<sup>15</sup> ; **NaiveBayes** (Friedman et al. 1997) : basé sur le théorème de Bayes et considérant une forte indépendance entre les caractéristiques ; **SMO** (*Sequential Minimal Optimization*) (Platt 1998) : basé sur une machine à vecteurs de support (SVM – *Support Vector Machines*, Cortes & Vapnik 1995), il fait en sorte d'augmenter la dimensionnalité des données jusqu'à ce que les caractéristiques sont différenciables dans une certaine dimension ; **Kstar** (Cleary & Trigg 1995) : basé sur l'algorithme des k plus proche voisins, et peut être utilisé avec différentes métriques de distance comme la distance de Mahalanobis ; **Ibk** (Aha et al. 1991) : basé sur l'algorithme des k plus proches voisins (avec k=1), il classe un exemple en prenant la classe du vecteur associé le plus proche dans l'ensemble d'apprentissage via l'une des métriques de distance ; **Jrip** : basé sur une méthode d'apprentissage des règles appelée « RIPPER<sup>16</sup> » (Cohen 1995), il crée une liste ordonnée de règles pour classer les exemples. Les résultats de test ont montré d'une part qu'aucun des classificateurs n'a pu se différencier par rapport aux autres en termes de performance, et d'autre part que les jeux de données basés uniquement sur le message offrent de meilleurs résultats que ceux obtenus avec les jeux de données basés sur l'un des autres caractéristiques. Cependant, comme le montre le tableau 3.2, la combinaison du message du commit et l'auteur a permis d'avoir le meilleur résultat avec un score F1 de 0.51. Il a été remarqué également que les projets qui ont permis d'avoir les meilleurs résultats, se caractérisent par leur adoption d'une terminologie claire pour décrire les opérations touchant le système, et par leur formulation du message du commit suivant cette terminologie.

---

<sup>15</sup> <https://www.cs.waikato.ac.nz/ml/weka/>

<sup>16</sup> Repeated Incremental Pruning to Produce Error Reduction

Tableau 3.2 – Le meilleur classificateur (moyenne des projets) dans chaque jeu de données.

Jeux de données	Classificateur	Exactitude	Score F1	ROC
Message	SMO	52.07	0.50	0.68
Auteur et Modules	J48	50.84	0.47	0.63
Auteur et message	SMO	53.27	0.51	0.70
Auteur, message et modules	Jrip	53.51	0.47	0.62
Message et modules	J48	52.59	0.50	0.66
Auteur	SMO	51.27	0.45	0.63
Modules	J48	51.55	0.48	0.63

Par la suite, Levin & Yehudai (2017) ont fait une comparaison entre trois modèles, le premier se basant sur le code modifié, le deuxième sur la fréquence d’occurrence de certains mots dans le message du commit, et le troisième qui combine les caractéristiques des deux premiers modèles. Contrairement aux études précédentes (Mockus & Votta 2000, Amor et al. 2006, Hindle et al. 2008), l’enjeu principal de l’étude était d’améliorer les résultats trouvés, tout en utilisant un jeu de données constitué de commits provenant de plusieurs projets. Dans leur approche, 11 projets<sup>17</sup> à code source ouvert ont été sélectionnés à partir de GitHub suivant quelques critères (le langage de programmation Java, la taille du projet d’au moins 2MB, cloné par au moins 60 utilisateurs...). À partir de ces projets, 1151 commits (environ 100 par projet) seront classifiés manuellement selon les types d’activités de maintenance définis par Swanson (1976) : corrective, perfective et adaptative. Pour le modèle basé sur l’analyse du message du commit, les caractéristiques ont été extraites à partir des messages, et sont les 20 mots les plus fréquemment utilisés dans chacune des activités de maintenance. Concernant le modèle basé sur le code modifié, 48 caractéristiques sont extraites en utilisant l’outil ChangeDistiller (Fluri et al. 2007, Gall et al. 2009). Ce dernier prend en entrée deux révisions consécutives pour chaque fichier modifié puis génère 48 types

<sup>17</sup> RxJava, IntelliJ Community Edition, HBase, Drools, Kotlin, Hadoop, Elasticsearch, Restlet, OrientDB, Camel et Spring Framework

d'éditions définis selon la taxonomie de Fluri pour les langages de programmation orientés objet (Fluri & Gall 2006). Parmi ces 48 types, on trouve : « *statement\_delete* », « *statement\_insert* », « *removed\_class* », « *additional\_class* », etc. Une fois toutes les caractéristiques ont été extraites, l'entraînement des modèles s'est fait en utilisant trois algorithmes de classification qui sont les forêts aléatoires (RF – *Random Forest*, Breiman 2001), l'arbre de décision J48 et le GBM (*Gradient Boosting Machine*, Friedman 2001). Suivant une notion appelée modèle composé (Saar-Tsechansky & Provost 2007) lors de l'entraînement, le meilleur résultat obtenu était de 76% en exactitude et de 63% en Kappa, et ce, en utilisant le classificateur RF avec le modèle composé qui utilise un modèle prenant en entrée les caractéristiques du message si le message du commit comporte au moins l'un des 20 mots prédéfinis, sinon il utilise un modèle qui prend entrée la combinaison des caractéristiques de deux données : message et code modifié.

Mariano et al. (2019) ont proposé une amélioration à l'étude de Levin & Yehudai (2017), qu'on vient juste de présenter, en testant trois nouvelles caractéristiques concernant le code modifié. Ces métadonnées, extraites en utilisant l'outil GraphQL API<sup>18</sup> de GitHub, sont le nombre de LOC ajoutées, de celles supprimées et le nombre de fichiers modifiés. Dans leur approche, ils se sont basés sur le même jeu de données et ont suivi le même processus de Levin & Yehudai (2017) pour la représentation du code modifié et du message du commit. Pour le classificateur, ils ont testé également le RF ainsi que XGBoost (Chen et al. 2015) au lieu de l'arbre de décision J48 et GBM, vu qu'il surpasse les deux en termes de performance (Chen & Guestrin 2015, Chen & Guestrin 2016). Au final, les résultats obtenus ont montré une amélioration par rapport à l'étude de base en atteignant 77.32% en exactitude et 64.61% en Kappa, et ce, en utilisant le RF avec les caractéristiques concernant le message, le code modifié et le nombre de LOC ajoutées et supprimées.

Suivant la même voie, Hönel et al. (2020) ont démontré à travers leur étude l'importance d'une nouvelle notion qui est la densité du commit. Ils ont défini cette dernière

---

<sup>18</sup> <https://docs.github.com/en/graphql>

comme étant le ratio entre la taille nette et brute d'un commit, sachant que la taille d'un commit est le nombre de fichiers ou de LOC modifiés, la taille brute est la taille sans tenir compte du fait qu'une ligne affecte ou non une fonctionnalité, tandis que la taille nette est la taille brute moins le nombre de fichiers ou de LOC modifiées qui n'ont pas contribué à la modification d'une fonctionnalité. Leur travail suit la même approche de Levin & Yehudai (2017) et utilise le même jeu de données avec l'ajout de 22 caractéristiques qui concernent la taille brute et nette d'un commit. Dans une expérience intéressante ayant comme objectif de déterminer les caractéristiques qui contribueront le plus durant la classification, la méthode d'élimination récursive des caractéristiques (RFE – *Recursive Feature Elimination*, en anglais) a été appliquée sur le nouveau jeu de données comportant les caractéristiques des trois types de données : le message, le code modifié et la taille du commit. Le résultat de cette expérience a montré que parmi les dix caractéristiques les plus pertinentes : trois concernent la taille (parmi 22), quatre liés aux messages (parmi 20) et trois liée aux changements de code (parmi 48). Concernant le processus de classification, ils ont suivi la même démarche de Levin & Yehudai (2017) en faisant différentes combinaisons de caractéristiques suivant la notion du modèle composé, et en testant plusieurs classificateurs : **SVM** (*Support Vector Machine* – Cortes & Vapnik 1995), **LogitBoost** (Friedman et al. 2000), **avNNet** (*Averaged Neural Network*), **GBM** (*Gradient Boosting Machine* – Friedman 2001), **xgbTree** (*eXtreme Gradient Boosting Tree* – Chen et al. 2015), **LDA** (*Linear Discriminant Analysis* – Mika et al. 1999), **MDA** (*Mixture Discriminant Analysis* – Hastie & Tibshirani 1996), **l'arbre de décision C5.0** et **naïf bayésien** (Friedman et al. 1997). Au final, le meilleur résultat était de 89% en exactitude et 82% en Kappa, et ce, en utilisant le classificateur LogitBoost avec le modèle composé qui utilise un modèle prenant en entrée toutes les caractéristiques si le message du commit comporte au moins l'un des 20 mots prédéfinis sinon il utilise un modèle qui prend en entrée que les 20 caractéristiques du message. Ce résultat montre une amélioration par rapport à l'étude de base (Levin & Yehudai 2017), en notant également un changement significatif dans les commits ayant une grande densité passant de la classe correctrice vers celle perfective.

Dans une autre étude, Gharbi et al. (2019) ont exploité le message du commit sur lequel ils ont appliqué des techniques de TALN comme la suppression des mots vides, la racinisation et la lemmatisation. Ensuite, ils ont extrait les 50 mots les plus importants à partir de ces messages en utilisant la méthode TF-IDF (*Term frequency–inverse document frequency*), pour finalement représenter chaque commit sous forme d’un vecteur terme-fréquence (*term-frequency vector*, en anglais). Cette phase de prétraitement leur a permis de supprimer, par exemple, certains mots qui sont liés soit aux domaines d’application des projets sélectionnés, soit aux informations concernant les auteurs des commits. Ayant sélectionnés un grand nombre de commits, 29 604 à partir de 12 projets<sup>19</sup> développés avec différents langages de programmation (Java, Python, PHP, Ruby, C++, C, et C#), et vu le temps important qu’il faudrait pour les étiqueter, ils ont choisi d’adopter une approche d’apprentissage actif pour leur modèle. Cette approche commence par l’étiquetage manuel d’une petite partie du jeu de données (1200 commits) appelée « *seeds* » qui représente l’ensemble d’entraînement initial. Ensuite, ils ont choisi la régression logistique multinomiale comme classificateur et le MLALAL<sup>20</sup> comme stratégie de requête utilisée à chaque itération du processus de l’apprentissage actif. Lors de chaque itération, cette stratégie de requête interroge un oracle pour étiqueter un sous-ensemble de commit non étiquetés encore, ensuite ce sous-ensemble sera ajouté à l’ensemble d’entraînement et le modèle sera entraîné à nouveau puis tester sur l’ensemble de test. Le meilleur résultat obtenu suivant cette approche est de 0.05 en perte de Hamming et de 45.79% en score F1.

Parmi les premières études à utiliser les modèles de langues pré-entraînés pour l’extraction des caractéristiques du message du commit, on trouve Zafar et al. (2019) qui se sont focalisés uniquement sur la détection des changements correctifs, et Sarwar et al. (2020) qui ont étudié les trois types d’activité de maintenance. Dans cette dernière étude, les auteurs ont commencé par sélectionner 2037 commits, parmi les 100 meilleurs projets dans chacun

---

<sup>19</sup> *geocaching, anuto, adam.aslfms, filemanager, android.reddit, android.keeppass, faircode.netguard, fastaccess.github, ccrama.redditslide, moztumbler, keychain et org.videolan.vlc.*

<sup>20</sup> Multi-label Active Learning with Auxiliary Learner

des 17 langages de programmation les plus utilisés sur GitHub. Ces commits qu’ils ont étiquetés manuellement selon les définitions des classes proposées par Swanson (1976) en plus d’avoir mis en place un ensemble de règles à suivre pour certains cas de changement de code (voir tableau 3.3). Ensuite, ils ont choisi le modèle DistilBERT (Sanh et al. 2019) qui est une variante légère du modèle BERT. La différence entre les deux, c’est que DistilBERT possède 40% de paramètres en moins, s’exécute 60% plus rapidement et préserve 97% des performances de BERT. Pour effectuer la classification, ils ont réentraîné DistilBERT en lui ajoutant un classificateur d’une couche qui prend en entrée le vecteur de plongement généré. Au final, le meilleur résultat obtenu était de 87% en score  $F1_{\text{micro}}$  avec une perte de Hamming de 0.11, et ce, avec les valeurs d’hyperparamètres suivants : le nombre d’époques = 4, le taux d’apprentissage =  $2^{-05}$ , la longueur maximale du message = 512 et la taille du lot = 8. Ce même modèle a été testé sur deux autres jeux de données publiquement disponibles, et les résultats étaient comme suit : un score  $F1_{\text{micro}}$  de 71% avec une perte de Hamming de 0.22 pour le jeu de données de Gharbi et al. (2019) et un score  $F1_{\text{micro}} = 50\%$  avec une perte de Hamming de 0.31 pour le jeu de données de Mauczka et al. (2015).

Tableau 3.3 – Les règles définies par Sarwar et al. (2020) pour l’étiquetage du jeu de données

Perfectif	Le message contient le tag ‘perfective’.
	Une amélioration des performances du système.
	Une suppression d’une fonction, une variable ou un fichier qui est dupliqué ou qui n’est plus utilisé.
	Le réusinage du code source, comme l’utilisation d’une convention de dénomination ou la réorganisation du code.
	La simplification d’une partie du code responsable de la gestion d’erreurs.
	Toute amélioration apportée à l’interface utilisateur, à la gestion de la mémoire ou à la sécurité du système.
Correctif	Le message contient uniquement la phrase ‘‘Fix Issue #[NUM]’’
	Une correction des coquilles dans le code. La correction de celles qui concernent la documentation n’entre pas dans le cadre des changements correctifs.

	La correction d'une erreur de compilation
	Une correction des problèmes non esthétiques de l'interface graphique.
	Une correction des problèmes liés à une mauvaise allocation de la mémoire.
Adaptatif	Le message contient le terme "feat()".
	Le réusinage du code afin de l'adapter à un nouvel environnement.
	Le support d'une nouvelle technologie, un outil ou une librairie, etc. Par exemple, un message disant : « <i>add core platform support for android devices</i> ».
	Ajout d'une fonctionnalité, d'un cas de test ou API.
	Toute modification nécessaire pour faire fonctionner un nouvel API.

Suivant la même méthode, Ghadhab et al. (2021) ont utilisé le modèle BERT pour générer un vecteur représentant les caractéristiques du message du commit. En plus du message, le code modifié a été exploité également en utilisant trois outils, appelés distillateurs, qui vont générer 70 caractéristiques décrivant quelques types d'édérations du code (voir figure 3.1). Ces distillateurs qui supportent uniquement du code en Java sont : l'outil RefactoringMiner (Tsantalis et al. 2018) qui détecte les opérations de réusinage du code, l'outil FixMiner (Koyuncu et al. 2020) qui détecte si un patron de fixation de bogue est présent dans les changements apportés, et l'outil ChangeDistiller (Fluri et al. 2007) utilisé auparavant par Levin & Yehudai (2017) qui génère les 48 types d'édérations selon la taxonomie de Fluri pour les langages de programmation orientés objet. Dans leur approche, un jeu de données de 1793 commits collectés à partir de trois jeux de données (Mauczka et al. 2015, Levin & Yehudai 2017, AlOmar et al. 2019) couvrant plusieurs projets développés en Java, sera utilisé pour évaluer sept modèles sous forme de réseaux de neurones profond (DNN) qui sont : (1) DNN@BERT : il prend en entrée le vecteur de plongement généré par BERT, (2) DNN@BERT+Refact\_cc : il prend en entrée le vecteur de plongement en plus des caractéristiques générées par RefactoringMiner, (3) DNN@BERT+CD\_cc : il prend en entrée le vecteur de plongement en plus des caractéristiques générées par ChangeDistiller, (4) DNN@BERT+Fix\_cc : il prend en entrée le vecteur de plongement en plus des caractéristiques générées par FixMiner, (5) DNN@BERT+All\_cc : il prend en entrée toutes



les caractéristiques, et finalement (6) DNN@KW+CD\_cc : il prend en entrée les caractéristiques générées par ChangeDistiller en plus d'un vecteur « *multi-hot* » représentant l'occurrence d'une liste de mots qui a été extraite à partir de tous les messages. Au final, le modèle (4) a permis d'obtenir le meilleur résultat avec une exactitude de 79,66% et un score  $F1_{macro}$  de 79,6%.

ChangeDistiller code changes
Adding Attribute Modifiability(7)/Method Overidability(6)
Additional Class(237)/Functionality(237)/Object State(751)
Alternative Part Delete(145)/Insert(238)
Attribute(82)/Method(148)/Class Renaming(4)
Attribute Type Change (85)
Comment Delete(218)/Insert(294)/Move(109)/Update(48)
Decreasing(103)/Increasing(158) Accessibility Change
Doc Delete(62)/Insert(177)/Update(204)
Parameter Delete(130)/Insert(218)/Ordering Change(16)/Renaming(67)/Type Change(106)
Parent Class Change(45)/Delete(9)/Insert(23)
Parent Interface Change(1)/Insert(45)/Delete(30)
Removed Functionality(347)/Class(255)/Object State(255)
Removing Attribute Modifiability(19)/Class Derivability(5)/ Method Overridability(9)
Return Type Change(141)/Delete(81)/Insert(11)
Condition Expression Change(388)
Statement Delete(730)/Insert(921)/Ordering Change(199)/ Parent Change(458)/Update(845)
RefactoringMiner code changes
Extract Method(153)/Class(24)/Subclass(5)/Superclass(33)/Interface(10)/variable(87)
Extract and move method(40)
Inline Method(23)/Variable(9)
Rename Variable(116)
Move Method(62)/Attribute(39)/Class(5)/Source folder(7)
Pull Up Method(33)/Attribute(25)
Push Down Method(8)/Attribute(6)
Change Package(12)
Parametrize Variable(17)
Move and rename Class(18)
Replace Variable with attribute(16)
Change variable type(66)
Fixminer code changes
Code patches(123)

Figure 3.1 – Les caractéristiques extraites à partir des changements de code.  
*Source: Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. (Ghadhab et al. 2021)*

### 3.2.2 La représentation des changements de code

Dans sa forme naturelle, le code source ne peut pas être traité directement par un modèle d'apprentissage automatique ou profond. Il est donc nécessaire de le transformer en une représentation intermédiaire avant de procéder à son exploitation. Les études faites à ce sujet soulignent que le code source est très riche en termes de sa structure, et donc ne doit pas être traité simplement comme une liste de jetons ou un texte (Allamanis et al. 2018). De ce fait, on trouve plusieurs techniques qui s'intéressent souvent à trois aspects du code qui sont la syntaxe, la structure et la sémantique. D'ailleurs, un exemple des méthodes les plus fréquemment utilisées est l'arbre de la syntaxe abstraite (AST- *Abstract Syntax Tree*, en anglais) qui permet de représenter un code dans un langage de programmation sous forme d'une structure arborescente hiérarchique (voir la figure 3.2). En effet, les études utilisant cette technique s'intéressent plus à l'analyse d'une fonction, d'une classe ou de la totalité d'un fichier de code parce que son utilisation nécessite que le code soit analysable (*parsable*, en anglais) (Hoang et al. 2020). Ce qui n'est pas le cas quand on fait l'analyse des changements de code, car on parle plutôt d'un code qui est sous forme d'ajouts et suppressions touchant des LOC dans différentes parties d'un ou plusieurs fichiers. De ce fait, deux études (PatchNet (Hoang et al. 2020) et CC2Vec (Hoang et al. 2019)), que nous détaillerons dans cette section, ont proposé des approches où les changements de code sont encodé d'abord sous un format (matrice 3D) prenant en considération la structure hiérarchique des lignes ajoutées et supprimées. Ensuite, à partir de ce format, les caractéristiques des changements seront extraites en utilisant un réseau de neurones (CNN ou RNN). Nous présenterons plus en détail le modèle CC2Vec, d'abord vu que son fonctionnement n'est pas limité par la contrainte du langage de programmation du code modifié, et puis parce qu'il s'agit d'un modèle qui n'est pas spécifique à une tâche précise, mais qui a comme objectif principal l'extraction des caractéristiques, comme c'est le cas des modèles de langue pour le langage naturel.

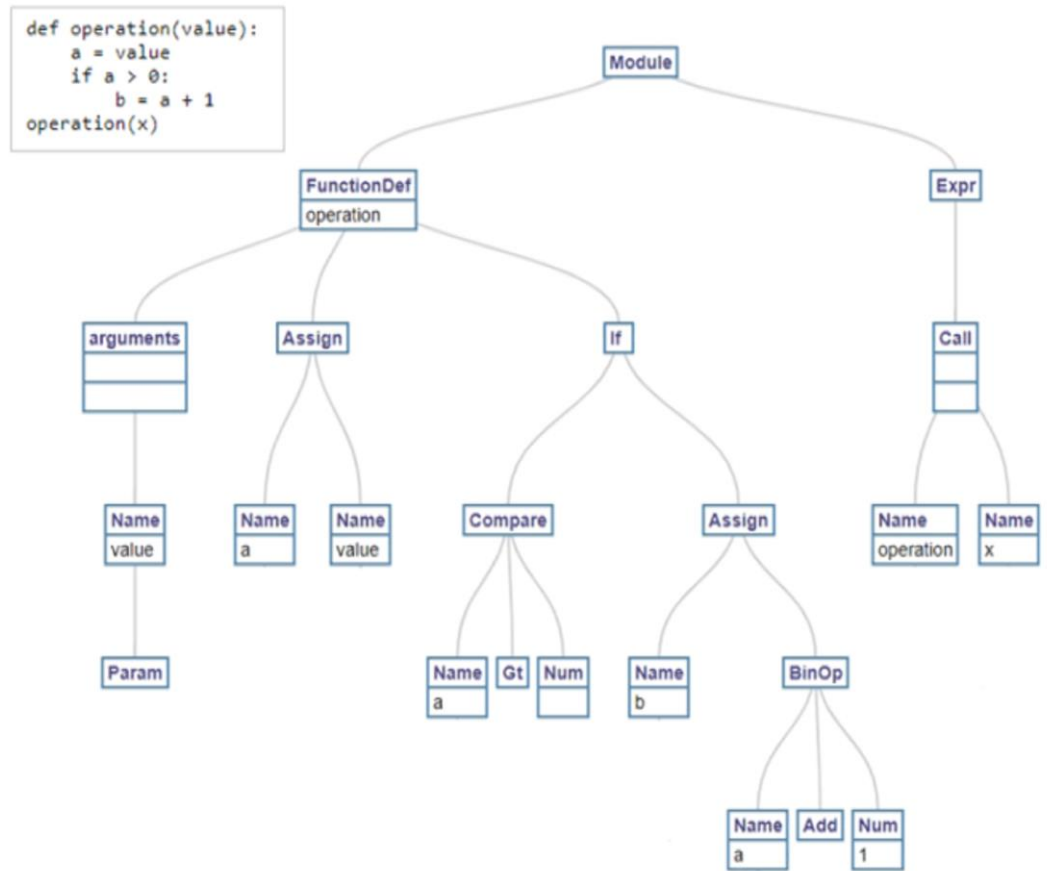


Figure 3.2 – Un exemple de l’arbre de la syntaxe abstraite d’une fonction en Python.  
(Schéma généré avec l’outil : Python AST Visualizer<sup>21</sup>)

Hoang et al. (2019) ont proposé PatchNet qui apprend la représentation sémantique des correctifs (en langage C) en déterminant s’ils sont stables ou non. Leur approche exploite le message du commit et le code modifié, qui seront tous les deux représentés sous forme de vecteurs de plongement puis concaténés formant ainsi un vecteur global du correctif. Concernant l’exploitation du message, elle est faite en utilisant quelques techniques de TALN comme la racinisation et l’élimination des mots vides, et en encodant le message sous

<sup>21</sup> <https://vpyast.appspot.com/>

la forme d'une matrice à deux dimensions, c.-à-d., en visualisant le message comme une séquence de vecteurs où chaque vecteur représente un mot apparaissant dans le message. Ensuite, ils ont utilisé une couche convolutive suivie d'une opération de max-pooling pour extraire les caractéristiques les plus importantes du message. Concernant les changements de code, c'est un autre module de PatchNet qui prend en entrée deux matrices à trois dimensions dont l'une représente les ajouts de code et l'autre représente les suppressions. Les dimensions de ces matrices font référence au nombre de « hunks »<sup>22</sup> dans un fichier, au nombre de lignes dans chaque « hunk » et au nombre de mots dans chaque ligne. Ensuite, chacune des deux matrices sera l'entrée d'un 3D-CNN (*Three-dimensional Convolutional Neural Network*, en anglais) qui, pareil au module traitant le message, générera les caractéristiques du code modifié sous forme d'un vecteur de plongement. Finalement, c'est le module de classification constitué de couches entièrement connectées qui va prendre en entrée la concaténation des deux vecteurs de plongements, et déterminera si le correctif est stable. Même s'il s'agit d'une tâche différente de celle que nous étudions, ce qui est à retenir des résultats obtenus c'est que la performance du modèle a diminué de 15-20% quand la classification est faite uniquement en se basant sur le message. Ce qui montre l'efficacité du processus suivie pour l'extraction des caractéristiques du code modifié.

L'étude qui nous intéresse le plus est celle de Hoang et al. (2020) dans laquelle ils ont proposé, CC2Vec, un modèle de réseau de neurones qui apprend la représentation distribuée des changements de code, en prédisant les mots qui sont présents dans le message du commit. Comme on peut voir à travers la figure 3.3, chaque fichier comportant normalement des ajouts et suppressions de code sera d'abord encodé sous forme de deux matrices à trois dimensions (comme dans PatchNet). Ensuite, chaque matrice sera l'entrée d'un HAN (*hierarchical attention network*, en anglais) qui va générer le vecteur de plongement du code ajouté ( $e_a$ ) ou supprimé ( $e_s$ ) dans un fichier. L'architecture du HAN se compose de trois parties constituées de manière hiérarchique : un encodeur de mots, un encodeur de la ligne et

---

<sup>22</sup> « *Hunk* » : un ensemble regroupant des LOC ajoutées (ou supprimées) qui sont successives.

un encodeur de « hunk ». Et la fin de chacune de ces parties, on trouve une couche d'attention qui permet donner plus d'importance aux mots (lignes ou « hunks ») contribuant le plus à l'amélioration de la fonction objective du modèle CC2Vec.

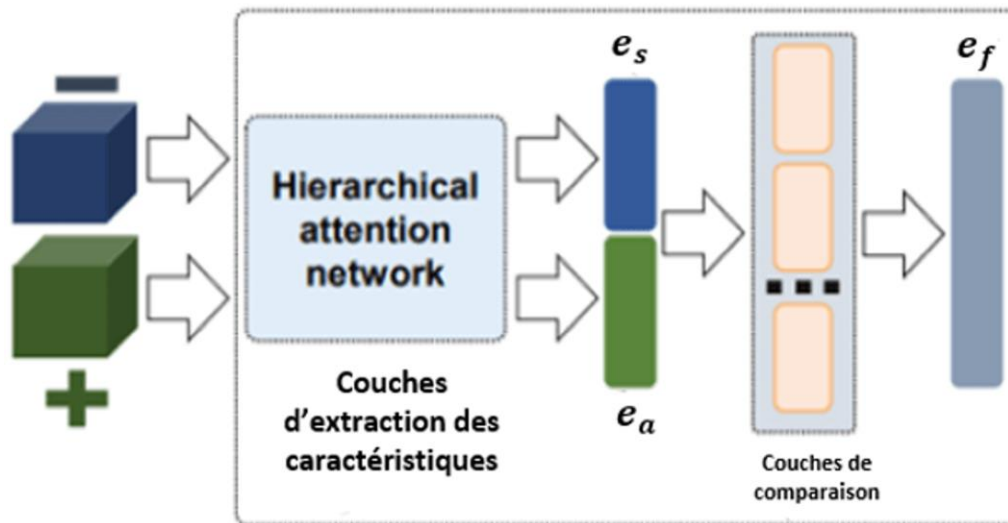


Figure 3.3 – Processus de génération du vecteur de plongement des changements dans un fichier. *Source : Universal Sentences Encoder. (Hoang et al. 2020)*

Plus précisément, et comme c'est expliqué dans leur article (Hoang et al. 2020), supposons que le code supprimé (ou ajouté) dans un fichier contient une séquence de « hunks »  $H = [t_1, t_2, \dots, t_{\mathcal{T}}]$ , chaque « hunk »  $t_i$  se compose d'une suite de lignes  $[s_{i1}, s_{i2}, \dots, s_{iL}]$ , et chaque ligne  $s_{ij}$  se compose d'une suite de mots  $[w_{ij1}, w_{ij2}, \dots, w_{ijW}]$ , où  $w_{ijk}$  avec  $k \in [1, W]$  représente le mot de la  $j$ -ième ligne du  $i$ -ième « hunk ». Comme on peut voir sur la figure 3.4, la hiérarchie du HAN commence au niveau de l'encodeur de mots qui prend en entrée une ligne  $s_{ij}$  sous forme d'une séquence de mots  $w_{ijk}$ , ces mots dont la représentation est retournée à partir d'une matrice de plongement de mots  $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}^c| \times d}$ , où

$\mathcal{V}^c$  est le vocabulaire contenant tous les mots qui se trouvent dans les changements de code et  $d$  est la dimension de la représentation du mot. Cette représentation de chaque mot d'une ligne de code est donnée comme suit :

$$\overline{w_{ijk}} = \mathbf{W}[w_{ijk}] \quad (23)$$

Où  $\overline{w_{ijk}} \in \mathbb{R}^d$  indique la représentation vectorielle du mot  $w_{ijk}$  dans la matrice de plongement  $\mathbf{W}$ . Une fois la représentation de chaque mot constituant une ligne de code est faite, c'est un RNN de type GRU bidirectionnel (24) qui va prendre entrée la séquence de mots et va apprendre des informations sur le contexte d'utilisation de chaque mot. Pour capturer ces informations, une partie du GRU bidirectionnel lit la ligne  $s_{ij}$  de  $w_{ij1}$  à  $w_{ijW}$  et l'autre partie lit la ligne  $s_{ij}$  de  $w_{ijW}$  à  $w_{ij1}$ .

$$\begin{aligned} \overrightarrow{h_{ijk}} &= \overrightarrow{GRU}(\overline{w_{ijk}}), k \in [1, W] \\ \overleftarrow{h_{ijk}} &= \overleftarrow{GRU}(\overline{w_{ijk}}), k \in [W, 1] \end{aligned} \quad (24)$$

Ainsi, la nouvelle représentation d'un mot  $w_{ijk}$  sera nommée  $h_{ijk}$ , qui est tout simplement la concaténation des deux états cachés du GRU bidirectionnel ( $\overrightarrow{h_{ijk}}$  et  $\overleftarrow{h_{ijk}}$ ). En sortant de l'encodeur de mots, c'est au tour du mécanisme d'attention. Dans ce dernier,  $h_{ijk}$  passera à travers une couche entièrement connectée ( $\mathcal{W}_w$ ) afin d'obtenir sa représentation cachée  $u_{ijk}$ , et ce suivant :

$$u_{ijk} = \text{ReLU}(\mathcal{W}_w h_{ijk} + b_w) \quad (25)$$

Ensuite, un vecteur de contexte de mot ( $u_w$ ) est créé puis initialisé de manière aléatoire. Ce vecteur est appris pendant l'entraînement du modèle, et permet d'attribuer plus d'importance à certains mots à travers un poids normalisé  $\alpha_{ijk}$  calculé comme suit :

$$\alpha_{ijk} = \frac{\exp(u_{ijk}^T u_w)}{\sum_k \exp(u_{ijk}^T u_w)} \quad (26)$$

Finalement, le vecteur de plongement de chaque ligne de code  $s_{ij}$  est calculé comme étant la somme pondérée des vecteurs de plongement de chaque mot, et ce en fonction de leur importance :

$$\overline{s}_{ij} = \sum_k \alpha_{ijk} h_{ijk} \quad (27)$$

Suivant la même logique (voir figure 3.4), l'encodeur de phrase prendra en entrée les vecteurs de plongement de chaque ligne de code qui constituent un « hunk », ensuite l'encodeur de « hunk » prendra en entrée les vecteurs de plongements de chaque « hunk » du fichier et génèrera au final le vecteur de plongement ( $e$ ) qui est la représentation distribuée du code ajouté (ou supprimé) dans un fichier.

Ce processus qu'on vient de détailler sera appliqué sur chacun des fichiers modifiés, et leurs vecteurs résultants seront concaténés pour former un seul vecteur pour le code ajouté (ou supprimé) dans un commit. Donc, en ayant deux vecteurs de plongement, l'étape suivante est de les fusionner pour en avoir qu'un seul vecteur final, cette fusion se fait à travers une couche de comparaison qui comporte cinq fonctions : NT (*Neural Tensor Network*), NN (*Neural Network*), la somme directe des deux matrices représentant la similarité cosinus et la distance euclidienne, la soustraction, et la multiplication. Chacune de ces fonctions prend en entrée les deux vecteurs et génère en sortie un vecteur ayant les caractéristiques qui relient le

code ajouté à celui supprimé. Au final, les cinq vecteurs résultants seront concaténés pour former le vecteur de plongement d'un commit.

Par la suite, ce vecteur final sera l'entrée d'une couche entièrement connectée, suivie d'une couche de sortie avec une fonction d'activation sigmoïde pour obtenir la distribution de probabilité sur une liste de mots précédemment extraits à partir de tous les messages des commits. Durant ce processus d'entraînement du modèle CC2Vec, les paramètres optimisés sont : la matrice de plongement située au niveau de l'encodeur de mots, les états cachés dans les trois encodeurs, les trois vecteurs de contexte, et les poids et biais du : NT, NN, la couche cachée et la couche de prédiction des mots (Hoang et al. 2020).

Concernant les performances de cette approche, les auteurs ont entraîné CC2Vec avec le jeu de données de l'étude de Hoang et al. (2019) (PatchNet), puis ils ont généré les vecteurs de plongements de chaque correctif. Ensuite, ils ont entraîné PatchNet en incluant au niveau du module de classification, le vecteur généré par CC2Vec avec les deux autres vecteurs de PatchNet. L'ajout de cette information a permis d'améliorer les résultats obtenus dans PatchNet de 5.22%, 9.18%, 4.37% et 6.51% en termes d'exactitude, précision, score F1 et AUC (*Area Under the Curve*). Donc, ceci veut dire que le modèle CC2Vec permet d'extraire des représentations qui peuvent être utiles même pour des tâches qui sont différentes de celle sur laquelle il est entraîné.



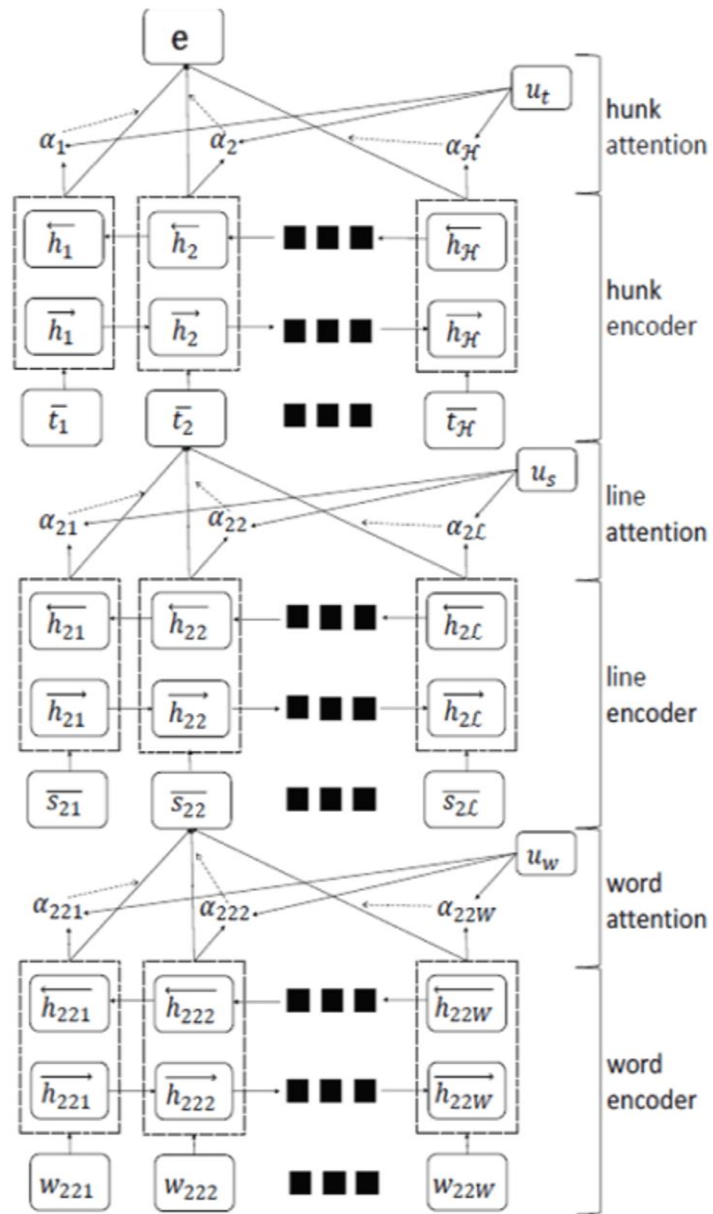


Figure 3.4 – L'architecture du HAN.

Source : *Universal Sentences Encoder. (Hoang et al. 2020)*

### 3.3 SYNTHÈSE

L'information qui a été exploitée principalement dans la classification par activité de maintenance est le message du commit que chaque étude a choisi une manière de l'encoder. Par exemple, pour les études les plus récentes, les plongements générés par les modèles de langues étaient les plus utilisés du fait de leur performance dans différents domaines d'application. Il a été remarqué également que l'ajout de quelques informations qui caractérisent chaque type d'activité de maintenance a permis une amélioration des résultats. Concernant l'exploitation du code modifié, les rares études utilisant cette donnée se sont limitées à l'utilisation d'outils qui extraient quelques types d'édition (selon la taxonomie de Fluri (Fluri & Gall 2006), par exemple). Dans leur fonctionnement, ces outils reposent sur l'AST qui est à la base spécifique à un langage de programmation précis (Java, pour la majorité). C'est ce qui explique d'ailleurs pourquoi les études se limitent à un jeu de données qui contient des commits appartenant à des projets implémentés uniquement en Java. À la fin du chapitre, on a vu comment l'approche suivie par CC2Vec conduit à la génération d'un vecteur de plongement du code modifié, indépendamment du langage de programmation. On peut donc envisager l'utilisation de ce genre de représentation qui, à notre connaissance, n'a pas encore été expérimenté dans le cadre de la classification par activités de maintenance.



## CHAPITRE 4

### L'ÉTUDE PROPOSÉE

Nous présenterons dans ce chapitre la démarche suivie. En effet, nous commencerons par expliquer nos choix sur le plan des architectures de modèle que nous allons tester, ainsi que le processus suivi. Ensuite, nous aborderons la phase de constitution de notre jeu de données, en commençant par le choix des projets et en passant par l'étiquetage manuel, jusqu'à en arriver à l'analyse de la distribution des classes. Et pour finir, nous présenterons les résultats trouvés ainsi que les conclusions et les perspectives à venir.

#### 4.1 PRÉSENTATION DE L'APPROCHE

L'approche développée dans cette étude consiste à analyser les commits afin de déterminer à quel(s) type(s) d'activité de maintenance appartiennent les changements de code qui y sont effectués. Cette analyse passe à travers les informations de chacun des commits, à savoir : le message du commit, le titre de la proposition d'un changement, les fichiers modifiés, le code modifié, les discussions entre les contributeurs quand il s'agit d'un *pull-request*. Etc. Toutes ces informations peuvent être divisées en deux catégories : des données textuelles en langage naturel et des données sous forme de code en langage de programmation. Avant de procéder à la classification, l'enjeu majeur est de pouvoir trouver une représentation de ces deux catégories de données de telle sorte que leurs caractéristiques les plus importantes ne soient pas perdues. De ce fait, nous avons choisi de baser notre approche sur le concept des plongements à travers l'application de l'apprentissage par transfert, que ce soit pour les données textuelles en utilisant le modèle pré-entraînée USE, ou pour les données sous forme de code en entraînant le modèle CC2Vec. En plus de ces deux vecteurs de plongements qui seront générés, nous avons également décidé d'enrichir notre représentation finale du commit par des métadonnées extraites à partir du code modifié.

Comme discuté dans le chapitre 3, toutes les études se basent principalement sur le message du commit, puis essayent d'ajouter soit des caractéristiques décrivant quelques types d'édits (par exemple : la taxonomie de Fluri (Fluri & Gall 2006)), soit des valeurs quantitatives liés au code modifié comme le nombre de fichiers modifiés, le nombre de LOC ajoutées/supprimées. Par conséquent, nous avons bien évidemment considéré le modèle basé sur le message du commit (ou de manière générale sur des données textuelles) comme modèle de base puis nous allons voir si l'ajout de nouvelles informations va nous permettre d'atteindre un meilleur résultat. De ce fait, nous avons choisi de tester ces quatre modèles :

- **Modèle 1 :** C'est notre modèle de base. Il s'agit du modèle pré-entraîné USE auquel nous ajoutons un classificateur dont la couche finale est sous forme de 3 neurones avec une fonction d'activation sigmoïde. Ce modèle prend en entrée des données textuelles et génère un vecteur de plongement qui sera l'entrée du classificateur.
- **Modèle 2 :** Il a la même architecture que « le modèle 1 » en plus de trois caractéristiques qui seront concaténées avec le vecteur de plongement généré par le modèle USE. Ces nouvelles caractéristiques sont : le nombre de fichiers modifiés et le nombre de LOC ajoutées/supprimées.
- **Modèle 3 :** Il a la même architecture que « le modèle 1 » avec une concaténation du vecteur généré par le modèle USE avec le vecteur de plongement représentant le code modifié, ce vecteur sera généré en utilisant le modèle CC2Vec.
- **Modèle 4 :** Il a la même architecture que « le modèle 1 » et regroupent toutes les caractéristiques mentionnées dans les modèles 1, 2 et 3.

La figure 4.1 ci-dessous est une représentation de l'architecture proposée qui va inclure les entrées représentées en bleu en fonction de ce qui a été définie pour chacun des quatre modèles.

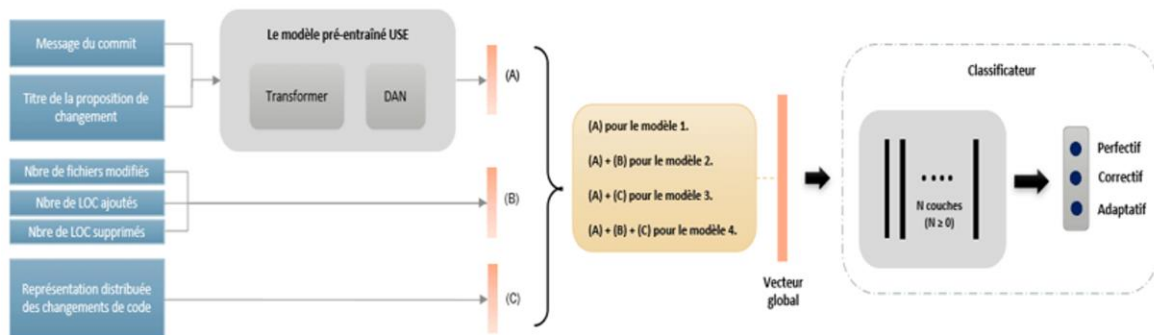


Figure 4.1 – Une représentation de l'architecture utilisée pour tester nos 4 modèles.

Pour le processus qui sera suivi durant cette approche, et comme on peut voir à travers la figure 4.2, il commence par plusieurs entraînements de notre modèle de base qui est le « Modèle 1 ». À la fin de cette première étape, nous obtiendrons non seulement le modèle le plus performant en termes d'utilisation des données textuelles, mais nous aurions également fait deux conclusions : la première concerne l'architecture d'encodeur du modèle USE (Transformer ou DAN) qui permet d'avoir les meilleurs résultats, et la deuxième concerne les valeurs optimales de quelques hyperparamètres. Du coup, nous garderons pour la suite la même configuration pour l'entraînement des modèles 2, 3 et 4. Finalement, dès que nous obtiendrons le meilleur modèle à partir des quatre modèles proposés, nous évaluerons par la suite sa performance sur l'ensemble de tests.

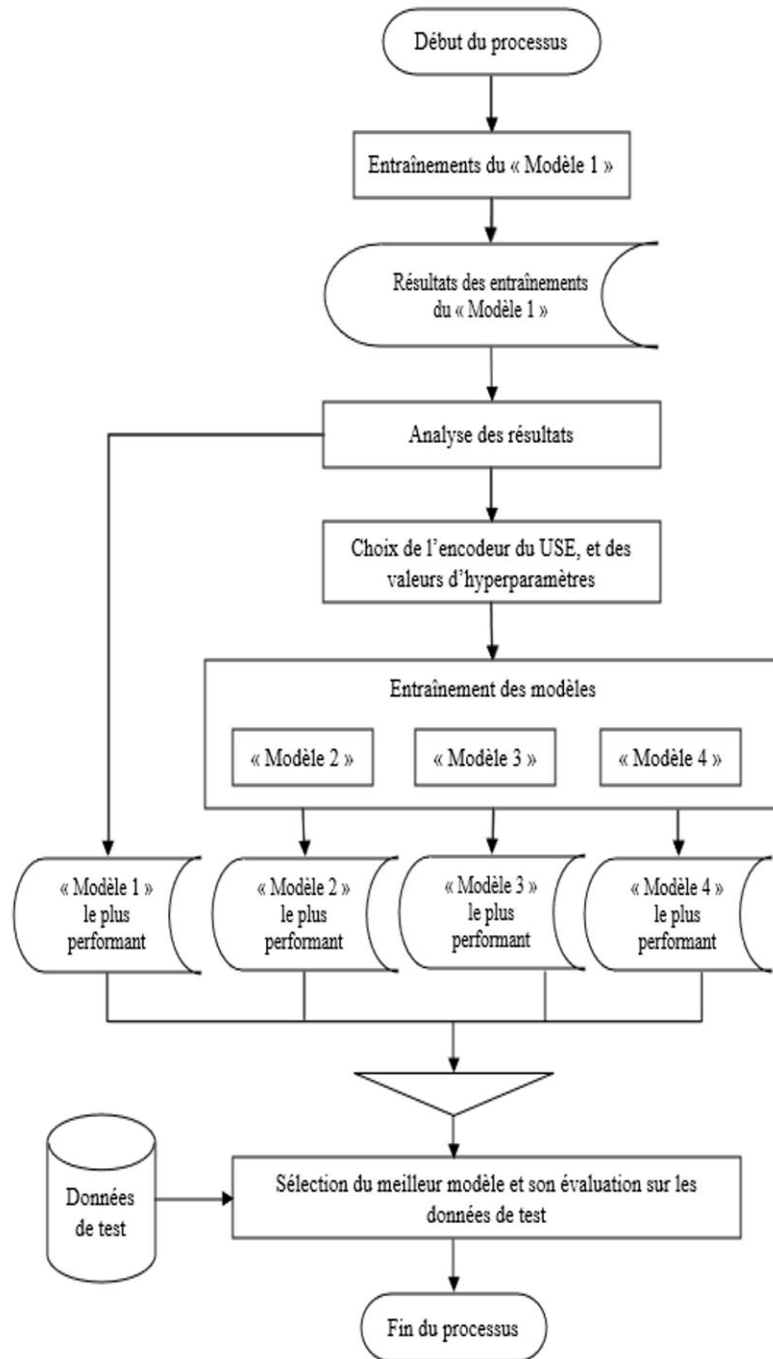


Figure 4.2 – Processus de l'approche suivie

Concernant le modèle 1 qui prend uniquement des données textuelles en entrée, nous avons choisi le message du commit comme information principale puis nous testerons également si l'ajout du titre de la proposition d'un changement (appelé « *issue*<sup>23</sup> » sur GitHub) en plus du message permettra l'amélioration des résultats. L'utilisation de cette information est motivée par le fait que dans certains cas, le message du commit ne fait pas référence à l'intérêt derrière les changements effectués et contient simplement une description des modifications apportées. Un exemple de ce genre de cas est le commit ([lien](#)) du projet « Create-React-App » dont :

- Le message du commit est « *Rename Overlay to ErrorOvelay* »
- Le titre de la proposition du changement est « *Module not found : Can't resolve './components/Overlay' on starting server* »

Ce commit doit être normalement classé comme correctif, car il s'agit d'un bogue où le module *'./components/Overlay'* n'a pas pu être chargé lors du démarrage du serveur. Ici, si on se base uniquement sur le message, le commit va être mal classé comme perfectif vu que le message fait simplement référence à une action de réusinage du code où la classe *'Overlay'* a été renommée par *'ErrorOverlay'*. D'autre part, rappelons que le modèle USE offre la possibilité de choisir entre deux architectures d'encodeur : Transformer ou DAN. La première architecture se compose de 6 couches cachées et offre une meilleure précision, mais nécessite beaucoup de ressources de calcul, et la deuxième architecture se compose de 4 couches cachées et offre une précision légèrement réduite par rapport à la première architecture, mais avec un meilleur temps d'inférence. De ce fait, nous avons quand même décidé de tester les deux architectures dans ce « modèle 1 », de comparer leurs performances et de choisir l'architecture la plus performante afin qu'elle soit réutilisée dans les modèles 2, 3 et 4. Ensuite, comme nous avons choisi d'utiliser le modèle pré-entraîné USE en appliquant l'apprentissage par transfert, un nouvel hyperparamètre qui est le nombre de couches à

---

<sup>23</sup> <https://docs.github.com/en/issues/tracking-your-work-with-issues/quickstart>



réentraîner doit être pris en considération en parallèle avec les autres hyperparamètres généraux qui sont le taux d'apprentissage et la taille du lot. Et donc, afin d'obtenir le meilleur ajustement des valeurs pour les trois hyperparamètres, et en même temps de déterminer l'architecture d'encodeur la plus performante, nous avons entraîné chacune des deux architectures avec différentes combinaisons de valeurs pour les hyperparamètres.

Concernant le modèle 2 dans lequel notre classificateur prendra en entrée le vecteur de plongement généré par le modèle USE en plus de trois caractéristiques qui sont le nombre de fichiers modifiés et le nombre de LOC ajoutées/supprimées. Le choix de ces données quantitatives repose sur le fait que les commits correctifs et ceux adaptatifs se caractérisent par certains aspects qui peuvent permettre à notre modèle de mieux identifier ces deux classes. Dans leurs définitions (voir chapitre 1) et même en pratique à travers quelques études (Mockus & Votta 2000, Mariano et al. 2019), les changements correctifs se présentent sous forme de petites modifications ayant le but de corriger un bogue, tandis que les changements adaptatifs ont tendance à ajouter/supprimer plusieurs LOC vu qu'elles touchent souvent l'infrastructure du projet, que ça soit en ajoutant un nouveau module ou en mettant à jour plusieurs modules à la fois pour assurer une compatibilité entre eux.

Concernant le modèle 3, notre classificateur prendra en entrée deux vecteurs de plongement, le premier est généré par le modèle USE et le deuxième, qui représente le code modifié, sera généré par le modèle CC2Vec. Comme cela a été expliqué plus en détail dans le chapitre 3, CC2Vec est un modèle de réseau de neurones qui apprend la représentation distribuée des changements de code en prédisant les mots qui constituent le message du commit. C'est-à-dire que le modèle CC2Vec considère les mots constituant le message du commit comme une étiquette sémantique fournie par l'auteur du commit. Certes, CC2Vec est indépendant du langage de programmation dans son fonctionnement, mais en réalité, il faut d'abord trouver un moyen pour construire un dictionnaire de code pour sa matrice de plongement. Ce dictionnaire est un ensemble de jetons extraits à partir du code modifié, ce code qui normalement peut être dans n'importe quel langage de programmation.

Donc avant de procéder à l'entraînement de CC2Vec, nous devons d'abord commencer par construire un dictionnaire de code à partir des changements de code de notre jeu de données. Ensuite, pour la couche de sortie du modèle, nous devons extraire la liste de tous les mots utilisés dans les messages des commits.

- Extraction des mots à partir des messages : deux techniques traditionnelles de TALN seront appliquées sur les messages du commit. La première étant la tokenisation, elle permet de séparer un texte en unités plus petites appelées jetons. Ces jetons peuvent être des mots, des caractères ou des sous-mots. La deuxième technique étant la racinisation, elle permet d'enlever les préfixes/suffixes des mots pour ne garder que la racine, ainsi, on évite d'avoir plusieurs mots qui pourraient vouloir dire la même chose. Ces deux techniques sont disponibles à travers la librairie NLTK<sup>24</sup>.
- Construction d'un dictionnaire du code : vu que nous avons choisi d'utiliser plus d'un langage de programmation pour construire notre jeu de données, il fallait trouver un moyen pour prendre en considération ce facteur. Pour cela, nous avons choisi d'exploiter la méthode de tokenisation de l'approche CodeBERT (Feng et al. 2020) qui est à la base un modèle pré-entraîné qui supporte à la fois deux types de données, ceux en langage naturel et ceux en l'un des langages de programmation suivants : Python, Java, JavaScript, PHP, Ruby, Go. Et donc, pour chaque commit, nous avons appliqué la tokenisation sur chacune de ces lignes de code modifiées.

Une fois ces deux étapes sont faites, nous pouvons commencer à ce moment-là l'entraînement du modèle CC2Vec, puis passer à la génération des représentations distribuées du code modifié de chacun de nos commits. Finalement, nous procéderons à l'entraînement de notre « modèle 3 » en concaténant les changements de code, qui sont maintenant sous forme d'un vecteur de plongement, avec le vecteur généré par le modèle USE.

---

<sup>24</sup> <https://www.nltk.org/>

## 4.2 CONSTITUTION DU JEU DE DONNÉES

### 4.2.1 Le choix des projets

Quand il s'agit de compréhension d'un code, il est tout à fait normal que même l'auteur d'un commit puisse se poser parfois des questions sur des changements qu'il avait faits il y a un certain temps. Donc, on peut déjà imaginer la difficulté qu'aura une personne, qui ne fait pas partie des contributeurs d'un projet, à comprendre les détails du code modifié. Dans notre cas, pour minimiser au moins l'effet de cette contrainte et pour construire un jeu de données avec un nombre important de commits, nous avons sélectionné deux projets qui œuvrent dans le domaine du développement d'application Web : « Create-React-App<sup>25</sup> » et « Flask<sup>26</sup> ». Ces deux projets disponibles sur GitHub sont à code source ouvert et sont toujours actifs.

Le choix de ces deux projets repose principalement sur l'expérience que nous avons acquise dans ce domaine de développement d'application Web. Cette expérience qui va nous permettre de bien comprendre l'intérêt des changements effectués surtout quand les informations disponibles (message du commit, la proposition de changements...etc.) n'en disent pas beaucoup, ou qu'elles comportent des notions qui nécessitent certaines connaissances dans le domaine. Ensuite, le choix d'un projet en JavaScript (Create-React-App) et d'un autre en Python (Flask) fait partie des objectifs que nous essayons d'atteindre en proposant une approche indépendante du langage de programmation afin de remédier aux limites de la majorité des études reliées qui ont exploité les changements de code en langage Java puisqu'elles utilisent des outils de traitement de code qui ne supportent que ce langage de programmation. Donc, les deux projets choisis sont :

---

<sup>25</sup> <https://github.com/facebook/create-react-app>

<sup>26</sup> <https://github.com/pallets/flask>

- « **Create-React-App** » : il fournit un environnement de développement d'application Web avec une sélection d'outils préconfigurés, tout en offrant une certaine flexibilité et des options de personnalisation. En fait, il offre des fonctionnalités pratiques, telles que : un environnement de développement en temps réel (une mise à jour directe de l'application dans le navigateur lors d'une modification du code), un code passe-partout (*boilerplate code*, en anglais) et la gestion des dépendances entre les librairies. En gros, il permet aux développeurs qui découvrent les outils/bibliothèques JavaScript modernes de se concentrer sur la logique d'affaire de leur projet sans avoir à perdre du temps sur comment doit se faire la configuration de l'environnement du projet.
- « **Flask** » : un micro-framework d'application Web. Il est conçu pour faciliter et accélérer la mise en œuvre des applications, tout en facilitant la possibilité de faire augmenter leur évolutivité (*scalability*, en anglais). Il se base essentiellement sur deux modules : « *werkzeug* » qui est une librairie offrant tous les services nécessaires pour le WSGI (*Web Server Gateway Interface*), et « *jinja* » qui est un moteur de template.

#### 4.2.2 Extraction des commits

Une fois les projets choisis, nous passons à la phase de sélection puis d'extraction des commits. Parmi l'ensemble des commits, nous avons sélectionné au total 1857 commits : 922 à partir de « Create-React-App » et 935 de « Flask ». Durant cette phase, nous avons éliminé pas mal de commit pour les raisons suivantes :

- Les commits sans message significatif (tel que : « *update file.js* »).
- Les commits sous forme de *Releases*.
- Les commits qui ne contient aucun fichier JavaScript ou Python.

Un fois les commits sélectionnés, nous passons à la phase d'extraction en utilisant l'outil jsdom<sup>27</sup>. Ce dernier est un robot d'indexation qui va prendre en entrée l'URL (*Uniform Resource Locator*) du commit sur GitHub et va retourner le contenu de la page web sous forme HTML (*HyperText Markup Language*). Le contenu de cette page HTML sera analysé afin d'extraire les informations dont nous avons besoin, c.-à-d., le message du commit et les changements de code. Finalement, pour chaque commit, un fichier texte est généré. Ce fichier, dont la forme est montrée à la figure 4.3, correspond à un exemple de notre jeu de données. Concernant le titre de la proposition de changement (*issue*) qui se trouve dans une autre page WEB, il a été récupéré manuellement.

```
id| https://github.com/facebook/create-react-app/commit/1d5a27736766c6fa9f6ae27c8fdaab4255fe735d
message| Enable watchContentBase for webpack-dev-server
issue| Force page reload in Development when external files have changed
label|

==>| packages/react-dev-utils/webpackHotDevClient.js
    @@ -252,6 +252,10 @@ connection.onmessage = function(e) {
*|252|         case 'ok':
*|253|             handleSuccess();
*|254|             break;
+|255|         case 'content-changed':
+|256|             // Triggered when a file from `contentBase` changed.
+|257|             window.location.reload();
+|258|             break;
*|255|         case 'warnings':
*|256|             handleWarnings(message.data);
*|257|             break;

==>| packages/react-scripts/scripts/start.js
    @@ -249,6 +249,8 @@ function runDevServer(host, port, protocol) {
*|249|         // for some reason broken when imported through Webpack. If you just want to
*|250|         // use an image, put it in `src` and `import` it from JavaScript instead.
*|251|         contentBase: paths.appPublic,
+|252|         // By default files from `contentBase` will not trigger a page reload.
+|253|         watchContentBase: true,
*|252|         // Enable hot reloading server. It will provide /sockjs-node/ endpoint
*|253|         // for the WebpackDevServer client so it can learn when the files were
*|254|         // updated. The WebpackDevServer client is included as an entry point
```

Figure 4.3 – Un exemple du fichier généré pour chaque commit.

---

<sup>27</sup> <https://github.com/jsdom/jsdom>

### 4.2.3 La phase d'étiquetage

Une fois notre jeu de données est construit, nous avons procédé à l'étiquetage manuel de ces exemples. Cette étape nécessite deux choses essentielles : la définition des règles sur lesquels nous nous baserons pour attribuer un commit à la classe ou les classes correspondantes ; et la définition d'un protocole qui va nous garantir à la fin que notre étiquetage est fiable et non biaisé par l'interprétation humaine. Pour commencer, nous avons défini nos activités de maintenance sous forme de trois classes<sup>28</sup> : perfectif, correctif et adaptatif, comme ça a été proposé par Swanson (1976). Ce dernier avait décrit ces classes comme suit :

- Perfectif : tout changement de code qui permet soit une amélioration des performances d'un système soit une restructuration du code à travers des modifications de réusinage.
- Correctif : tout changement de code qui a comme but de régler un dysfonctionnement dans un système.
- Adaptatif : tout changement de code qui permet de garantir une compatibilité entre les composantes d'un système.

C'est vrai que ces définitions couvrent la majorité des opérations de modification de code possible, mais elles restent quand même insuffisantes pour certains cas de changements de code qui peuvent être présents dans notre jeu de données. D'ailleurs, comme déjà discuté dans le chapitre 3, Hindle et al. (2008) ont choisi d'ajouter une nouvelle classe, appelée implémentation, où ils ont mis tout changement de code qui concerne l'ajout d'une nouvelle fonctionnalité. Apparemment, cette dernière n'a pas été évoquée par Swanson (1976) du fait

---

<sup>28</sup> Certes, la classe préventive est citée aussi dans certaines sources, mais elle partage plusieurs traits en commun avec la classe perfective en termes d'amélioration d'un système (Thakur 2023), et parfois même avec celle corrective en corrigeant un dysfonctionnement qui n'affecte pas actuellement le système, mais qui causera un sérieux problème dans le futur (Sanjoy 2023).

qu’il s’est plus intéressé à la maintenance des systèmes déjà conçus (Hindle et al. 2008). De ce fait, pour ce cas d’ajout d’une nouvelle fonctionnalité, nous avons décidé de suivre la définition de Pressman & Maxim (2020) qui l’ont considéré comme un changement perfectif. En plus de ça, nous avons défini quelques règles (voir tableau 4.1) pour des cas de changements qui peuvent créer confusion lors de l’étiquetage.

Tableau 4.1 – Les règles que nous avons définies pour certains cas de changements

Cas	Classe(s)
Ajout des tests (unitaires, fonctionnel...etc.)	Perfectif
Correction d’un bogue + Modification des tests correspondants	Correctif
Correction d’un bogue + Ajout des tests correspondants	Correctif et perfectif
Modification touchant l’infrastructure (compatibilité entre les paquetages, etc.) + Ajout des tests correspondants	Adaptatif et perfectif
Modification touchant l’infrastructure + Modification des tests correspondants	Adaptatif
Correction des coquilles ( <i>typos</i> , en anglais) dans la documentation et dans les messages d’avertissement ou dans les fichiers de journalisation (log)	Perfectif <sup>29</sup>
Correction des coquilles dans le code (nom d’une méthode, une variable...etc.)	Correctif

Une fois toutes les règles spécifiées, nous passons à la définition du protocole à suivre pour garantir une certaine fiabilité de notre étiquetage. Contrairement à l’étiquetage d’un jeu de données sous forme d’images ou de texte qui peut être réalisé par n’importe quelle personne, la classification manuelle d’un commit nécessite, d’une part, une connaissance dans le domaine d’informatique et plus précisément en programmation logicielle, et d’autre part, beaucoup plus de temps vu le nombre important d’informations à vérifier avant de classer un exemple. Ces informations qui sont le message, le titre de la proposition de changement, le code modifié, le nom des fichiers modifiés et leurs rôles, et les discussions

<sup>29</sup> Suivant le raisonnement de Zafar et al. (2019) qui ne considèrent pas ce cas comme étant correctif du fait que la modification de la documentation ou des commentaires n’a aucun effet fonctionnel sur le code.

entre les contributeurs, etc. Donc, pour minimiser au maximum les risques liés aux erreurs qui peuvent être fréquentes d'ailleurs à cause de la complexité de la tâche, nous avons fait appel à 10 personnes ayant une expérience en programmation entre 5 à 10 ans pour nous aider dans cette étape d'étiquetage.

Tout d'abord, nous avons décidé de développer une application (voir figure 4.4) qui va présenter les informations d'un commit de manière beaucoup plus lisible et facile à comprendre, contrairement à si nous avions travaillé directement avec le fichier texte généré pour chacun des commits (voir figure 4.3). Ensuite, nous nous sommes renseignés auprès des participants concernant leur préférence et leur familiarisation avec les deux projets « Create-React-App » qui est en langage JavaScript et « Flask » qui est en langage Python, puis nous avons essayé, au maximum, de leur affecter les commits en fonction de leur choix. Il est important de prendre en considération ce facteur pour permettre aux participants de gagner du temps et d'éviter aussi à ce qu'un commit soit mal classifié à cause d'une mauvaise compréhension de la syntaxe du langage de programmation. Finalement, nous avons procédé à l'étiquetage de la manière suivante : dans une première étape, l'ensemble du jeu de données (1857 commits) a été classifié par moi-même et chaque groupe de deux personnes s'est vu recevoir les mêmes ~371 commits, ce qui fait que chaque commit a été étiqueté par trois personnes. Les participants avaient reçu d'avance un document comportant un guide d'utilisation de l'application, les définitions des trois classes d'activité de maintenance, la structure des deux projets avec le rôle de chaque dossier (voir annexe A) et quelques exemples étiquetés avec des explications. Dans une deuxième étape, nous avons comparé les trois étiquettes de chaque commit puis nous avons programmé plusieurs rencontres pour résoudre au maximum les désaccords trouvés. En raison de la disponibilité limitée des participants et du temps nécessaires pour discuter de tous les cas de désaccords, nous avons décidé d'abord de résoudre tous les commits sur lesquels les trois étiquettes étaient différentes, puis dans un deuxième temps d'essayer de résoudre le maximum possible pour le reste des commits. À la fin de ce processus, nous nous sommes retrouvés avec un jeu de données dont 83% des commits avaient un accord total.



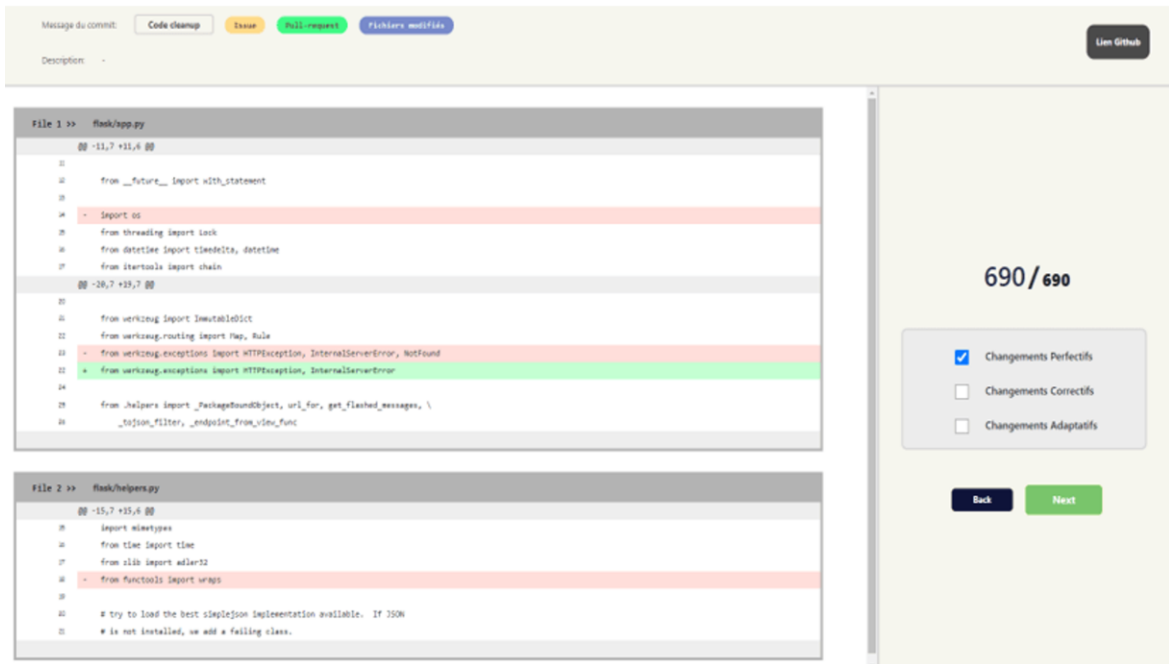


Figure 4.4 – Aperçu de l’application développée pour effectuer l’étiquetage.

Une fois notre jeu de données a été étiqueté et vérifié, l’étape finale est de préparer les ensembles de données pour nos modèles. Rappelons qu’il est recommandé de répartir le jeu de données en trois ensembles : d’entraînement, de validation et de test. Comme c’est détaillé dans le tableau 4.2, nous avons donc attribué 80% des exemples à l’ensemble d’entraînement utilisé pour ajuster les poids de nos modèles, 10% à l’ensemble de validation utilisé pour garantir une meilleure optimisation du modèle, et 10% pour l’ensemble de tests utilisé pour confirmer la généralisation des résultats avec des exemples non vus auparavant par les modèles.

Tableau 4.2 – La répartition du jeu de donnée sur les trois ensembles

	Nombre de commits	Pourcentage	Entraînement	Validation	Test
Perfectif	965	51.96	773	96	96
Correctif	403	21.71	323	40	40
Adaptatif	372	20	298	37	37
Perfectif et correctif	72	3.88	58	7	7
Perfectif et adaptatif	42	2.27	34	4	4
Correctif et adaptatif	3	0.18	3	0	0
Perfectif, correctif et adaptatif	0	0	0	0	0
Total	1857	100	1489	184	184

### 4.3 GESTION DES CLASSES ASYMÉTRIQUES

L'apprentissage à partir de données dont la distribution des classes est éloignée de la distribution uniforme est une situation fréquente dans plusieurs domaines d'application. Parmi les domaines qui en souffrent le plus, on trouve : la détection des fraudes (Cieslak et al. 2006), le diagnostic médical (Mazurowski et al. 2008), la gestion des risques bancaires (Huang et al. 2006), la détection des marées noires (Kubat et al. 1998), etc. Notre tâche de classification par activité de maintenance n'en fait pas l'exception, et en souffre également. En effet, comme nous avons vu dans les statistiques présentées dans chapitre 1, les changements perfectifs sont d'habitude ceux qui surviennent le plus fréquemment durant le processus de maintenance logicielle. Une chose qui a été bien confirmée après avoir effectué l'étiquetage de notre jeu de données comme le montre le tableau 4.2.

Pour faire face à cette contrainte, plusieurs méthodes ont été proposées. Le choix de la méthode ou des méthodes à utiliser dépend principalement du problème à traiter. Ces

méthodes peuvent être regroupées en deux catégories : la première catégorie agit directement sur le jeu de données et consiste à rééquilibrer la distribution des classes en effectuant du rééchantillonnage. Ce dernier vient soit sous forme d'un sous-échantillonnage (*undersampling*, en anglais) qui consiste à supprimer aléatoirement des exemples de la classe ou les classes abondantes, soit sous forme d'un suréchantillonnage (*oversampling*, en anglais) qui consiste à ajouter de nouveaux exemples, parfois créés artificiellement, aux classes les moins représentées. La deuxième catégorie, qui entre dans le cadre l'apprentissage sensible au coût (*Cost Sensitive Learning*, en anglais), intervient lors de l'entraînement du modèle au niveau du calcul de la fonction de coût. L'une des méthodes de cette catégorie est l'équilibrage des poids qui permet de donner plus d'importance aux classes les moins représentées, et ce en compensant le manque d'exemples qu'ils ont par un poids plus important. Et ainsi, le coût sera calculé différemment en fonction du poids de chaque classe.

Dans notre contexte, les deux méthodes de rééchantillonnage ne seront pas utilisées pour la simple raison que l'application du sous-échantillonnage peut nous faire perdre des exemples ayant des caractéristiques utiles pour la détection d'une classe donnée, et de limiter également la diversité des exemples au sein d'une même classe. Quant au suréchantillonnage, il va falloir analyser un nouveau projet à code source-ouvert avant de procéder à l'extraction de ces commits. Malheureusement, c'est un travail qui nécessitera beaucoup de temps et d'effort, surtout pour les participants à l'étiquetage, du fait qu'ils devaient comprendre l'architecture de ce nouveau projet. Puis, pour la deuxième solution qui est la génération d'exemples, il n'est pratiquement pas possible de générer de nouveaux commits artificiellement. Par conséquent, nous avons choisi d'utiliser la méthode d'équilibrage des poids pour régler ce problème d'asymétrie entre les classes. Nous avons donc calculé le poids de chacune des trois classes suivant la méthode définie par Tensorflow<sup>30</sup>, et nous avons obtenu les poids suivants : perfectif = 0.57, correctif = 1.29 et adaptatif = 1.48.

---

<sup>30</sup> [https://www.tensorflow.org/tutorials/structured\\_data/imbalanced\\_data#class\\_weights](https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#class_weights)

#### 4.4 CHOIX DES MÉTRIQUES

Comme nous avons expliqué dans le chapitre 1 concernant les métriques d'évaluation utilisées dans le contexte multiétiquettes, il faut tenir compte du fait que la prédiction faite pour chaque commit peut être totalement correcte, partiellement correcte ou totalement incorrecte. Par conséquent, nous avons choisi d'utiliser deux métriques : le score  $F1_{micro}$  qui est très souvent utilisé, car il reflète la vraie performance d'un modèle surtout quand on dispose d'un jeu de données avec des classes asymétriques. Mathématiquement, le score  $F1_{micro}$  est la moyenne harmonique des deux valeurs de la  $précision_{micro}$  et le  $rappel_{micro}$  qui sont définies comme suit :

$$Précision_{micro} = \frac{\sum_{\lambda=1}^{|L|} VP_{\lambda}}{\sum_{\lambda=1}^{|L|} VP_{\lambda} + \sum_{\lambda=1}^{|L|} FP_{\lambda}}$$

$$Rappel_{micro} = \frac{\sum_{\lambda=1}^{|L|} VP_{\lambda}}{\sum_{\lambda=1}^{|L|} VP_{\lambda} + \sum_{\lambda=1}^{|L|} FN_{\lambda}}$$

Avec : L : nombre de classes.

$VP_{\lambda}$ ,  $FP_{\lambda}$  et  $FN_{\lambda}$  : les résultats de prédictions pour la classe  $\lambda$ .

C'est vrai qu'il est recommandé d'utiliser une seule métrique pour évaluer la performance d'un modèle, mais comme le contexte de la classification multiétiquettes nous oblige à vérifier d'autres aspects au sein de chaque prédiction, nous avons choisi de rapporter aussi la perte de Hamming, qui dans sa définition, nous permet de mesurer à quel point le vecteur prédit est proche du vecteur réel. Ces deux vecteurs qui sont comparés sont de taille 3 dont chaque élément fait référence à l'une des trois classes : perfectif, correctif et adaptatif.

## 4.5 TECHNOLOGIES UTILISÉES

Depuis sa création en 1991, Python n'a pas cessé de se développer jusqu'à parvenir à se classer parmi le top cinq des langages de programmation le plus utilisés. L'un de ses points fort est sa richesse en termes de bibliothèques avec lesquelles il est compatible. Ces bibliothèques couvrent plusieurs disciplines comme la science des données, le traitement des images, le TALN, la visualisation des données, etc. D'ailleurs ces dernières années, un grand nombre de ses bibliothèques, comme Tensorflow, Pytorch et Keras, sont les plus utilisées dans le monde de l'apprentissage automatique et profond.

En pratique, la contrainte à laquelle nous étions confrontés à ce niveau est la compatibilité entre l'ensemble de bibliothèques dont nous aurions besoin pour atteindre nos objectifs. En effet, nous avons choisi la bibliothèque Tensorflow<sup>31</sup> 2.1 de Google pour l'entraînement de nos modèles de classification, ce choix est fait principalement parce que le modèle pré-entraîné USE n'est disponible qu'à travers la plateforme TensorFlow-Hub<sup>32</sup>. Cette dernière est un dépôt de modèles pré-entraînés qui sont prêts à être utilisés dans le cadre l'apprentissage par transfert. Ensuite, pour l'entraînement du modèle CC2Vec qui va nous permettre de générer les représentations distribuées du code modifié, il a été fait en utilisant la bibliothèque Pytorch<sup>33</sup>. Ici, nous étions obligés de suivre la même configuration de bibliothèques mentionnée dans le guide pour faire fonctionner correctement ce modèle. D'ailleurs, ces deux bibliothèques, que nous avons utilisées, offrent une documentation très riche et possèdent une communauté active en permanence.

Concernant la configuration de la machine utilisée, nous avons réalisé les entraînements sur un GPU NVIDIA QUADRO RTX600028 et un CPU Intel Xeon W-214529 disposant de 64Go de mémoire vive.

---

<sup>31</sup> <https://www.tensorflow.org>

<sup>32</sup> <https://tfhub.dev>

<sup>33</sup> <https://pytorch.org>

## 4.6 LA VALIDATION DE L'APPROCHE

### 4.6.1 Les résultats

Comme déjà décrit, la première étape de l'approche concerne le modèle 1 qui fait la classification des commits par activité de maintenance en se basant sur des données textuelles. Nous avons considéré ce modèle comme étant un modèle de base, car c'est à partir de lui que nous déterminerons principalement l'encodeur (DAN ou Transformer) du USE que nous utiliserons pour le reste des modèles.

Afin de déterminer la meilleure configuration pour le modèle, c.-à-d., l'encodeur le plus performant ainsi que les hyperparamètres (nombre de couches à réentraîner, taux d'apprentissage, la taille du lot) qui lui sont associés, nous avons pris le message du commit comme entrée et nous avons testé l'ensemble des combinaisons possibles, ce qui nous a fait au total : 240 tests pour l'architecture avec l'encodeur DAN comme le montre la figure 4.5 (et 336 tests pour le Transformer puisqu'il comporte 2 couches de plus). Pour ce qui est du classificateur, nous avons ajouté une seule couche ayant 3 neurones avec une activation sigmoïde qui retourne pour chaque exemple, une valeur entre 0 et 1 pour chacune des 3 classes représentant les types d'activité de maintenance. Pour ce qui est de la fonction de coût, nous avons utilisé l'entropie croisée binaire (BCE – *Binary Cross Entropy*, en anglais) avec ADAM (Kingma & Ba 2014) comme optimisateur. Et puis, nous avons choisi la valeur 50 comme nombre d'époques (*epochs*), c.-à-d., le nombre de cycles d'entraînement pour chaque configuration de modèle.

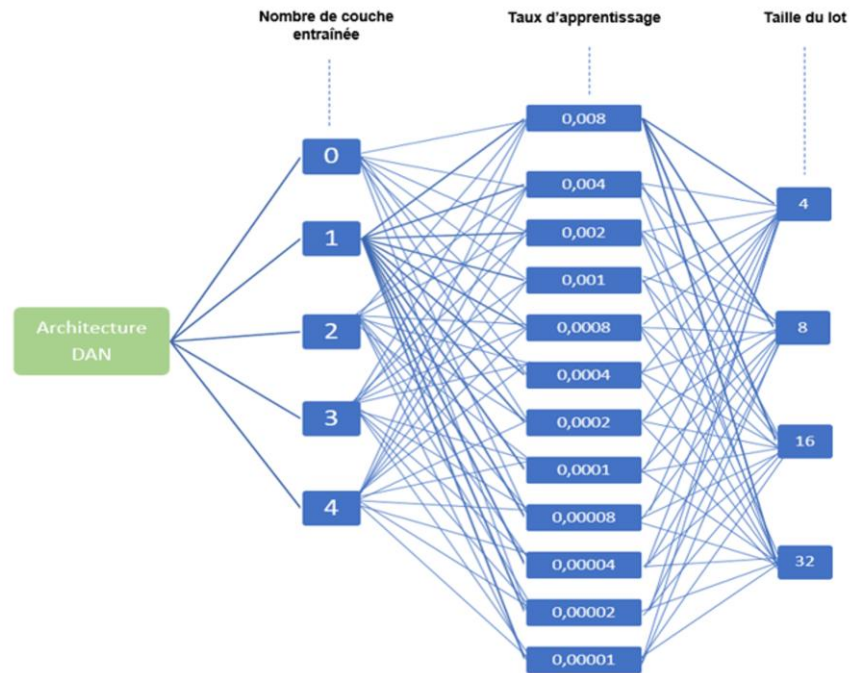


Figure 4.5 – Combinaisons de valeurs testées pour les hyperparamètres (la variante DAN du USE)

Les résultats obtenus durant cette première phase confirment ce que nous avons rapporté théoriquement dans le chapitre 2, sur le fait que la variante Transformer du modèle pré-entraîné USE dépasse celle du DAN en termes de performance. Comme on peut voir dans le tableau 4.3 montrant le meilleur score  $F1_{\text{micro}}$  de validation obtenu pour chaque nombre de couches réentraînées, le score  $F1_{\text{micro}}$  s’améliore en augmentant le nombre de couches réentraînées passant ainsi de 67% à 73.1% pour le modèle avec l’encodeur DAN, et de 73.5% à 74.8% pour le Transformer. Et en notant que le meilleur résultat concerne le modèle où toutes les couches du Transformer sont réentraînées. Par conséquent, nous utiliserons pour le reste de notre approche, l’encodeur Transformer avec les hyperparamètres suivants :

- Nombre de couches réentraînées : 6
- Taux d’apprentissage = 0.00008
- Taille du lot = 4

Tableau 4.3 – Les meilleurs résultats pour chaque valeur de nombre de couches entraînées pour le DAN et Transformer

Encodeur	Nombre de couches réentraînées	Taux d'apprentissage	Taille du lot	Score $F1_{\text{micro}}$ de validation
DAN	0	0.004	4	67
	1	0.002	4	71
	2	0.0001	4	73
	3	0.004	16	72.8
	4	0.002	8	73.1
Transformer	0	0.00002	4	62
	1	0.0001	4	73.5
	2	0.00002	8	73.4
	3	0.00004	32	73.8
	4	0.00004	4	74.3
	5	0.0001	8	74.3
	6	0.00008	4	74.8

La figure 4.6 montre un aperçu des résultats de l'entraînement et de validation (coût et score  $F1_{\text{micro}}$ ) par époque pour la configuration qui a permis d'obtenir le meilleur résultat. On constate que le meilleur modèle est celui de l'époque 17 atteignant un score  $F1_{\text{micro}}$  de 74.8%. Mais le problème, c'est qu'on peut voir clairement à partir de l'époque 9 que, certes, le coût d'entraînement continue de diminuer, mais on constate en contrepartie que le coût de validation commence à augmenter. Ce qui veut dire que ce meilleur modèle (époque 17) entre dans le cadre de ce qu'on appelle : une phase de surapprentissage. Ce qui nous mène à penser à utiliser l'une des techniques de régularisation précédemment présentées dans le chapitre 1.



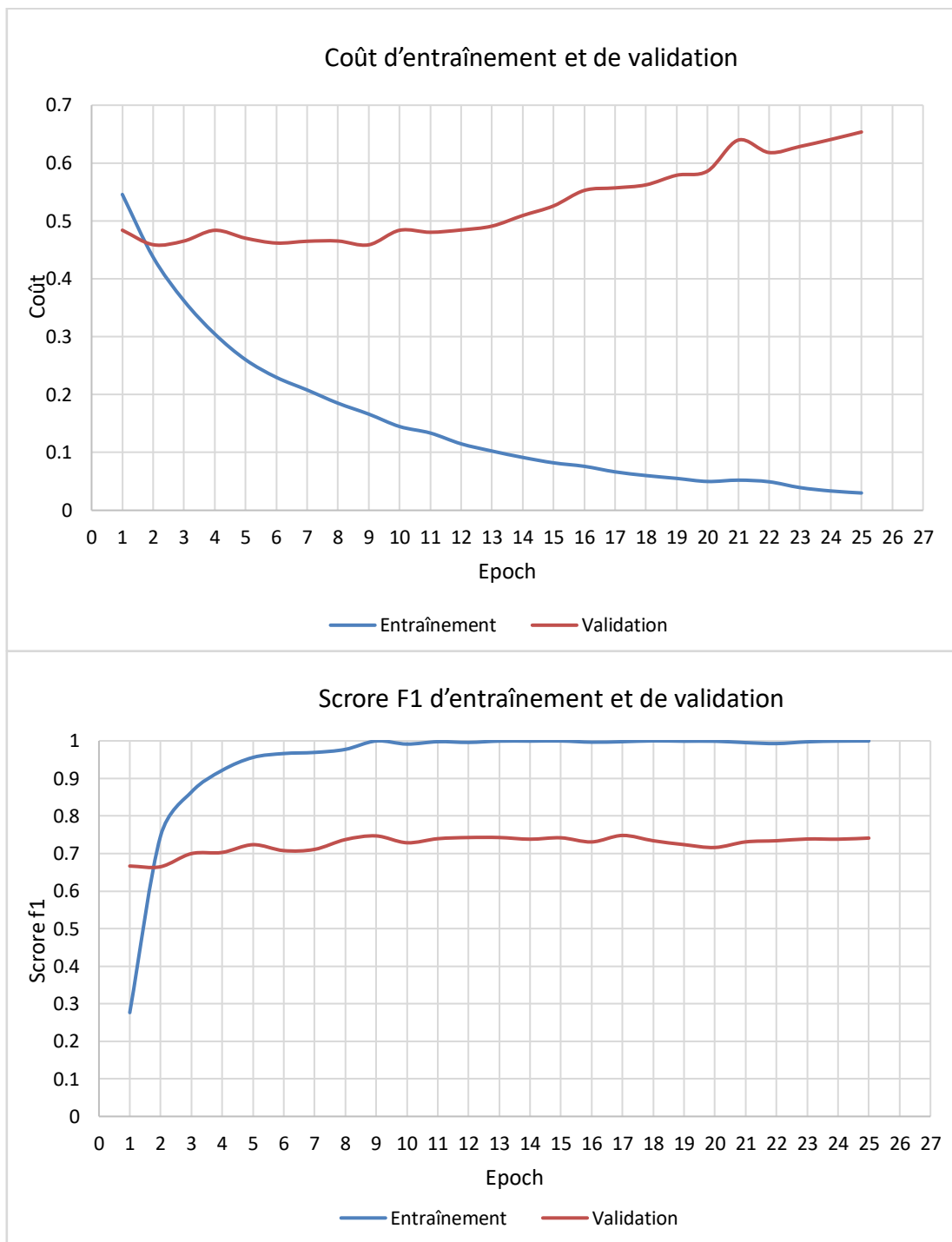


Figure 4.6 – Résultats d'entraînement et de validation du modèle 1

Pour pallier ce problème de surapprentissage, nous avons choisi d'utiliser les pénalités de normes de paramètre  $L_1$  et  $L_2$  comme technique de régularisation, et ce, en leur attribuant différentes valeurs et en gardant la même configuration d'hyperparamètres que celle du modèle précédemment sélectionné. Même si la pénalité  $L_2$  est généralement plus communément utilisée que  $L_1$  (Goodfellow et al. 2016), nous avons choisi quand même de les tester séparément dans un premier lieu. Ensuite, en analysant le comportement du modèle sur les cinq résultats obtenus, nous avons choisi les valeurs qui nous semblent appropriées pour la combinaison des deux pénalités. Le tableau 4.4 présente les valeurs testées, tandis que les graphiques y correspondant sont disponibles dans l'annexe B.

Tableau 4.4 – Les différentes valeurs de  $L_1$  et  $L_2$  testées

Régularisation	$L_1$	$L_2$
1	0.009	0
2	0.02	0
3	0	0.08
4	0	0.2
5	0	0.4
6	0.01	0.1
7	0.02	0.1

La figure ci-dessous représente les résultats de l'entraînement et de validation (coût et score  $F1_{\text{micro}}$ ) avec les valeurs  $L_1 = 0.02$  et  $L_2 = 0.1$  qui ont permis d'avoir le meilleur modèle. Avec 73.6% en score  $F1_{\text{micro}}$ , ce score représente, pour le moment, le meilleur résultat obtenu pour notre modèle 1 qui est basé uniquement sur des données textuelles. En effet, ce meilleur modèle correspond à l'époque 13 dans laquelle on peut voir que le score  $F1_{\text{micro}}$  est à sa valeur maximale, et en même temps, on peut voir qu'il ne s'agit pas d'une phase de surapprentissage, vu que le coût de validation était en baisse du début jusqu'à l'époque 15 où on constate que son écart avec le coût d'entraînement commence à augmenter.

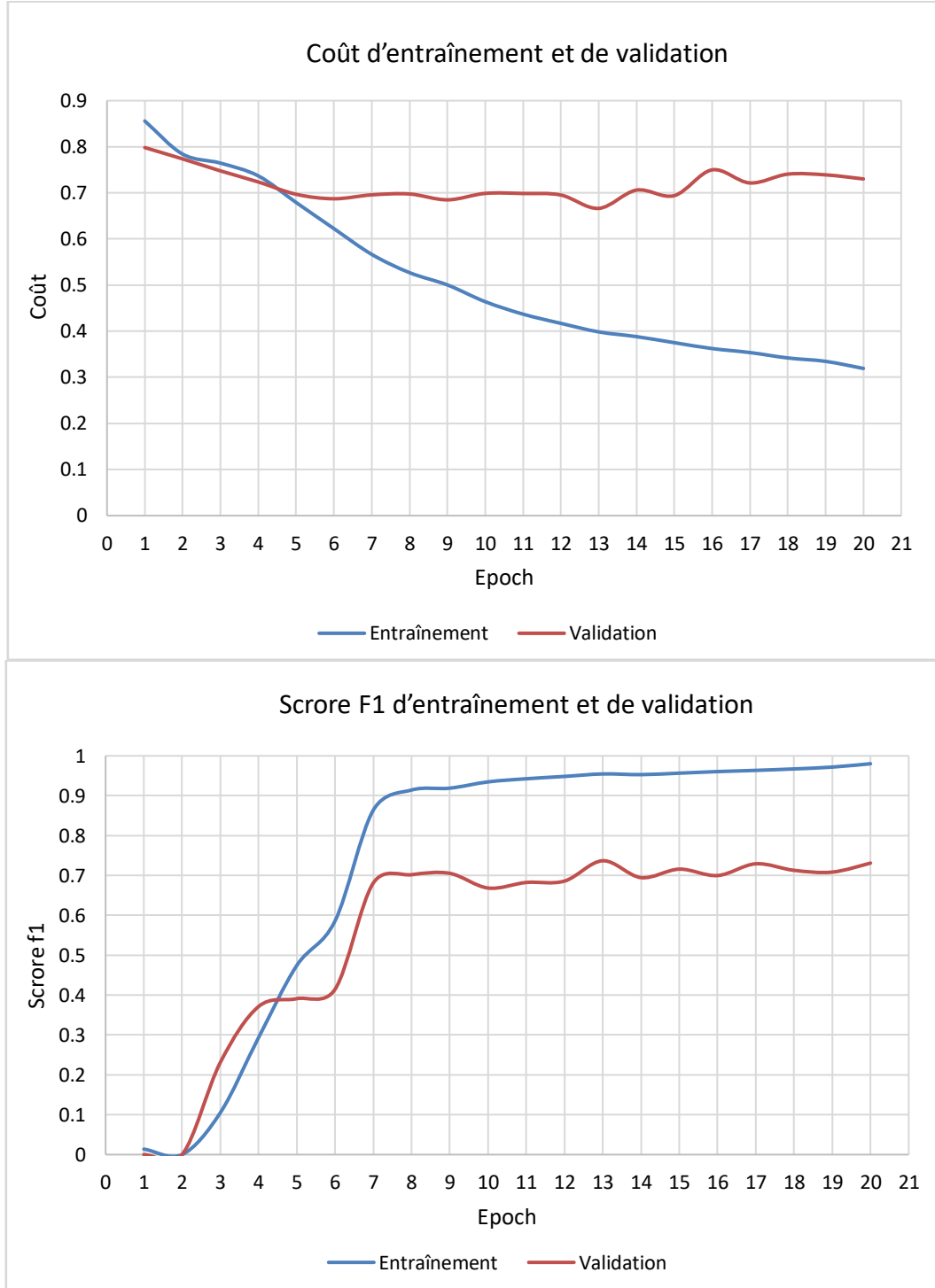


Figure 4.7 – Résultats d'entraînement et de validation du modèle 1 avec pénalité de norme  $L_1$  (0.02) et  $L_2$  (0.1)

Afin d'améliorer encore plus le résultat du modèle 1, et comme nous l'avons expliqué dans la description de l'approche, nous avons pensé à ajouter une nouvelle information textuelle au message du commit, qui est le titre de la proposition d'un changement (appelé « Issue » sur GitHub). Cette dernière représente l'information principale sur laquelle se basent les contributeurs aux projets avant d'apporter des changements au code, et est souvent associée aux commits qui contribuent à sa réalisation. Ce titre peut représenter, par exemple, une notification d'un bogue trouvé, ou bien parfois une proposition pour une amélioration à apporter au système.

Nous avons donc gardé les mêmes hyperparamètres, mais nous avons utilisé cette fois comme entrée du modèle, une concaténation des deux phrases : le message du commit et le titre de la proposition de changement. Le résultat de ce test est présenté à la figure 4.8 où on peut constater que le meilleur modèle concerne l'époque 13 avec un score  $F1_{\text{micro}}$  de 79.4%, ce qui représente une amélioration de 4.6% par rapport au modèle prenant en entrée le message du commit uniquement. Ici encore, on constate que ce meilleur modèle rentre dans la phase de surapprentissage à partir de l'époque 8. De ce fait, nous procéderons de la même manière qu'auparavant en testant différentes valeurs des pénalités de normes de paramètre  $L_1$  et  $L_2$  pour ce modèle. Les graphiques correspondants à ces tests sont disponibles dans l'annexe C.

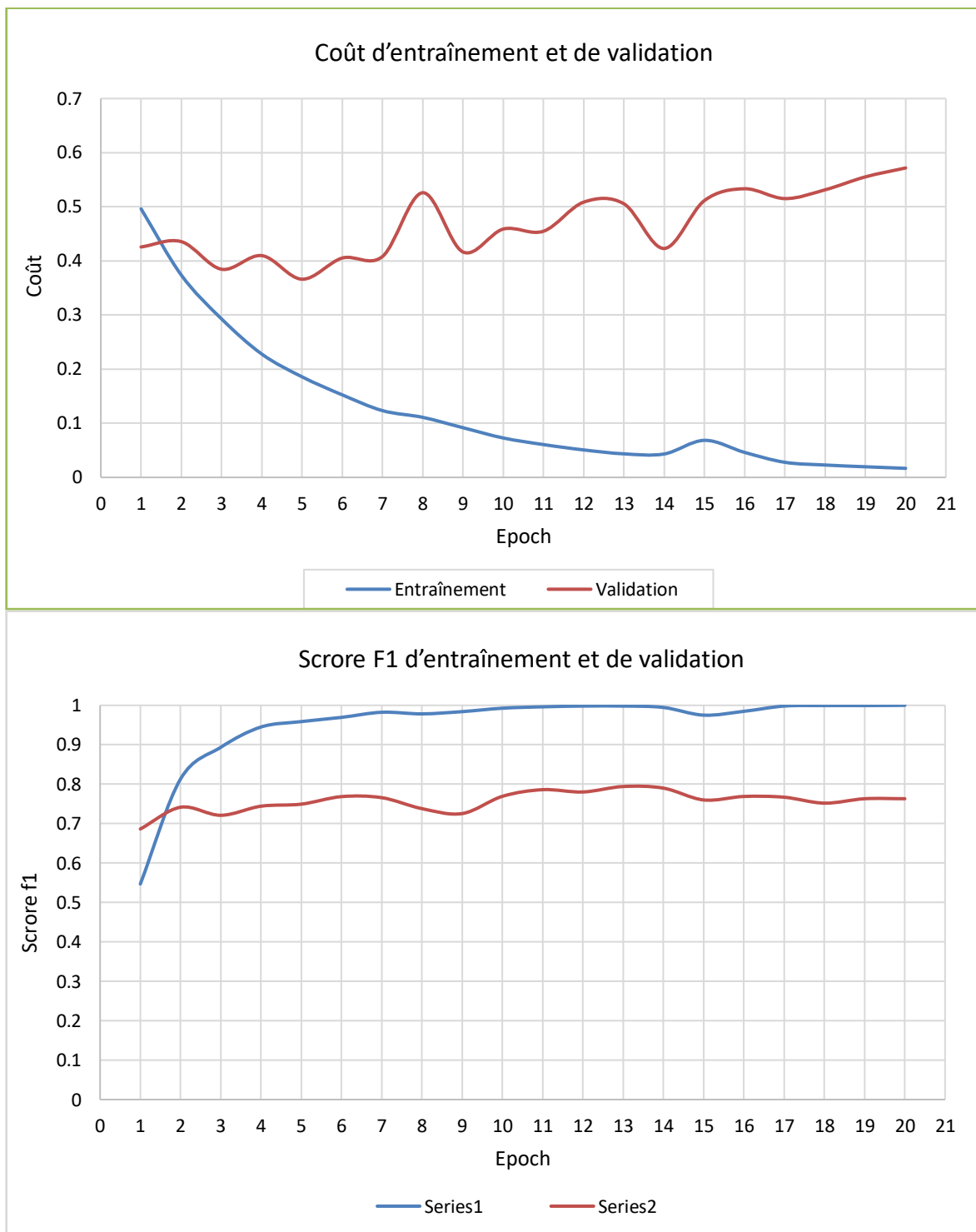


Figure 4.8 – Résultats d'entraînement et de validation du modèle 1 après ajout du titre de la proposition de changement

Les résultats de cette phase de régularisation (voir figure 4.9) nous ont permis d'avoir un score  $F1_{\text{micro}}$  de 79.2% au niveau de l'époque 17, avec  $L_1 = 0.02$ . Ce qui représente finalement le meilleur résultat obtenu pour le modèle 1 qui est basé sur des données textuelles.

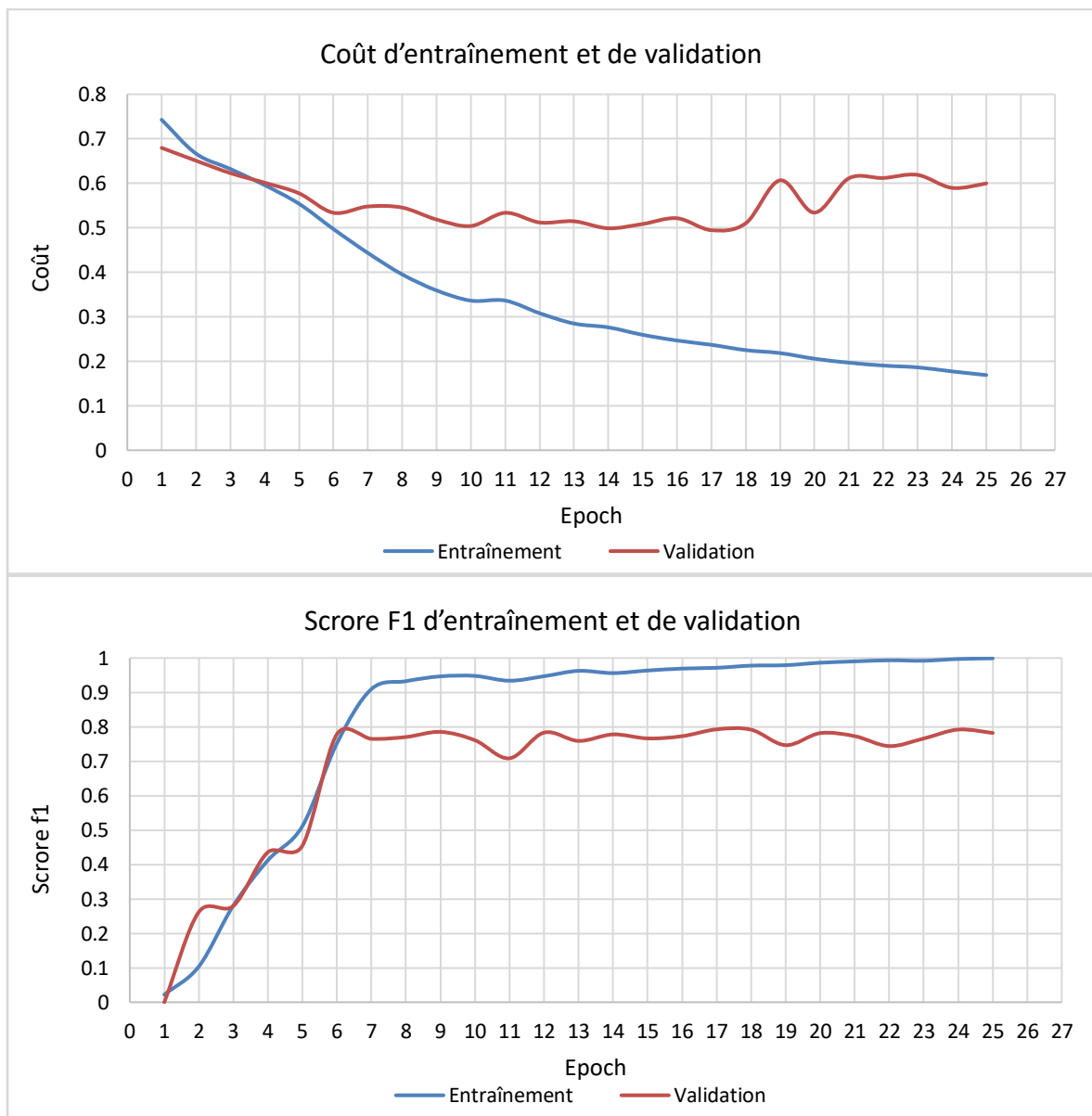


Figure 4.9 – Résultats d'entraînement et de validation du meilleur modèle 1 obtenu



+ une couche de sortie), mais nous avons eu des résultats inférieurs à ceux obtenus avec un classificateur d'une seule couche. Les meilleurs scores  $F1_{micro}$  obtenus étaient de 78%, 76.1%, 78.6% et 78.8% respectivement dans chacun des modèles 1, 2, 3 et 4. Les graphiques correspondants à ces tests sont disponibles dans l'annexe D.

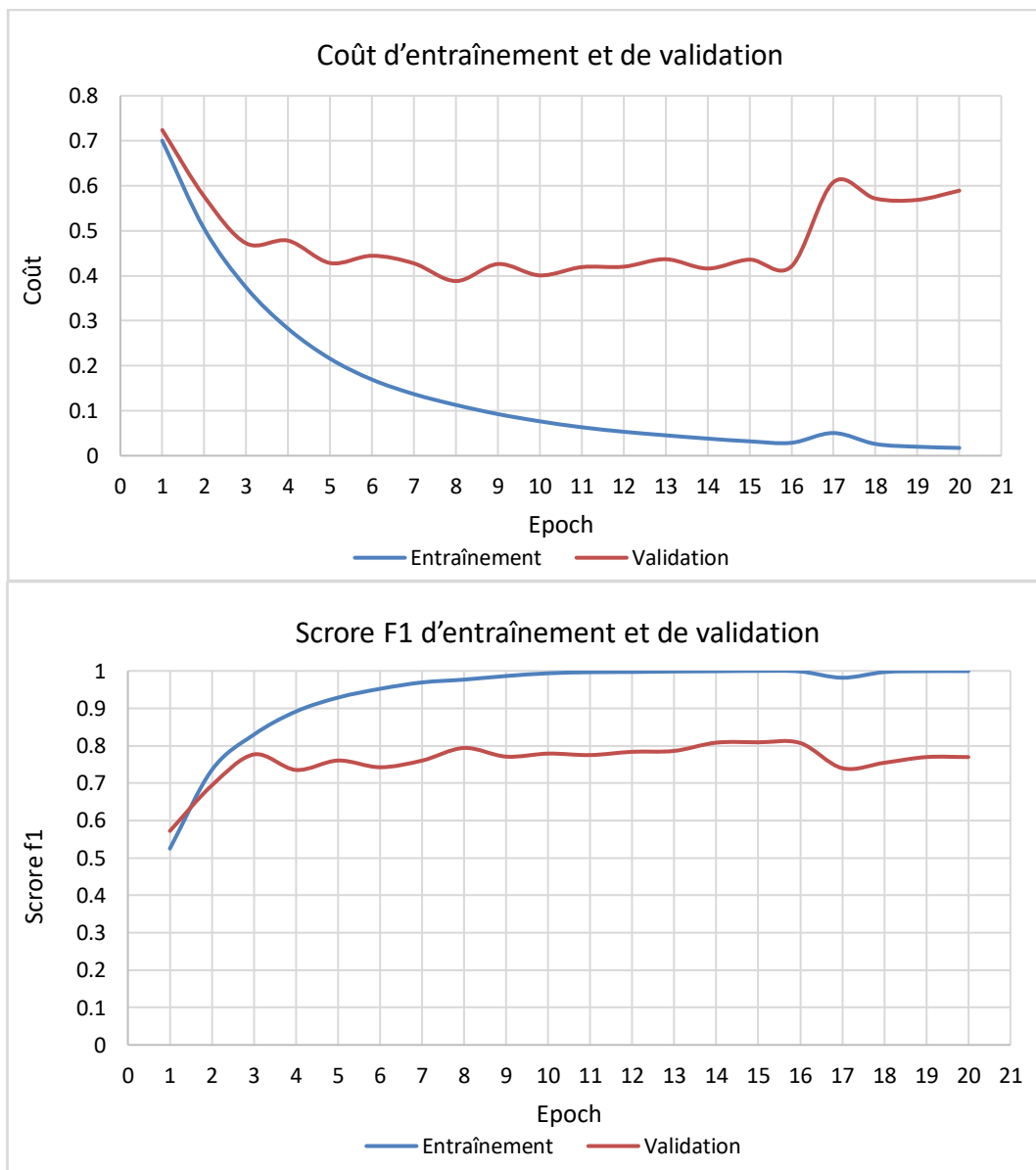


Figure 4.10 – Résultats d'entraînement et de validation du modèle 2



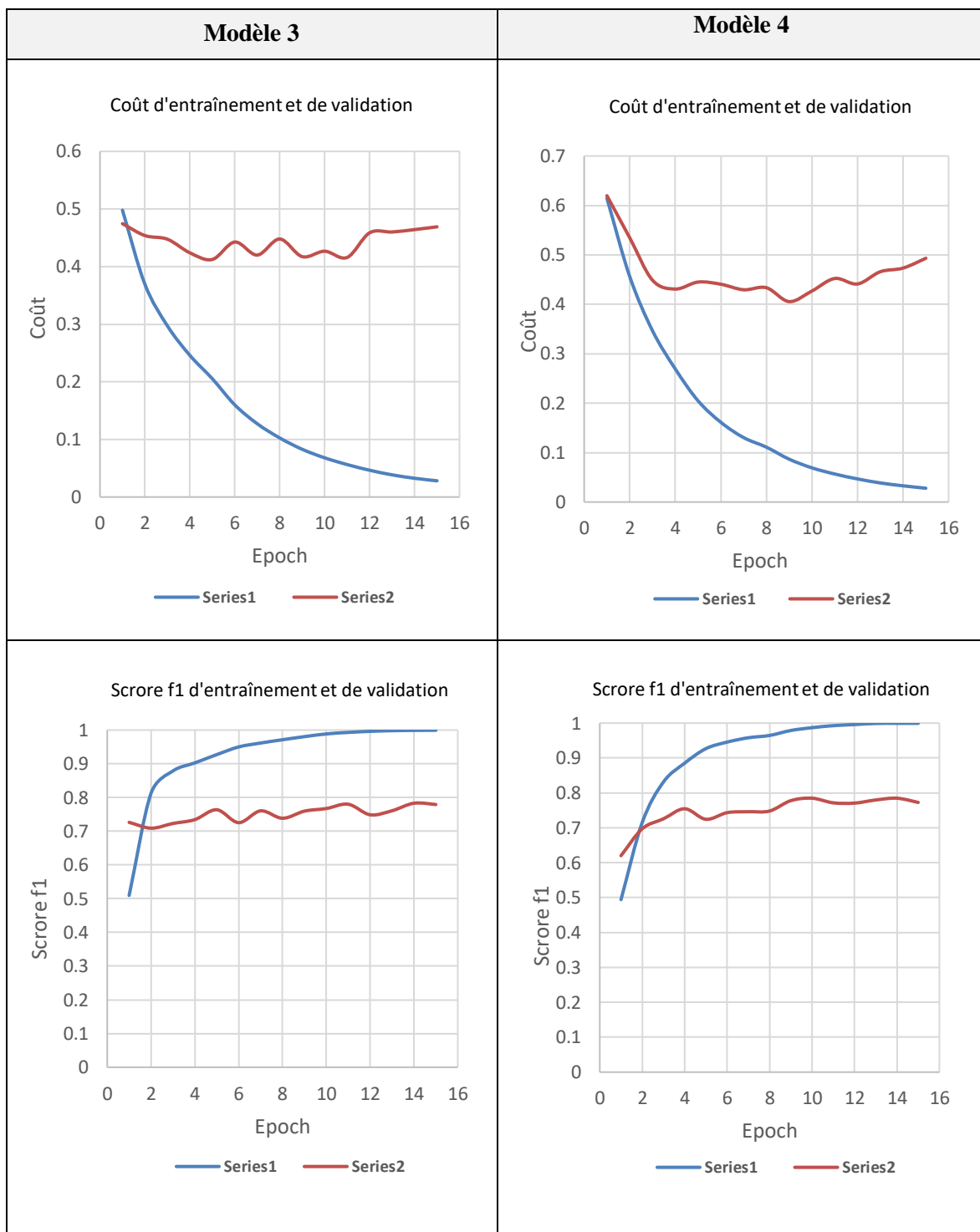


Figure 4.11 – Résultats d'entraînement et de validation du modèle 3 et 4

On passe maintenant à l'étape finale du processus, dans laquelle le meilleur modèle obtenu sera évalué sur les données de test. Le résultat obtenu est de 77.1% en score  $F1_{\text{micro}}$  et 0.16 en perte de Hamming. Pour avoir une idée détaillée sur cette performance, le tableau 4.5 montre la capacité du modèle dans la détection de chacune des trois classes.

Tableau 4.5 – Les résultats du modèle 2 (meilleur modèle) sur les données de test

	Précision	Rappel	Score $F1_{\text{micro}}$
Perfectif	79%	82%	80%
Correctif	88%	81%	84%
Adaptatif	62%	59%	60%
	77.3%	76.9%	77.1%

À partir du tableau ci-dessus, nous avons obtenu de bons résultats pour deux des trois types d'activités de maintenance (perfectif et correctif). En revanche, le modèle performe moins dans la détection des changements adaptatifs. C'est ce qui explique d'ailleurs la baisse dans la moyenne micro du score F1 sur les données de test par rapport à celle sur les données de validation.

D'autre part, nous avons choisi de tester notre modèle sur le jeu de données de Mauczka et al. (2015) afin de comparer nos résultats avec ceux de l'étude de Sarwar et al. (2020) qui ont suivi eux aussi une approche de classification multiétiquettes. Le tableau 4.6 présente les résultats trouvés, où on constate une amélioration de 4% en score F1 par notre modèle.

Tableau 4.6 – Comparaison entre notre modèle et celui de Sarwar et al. (2020) par rapport au jeu de données de Mauczka et al. (2015)

Modèle	Précision	Rappel	Score $F1_{\text{micro}}$	Perte de Hamming
Sarwar et al. (2020)	48%	52%	50%	0.31
Notre modèle	55%	54%	54%	0.31

## 4.6.2 Discussion et proposition de travaux futurs

À partir des résultats obtenus, on peut voir comment l'exploitation de l'information concernant le titre de la tâche à faire, en plus du message du commit a permis d'améliorer le processus de classification par activités de maintenance pour le modèle 1. Et ce, rappelons-le, du fait que les développeurs ont tendance à ne décrire dans le message du commit que les modifications appliquées au code, et à ne mentionner la raison derrière ce changement qu'à travers un identifiant (#890, par exemple) qui fait référence à la tâche à faire. C'est le cas pour le commit « A » (voir figure 4.12) dont le message est « *Handle different sep across os* » et le titre de la proposition de changement est « *Catch-all route doesn't work when static\_folder ends in forwardslash* » (voir figure 4.13). Ce commit a été classifié comme étant perfectif par le modèle 1 en se basant uniquement sur le message et comme étant correctif par le même modèle, mais en tenant compte du titre de la proposition de changement. Cela est dû au fait que le message ne fait aucune référence à une correction contrairement à la proposition de changement qui montre qu'il s'agit d'un dysfonctionnement dans le code. Le même constat s'applique aussi pour le commit « B » (voir figure 4.14 et 4.15). En effet, son message est « *safe\_join on Windows uses posixpath* » et le titre de la proposition de changement est « *failing test for safe\_join in windows* » (voir figure 4.16). En tenant compte de cette dernière information qui fait référence à un bogue dans l'implémentation de la fonction « *safe\_join* », le modèle 1 exploitant aussi la proposition de changement a pu le classifier correctement.

Ensuite, nous avons vu comment l'ajout des trois caractéristiques qui sont le nombre de LOC ajoutées et supprimées et le nombre de fichiers modifiés a permis d'améliorer encore plus les résultats. Par exemple le commit « C » (voir figure 4.17) dont son message est « *Added a testcase for 404 errors caused by the routing system* » et ayant 12 LOC ajoutées dans un seul fichier. Ce commit est perfectif puisque l'ajout d'une fonction « *test\_before\_request\_and\_routing\_errors* » représente l'implémentation d'un cas de test. Cependant, il a été classifié par le modèle 1 comme étant correctif, probablement parce que le mot « *errors* » lié souvent à un dysfonctionnement a eu plus d'influence lors de la

classification que l'expression « *added a testcase* ». Mais avec l'exploitation des informations quantitatives liées au code modifié, le modèle 2 a pu classifier correctement ce commit (12 LOC ajoutées et aucune ligne supprimée). Cette information, qui montre l'ajout d'un nouveau code uniquement, ne caractérise pas la majorité des changements correctifs qui nécessitent souvent une modification du code existant impliquant à la fois des LOC ajoutées et supprimées. Le commit perfectif « D » (voir figure 4.18 et 4.19) est un autre exemple. En effet, son message est « *Add postcss-preset-env, remove autoprefixer* » et il a 3 fichiers modifiés avec 11 LOC ajoutées et 7 LOC supprimées, a été correctement classifié par le modèle 2 contrairement au modèle 1 qui l'a considéré comme étant adaptatif. D'après son message, le changement peut être perfectif ou bien adaptatif (ou les deux). Mais en se basant également sur les données quantitatives liées au code modifié, il est très peu probable que le changement soit adaptatif étant donné que ce type de changement implique souvent des modifications de grandes tailles.

**Handle different sep across os**

main (#3456)  
2.3.2 ... 1.1.2

frostming committed on Feb 12, 2020 Unverified

Showing 1 changed file with 3 additions and 1 deletion.

```

src/flask/helpers.py
@@ -1000,6 +1000,8 @@ def static_folder(self):
1000 1000
1001 1001     @static_folder.setter
1002 1002     def static_folder(self, value):
1003 +         if value is not None:
1004 +             value = value.rstrip("/\\")
1003 1005         self._static_folder = value
1004 1006
1005 1007     @property
@@ -1013,7 +1015,7 @@ def static_url_path(self):
1013 1015         return self._static_url_path
1014 1016
1015 1017         if self.static_folder is not None:
1016 -             basename = os.path.basename(self.static_folder.rstrip("/"))
1018 +             basename = os.path.basename(self.static_folder)
1017 1019         return ("/" + basename).rstrip("/")
1018 1020
1019 1021     @static_url_path.setter


```

Figure 4.12 – Le commit « A ». *Source* : <https://github.com/pallets/flask/commit/e6178fe489b7828acc2bb8fd4b56a70b11ab6c6a>

## Catch-all route doesn't work when static\_folder ends in forwardslash #3452

New issue

 Closed Stj2nd opened this issue on Dec 19, 2019 · 2 comments · Fixed by #3456

 Stj2nd commented on Dec 19, 2019 · edited

I am trying to serve a React app with Flask. The React app uses React Router which means some routes should simply be handled by React. To achieve this one can usually use a catch-all route that returns the index.html. I first couldn't get the catch-all route to work at all, and then I got it working by removing the trailing slash for the `static_folder` parameter when calling Flask.

### Expected Behavior

When one enter a path with trailing forward slash in the `static_folder` parameter for flask.Flask() it shouldn't affect whether a catch-all route goes into effect or not.

In the below code remove the trailing slash for the `static_folder` and you are able to reach the catch-all route (home function) by entering a random route like `/something`.

```
from flask import Flask, render_template

app = Flask(__name__, static_folder="frontend/build/", template_folder="frontend/build")

@app.route("/")
def route1():
    return "route1"

@app.route("/<path:path>")
def home(path):
    return "routeAny"
```

### Actual Behavior

Steps to reproduce

1. Run the above flask app
2. Open browser and verify that it renders the text "route1" for route `/`
3. Browse to `/something` and see that you get 404 Not Found.

### Environment

- Python version: 3.7.
- Flask version: 1.1.1
- Werkzeug version: 0.16.0

Assignees

No one assigned

Labels

None yet

Projects


None yet

Milestone

1.1.2

Development

Successfully merging a pull request may close this issue.

 Fix a bug that the static endpoint will unexpecte...

2 participants




Figure 4.13 – La proposition de changement implémentée par le commit « A ».  
Source : <https://github.com/pallets/flask/issues/3452>

safe\_join on Windows uses posixpath

fixes #2033  
closes #2059

main  
2.3.2 0.12.2

dauidism authored and unitaker committed on May 16, 2017

Showing 3 changed files with 33 additions and 21 deletions.

Filter changed files

- CHANGES
- flask
  - helpers.py
  - tests
    - test\_helpers.py

```

@@ -15,6 +15,13 @@ Major release, unreleased
15 15     method returns compressed response by default, and pretty response in
16 16     debug mode.
17 17
18 + Version 0.12.2
19 + -----
20 +
21 + Bugfix release
22 +
23 + - Fix a bug in `safe_join` on Windows.
24 +
18 25     Version 0.12.1
19 26     -----
20 27

```

```

@@ -619,18 +619,24 @@ def wiki_page(filename):
619 619     :raises: :class:`~werkzeug.exceptions.NotFound` if one or more passed
620 620         paths fall out of its boundaries.
621 621     """
622 +
623 +     parts = [directory]
624 +
625     for filename in pathnames:
626         if filename != '':
627             filename = posixpath.normpath(filename)
628 -         for sep in _os_alt_seps:
629 -             if sep in filename:
630 -                 raise NotFound()
631 -         if os.path.isabs(filename) or \
632 -             filename == '..' or \
633 -             filename.startswith('../'):
634 +
635 +         if {
636 +             any(sep in filename for sep in _os_alt_seps)

```

Figure 4.14 – Le commit « B ». Source : <https://github.com/pallets/flask/commit/f7c35bf0d51d1ae02709e39fe29110e12f64fb87>

Showing 3 changed files with 33 additions and 21 deletions.

Filter changed files

CHANGES

flask

helpers.py

tests

test\_helpers.py

```
630 +         any(sep in filename for sep in _os_alt_seps)
631 +         or os.path.isabs(filename)
632 +         or filename == '..'
633 +         or filename.startswith('../')
634 +     ):
631 635         raise NotFound()
632 -     directory = os.path.join(directory, filename)
633 -     return directory
636 +
637 +     parts.append(filename)
638 +
639 +     return posixpath.join(*parts)
634 640
635 641
636 642 def send_from_directory(directory, filename, **options):
```

```
tests/test_helpers.py
@@ -846,21 +846,20 @@ def generate():
846 846
847 847
848 848 class TestSafeJoin(object):
849 -
850 849 def test_safe_join(self):
851 850     # Valid combinations of *args and expected joined paths.
852 851     passing = (
853 -         (('a/b/c',), 'a/b/c'),
854 -         (('/', 'a/', 'b/', 'c/'), '/a/b/c/'),
855 -         (('a', 'b', 'c'), 'a/b/c'),
856 -         (('a', 'b/c'), '/a/b/c'),
857 -         (('a/b', 'X/./c'), 'a/b/c'),
858 -         (('a/b', 'c/X/.'), 'a/b/c'),
852 +         (('a/b/c',), 'a/b/c'),
853 +         (('/', 'a/', 'b/', 'c/'), '/a/b/c/'),
854 +         (('a', 'b', 'c'), 'a/b/c'),
855 +         (('a', 'b/c'), '/a/b/c'),
856 +         (('a/b', 'X/./c'), 'a/b/c'),
857 +         (('a/b', 'c/X/.'), 'a/b/c'),
859 858     # If last path is '' add a slash
860 -         (('a/b/c', ''), '/a/b/c/'),
859 +         (('a/b/c', ''), '/a/b/c/'),
861 860     # Preserve dot slash
862 -         (('a/b/c', './'), 'a/b/c/'),
863 -         (('a/b/c', 'X/.'), 'a/b/c'),
861 +         (('a/b/c', './'), 'a/b/c/'),
862 +         (('a/b/c', 'X/.'), 'a/b/c/'),
864 863     # Base directory is always considered safe
865 864         (('..', 'a/b/c', './a/b/c'),
866 865         (('..',), '..'),
@@ -874,12 +873,12 @@ def test_safe_join_exceptions(self):
874 873     failing = (
875 874         # path.isabs and '..' checks
876 875         ('/a', 'b', 'c'),
877 -         ('/a', './b/c'),
876 +         ('/a', './b/c'),
878 877         ('/a', '..', 'b/c'),
879 878         # Boundaries violations after path normalization
880 -         ('/a', 'b/./b/././c/'),
879 +         ('/a', 'b/./b/././c/'),
881 880         ('/a', 'b', 'c/././'),
882 -         ('/a', 'b/././c/'),
881 +         ('/a', 'b/././c/'),
883 882     )
884 883
885 884     for args in failing:
```

Figure 4.15 – La suite du commit « B ». Source : <https://github.com/pallets/flask/commit/f7c35bf0d51d1ae02709e39fe29110e12f64fb87>



## failing test for safe\_join in windows #2033



infinite-Joy opened this issue on Sep 21, 2016 · 0 comments · Fixed by #2284



infinite-Joy commented on Sep 21, 2016

This seems like a bug on windows:  
Test run on windows gave me the following error

```
> py.test tests\test_helpers.py
+[1m===== test session starts =====
====+[0m
platform win32 -- Python 2.7.10, pytest-3.0.2, py-1.4.31, pluggy-0.3.1
rootdir: , inifile: setup.cfg
collected 42 items +[0m+[1m
+[0m
tests\test_helpers.py .....F.

===== FAILURES =====
+[1m+[31m_____ TestSafeJoin.test_safe_join _____
+[0m

self = <test_helpers.TestSafeJoin object at 0x02F3F790>

+[1m def test_safe_join(self):+[0m
+[1m     # Valid combinations of *args and expected joined paths.+[0m
+[1m     passing = (+[0m
+[1m         (('a/b/c', ), 'a/b/c'),+[0m
+[1m         (('/', 'a/', 'b/', 'c/'), '/a/b/c'),+[0m
+[1m         (('a', 'b', 'c', ), 'a/b/c'),+[0m
+[1m         (('a', 'b/c', ), '/a/b/c'),+[0m
+[1m         (('a/b', 'X/./c'), 'a/b/c'),+[0m
+[1m         (('a/b', 'c/X/..'), '/a/b/c'),+[0m
+[1m         # If last path is '' add a slash+[0m
+[1m         (('a/b/c', ''), '/a/b/c/'),+[0m
+[1m         # Preserve dot slash+[0m
+[1m         (('a/b/c', './'), '/a/b/c./'),+[0m
+[1m         (('a/b/c', 'X/..'), 'a/b/c./'),+[0m
+[1m         # Base directory is always considered safe+[0m
+[1m         (('./', 'a/b/c'), '../a/b/c'),+[0m
+[1m         (('../', ), '/../'),+[0m
+[1m     )+[0m
+[1m     for args, expected in passing:+[0m
+[1m>         assert flask.safe_join(*args) == expected+[0m
+[1m+[31mE         assert '/a\\b\\c' == '/a/b/c'+[0m
+[1m+[31mE         - /a\\b\\c+[0m
+[1m+[31mE         + /a/b/c+[0m

+[1m+[31mtests\test_helpers.py+[0m:803: AssertionError
+[1m+[31m===== 1 failed, 41 passed in 1.33 seconds =====
=====+[0m
```

In windows the directories on an external location can be accessed using four back-slashes "\\" from python code. So I would suggest to return "\\a\b\c" if the env is windows. Of course this line of thinking can be naïve and maybe there is a better approach there.

Figure 4.16 – La proposition de changement implémentée par le commit « B ».

Source : <https://github.com/pallets/flask/issues/2033>

Added a testcase for 404 errors caused by the routing system

main

2.3.2 ... 0.8

 mitsuhiko committed on Aug 4, 2011

Showing 1 changed file with 12 additions and 0 deletions.

```

12 tests/flask_tests.py
@@ -564,6 +564,18 @@ def error():
564 564     assert rv.status_code == 500
565 565     assert 'internal server error' == rv.data
566 566
567 +     def test_before_request_and_routing_errors(self):
568 +         app = flask.Flask(__name__)
569 +         @app.before_request
570 +         def attach_something():
571 +             flask.g.something = 'value'
572 +         @app.errorhandler(404)
573 +         def return_something(error):
574 +             return flask.g.something, 404
575 +         rv = app.test_client().get('/')
576 +         assert rv.status_code == 404
577 +         assert rv.data == 'value'
578 +
567 579     def test_user_error_handling(self):
568 580         class MyException(Exception):
569 581             pass

```

Figure 4.17 – Le commit « C ». Source : <https://github.com/pallets/flask/commit/fafcc02f261e90a006efd5cc4dfa876531af9594>

**Add postcss-preset-env, remove autoprefixer (#4716)**

\* add postcss-preset-env, remove autoprefixer

\* limit to stage 4

main (#4716)  
v5.0.1 babel-plugin-named-asset-import@0.2.1

heygrady authored and Timer committed on Sep 25, 2018

Showing 3 changed files with 11 additions and 7 deletions.

Filter changed files

- packages/react-scripts
  - config
    - webpack.config.dev.js
    - webpack.config.prod.js
    - package.json

```

@@ -8,7 +8,6 @@
 8 8 // @remove-on-eject-end
 9 9 'use strict';
10 10
11 - const autoprefixer = require('autoprefixer');
12 11 const path = require('path');
13 12 const webpack = require('webpack');
14 13 const HtmlWebpackPlugin = require('html-webpack-plugin');

@@ -57,8 +56,11 @@ const getStyleLoaders = (cssOptions, preProcessor) => {
 57 56 ident: 'postcss',
 58 57 plugins: () => [
 59 58   require('postcss-flexbugs-fixes'),
 60 -   autoprefixer({
 61 -     flexbox: 'no-2009',
 62 +   require('postcss-preset-env')({
 63 +     autoprefixer: {
 64 +       flexbox: 'no-2009',
 65 +     },
 66 +     stage: 4,
 67   }
 68   ),
 69   ],
 70   },
 71   ],
 72   },
 73   ],
 74   ],
 75   ],
 76   ],
 77   ],
 78   ],
 79   ],
 80   ],
 81   ],
 82   ],
 83   ],
 84   ],
 85   ],
 86   ],
 87   ],
 88   ],
 89   ],
 90   ],
 91   ],
 92   ],
 93   ],
 94   ],
 95   ],
 96   ],
 97   ],
 98   ],
 99   ],
100  ],

```

Figure 4.18 – Le commit « D ». Source : <https://github.com/facebook/create-react-app/commit/fb6e6f70a312247c10d2ad8db0e6dc56cbc6302f>

Showing 3 changed files with 11 additions and 7 deletions.

Filter changed files

- packages/react-scripts
  - config
    - webpack.config.dev.js
    - webpack.config.prod.js
    - package.json

```
packages/react-scripts/config/webpack.config.prod.js
@@ -8,7 +8,6 @@
8 8 // @remove-on-eject-end
9 9 'use strict';
10 10
11 - const autoprefixer = require('autoprefixer');
12 11 const path = require('path');
13 12 const webpack = require('webpack');
14 13 const HtmlWebpackPlugin = require('html-webpack-plugin');

@@ -69,8 +68,11 @@ const getStyleLoaders = (cssOptions, preProcessor) => {
69 68 ident: 'postcss',
70 69 plugins: () => [
71 70 require('postcss-flexbugs-fixes'),
72 - autoprefixer({
73 - flexbox: 'no-2009',
74 + require('postcss-preset-env')({
75 + autoprefixer: {
76 + flexbox: 'no-2009',
77 + },
78 + stage: 4,
79 76 }]),
80 77 ],
81 78 sourceMap: shouldUseSourceMap,

packages/react-scripts/package.json
@@ -24,7 +24,6 @@
24 24 "@babel/core": "7.1.0",
25 25 "@babel/runtime": "7.0.0",
26 26 "@svgr/webpack": "2.4.1",
27 - "autoprefixer": "9.1.5",
28 27 "babel-core": "7.0.0-bridge.0",
29 28 "babel-eslint": "9.0.0",
30 29 "babel-jest": "23.6.0",

@@ -55,6 +54,7 @@
55 54 "optimize-css-assets-webpack-plugin": "5.0.1",
56 55 "postcss-flexbugs-fixes": "4.1.0",
57 56 "postcss-loader": "3.0.0",
58 + "postcss-preset-env": "6.0.6",
59 58 "postcss-safe-parser": "4.0.1",
60 59 "promise": "8.0.2",
61 60 "raf": "3.4.0",
```

Figure 4.19 – La suite du commit « D ». Source : <https://github.com/facebook/create-react-app/commit/fb6e6f70a312247c10d2ad8db0e6dc56cbc6302f>

En termes de généralisation du modèle final, nous avons pu améliorer de 4% le score  $F1_{\text{micro}}$  obtenu par Sarwar et al. (2020) sur le jeu de données de Mauczka et al. (2015). Dans le futur, nous pensons à tester, d'une part, une architecture concaténant les deux vecteurs de plongements résultants des modèles de plongement de mots et de phrases, et d'autre part, nous envisageons d'exploiter d'autres informations qui nous étaient utiles lors de l'étiquetage manuel comme le nom des fichiers (ou dossiers) modifiés, la description détaillée de la tâche à faire, les conversations entre les collaborateurs lors d'un pull-request, etc.

D'autre part, nous avons vu comment la prédiction du type de changements adaptatifs a pénalisé la moyenne micro du score F1 lors de la phase de test. Ceci nous mène à penser que les données de test contiennent des commits adaptatifs ayant certains aspects ou caractéristiques qui ne sont pas présents dans ceux constituant les données d'entraînement. Deux solutions peuvent être envisagées dans le futur : utiliser la validation croisée pour inclure également les données de validation dans le processus d'entraînement, et chercher à augmenter le nombre de commits adaptatifs dans notre jeu de données afin d'assurer une meilleure diversification de cette classe dans les trois ensembles de jeu de données. Il faut aussi faire, en parallèle, une étude approfondie sur ce type d'activité de maintenance afin de bien comprendre encore plus sur les traits spécifiques à ce type de changement et de les exploiter.

En ce qui concerne les représentations distribuées générées par CC2Vec, nous avons remarqué qu'elles n'avaient contribué à aucune amélioration des résultats. Soit qu'ils ne sont pas utiles dans le contexte de la classification par activité de maintenance, soit que notre jeu de données n'est pas assez grand pour pouvoir extraire les meilleures représentations et donc qu'il faut envisager l'utilisation d'environ 82 000 commits comme c'est le cas avec l'expérimentation faite sur PatchNet (Hoang et al. 2020). Cette partie de ce travail est faite, à la base, à titre exploratif, mais elle nous sera utile dans un autre sujet relié dans lequel nous serons amenés à transformer les LOC modifiées en des représentations distribuées afin d'effectuer du regroupement (*clustering*, en anglais). Ce sujet concernera le démêlage des commits enchevêtrés contenant plus d'une seule activité de maintenance.

## 4.7 SYNTHÈSE

Ce dernier chapitre du mémoire a été consacré à la description de notre approche ainsi que de son implémentation. Nous avons d'abord détaillé le protocole suivi pour la construction de notre jeu de données, et particulièrement le processus d'étiquetage. Ensuite en ayant des classes asymétriques, nous avons expliqué le choix de la méthode adoptée pour faire face à cette contrainte. Finalement, nous avons présenté les résultats obtenus pour les quatre modèles de notre approche, et nous avons vu comment le modèle exploitant le message du commit, le titre de la tâche à faire ainsi que l'information sur le nombre de LOC ajoutées et supprimées et le nombre de fichiers modifiés, a permis d'atteindre le meilleur score.

## CONCLUSION GÉNÉRALE

Avec l'avènement de nouvelles approches de développement logiciel comme le DevOps, la manière avec laquelle les développeurs travaillent a connu un changement important. On s'est donc retrouvé avec un environnement où les changements de code sont devenus de plus en plus fréquents, et dans lesquels les tâches de planification, d'implémentation, de test et de mise en production se font conjointement durant le développement d'un logiciel. Ceci amène à trouver un moyen d'assurer un bon suivi de ces opérations effectuées sur le code source. De ce fait, nous avons présenté une approche de classification multiétiquettes des commits par types d'activité de maintenance (perfectif, correctif et adaptatif). Notre approche est basée sur de l'apprentissage profond, et principalement sur de l'apprentissage par transfert. En effet, nous avons utilisé le modèle pré-entraîné USE pour exploiter les données textuelles d'un commit, le modèle CC2Vec pour générer des représentations distribuées du code modifié, et un classificateur sous forme de couches entièrement connectées.

Dans ce mémoire, nous avons commencé par décrire le contexte de notre sujet en abordant différentes notions liées à la maintenance logicielle. Nous avons d'abord défini les différents types d'activité de maintenance, les systèmes de contrôle de versions et le processus de changement de code, en évoquant notamment la culture DevOps. Ensuite, comme nous étions menés à exploiter des données textuelles, nous avons parlé des techniques utilisées pour traiter des données séquentielles, et puis nous avons vu à la fin comment les modèles à base de transformateur sont considérés aujourd'hui comme référence en termes de performance. Par la suite, nous avons commencé par un examen des études faites dans ce sujet, pour finalement proposer notre approche qui consiste sur une évaluation de quatre modèles dont la différence réside dans l'information exploitée à partir du commit.

Avant d'aborder l'entraînement des modèles, nous avons contribué à travers ce travail dans la construction d'un nouveau jeu de données (1857 commits), et ce, dû à leur rareté dans ce sujet de recherche. Nous avons défini tout un protocole afin de réussir la tenue de ce

processus manuel, et de garantir une certaine fiabilité pour ce jeu de données. Les étapes suivies consistaient dans l'étude de la structure des projets exploités (« *Create-React-App* » et « *Flask* »), le développement d'une application Web pour faciliter l'étiquetage, la définition de quelques règles à suivre durant l'étiquetage pour certains cas de changement, et une dernière étape de validation des étiquettes par les participants, vu que chaque commit a été étiquetée par trois personnes. Ensuite, nous avons vu comment la distribution des classes de notre jeu de données était asymétrique, et donc nous avons opté pour la méthode d'équilibrage des poids pour régler ce problème.

Après avoir évalué les quatre modèles de notre approche, nous avons remarqué une amélioration dans le score  $F1_{\text{micro}}$  en incluant le titre de la tâche à faire avec le message du commit, ainsi que les trois caractéristiques qui sont le nombre de LOC ajoutées et supprimées, et le nombre de fichiers modifiés. En contrepartie, nous avons remarqué que les représentations distribuées générées par CC2Vec n'ont contribué à aucune amélioration. Ceci nous amène à penser dans le futur à entraîner CC2Vec avec un jeu de données beaucoup plus large avant de pouvoir trancher définitivement sur son utilité dans notre contexte. Finalement lors de la phase de test, nous avons obtenu un score F1 de 80%, 84% et 60%, respectivement, pour nos trois classes : perfective, corrective et adaptative. Avec une moyenne-micro du score F1 de 77.1% et une perte de Hamming de 0.16. Ces résultats sont satisfaisants pour les deux classes perfectives et correctives, mais le sont moins pour la classe adaptative. Ceci nous mène à se focaliser beaucoup plus sur ce type d'activité de maintenance dans les travaux futurs, afin de trouver de nouvelles caractéristiques qui le distingue par rapport aux autres classes. Nous avons également testé notre modèle final sur le jeu de données de Mauczka et al. (2015), et nous avons remarqué une amélioration de 4% en score  $F1_{\text{micro}}$ , par rapport au score obtenu par le modèle Sarwar et al. (2020).



## ANNEXE A – STRUCTURE DES DEUX PROJETS UTILISÉS DANS NOTRE JEU DE DONNÉES

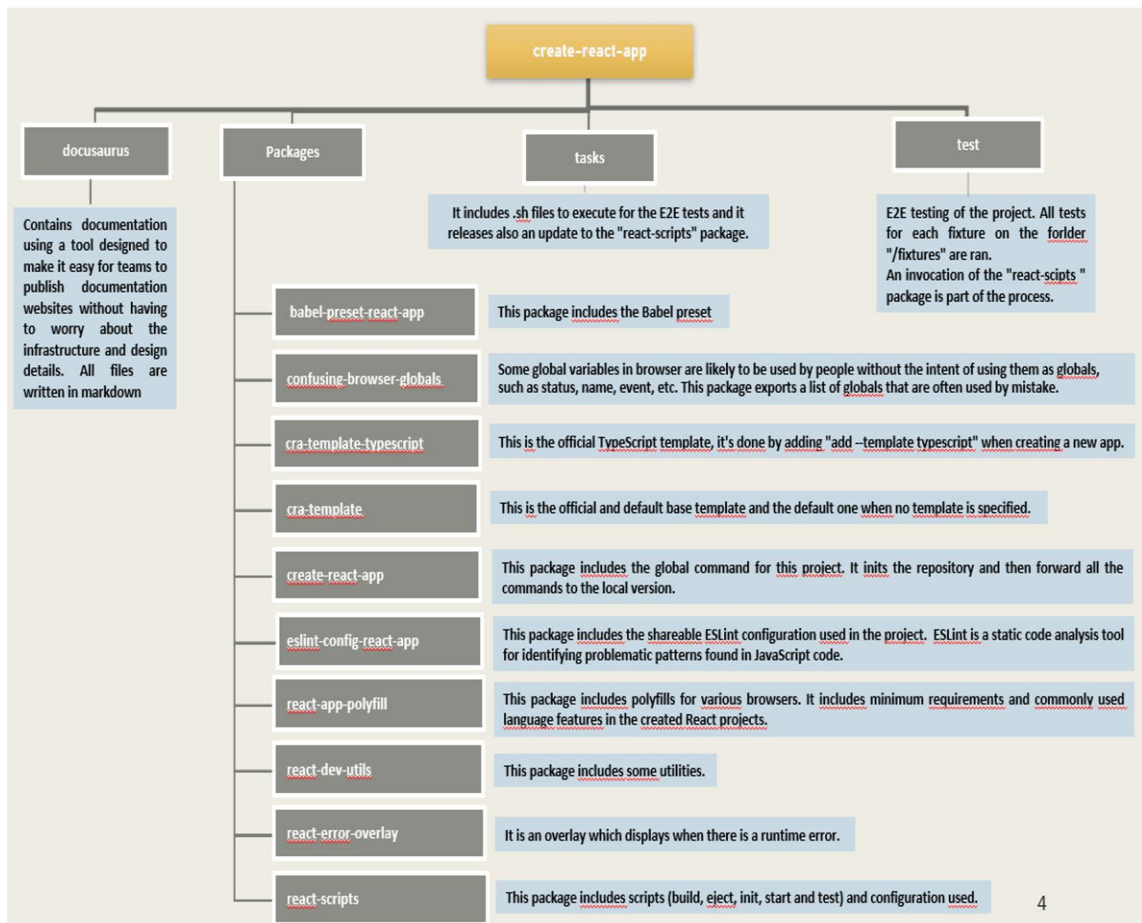


Figure A.1 – Principaux dossiers et fichiers du projet « Create-React-App » avec leur rôle.

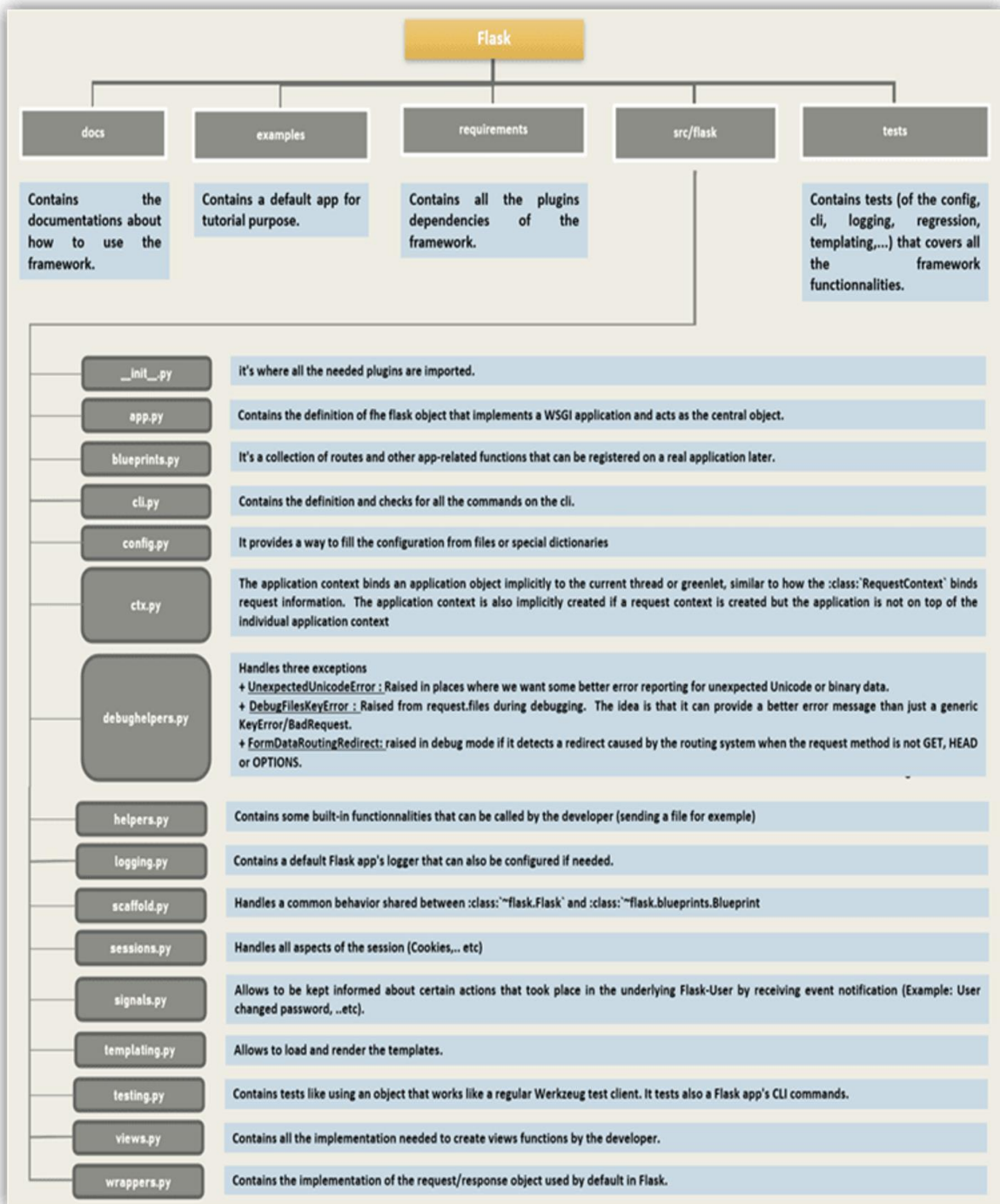
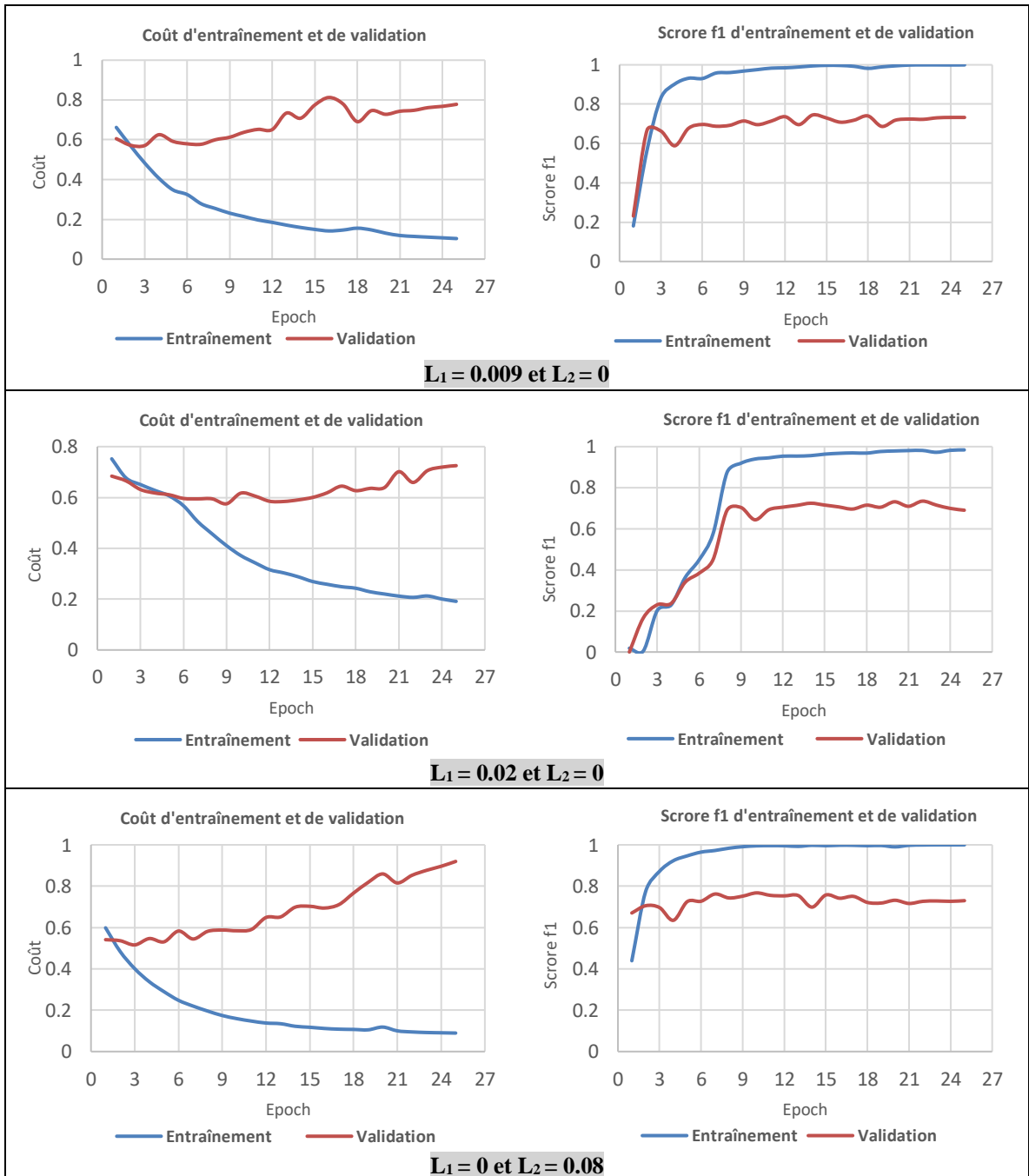
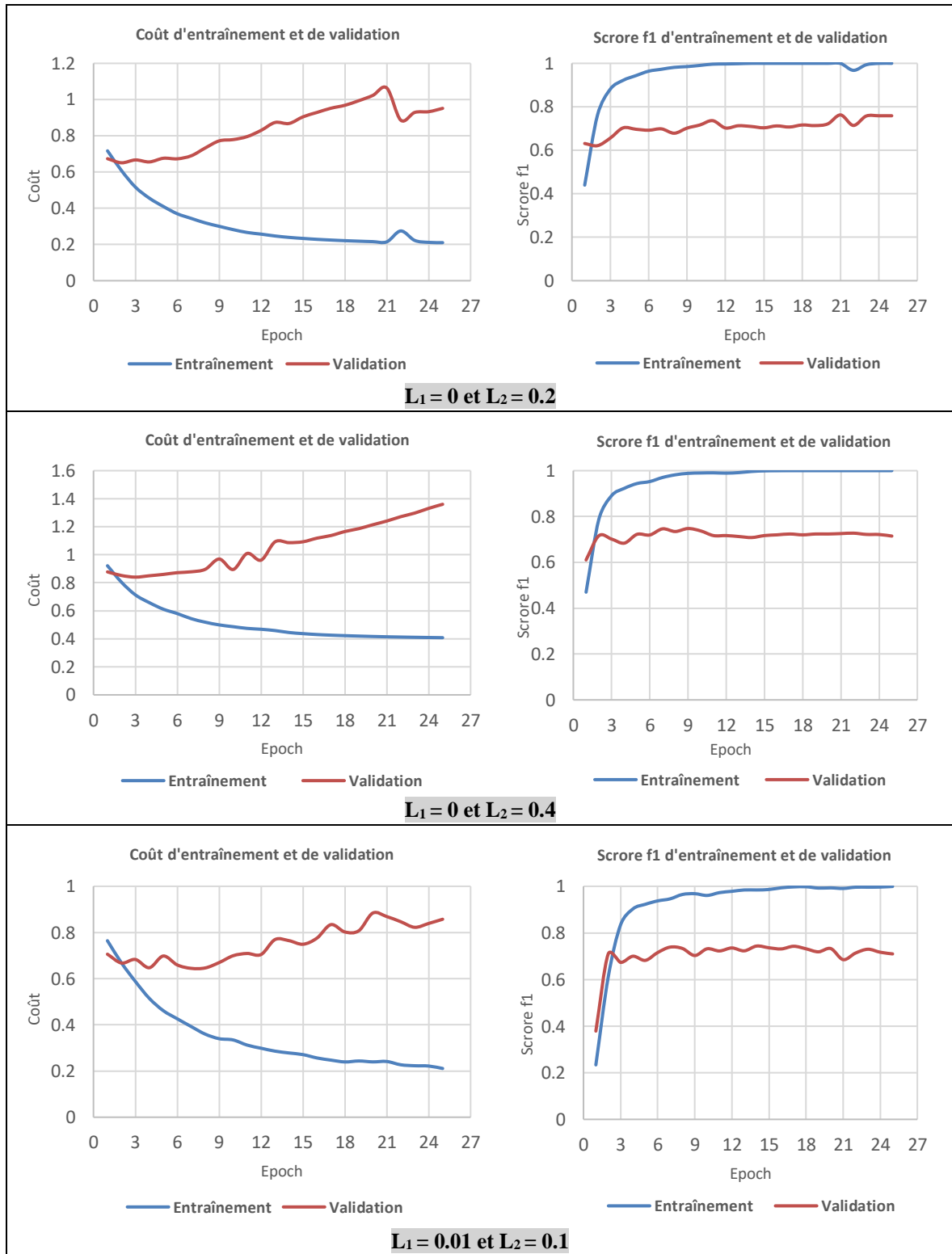


Figure A.2 – Principaux dossiers et fichiers du projet « Flask » avec leur rôle

## ANNEXE B

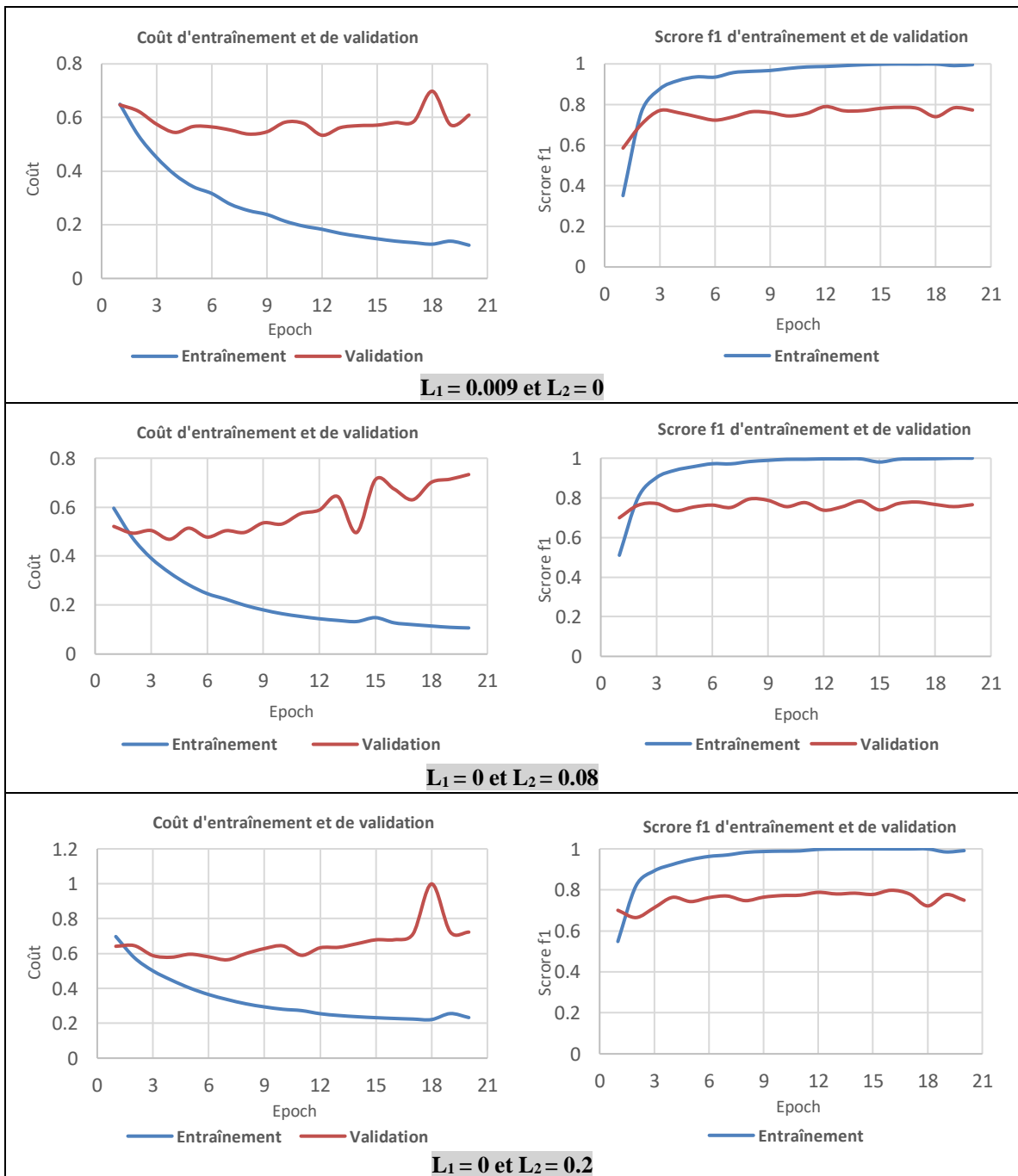
Cette annexe concerne la phase de régularisation du modèle 1 prenant comme entrée le message du commit. Les graphiques ci-dessous représentent les résultats des différentes valeurs testées pour les pénalités de normes de paramètre  $L_1$  et  $L_2$ .

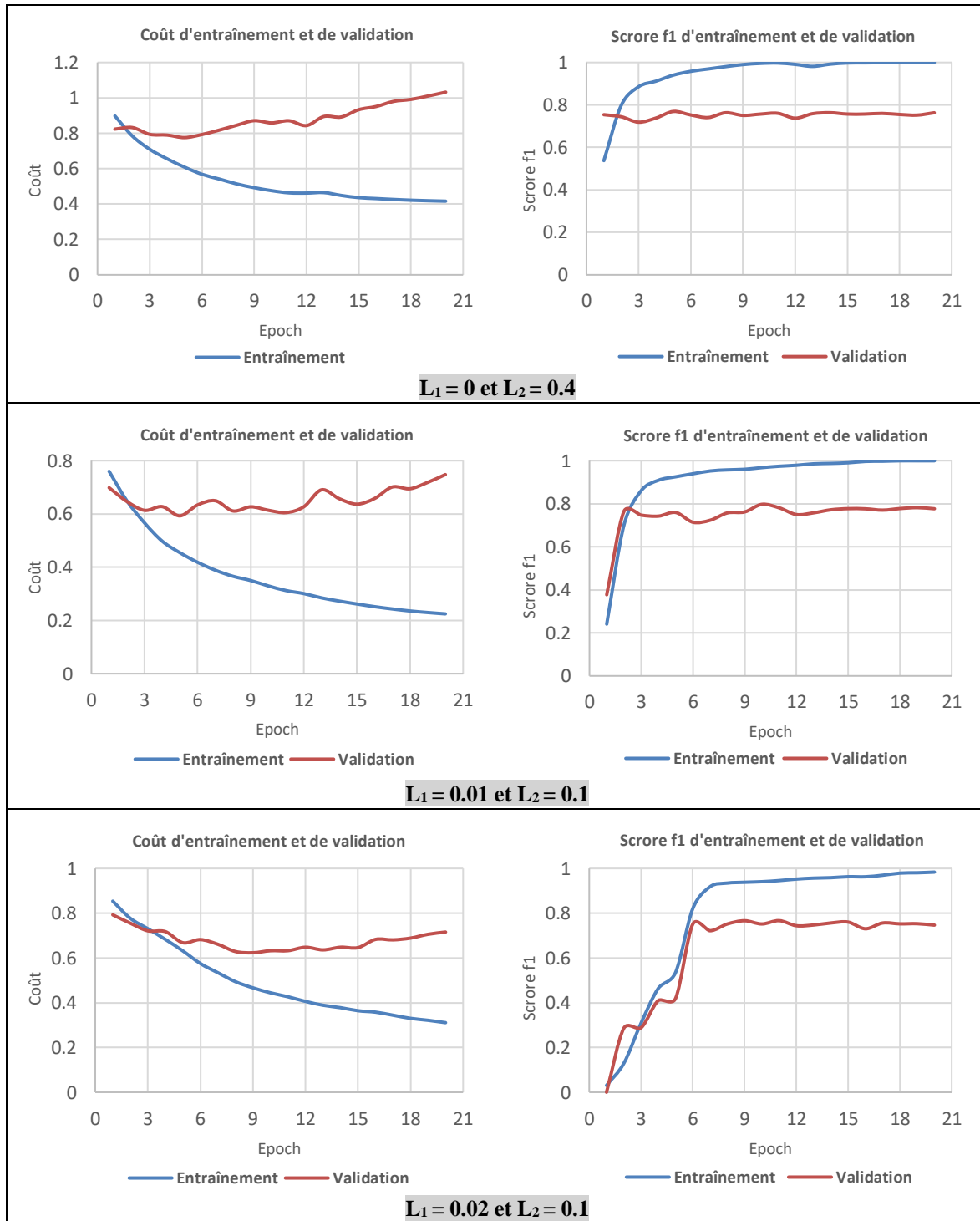




## ANNEXE C

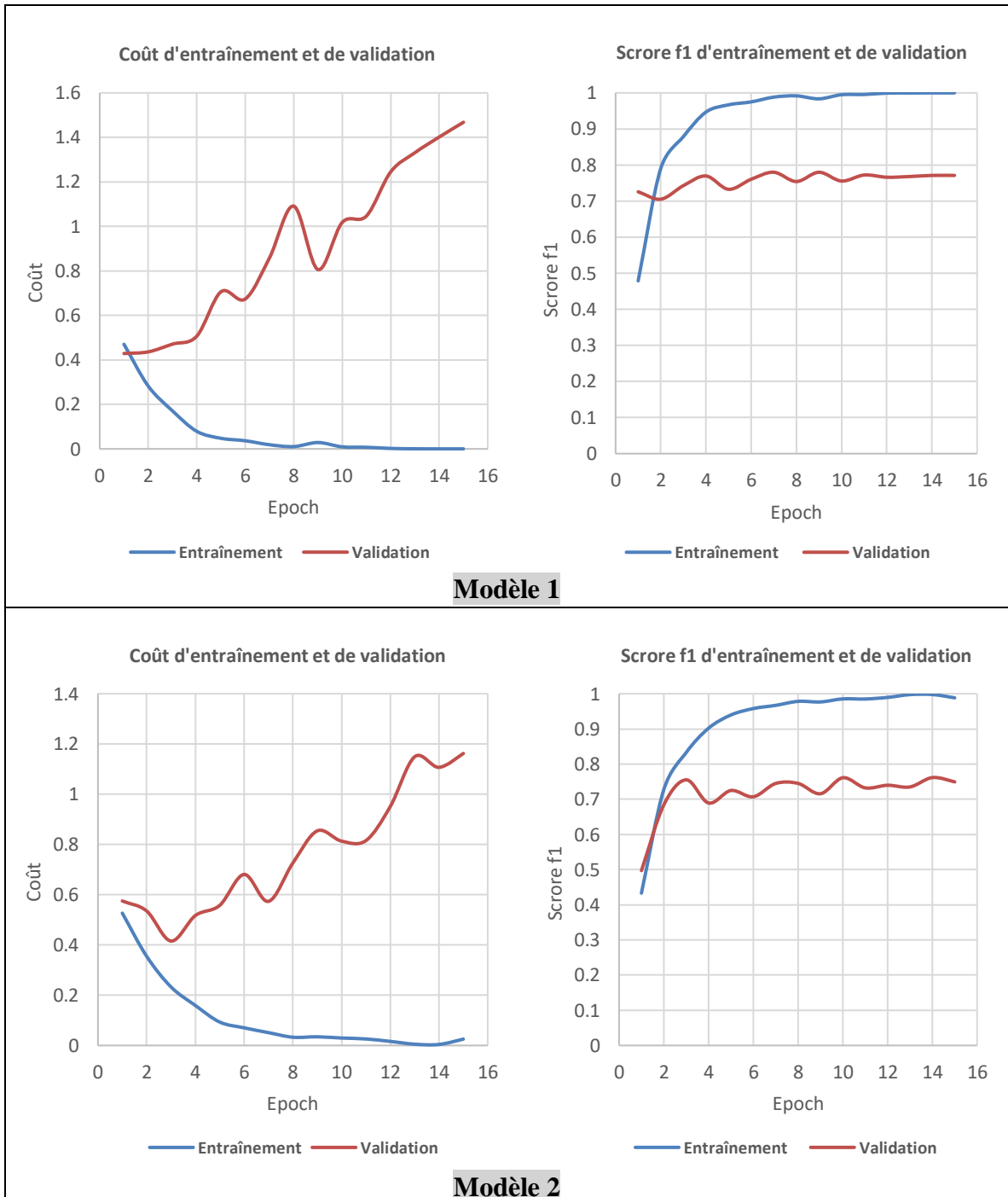
Cette annexe concerne la phase de régularisation du modèle 1 prenant également comme entrée le titre de la proposition de changement. Les graphiques ci-dessous représentent les résultats des différentes valeurs testées pour les pénalités de normes de paramètre  $L_1$  et  $L_2$ .

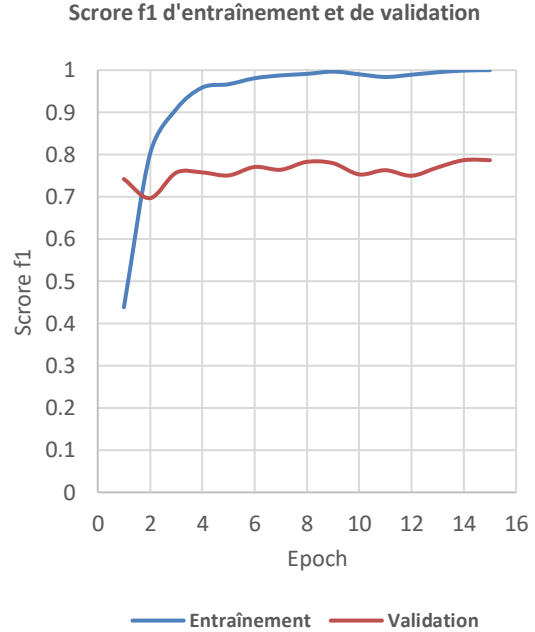
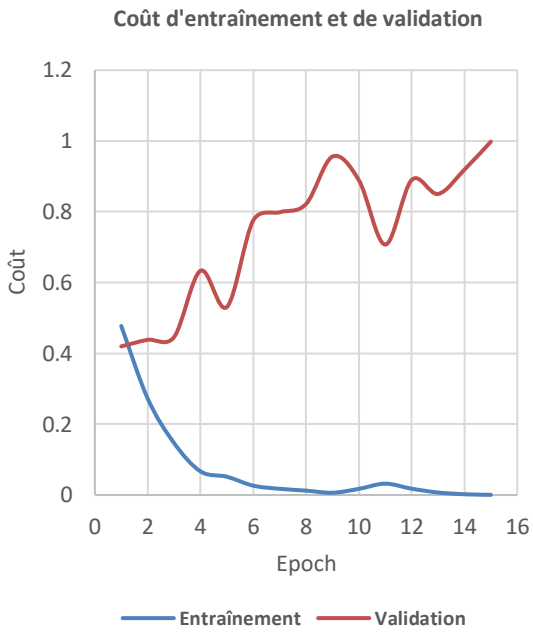




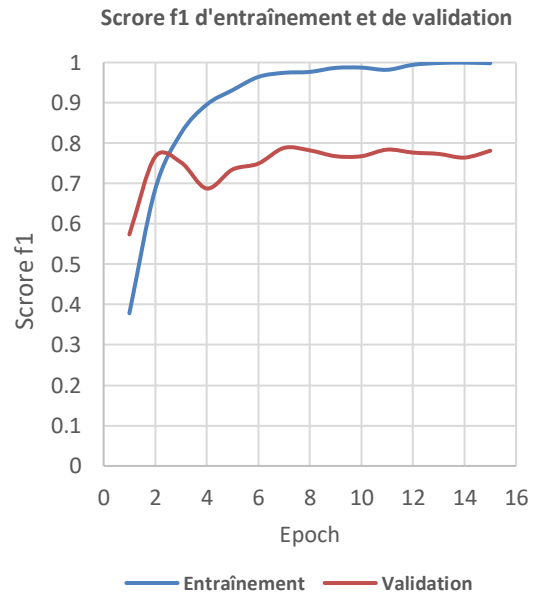
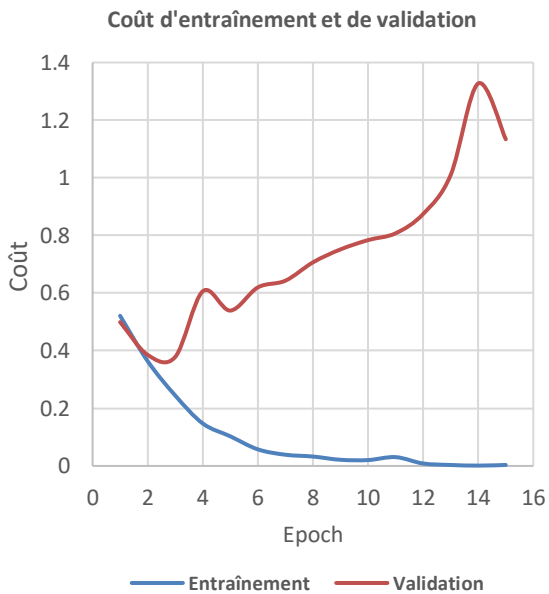
## ANNEXE D

Cette annexe concerne les résultats d'entraînement et de validation pour les quatre modèles en utilisant un classificateur de 5 couches.





**Modèle 3**



**Modèle 4**



## ANNEXE E – LE CODE DU PROGRAMME

```
1 def read_args():
2     parser = argparse.ArgumentParser()
3
4     parser.add_argument('-dataset', type=str, default="notre dataset", help='dataset to use')
5     parser.add_argument('-model', type=int, default=1, help="Depends on the commit's informations used")
6
7     # Hyperparameters values
8     parser.add_argument('-epochs', type=int, default=15, help='the number of epochs')
9     parser.add_argument('-batch', type=int, default=4, help='the batch size')
10    parser.add_argument('-lr', type=float, default=0.00008, help='learning rate')
11    parser.add_argument('-layer_num', type=int, default=0, help='Layer fine-tuning')
12    parser.add_argument('-l1', type=float, default=0, help='l1 regulazition')
13    parser.add_argument('-l2', type=float, default=0, help='l2 regulazition')
14
15    # Scripts config
16    parser.add_argument('-train', action='store_true', help='training phase')
17    parser.add_argument('-test', action='store_true', help='testing phase')
18    parser.add_argument('-load_model', type=str, default=None, help='trained model file')
19    parser.add_argument('-result_file', type=str, default="results", help= File that contains the results')
20    parser.add_argument('-pre_processing', action='store_true', help='pre-processing phase')
21
22    return parser
```

Figure E.1 – La configuration des scripts lancés pour exécuter les entraînements et tests

```
1 def load_commits(rootdir, args):
2     commits, features, code_vectors, labels = [], [], [], []
3     vectors = pickle.load(open('code-change-vector', 'rb'))
4     for 129ilename in os.listdir(rootdir):
5         text = ''
6         data = [0, 0, 0] #[files_modified, LOC_added, LOC_deleted]
7         with open(os.path.join(rootdir, 129ilename), "r", encoding="utf-8") as file:
8             for I, line in enumerate(file, start=1):
9                 tks = unicodedata.normalize("NFKD", line.strip()).split("|")
10                if tks[0] == '→': data[0] += 1
11                if tks[0] == '+': data[1] += 1
12                if tks[0] == '-': data[2] += 1
13                if I == 1:
14                    if tks[1] in vectors:
15                        code_vectors.append(vectors[tks[1]])
16                    else:
17                        code_vectors.append(np.zeros((84,)))
18                if tks[0] in ['message', 'description', 'issue']: text += '. ' + tks[1]
19                if tks[0] == 'label':
20                    two_labels = tks[1].split("/")
21                    am = [0, 0, 0]
22                    for index, lbl in enumerate(["perfectif", "correctif", "adaptif"], start=0):
23                        if lbl in two_labels[1]: am[index] = 1
24                    labels.append(am)
25                commits.append(text)
26                features.append([x/1 for x in data]) #data
27    if args.model == 2: training_data = [np.array(commits), np.array(features)]
28    elif args.model == 3: training_data = [np.array(commits), np.array(code_vectors)]
29    elif args.model == 4: training_data = [np.array(commits), np.array(features), np.array(code_vectors)]
30    else: training_data = np.array(commits)
31    return training_data, np.array(labels)
```

Figure E.2 – La préparation des données

```

1 def create_model_1(args):
2     use_object = hub.load("https://tfhub.dev/google/universal-sentence-encoder-large/5") #Transformer
3     #use_object = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4") #DAN
4
5     regularization = tf.keras.regularizers.L1L2(l1=args.l1, l2=args.l2)
6     text = layers.Input(shape=(), dtype=tf.string)
7     emb_layers = keras.Sequential()
8     emb_layers.add(hub.KerasLayer(use_object, trainable=True, activity_regularizer=regularization))
9
10    #### DAN : 30, L1=60, L2=90, L3=120, L4=150, 180
11    #### Transformer : 240, L1=660, L2=1080, L3=1500, L4=1920, L5=2340, L6=2760, 3000
12    layer_num = args.layer_num
13    to_freeze = emb_layers.layers[0].weights[:layer_num]
14    to_train = emb_layers.layers[0].weights[layer_num:]
15    emb_layers.layers[0]._trainable_weights = []
16    for x in to_train: emb_layers.layers[0]._trainable_weights.append(x)
17    for y in to_freeze: emb_layers.layers[0]._non_trainable_weights.append(y)
18
19    text_emb = emb_layers(text)
20    #fc1 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(text_emb)
21    #fc2 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc1)
22    #fc3 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc2)
23    #fc4 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc3)
24    final = layers.Dense(3, activation='sigmoid')(text_emb)
25    model = keras.Model(inputs=[text], outputs=[final])
26    return model

```

Figure E.3 – La fonction de création du modèle 1

```

1 def create_model_2(args):
2     use_object = hub.load("https://tfhub.dev/google/universal-sentence-encoder-large/5") #Transformer
3     regularization = tf.keras.regularizers.L1L2(l1=args.l1, l2=args.l2)
4     text = layers.Input(shape=(), dtype=tf.string)
5     features = layers.Input(shape=(3,), name='Features')
6     emb_layers = keras.Sequential()
7     emb_layers.add(hub.KerasLayer(use_object, trainable=True, activity_regularizer=regularization))
8     text_emb = emb_layers(text)
9     merged = layers.concatenate([text_emb, features])
10    #fc1 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(merged)
11    #fc2 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc1)
12    #fc3 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc2)
13    #fc4 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc3)
14    final = layers.Dense(3, activation='sigmoid')(merged)
15    model = keras.Model(inputs=[text, features], outputs=[final])
16    return model

```

Figure E.4 – La fonction de création du modèle 2

```

1 def create_model_3(args):
2     use_object = hub.load("https://tfhub.dev/google/universal-sentence-encoder-large/5") #Transformer
3     regularization = tf.keras.regularizers.L1L2(l1=args.l1, l2=args.l2)
4
5     text = layers.Input(shape=(), dtype=tf.string)
6     features = layers.Input(shape=(84,), name='Features')
7     emb_layers = keras.Sequential()
8     emb_layers.add(hub.KerasLayer(use_object, trainable=True, activity_regularizer=regularization))
9     text_emb = emb_layers(text)
10    merged = layers.concatenate([text_emb, features])
11    #fc1 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(merged)
12    #fc2 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc1)
13    #fc3 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc2)
14    #fc4 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc3)
15    final = layers.Dense(3, activation='sigmoid')(merged)
16    model = keras.Model(inputs=[text, features], outputs=[final])
17    return model

```

Figure E.5 – La fonction de création du modèle 3

```

1 def create_model_4(args):
2     use_object = hub.load("https://tfhub.dev/google/universal-sentence-encoder-large/5") #Transformer
3     regularization = tf.keras.regularizers.L1L2(l1=args.l1, l2=args.l2)
4     text = layers.Input(shape=(), dtype=tf.string)
5     size_feats = layers.Input(shape=(3,), name='size_feats')
6     code_feats = layers.Input(shape=(84,), name='code_feats')
7     emb_layers = keras.Sequential()
8     emb_layers.add(hub.KerasLayer(use_object, trainable=True, activity_regularizer=regularization))
9     text_emb = emb_layers(text)
10    merged = layers.concatenate([text_emb, size_feats, code_feats])
11    #fc1 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(merged)
12    #fc2 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc1)
13    #fc3 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc2)
14    #fc4 = layers.Dense(500, activation='relu', activity_regularizer=regularization)(fc3)
15    final = layers.Dense(3, activation='sigmoid')(merged)
16    model = keras.Model(inputs=[text, size_feats, code_feats], outputs=[final])
17    return model

```

Figure E.6 - La fonction de création du modèle 4

```

1 rootdir = 'raw_commits/notre_dataset/all_commits'
2 all_commits = []
3 dict_text, dict_code = {}, {}
4 tokenizer = AutoTokenizer.from_pretrained("microsoft/codebert-base")
5 model = AutoModel.from_pretrained("microsoft/codebert-base")
6
7 def message_normalization(commit_msg):
8     stemmer = PorterStemmer()
9     stopWords = set(stopwords.words('english'))
10    tokenizer = RegexpTokenizer(r"\w+")
11
12    words = tokenizer.tokenize(commit_msg)
13    wordsFiltered = []
14    for w in words:
15        if w not in stopWords:
16            wordsFiltered.append(w)
17    final = [stemmer.stem(word) for word in wordsFiltered]
18
19    ## Update dict_text
20    for w in final:
21        if w not in dict_text:
22            dict_text[w] = (len(dict_text))
23    return ','.join(final)
24
25 def code_normalization(loc):
26    code_tokens= [tokenizer.cls_token] + tokenizer.tokenize(loc)
27    tokens_ids= tokenizer.convert_tokens_to_ids(code_tokens)
28
29    final = []
30    ## Update dict_code
31    for index, token in enumerate(code_tokens):
32        if token not in dict_code:
33            dict_code[token] = len(dict_code) #tokens_ids[index]
34        final.append(str(dict_code[token]))
35    return ','.join(final)

```

Figure E.7 - La construction des dictionnaires pour le modèle CC2Vec

```

1 if __name__ == '__main__':
2     logging.getLogger("tensorflow").setLevel(logging.ERROR)
3     params = read_args().parse_args()
4     path, weights = get_dataset(params.dataset)
5     my_metrics = ["TruePositives", "FalsePositives", "TrueNegatives", "FalseNegatives", "Precision", "Recall",
6                 _hamming_loss]
7
8     if params.train is True:
9         train_data, train_label = load_commits(path + 'train_data', params)
10        val_data, val_label = load_commits(path + 'val_data', params)
11        checkpoint_path = "D:/Marouane/epochs/cp-{epoch:02d}.ckpt"
12        mc = keras.callbacks.ModelCheckpoint(checkpoint_path, save_weights_only=True, save_freq='epoch')
13        if params.model == 2:
14            model = create_model_2(params)
15        elif params.model == 3:
16            model = create_model_3(params)
17        elif params.model == 4:
18            model = create_model_4(params)
19        elif params.model == 0:
20            model = create_model_0(params)
21        elif params.model == -1:
22            model = create_model_benchmark(params)
23        else:
24            model = create_model_1(params)
25
26        model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=params.lr),
27                    metrics= my_metrics)
28        results = model.fit(train_data, train_label, validation_data=(val_data, val_label), shuffle=True,
29                            epochs=params.epochs, batch_size=params.batch, class_weight=weights, callbacks=[mc])
30        download_results(results, params.result_file)
31        exit()
32    elif params.test is True:
33        test_data, test_label = load_commits(path + 'test_data', params)
34        if params.model == 2:
35            model = create_model_2(params)
36        elif params.model == 3:
37            model = create_model_3(params)
38        elif params.model == 4:
39            model = create_model_4(params)
40        elif params.model == 0:
41            model = create_model_0(params)
42        elif params.model == -1:
43            model = create_model_benchmark(params)
44        else:
45            model = create_model_1(params)
46        model.load_weights('D:/Marouane/epochs/' + params.load_model)
47        model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate=params.lr),
48                    metrics= my_metrics)
49        preds = model.evaluate(test_data, test_label)

```

Figure E.8 - La partie responsable de l'entraînement et test du modèle

## RÉFÉRENCES BIBLIOGRAPHIQUES

Afshine, A. and Shervine, A. (2019). "CS 230 — Deep Learning." Retrieved 2022-11-01, from <https://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-recurrents>.

Aha, D. W., Kibler, D. and Albert, M. K. (1991). "Instance-based learning algorithms." *Machine learning* **6**: 37-66.

Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, C. (2018). "A survey of machine learning for big code and naturalness." *ACM Computing Surveys (CSUR)* **51**(4): 1-37.

AlOmar, E., Mkaouer, M. W. and Ouni, A. (2019). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor), IEEE.

AlOmar, E. A., Mkaouer, M. W. and Ouni, A. (2021). "Toward the automatic classification of self-affirmed refactoring." *Journal of Systems and Software* **171**: 110821.

Amor, J. J., Robles, G., Gonzalez-Barahona, J. M. and Navarro, A. (2006). Discriminating development activities in versioning systems: A case study. Proceedings PROMISE.

Ankit, U. (2022). "Transformer Neural Networks: A Step-by-Step Breakdown." Retrieved 2022-11-01, from <https://builtin.com/artificial-intelligence/transformer-neural-network>.

Anupadhyay. (2021). "Centralized vs Distributed Version Control: Which One Should We Choose?" Retrieved 2022-11-01, from <https://www.geeksforgeeks.org/centralized-vs-distributed-version-control-which-one-should-we-choose>.

Aponyi, A. (2021). "What are Sentence Embeddings and Their Applications?" Retrieved 2022-11-01, from <https://www.taus.net/resources/blog/what-are-sentence-embeddings-and-their-applications>.

Arima, R., Higo, Y. and Kusumoto, S. (2018). A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects? Proceedings of the 15th International Conference on Mining Software Repositories.

Arora, S., Liang, Y. and Ma, T. (2017). A simple but tough-to-beat baseline for sentence embeddings. International conference on learning representations.

Bahdanau, D., Cho, K. and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *3rd International Conference on Learning Representations (ICLR 2015)*. <http://arxiv.org/abs/1409.0473>.

- Bishop, C. M. and Nasrabadi, N. M. (2006). Pattern recognition and machine learning, Springer.
- Bowman, S. R., Angeli, G., Potts, C. and Manning, C. D. (2015). A large annotated corpus for learning natural language inference. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing: 632--642.
- Breiman, L. (2001). "Random forests." Machine learning **45**: 5-32.
- Buse, R. P. and Weimer, W. R. (2010). Automatically documenting program changes. Proceedings of the IEEE/ACM international conference on Automated software engineering.
- Cambria, E. and White, B. (2014). "Jumping NLP curves: A review of natural language processing research." IEEE Computational intelligence magazine **9**(2): 48-57.
- Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I. and Specia, L. (2017). Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. Proceedings of the 11th International Workshop on Semantic Evaluation, Association for Computational Linguistics: 1--14.
- Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S. and Tar, C. (2018). "Universal sentence encoder." arXiv preprint arXiv:1803.11175.
- Chen, T. and Guestrin, C. (2015). Xgboost: Reliable large-scale tree boosting system. Proceedings of the 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA.
- Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining.
- Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I. and Zhou, T. (2015). "Xgboost: extreme gradient boosting." R package version 0.4-2 **1**(4): 1-4.
- Cieslak, D. A., Chawla, N. V. and Striegel, A. (2006). Combating imbalance in network intrusion datasets. GrC.
- Cleary, J. G. and Trigg, L. E. (1995). K\*: An instance-based learner using an entropic distance measure. Machine Learning Proceedings 1995, Elsevier: 108-114.
- Cohen, J. (1960). "A coefficient of agreement for nominal scales." Educational and psychological measurement **20**(1): 37-46.

- Cohen, W. W. (1995). Fast effective rule induction. Machine learning proceedings 1995, Elsevier: 115-123.
- Conneau, A., Kiela, D., Schwenk, H., Barrault, L. and Bordes, A. (2017). Supervised learning of universal sentence representations from natural language inference data. Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics: 670--680.
- Cortes, C. and Vapnik, V. (1995). "Support-vector networks." Machine learning **20**: 273-297.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K. and Harshman, R. (1990). "Indexing by latent semantic analysis." Journal of the American society for information science **41**(6): 391-407.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T. and Jiang, D. (2020). Codebert: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics: 1536--1547.
- Fluri, B. and Gall, H. C. (2006). Classifying change types for qualifying change couplings. 14th IEEE International Conference on Program Comprehension (ICPC'06), IEEE.
- Fluri, B., Wursch, M., Pinzger, M. and Gall, H. (2007). "Change distilling: Tree differencing for fine-grained source code change extraction." IEEE Transactions on software engineering **33**(11): 725-743.
- Friedman, J., Hastie, T. and Tibshirani, R. (2000). "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)." The annals of statistics **28**(2): 337-407.
- Friedman, N., Geiger, D. and Goldszmidt, M. (1997). "Bayesian network classifiers." Machine learning **29**: 131-163.
- Frreira, D. (2020). "What Are Sentence Embeddings and why Are They Useful?" Retrieved 2022-11-01, from <https://engineering.talkdesk.com/what-are-sentence-embeddings-and-why-are-they-useful-53ed370b3f35>.
- Gall, H. C., Fluri, B. and Pinzger, M. (2009). "Change analysis with evolizer and changedistiller." IEEE software **26**(1): 26-33.



Ghadhab, L., Jenhani, I., Mkaouer, M. W. and Messaoud, M. B. (2021). "Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model." Information and Software Technology **135**: 106566.

Gharbi, S., Mkaouer, M. W., Jenhani, I. and Messaoud, M. B. (2019). On the classification of software change messages using multi-label active learning. Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). Deep learning, MIT press.

Google-Trends. (2023). "SCCV vs SDCV." Retrieved 2023-07-07, from <https://trends.google.com/trends/explore?date=2015-05-20%202023-06-20&q=Centralized%20version%20control%20system,distributed%20version%20control%20system&hl=fr>.

Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE.

Hindle, A., German, D. M. and Holt, R. (2008). What do large commits tell us? a taxonomical study of large commits. Proceedings of the 2008 international working conference on Mining software repositories.

Hoang, T., Kang, H. J., Lo, D. and Lawall, J. (2020). CC2Vec: Distributed representations of code changes. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.

Hoang, T., Lawall, J., Oentaryo, R. J., Tian, Y. and Lo, D. (2019). PatchNet: a tool for deep patch classification. 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE.

Holt, R. C., Hindle, A. and Godfrey, M. W. Release Pattern Discovery via Partitioning: Methodology and Case Study. International Conference on Mining Software Repositories.

Hönel, S., Ericsson, M., Löwe, W. and Wingkvist, A. (2020). "Using source code density to improve the accuracy of automatic commit classification into maintenance activities." Journal of Systems and Software **168**: 110673.

Hotelling, H. (1933). "Analysis of a complex of statistical variables into principal components." Journal of educational psychology **24**(6): 417.

Hu, M. and Liu, B. (2004). Mining and summarizing customer reviews. Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining.

Huang, Y.-M., Hung, C.-M. and Jiau, H. C. (2006). "Evaluation of neural networks and data mining methods on a credit assessment task for class imbalance problem." Nonlinear Analysis: Real World Applications **7**(4): 720-747.

Huo, X., Li, M. and Zhou, Z.-H. (2016). Learning unified features from natural and programming languages for locating buggy source code. IJCAI.

Iyyer, M., Manjunatha, V., Boyd-Graber, J. and Daumé III, H. (2015). Deep unordered composition rivals syntactic methods for text classification. Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers).

Jurafsky, D. and Martin, J. (2009). Speech and Language Processing (2nd Edition). Prentice-Hall inc.

Kamiya, T., Kusumoto, S. and Inoue, K. (2002). "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." IEEE transactions on software engineering **28**(7): 654-670.

Kingma, D. P. and Ba, J. (2014). "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980.

Kirinuki, H., Higo, Y., Hotta, K. and Kusumoto, S. (2014). Hey! are you committing tangled changes? Proceedings of the 22nd International Conference on Program Comprehension.

Koyuncu, A., Liu, K., Bissyandé, T. F., Kim, D., Klein, J., Monperrus, M. and Le Traon, Y. (2020). "Fixminer: Mining relevant fix patterns for automated program repair." Empirical Software Engineering **25**: 1980-2024.

Kubat, M., Holte, R. C. and Matwin, S. (1998). "Machine learning for the detection of oil spills in satellite radar images." Machine learning **30**: 195-215.

Lehman, M. M. and Belady, L. A. (1985). Program evolution: processes of software change, Academic Press Professional, Inc.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. and Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view. Proceedings Fourth International Software Metrics Symposium, IEEE.

Levin, S. and Yehudai, A. (2016). Using temporal and semantic developer-level information to predict maintenance activity profiles. 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE.

Levin, S. and Yehudai, A. (2017). Boosting automatic commit classification into maintenance activities by utilizing source code changes. Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering.

Li, X. and Roth, D. (2002). Learning question classifiers. COLING 2002: The 19th International Conference on Computational Linguistics.

Manning, C. (2022). CS224n: Natural Language Processing with Deep Learning. Lecture slides and notes, Stanford University.

Manning, C. D. (2008). Introduction to information retrieval, Syngress Publishing.

Mariano, R. V., dos Santos, G. E., de Almeida, M. V. and Brandão, W. C. (2019). Feature changes in source code for commit classification into maintenance activities. 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), IEEE.

Matsuda, J., Hayashi, S. and Saeki, M. (2015). Hierarchical categorization of edit operations for separately committing large refactoring results. Proceedings of the 14th International Workshop on Principles of Software Evolution.

Mauczka, A., Brosch, F., Schanes, C. and Grechenig, T. (2015). Dataset of developer-labeled commit messages. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE.

Mazurowski, M. A., Habas, P. A., Zurada, J. M., Lo, J. Y., Baker, J. A. and Tourassi, G. D. (2008). "Training neural network classifiers for medical decision making: The effects of imbalanced datasets on classification performance." Neural networks **21**(2-3): 427-436.

Mikolov, T., Kai, C., Greg, C. and Jeffrey, D. (2013b). Efficient Estimation of Word Representations in Vector Space. 1st International Conference on Learning Representations, (ICLR 2013).

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J. (2013a). "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems **26**.

Miller, G. A. (1995). "WordNet: a lexical database for English." Communications of the ACM **38**(11): 39-41.

Mitchell, T. M. (1997). Machine learning, McGraw-hill New York.

Mockus, A. and Votta, L. G. (2000). Identifying Reasons for Software Changes using Historic Databases. icism.

Mondal, A. K., Roy, B. and Schneider, K. A. (2019). An exploratory study on automatic architectural change analysis using natural language processing techniques. 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE.

Nayrolles, M. and Hamou-Lhadj, A. (2018). CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. Proceedings of the 15th International Conference on Mining Software Repositories. Gothenburg, Sweden, Association for Computing Machinery: 153–164.

Nguyen, A. T., Hilton, M., Codoban, M., Nguyen, H. A., Mast, L., Rademacher, E., Nguyen, T. N. and Dig, D. (2016). API code recommendation using statistical learning from fine-grained changes. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Nguyen, H. A., Nguyen, A. T. and Nguyen, T. N. (2013). Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. 2013 IEEE 24th international symposium on software reliability engineering (ISSRE), IEEE.

Otter, D. W., Medina, J. R. and Kalita, J. K. (2020). "A survey of the usages of deep learning for natural language processing." IEEE transactions on neural networks and learning systems **32**(2): 604-624.

Pang, B. and Lee, L. (2004). A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04).

Pang, B. and Lee, L. (2005). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05): 115–124.

Pennington, J., Socher, R. and Manning, C. D. (2014). Glove: Global vectors for word representation. Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP).

Platt, J. (1998). "Fast Training of Support Vector Machines using Sequential Minimal Optimization. MIT Press."

Powers, D. M. (2015). "What the F-measure doesn't measure: Features, Flaws, Fallacies and Fixes." arXiv preprint arXiv:1503.06410.

Pressman, R. and Maxim, B. (2020). Software Engineering: A Practitioner's Approach, McGraw-Hill Education.

- Quinlan, J. R. (2014). C4. 5: programs for machine learning, Elsevier.
- Ramage, D., Hall, D., Nallapati, R. and Manning, C. D. (2009). Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora. Proceedings of the 2009 conference on empirical methods in natural language processing.
- Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): 3982--3992.
- Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). "Learning representations by back-propagating errors." nature **323**(6088): 533-536.
- Saar-Tsechansky, M. and Provost, F. (2007). "Handling missing values when applying classification models."
- Sanh, V., Debut, L., Chaumond, J. and Wolf, T. (2019). "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." arXiv preprint arXiv:1910.01108.
- Sanjoy. (2023). "Software Engineering | Software Maintenance." Retrieved 2023-07-07, from <https://www.geeksforgeeks.org/software-engineering-software-maintenance/>.
- Sarwar, M. U., Zafar, S., Mkaouer, M. W., Walia, G. S. and Malik, M. Z. (2020). Multi-label classification of commit messages using transfer learning. 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE.
- Schwaber, K. and Sutherland, J. (2011). "The scrum guide." Scrum Alliance **21**(1): 1-38.
- Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y. and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. Proceedings of the 2013 conference on empirical methods in natural language processing.
- Sorower, M. S. (2010). "A literature survey on algorithms for multi-label learning." Oregon State University, Corvallis **18**(1): 25.
- Sristy, A. K. M. B. R. and Schneider, S. N. K. A. (2021). "ArchiNet: A Concept-token based Approach for Determining Architectural Change Categories."
- Swanson, E. B. (1976). The dimensions of maintenance. Proceedings of the 2nd international conference on Software engineering.
- Thakur, D. (2023). "Types of Software Maintenance." Retrieved 2023-07-07, from <https://ecomputernotes.com/software-engineering/types-of-software-maintenance>.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinianian, D. and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. Proceedings of the 40th international conference on software engineering.

Tsoumakas, G. and Vlahavas, I. (2007). Random k-labelsets: An ensemble method for multilabel classification. Machine Learning: ECML 2007: 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007. Proceedings 18, Springer.

Uzila, A. (2022). "Beautifully Illustrated: NLP Models from RNN to Transformer." Retrieved 2022-11-01, from <https://towardsdatascience.com/beautifully-illustrated-nlp-models-from-rnn-to-transformer-80d69faf2109>.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. and Polosukhin, I. (2017). "Attention is all you need." Advances in neural information processing systems **30**.

Wiebe, J., Wilson, T. and Cardie, C. (2005). "Annotating expressions of opinions and emotions in language." Language resources and evaluation **39**: 165-210.

Wieting, J., Bansal, M., Gimpel, K. and Livescu, K. (2016). Towards universal paraphrastic sentence embeddings. 4th International Conference on Learning Representations.

Williams, A., Nangia, N. and Bowman, S. (2018). A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers): 1112--1122.

Yang, Y. (1999). "An evaluation of statistical approaches to text categorization." Information retrieval **1**(1-2): 69-90.

Zafar, S., Malik, M. Z. and Walia, G. S. (2019). Towards standardizing and improving classification of bug-fix commits. 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE.

Zhang, M.-L. and Zhou, Z.-H. (2013). "A review on multi-label learning algorithms." IEEE transactions on knowledge and data engineering **26**(8): 1819-1837.