Department of Computer Science
University College London

Improving Software Project Health Using Machine Learning

Profir-Petru Pârțachi

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

7th December 2020

I, Profir-Petru Pârțachi, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. The work presented in this thesis is original work undertaken between September 2016 and July 2020 at University College London.

I list the papers that comprise this work in Section 1.3. In the same section, I clearly detail my contributions towards each paper. They represent Chapers 3, 4, and 5 respectively.

Date: 7th December 2020

Name: Profir-Petru Pârțachi

Signature:

Abstract

In recent years, systems that would previously live on different platforms have been integrated under a single umbrella. The increased use of GitHub, which offers pull-requests, issue tracking and version history, and its integration with other solutions such as Gerrit, or Travis, as well as the response from competitors, created development environments that favour agile methodologies by increasingly automating non-coding tasks: automated build systems, automated issue triaging *etc.* In essence, source-code hosting platforms shifted to continuous integration/continuous delivery (CI/CD) as a service. This facilitated a shift in development paradigms, adherents of agile methodology can now adopt a CI/CD infrastructure more easily. This has also created large, publicly accessible sources of source-code together with related project artefacts: GHTorrent and similar datasets now offer programmatic access to the whole of GitHub.

Project health encompasses traceability, documentation, adherence to coding conventions, tasks that reduce maintenance costs and increase accountability, but may not directly impact features. Overfocus on health can slow velocity (new feature delivery) so the Agile Manifesto suggests developers should travel light — forgo tasks focused on a project health in favour of higher feature velocity. Obviously, injudiciously following this suggestion can undermine a project's chances for success.

Simultaneously, this shift to CI/CD has allowed the proliferation of Natural Language or Natural Language and Formal Language textual artefacts that are programmatically accessible: GitHub and their competitors allow API access to their infrastructure to enable the creation of CI/CD bots. This suggests that approaches from Natural Language Processing and Machine Learning are now feasible and indeed desirable. This thesis aims to (semi-)automate tasks for this new paradigm and its attendant infrastructure by bringing to the foreground the relevant NLP and ML techniques.

Under this umbrella, I focus on three synergistic tasks from this domain: (1) improving the issue-pull-request traceability, which can aid existing systems to automatically curate the issue backlog as pull-requests are merged; (2) untangling commits in a version history, which can aid the beforementioned traceability task as well as improve the usability of determining a fault introducing commit, or cherry-picking via tools such as git bisect; (3) mixed-text parsing, to allow better API mining and open new avenues for project-specific code-recommendation tools.

Impact Statement

In the modern world, software is ubiquitous: from the personal computers most of us use to the mission critical systems over which we require fine control. Software quality is usually a result of a healthy software project, *i.e.* it is not just a function of the code, rather also of the process producing that code. Further, projects are not just source-code; indeed, they contain a wealth of documents written in English or other languages. Focusing strictly on source-code tells us only part of the story.

In this thesis, I define the notion of project health; a notion that was previously colloquially understood and that encompasses those processes that help project succeed and flourish. Under this umbrella, I focus on three tasks that could impede project success: pull-request-issue linking, commit separation into atomic patches, and mixed-text segmentation and pre-processing. I focus on tasks that would break assumptions made by researchers when proposing techniques in the first two, while the latter presents a new way of handling data enabling analysis which is aware of algorithmic and natural language information. By borrowing techniques and methodology from Machine Learning, I propose prototype systems to resolve these issues.

The work I present in this thesis follows the ethos of helping developers help themselves and us. We, as researchers, help developers maintain project health with its inherent benefits: easier onboarding, lower costs of maintenance, *etc.* This, in turn, can create better training data for researchers and may enable yet better automation techniques and research avenues for big code. Further, all tooling created during this thesis is open sourced and made available under the MIT license for developers or researchers to use directly.

Acknowledgements

A PhD is not an easy journey, but one I will look back on fondly. Along the way, I have had the support of many for whom I wish to dedicate the following paragraphs in thanks.

I would like to thank my supervisor, Dr Earl T. Barr, for the sage advice and importantly patience with me especially as I was learning the art of academic writing. I would also like to thank Earl for bearing with me as I fell ill throughout the journey and offering support during those times. The dialectic we had trying to formally prove a hypothesis with the clock ticking to the deadline will be a fond memory for years despite the stress of the moment then.

I would also like to thank Dr David R. White for the 'Desk Compensation Meetings' that set me on the path towards rigorous experimental design early in my PhD. That formed the support beams for my empirical studies throughout my doctoral work. I would also like to thank David for being there for me when I needed someone to listen to me and helping navigate the academic landscape with more confidence.

For the help and patience they showed, I would also like to thank my co-authors during my PhD research. I would like to thank Dr Santanu Dash, for helping me grok the rougher sides of Roslyn and enabling the work that followed from that. I would also like to thank Santanu for helping me pursue an idea that later became a paper by helping me with initial labelled data that would have been impossible without his CLANG knowledge. His help and advice on the subtilties of formal languages helped shape my work.

I extend similar gratitude to Miltos Allamanis and Christoph Treude, who have made invaluable contributions and were a pleasure to work with as co-authors. Miltos has helped me get a better start writing machine learning code, while Christoph helped me better understand how to do qualitative and manual quantitative analysis.

I would like to thank my lab mates in CREST and SSE that made the journey more fun: Leo Jeoffe, David Kelly, Bobby Bruce, DongGyun Han, Matheus Paixão, Carlos Gavidia, Giovani Guizzo, Vali Tawosi, Rebecca Mousa, Jie Zhang. The informal chats were invaluable and they all helped me through the journey. Discussing our work together helped me, and I hope them as well, better understand how to do proper science, to better understand how and when statistical techniques should be applied, how to formulate experiments, how to thoroughly explore hypothesises. I also extend my gratitude to my close friend, Tudor Haruța, with whom I oft shared my progress and who was always available to chat when I needed to take my mind off of research.

Last but not by any means least, I would like to thank my family, my dear older sister for always bearing with me and lending a ear to listen and a pair of eyes to double-check my English. My parents for supporting me through my PhD years and even donating time on the home PC when I needed extra compute resources on a tight deadline.

Our research is shaped by those in our network, those we talk to even casually, thus the sum total of the contributions to this research is beyond any quantifiable scope; I cannot hope to enumerate everyone who has contributed to my work in the multitude of ways that I have been supported along on this journey. To everyone, thank you.

Contents

List of Figures

List of Tables

1

Introduction

The software development process involves a multitude of aspects: the specification and design, the architecture of the system, the code writing itself, code version histories, issue tracking, code review. In an opensource project, external collaboration and their integration are also a part of it. Last but not least, the developers themselves and the project culture further define the development process. Some projects are vibrant; some moribund; others implode after a few commits. *Project health* determines project success. Project health encompasses: social, technical, and traceability aspects of a project. It is defined by the quality of code, documentation, version histories, issue tracking, specifications and requirements. It is further defined by the quality of the interlinking of these aspects. Finally, but not less important, it is defined by the culture of the project, be that explicitly codified in a file/manifesto or implicit in the interactions of developers. In this work, I focus only on the technical and traceability aspects of project health. It is of no surprise that these processes produce artefacts that span and intermix a profusion of formats, natural languages, and programming languages.

In recent years systems, which would previously live on different platforms, have been integrated under a single umbrella, thus the different aspects that can impact project health can be centrally observed. The proliferation of GitHub, which offers pull-requests, issue tracking and version history, and its integration with other solutions such as Gerrit (for Code Reviews), Travis (for continuous integration and delivery), as well as the response from competitors, such as GitLab and Atlassian who offer equivalent offerings, has led to leaner and faster development cycles: GitHub advertises their CI offerings with a promise to reduce time spent merging commits or debugging and increase time spend writing code [1]. This has also reduced the cost of entry and created large, publicly accessible sources of source-code together with related project artefacts, such as GHTorrent [2], CodeSearchNet [3] and similar datasets. Within this context,

developers can benefit from the automation of tasks, especially if they are often forgotten or ignored in favour of traveling light.

This shift in tooling has also facilitated a shift in development paradigms, developers can now more easily adopt a continuous integration/continuous delivery infrastructure. This has led to more projects adopting a more agile development process. Cleland-Huang *et al.* [4] and Ståhl *et al.* [5] observe that, for traceability to be adopted, within an agile framework, it must remain within the traveling light paradigm. Simultaneously, the shift to continuous improvement/continuous development has allowed for the proliferation of Natural Language or Natural Language and Formal Language textual artefacts that are programmatically accessible. This suggests that approaches from Natural Language Processing and Machine Learning are now feasible and indeed desirable. This thesis aims to (semi-)automate tasks for this new paradigm and its attendant infrastructure by bringing to the foreground the relevant NLP and ML techniques.

1.1 Problem Statements

Under Improving Project Health, I focus on three synergistic tasks: (1) improving the issue-pull-request traceability, which can aid existing systems to automatically curate the issue backlog as pull-requests are merged; (2) untangling commits in a version history, which can aid the beforementioned traceability task as well as improve the usability of determining a fault introducing commit, or cherry-picking via tools such as git bisect; (3) mixed-text parsing, to allow better API mining and open new avenues for project-specific code-recommendation tools.

Software development is a community task. The programming task itself is a component of a much wider process. It is, however, a task that is viewed exclusively when considered by developers at the cost of other administrative tasks. This manifests itself as, for example, in the case of traceability, which companies recognise as important, by organisations reaching an insufficient level of traceability [6]. These administrative tasks come with the promise of facilitating internal follow-ups, evaluation or improvement of the development process [7]. Agile practice, and especially Continuous Integration, by suggesting a lean approach to documentation, reduces the direct applicability of existing approaches, though work has been done to adapt to the new paradigm: Ståhl *et al.* [5] provide the Eiffel framework, which allows integrating different sources of traceability information in a light-weight manner. Indeed, the conflict arises between the traditional assumptions of Traceability and the agile team's tendencies of traveling quite light [4]. Indeed, *traveling light* means forgoing documentation, user stories not being punted to new sprints, or, more generally, administrative tasks that distract from a developer coding being ignored. If combined with backlog refinement [8], traveling light can manifest itself as a form of project amnesia. This is a form of technical debt, and will degrade project health in the long-run.

To help developers, tool smiths themselves require supporting artefacts and utilities. One of these is access to high quality data that can be used both to train a human intuition for a

task as well as statistical approaches to enable developers from less well maintained projects improve their situation. The proliferation of web-sites such as GitHub, BitBucket/JIRA, GitLab, and others that enable developers to have commits, issues, pull-request, code-reviews and other development processes under one umbrella also opens a potential resource to researchers and tool smiths that wish to study the properties of the inter-relationships of such artefacts. In this context, traceability can help developers maintain the collective memory of projects. When using such resources, the promise of automatic triaging of issues as commits resolving them are accepted into the mainline incentivises developers to maintain code-review/issue to commit links. I have, however, observed a lack of such links as far as GitHub is concerned, despite an encouragement of contributors to do so in the contribution guidelines (Section 3.7.1). Further, developers only partially recording such link introduces bias in the datasets researchers use.

Pull-request-Issue Traceability

Given a project, developers should receive traceability link suggestions when pertinent, *i.e.* when they are already performing a task within which a link can be recorded. For a pull-request-issue traceability, suggestions are to be made at issue triage and PR submission.

Additionally, links may not be clear if the fixes are presented as unfocused patches, hiding bug fixing beyond many lines of refactoring. Herzig *et al.* [9] (Herzig and Zeller [10]) show that tangled commits introduce bias in defect prediction. In our work, I aim to provide a diff time solution for commit untangling to help developers reduce this bias. The promise to developers is enabling tooling such as git bisect.

Commit Untangling

Given a commit, the patch should be decomposed into smaller patches such that each patch tackles a single task, *i.e.* would be linked to a single issue or implements a single stakeholder concern. On atomic commits, this process should be the identity function.

Despite the international nature of platforms such as GitHub, a significant portion of software development information is in English. This leads to projects in other natural languages to adopt English terms within their processes as observed by Treude *et al.* [11]. Further, such natural language can mix any natural languages it may use with formal languages. This same phenomenon is observable on mailing lists, such as the Linux Kernel Mailing List, and developer fora, such as StackOverflow. Further, despite access to mark-up to help developers demarcate code from English, in practice, even on StackOverflow submitters do forget to do so. This reduces the applicable research tools which use such formatting cues.

Mixed-text Parsing

Given an artefact that freely mixes natural languages and formal languages, separate the languages. Further, for each token, provide the tag in the language from which the token is taken. In natural languages, these tags are part-of-speech tags; in formal languages, these tags are the AST tags of the parents of the terminals.

The tasks range from more developer focused to more researcher focused; however, all tasks aim to always provide a benefit to developers, directly or indirectly.

Overall, the focus of this research is the creation of tools and meta-tools that (semi-)automate development tasks to enable project to remain healthy. PR-issue traceability focuses directly on improving the development process by reducing the administrative burden on the developers, while commit untangling and mixed-text parsing focus on enabling a wider range of tools to be applicable. The goal as a whole is an improvement of the development process by allowing more time to be dedicated to the programming task rather than tasks required to maintain the project health.

1.2 Contributions

Under the three synergistic tasks, I have found the following ways of improving the state-of-the-art, which can benefit both practitioners as well as researchers; This thesis:

1. Provides an online pull-request traceability link inference algorithm realised as Aide-mémoire. It solves an online, modern variant of the “Missing Links” problem proposed by Bachman *et al.* [12]. It departs from using commits in favour of pull-requests. Aide-mémoire suggests links when developers are already handling a PR or an issue.

2. Forwards the state-of-the-art in commit untangling both in accuracy and in speed and thus enables the creation of better statistical tooling for developers indirectly, while directly allowing them to improve the health of their version histories. I realise this as Flexeme. Flexeme offers precise untangling suggestions within 10 seconds; fast enough to be viable in the code-review process.

3. Formulates the mixed-text parsing problem, posing it as a simultaneous segmentation and tagging task. It proposes a solution, POSIT, which borrows from the Natural Language Processing and Linguistics literature that concerns itself with code-switching. It maps the borrowed results to the Software Engineering context and allows extending methods to languages other than English or resources that freely mix natural and formal languages.

The solutions I propose can help developers directly, by improving the state of the project artefacts, or indirectly by helping tool smiths. Further, these solutions interact synergistically,

Flexeme and POSIT improve the quality of the training data for Aide-mémoire, while also improving the general health of developer version histories (Flexeme) or developer issue trackers and fora (POSIT).

1.3 List of Papers

A long research project, like a PhD, is a close collaboration where it is difficult, often impossible, to separate each collaborator's contributions. What is clear is who took the lead on which aspects and tasks. The reader will also notice a mixed use of 'I' and 'we', I continue to use 'we' in the paper chapters when referring to work done in close collaboration with my co-authors to acknowledge their contribution. Further, the papers are presented as published or submitted for review spare figures and tables that required editing to fit into the thesis format. I now enumerate the tasks I lead in each of the papers in this thesis.

1. Aide-mémoire: Improving a Project's Collective Memory via Pull Request-Issue Links

Authors: Profir-Petru Pârțachi, David R. White, Earl T. Barr

Venue: Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM)

I determined and mined the dataset of GitHub projects. I implemented the proposed PR-Issue linker, performed the exploratory data analysis and feature engineering as well as wrote the evaluation and result analysis scripts. The experimental and EDA designs were done in close collaboration with David White who guided me patiently.

2. Flexeme: Untangling Commits Using Lexical Flows

Authors: Profir-Petru Pârțachi, Santanu Kumar Dash, Miltos Allamanis, Earl T. Barr

Venue: Proceedings of 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/FSE 2020)

I implemented in full the construction of our new structure, though the idea of the structure was from Earl Barr, and the details of its specification were worked out in close collaboration with him. I also implemented the necessary methods to construct the dataset, reproduced previous work in the area and implemented our proposed untangling algorithms as well as their evaluation on the constructed dataset. The RefiNym implementation was provided by Santanu Dash who helped me integrate it with Flexeme. The original PDG extraction implementation for C# code was provided by Miltos Allamanis. Santanu Dash and I performed the manual evaluations.

3. POSIT: Simultaneously Tagging Natural and Programming Languages

Authors: Profir-Petru Pârțachi, Santanu Kumar Dash, Christoph Treude, Earl T. Barr

Venue: Proceedings of 42nd International Conference on Software Engineering (ICSE '20)

I implemented the preprocessing scripts, the Neural Network that realises POSIT, the adaptation of the previous state-of-the-art to our problem, and the necessary evaluation scripts. The Code Comments corpus was provided by Santanu Dash. The original implementation of TaskNav was provided by Christoph Treude. The informal proof of context-sensitivity of mixed-text was worked on in close collaboration with Earl Barr without whom the proof would have not been finished in a timely manner. The manual evaluation of POSIT was done together with Santanu Dash, while the manual evaluations of TaskNav augmented with POSIT were done together with Christoph Treude.

1.4 Thesis Organisation

The remainder of the thesis is organised as follows.

Chapter 2 first presents literature on the areas encompassed by project health. It then focuses on the areas touched by the proposed tasks in the order they are presented in the thesis. It first discusses traceability and the "Missing Links" problem [12], it then focuses on the issues caused by tangled commits and approaches to the commit untangling, it concludes with an exploration of existing approaches to mixed-text: how do researchers tackle code when applying natural language techniques and how the natural language processing community handles the mixing of multiple natural languages.

Chapter 3 presents Aide-mémoire, a solution to online pull-request-issue linking problem. It extends the "Missing Links" problem due to Bachmann *et al.* [12] to pull-requests and proposes a solution that can live along side the code-review process. Aide-mémoire is evaluated on an extensive corpus of GitHub projects.

Chapter 4 presents Flexeme, a solution to the commit untangling problem. It shows that Flexeme improves the state-of-the-art while being fast enough to be viable during code-review. Flexeme is evaluated on an artificial corpus that is rigorously constructed to mimic tangled commits Herzig *et al.* [9] have observed developers make.

Chapter 5 presents POSIT, where the mixed-text parsing problem is introduced together with our proposed solution: POSIT. In it, we argue that the mixed-text parsing problem is context-sensitive in the general case and hence propose a distributional semantics approach, realised as a neural network, for the problem. We show that POSIT can both directly improve software fora by suggesting mixed code annotations, as well as aid other research tools: we show POSIT to improve TaskNav's [13] recall.

Chapter 6 concludes the thesis and provides directions for future work.

2

Literature Review

In this chapter, I first present the wider literature that concerns developer activities impacting project health in Section 2.1. I then focus on the subproblems tackled in the thesis and start by presenting Traceability and Commit-Issue linking literature in Section 2.2. Section 2.3 presents the literature around commit untangling. I conclude this chapter by describing the literature around mixed-text in Section 2.4.

2.1 The Many Faces of Project Health

As project health comprises different aspects of a project — from technical, through design, to social — it is natural that it touches many areas of active interest from researchers. Some of the more prominently researched areas are technical debt and human aspects of software engineering. I will first present technical debt literature and its relation to project health. I then will shift focus to the area of software evolution and maintenance, with a primary focus on how it helps projects remain active. I conclude this section by presenting recent research in human aspects of software engineering, with a focus on improving the development process by removing negative incentives.

Technical debt primarily arose as a concept from a metaphor relating it to financial debt and serving to capture the consequences of poor software development [14]. The core concept was that debt is not necessarily bad, it can be used as a asset to speed up development at a crucial point so long as it is repaid. While this provided an intuitive basis for the concept, the lack of a formal definition left the boundaries fuzzy.

Tom *et al.* [15] found that this created an apparently fragmented understanding from a number of both themes and anecdotal accounts that fail to fully capture the phenomenon. Later, Tom *et al.* conduct a multivocal literature review [16] within which they determine the dimensions of technical debt, its attributes, and provide a taxonomy for it. Tom *et al.* separate technical debt

into: code debt, design and architectural debt, environment debt, knowledge distribution and documentation debt, and testing debt:

- Code debt covers hacks or sloppy code that was written and would incur heavy maintenance costs.
- Design and architectural debt comes mainly in two flavours: an upfront design that did not factor maintenance or scale or piecemeal design that was done without the required refactoring; further, this debt could also arise as a consequence of a sub-optimal architectural solution, be it sub-optimal at the time of design or as technology progresses.
- Environmental debt covers software, hardware, dependencies, or even lack of automation. It manifests itself as suboptimal use of labour, reduced development velocity or even vulnerabilities.
- Knowledge distribution and documentation cost covers aspects related to having a high 'bus factor', when a complex system is maintained by few knowledgeable individuals and the process is not documented, a company risks a sudden increase of technical debt should the individuals leave.
- Testing debt covers the need for manual quality assurance when the task can be automated. While it shares the automation aspect with Environmental debt, it has the associated risk of brand damage if a faulty patch is pushed through to production.

As impact on project health, in the short term, technical debt allows a boost in project velocity; and should this debt never come due, you may feel free to incur high amounts [16]; however, in the long term, it can create brittle code, increase the cost of new features or products by decreasing reusability, and even increase production costs by decreasing long term project velocity as bandwidth is taken by debt repayment or simply attrition from the sub-optimal products and development pipelines. Tom *et al.* further found that this can have an impact on not just productivity, quality, and risk, but also developer morale as work time is taken on repaying debt and working in a brittle system that is difficult to reason about.

While Tom *et al.*'s [16] definition of technical debt covers some of the impacts from developers on project health: such as low morale, low documentation, sub-optimal tooling; it does not cover the scenario when the development process itself has negative incentives. Indeed, a development or maintenance process, such as bug triaging, may have hidden incentives that, when each participant behaves optimally for themselves, causes sub-optimal outcomes for the project, such as wasted development time. The nature of this issue lends itself well to game-theoretic approaches. Gavidia-Calderon *et al.* [17, 18] propose that anomalies in the software development process be modelled using Empirical Game Theory so that a process intervention can be devised

in a way that avoids negative incentives. They demonstrate their approach using simulation based examples that cover bug priority inflation, acquisition of technical debt in a simple fix or kugel example, and the tragedy of the test suite (the accumulation of test cases due to incorrect developer incentivisation).

When technical debt cannot be avoided, a risk associated with technical debt is its interest rate. The work I present in this thesis aims to, first, reduce the need to accrue technical debt by (semi-)automating tasks that would otherwise be forgone for a leaner project. When debt is acquired, the work herein aims to reduce the interest rate. Recasting the perspective into the framework presented by Gavidia-Calderon *et al.*, the thesis aims to reduce the cost of actions associated with good long-term project health in the pay-off matrix, *i.e.* to ensure that short horizon optimal decisions are also optimal in a long horizon setting.

2.2 Bridging the Gap between Issues and History

A method of combating accruing knowledge distribution debt is by maintain a high degree of traceability — interlinking of development and design artefacts by a 'are related/implements/specifies etc.' link. In this section, I focus on software traceability as an area, and the position of the commits to issues traceability task, which we generalise and propose a solution for in Chapter 3, within this wider context.

2.2.1 Modern Development

Modern development increasingly relies on tooling that integrates version control, issue tracking, wikis, continuous integration and continuous deployment under a single system. Notable examples are GitHub and Atlassian's JIRA. This new development paradigm poses new problems and opportunities. Kalliamvakou *et al.* [19] elucidates these, using GitHub as their archetypal example. A particular opportunity Kalliamvakou *et al.* identify is this paradigm's integration of Version Control and Issue Tracking with the potential to interlink them. This brings in opportunities for those following in the footsteps of Bachmann *et al.*'s Missing Links Problem [12], as they show that indeed this integration is not fully realised.

Lack of PR-issue links is an ongoing problem in modern software development. So much so, Agile practice specifies spring cleaning an issue (a user story in Agile terminology) backlog. During backlog refinement, developers remove stale stories and reprioritise and re-estimate remaining stories. When all stories are stale, this practice discards all sprint-related artefacts — issues, feature requests, user stories, as well their links — in favour of starting the next sprint from a clean slate [8]. Projects must resort to this spring cleaning all too often [20]. This practice loses user stories, documentation, issues, discussions, and other such artefacts associated with a sprint by design; this loss of information comes at the cost of higher maintenance costs, onboarding costs *etc.* as the information has to be recreated or reverse engineered. The loss of

documentation it entails it just one example. This can exacerbate the accumulation of knowledge distribution and documentation debt. Adoption of PR-issue inference tooling promises to reduce the need to resort to this drastic measure by enabling automatic issue triaging.

2.2.2 Software Traceability

Requirements engineering (RE) focuses on stakeholders, decision makers, and their artefacts: requirements, documentation, specification, and design or architectural documents. These artefacts tend to be natural language, text or speech, and often go unrecorded. When they are recorded, they exist in multiple formats, including spreadsheets, email, figures, and printed material. They further encompass developer artefacts, such as source code, pull-requests, commits, and issues, but do not focus on them. Further, within an agile context, documentation is often forgone in favour of staying lean and traveling light. This further complicates an already difficult task and suggests that tooling that would target such a paradigm should be light-weight [5].

Software traceability seeks to infer *traces* (*i.e.* links) between these heterogeneous artefacts [4]. Missing or hard-to-parse artefacts greatly complicate trace recovery, which is why much work on traceability seeks to provide tooling to decision makers to capture or parse these artefacts and persuade them to use it [21, 22, 23]. These tools must often record decision maker or developer interactions, with each other or their tools. They must also avoid being either disruptive, requiring the developer to switch contexts, or invasive, as when they require developers to change their workflow or use instrumented IDEs [24], raising privacy and deployability concerns. Inferring traces over these heterogeneous artefacts, as a consequence of their heterogeneity, can only leverage abstract, generic features.

As many of the artefacts within such systems are textual, work in this area borrowed techniques from Information Retrieval, including ourselves. Borg *et al.* [25] provide an excellent survey detailing the use of such methods within traceability. Further, Mills *et al.* [26] propose reducing the human effort required by IR based techniques via a classification step after the recommender that filters the suggestions. Finally, Ståhl *et al.* [5] propose a light-weight framework where different systems can emit traceability events, facilitating a standardisation of how such links are recorded by minimising the cost of integrating otherwise incompatible tools within a single eco-system, which they call the Eiffel framework.

Asuncion and Taylor [27] use record-replay and hypermedia to tackle software traceability as an online problem. Their online solution cannot employ offline information retrieval techniques (such as VSM, tf-idf or topic modelling) commonly used in retrospective approaches to traceability [28]. They focus on requirements and specifications, architectural modules, source files, and test cases, making no mention of PRs or issues beyond bug reports. Previous work required expertise in formal modelling [29, 30]. To eliminate this barrier to entry, they instrument developer or decision maker tools, raising the usual deployability and privacy concerns. Indeed,

the solution they present invasively instrumented specific versions of no less than five different tools, including general purpose tools like Firefox and MS Word. Many of these tools are under constant development; updating this instrumentation out-of-tree is extremely expensive, suggesting that Ståhl *et al.*'s light-weight approach is more likely to be maintainable. Asuncion *et al.* [24] in follow-up work incorporate an LDA model into their framework to improve the interpretability of their traces. To remain online, they recompute an LDA model on demand for each graph visualisation and search query.

Additionally, Falessi *et al.* [31] have done work towards quantifying the number of links that are left to be recovered. Their work assumes an offline, closed-world setting; they provide a framework that can help an analyst decide when to stop pursuing a traceability maintenance task. The aim of this work is to provide a quantifiable signal of when the traceability task should be finished as linking can be considered good enough.

Work in general traceability, due to its broader nature, deals with a profusion of formats and has, so far, either resorted to intrusive instrumentation [27, 24], or defining a protocol which tools must implement and communicate via [5]. The more limited scope of this work allows us to exploit known structures that developers using platforms such as GitHub naturally produce: namely issues and pull-requests. This enables Aide-mémoire to rely on API access from GitHub, or, with minimal additional effort, JIRA, rather than instrument the developer environment (Section 3.3.2).

2.2.3 Commit-Issue Link Inference

The *missing link problem* is the offline prediction problem of inferring missing commit-issue links given a version history and issue tracker archive. Bachman *et al.* were the first to formulate and quantify this problem [32, 12]. Their work aids developers indirectly by helping researchers and tool-smiths avoid the bias introduced by missing links that could undermine their techniques or tools. Specifically, they show that by assuming recorded links to be representative of all links, tools are biased to use code from more experienced developers, thus not learning from mistakes or bugs introduced by less experienced contributors.

Bachman *et al.* together with Apache Commons developers manually supplied missing links and published their Apache Commons corpus. Wo *et al.* are the first to attempt to automatically infer them. They propose ReLink [33], which measures the similarity of change logs and bug reports with cosine similarity on tf-idf vectors, learning a threshold for true links. Nguyen *et al.* exploited commit and issue tracker metadata in MLink [34] to improve recall over ReLink. They evaluated MLink on the Apache Commons corpus, making it the de facto standard. ReLink and MLink first consider only commit data to form an initial set of commit-issue links that they then filter. Prechlet and Pepper [35] dispense with the initial commit-only stage, and instead consider both commit and issue data from the start. They argue this bi-directional inference is more sound. Their BFLinks proposes two link predictors (based on bug and commit IDs) and a series of filters

to reduce the candidate set of links.

RCLinker [36] further improves recall. RCLinker relies on ChangeScribe [37] to produce textual descriptions of commits, especially those that lack commit messages. PRs have their own message and aggregate multiple commits and their messages. This fact alleviates the problem of sparse commit descriptions when considering issue-PR interlinking. PR descriptions for some projects can be of low quality, impacting those that would rely on them. Liu *et al.* [38] propose a tool, based on a bi-directional RNN with a copy network, to tackle PR summarisation, which can help side-step the low-quality PR descriptions. Section 3.6 provides a more detailed description of RCLinker.

Sun *et al.* [39] used non-source files in commits for commit-issue link inference. They argue that these files are important for capturing developer intent. They use the standard heuristics, such as checking for camelCase or snake_case, to determine the relevancy of a non-source file in a commit. As is conventional, they implement these heuristics as regexes. They use the resulting set of non-source files together with the co-committed source files to compute textual features. They scan a preset and fixed list of the similarity thresholds to find the maximum F1-score where Recall is at least 0.80. The procedure raises two unanswered questions. First, how did the authors determine the threshold granularity? Second, how does training FRLink on F1-score for a task whose performance is measured in terms of F1-score avoid overfitting? They report these choices allow FRLink to improve Recall over previous work while matching or improving F1-score.

Sun *et al.* evaluate FRLink on a new corpus of GitHub projects. This corpus differs from the Apache corpus used in prior work in the conventions governing commit messages: Apache messages tend to be descriptive [40], while FRLink's GitHub sample tends to contain exact matches, because copying issue text is common practice [41]. Sun *et al.* do not investigate the effect of this differing practice on their results. They specify their corpus in sufficient detail to reconstruct it. FRLink, the tool, however, is not available. When we tried to reproduce FRLink, using its description in Sun *et al.*'s paper, we were unable to reproduce the reported results. We contacted the authors for help explaining and correcting our reproduction without response. More recently, Sun *et al.* [42] treat existing links as labels and reformulate the missing link problem into a semi-supervised problem. As Bachmann *et al.* found, existing links are biased; Sun *et al.* do not discuss how they coped with this bias. They report that their solution, PULink, outperforms FRLink on FRLink's corpus. Like FRLink, PULink is not available.

Ruan *et al.* [41] empirically studied the state of commit-issue linking on GitHub Java projects and found only 42.2% to be linked. They propose DeepLink, a neural approach to the missing links problem. DeepLink trains a text embedding for non-source artefacts and a code embedding for source artefacts using the skip-gram model [43, 44], then passes each of these embeddings separately through an LSTM layer to obtain the final vector representations. They use cosine

similarity to compare the vectors, choosing the maximum similarity to represent the score of the commit-issue link. They show an improvement over FRLink in terms of F1-score, and further show that pre-processing heuristics similar to previous work, such as ReLink [33] or MLink [34], help DeepLink achieve a higher F1-score. They also spot that the FRLink corpus had commit logs and issue titles that are exact matches, introducing bias in the dataset which Ruan *et al.* handle, while FRLink does not. Since they did not evaluate DeepLink on Apache Commons or against RCLinker and DeepLink is not available, we do not know its performance relative to RCLinker.

Rath *et al.* [45] also tackle the missing link problem, but from within the requirement engineering community and without referencing the line of work stemming from Bachmann *et al.* The missing link inference work above uses a vector space model over unigrams for textual features. They opt instead for a n-gram model. They are the first to perform feature selection, using Weka's feature auto-selection. They report promising results but on a different dataset than Apache Commons.

Work focusing on the narrower traceability area of commit-issue linking has so far been offline, focusing on fixing the situation after the fact, and working at the commit level. The commit to issue mapping, however, is not one-to-one [46]. Indeed, current practice on GitHub suggests that the one-to-one mapping be from pull-requests, that may contain a feature branch, onto issues (Section 3.7.1). Still, Section 3.7.1 also shows that the state of pull-request-issue linking is non-ideal. To address these issues, we focused Aide-mémoire as an online pull-request-issue linking tool. Its online nature makes it complementary to offline approaches, while it has a lower recall, it can still improve the training data of offline approaches and allow for better link recovery. Its shift to pull-request focuses it on the modern development practice that is now common on platforms such as GitHub.

2.3 Towards Atomic Commits

A potential source of inaccuracy and noise for the task of maintaining links between commits and pull-requests is multi-concern commits. A developer, as part of a bug fix or indeed to facilitate said bug fix, can and often do mix refactoring changes with fixes. These are then committed as a single patch. Such commits represent tangled commits. The ideal state of a version history would be to contain only atomic commits, *i.e.* those that tackle a single task (would be linked to a single issue or implements a single stakeholder concern). Changes unrelated to the fix may confuse statistical tools that strive to maintain traceability links due to the added noise, thus approaches to (1) detect multi-concern commits and (2) slice multi-concern commits into atomic patches may aid reduce this source of noise. Further, this can have an added benefit for developers since tools such as `git bisect` would also benefit from such a segmentation of commits.

2.3.1 Impact of Tangled Commits

Tao *et al.* [47] were amongst the first to highlight the problem of change decomposition in their study on programmer code comprehension; they highlight the need for decomposition when many files are touched, multiple features implemented, or multiple bug fixes committed. The last is diagnosed by Murphy-Hill *et al.* [48] as a deliberate practice to improve programmer productivity. Tao *et al.* conclude that decomposition is required to aid developer understanding of code changes.

Independently, Herzig *et al.* [10, 9] investigate the impact of tangled commits on classification and regression tasks within software engineering research. The authors manually classify a corpora of real-world changesets as atomic, tangled or unknown, and find that the fraction of tangled commits in a series of version histories ranges from 7% to 20%; they also find that most projects contain a maximum of four tangled concerns per commit, which is consistent with previous findings by Kawrykow and Robillard [46]. They find that non-atomic commits significantly impact the accuracy of classification and regression tasks such as fault localisation.

In this thesis, we borrow the approach that Herzig *et al.* [9] propose for generating a plausible dataset of tangled commits which we use to compare previous approaches and our proposed approach to untangling commits (Section 4.5.1).

2.3.2 Untangling Commits into Atomic Patches

Research on the impact of both tangled commits and non-essential code changes prompted an investigation into changeset decomposition. Herzig *et al.* [10, 9] apply confidence voters in concert with agglomerative clustering to decompose changesets with promising results, achieving an accuracy of 0.58-0.80 on an artificially constructed dataset that mimics common causes of tangled commits. In contrast, Kirinuki *et al.* [49, 50] compile a database of atomic patterns to aid the identification of tangled commits; they manually classify the resulting decompositions as True, False, or Unclear, and find more than half of the commits are correctly identified as tangled. The authors recognise that employing a database introduces bias into the system and may necessitate moderation via heuristics, such as ignoring changes that are too fine-grained or add dependencies.

Other approaches rely on dependency graphs and use-define chains: Roover *et al.* [51] use a slicing approach to segment commits across a Program Dependency Graph, and correctly classify commits as (un)tangled in over 90% of the cases for the systems studied, excluding some projects where they are hampered by toolchain limitations. They propose, but do not implement, the use of System Dependency Graphs to reduce some of the limitations of their approach, such as being solely intraprocedural.

Barnett *et al.* [52] implement and evaluate a commit-untangling prototype. This prototype projects commits onto def-use chains, clusters the results, then classifies the clusters as trivial

or non-trivial. A cluster is trivial if its def-use chains all fall into the same method. Barnett *et al.* employ a mixed approach to evaluate their prototype. They manually investigated results with few non-trivial clusters (0-1), finding that their approach correctly separated 4 of 6 non-atomic commits, or many non-trivial clusters (> 5), finding that, in all cases, their prototype's sole reliance on def-use chains lead to excessive clustering. For results containing 2–5 clusters, they conducted a user-study. They found that 16 out of the 20 developers surveyed agreed that the presented clusters were correct and complete. This result is strong evidence that their lightweight and elegant approach is useful, especially to the tangled commits that Microsoft developers encounter day-to-day. During the interviews, multiple developers agreed that the changeset analysed did indeed tangle two different tasks, sometimes even confirming that developers had themselves separated the commit in question after review. In addition to validating their prototype, their interviews also found evidence for the need for commit decomposition tools. Because they use def-use chains and ignore trivial clusters, Barnett *et al.*'s approach can miss tangled concerns that are in the same method. Barnett *et al.*'s user study itself shows that this can matter: it reports that some developers disagreed with the classification of some changesets as trivial.

Dias *et al.* [53] take a more developer-centric approach and propose the EpiceaUntangler tool. They instrument the Eclipse IDE and use confidence voters over fine-grained IDE events that are later converted into a similarity score via a Random Forest Regressor. This score is used similarly to Herzig *et al.* [9]'s metrics, *i.e.* to perform agglomerative clustering. They take an instrumentation-based approach to harvest information that would otherwise be lost, such as changes that override earlier ones. This approach also avoids relying on static analysis. They report a high median success rate of 91% when used by developers during a two-week study. While Dias *et al.* sidestep static analysis, they require developers to use an instrumented IDE.

Previous literature proposed approaches were either simple and, by virtue of the simplicity, efficient [52], made use of heuristics to construct their similarities and decompose-recluster commits [9], or relied on heavy IDE instrumentation [53]. In Chapter 4, we propose an approach that does not rely on heuristics, instead uses similarities over a multi-version representation of code in a decompose-recluster method similar to Herzig *et al.* While slower than Barnett *et al.*, it is within 10 seconds (Section 4.4.3), hence compatible with CI/CD, while being more accurate (13% accuracy vs 81%). Further, Flexeme does not require project instrumentation and its static analysis can be done incrementally as a post-commit script.

2.3.3 Multiversion Representations of Code

Related work has considered multiversion representations of programs for static analysis. Kim and Notkin [54] investigate the applicability of different techniques for matching elements between different versions of a program. They examine different program representations, such as String, AST, CFG, Binary or a combination of these as well as the tools that work on them on two

hypothetical scenarios. They only consider the ability of the tools to match elements across versions and leave the compact representation of a multiversion structure as future work. Some of the conclusions from the matching challenges presented by Kim and Notkin [54] are echoed in Flexeme as well, we make use of the UNIX diff as it is stored within version histories; however, we also make use of line-span hints from the compilers for each version of the application to better facilitate matching nodes within a NFG.

Le *et al.* [55] propose a Multiversion Interprocedural Control Graph (MVICFG) for efficient and scalable multiversion patch verification over systems such as the PuTTY SSH client. Alexandru *et al.* [56] generalise the Le *et al.* MVICFG construction to arbitrary software artefacts by constructing a framework that creates a multiversion representation of concrete syntax trees for a git project. They adopt a generic ANTLR parser, allowing them to be language agnostic, and achieve scalability by state sharing and storing the multi-revision graph structure in a sparse data structure. They show the usefulness of such a framework by means of 'McCabe's Complexity', which they implement in this framework such that it is language agnostic, does not repeat computations unnecessarily and reuses the data stores in the sparse graph by propagating from child to parent node. Sebastian and Harald [57] propose a compact, multiversion AST that cleverly shares state across versions.

Previous work mostly considered multiversion representation of code for static analysis or the compute of software metrics. We instead focus on multiversion representations of code as a initial structure from which we can segment atomic commits. To that end, we propose two new multiversion graph representations of code, firstly, the δ-PDG, which generalises Le *et al.*'s MVICFG to PDGs, and the δ-NFG that further augments δ-NFGs with nameflow information (Section 4.3.1), which we hoped to enable us to better approximate the location of concerns in code via naming conventions.

2.3.4 Semantic Slicing of Version Histories

Features in a system often co-evolve, which tangles the changes made for a one high-level feature with others in a version history. To resurface feature-specific changes, one can dynamically slice a target version, then walk backwards in history while they can reverse the intra-version patch without conflict; at each version they reach, they add any commit that contains a hunk that touches the current slice to it. The goal of this semantic slicing of version histories is to find a minimal slice of a version history that captures the evolution of a feature. Li *et al.* [58, 59] first formulated and introduced this problem. Semantic slicing is a form of commit untangling backwards through history. This retrospective framing is why they treat the history as immutable. In this initial solution, Li *et al.* treat commits as atomic so their slices may contain noise introduced by tangled commits. To reduce this noise, Li *et al.*, in more recent work [60], unpack commits into single-file commits into a private, local history.

Flexeme, in contrast, is static and online: built from the ground up to rewrite commits as developer make history. As such Flexeme and semantic slicing are complementary: Flexeme would improve the signal to noise ratio of semantic slicing. An interesting direction for future work would be to use Flexeme to preprocess version histories prior to semantically slicing them as with Definer [60].

2.4 Discerning Text from Code

One of the meta-task that can aid the traceability aspect of this research is the segmentation of English, or natural languages in a more general sense, from formal languages. While online fora perform a best effort attempt at maintaining separate formatting for the two modalities of text, this signal remains noisy [61]. The work carried out in this thesis seeks to improve the situation by framing the task as a code-switching phenomenon and borrowing results from the relevant areas of the Natural Language Processing (NLP) community (Chapter 5).

In this section, we will focus on first providing how similar techniques were already employed within the Software Engineering (SE) community, followed by the more relevant for this task research from the NLP community, concluding with Ponzanelli *et al.*'s work on Stack Overflow which is closest to POSIT.

2.4.1 Part of Speech Tagging in Software Engineering

In SE research, part-of-speech tagging has been directly applied for identifier naming [62], code summarisation [63, 64], concept localisation [65], traceability-link recovery [66], and bug fixing [67]. The main intuition behind the statistical approaches in this are are due to the naturalness of code as discussed by Hindle *et al.* [68].

Operating directly on source code (not mixed text), Newman *et al.* [69] created source code equivalents for lexeme categories from natural languages. They looked at the behaviour exhibited by Proper Nouns, Nouns, Pronouns, Adjectives, and Verbs and derived similar notions for source-code from 1) Abstract syntax trees, 2) how the tokens impact memory, 3) where they are declared, and 4) what type they have. They report the prevalence of these categories in source-code. Their goal was to map these code categories to PoS tags, thereby building a bridge for applying NLP techniques to code for tasks such as program comprehension.

Treude *et al.* [11] described the challenges of analysing software documentation written in Portuguese which commonly mixes two natural languages (Portuguese and English) as well as code. They suggested the introduction of a new part-of-speech tag called Lexical Item to capture cases where the "correct" tag cannot be determined easily due to language switching.

Instead of adapting code to part-of-speech akin to Newman *et al.* [69], in this thesis, we instead wish to treat the problem as a multi-lingual parsing problem, *i.e.* we want to know what part of speech or part of code each token is for all languages that we may consider. In POSIT, we

realised a first step towards this by considering the single natural language and a single C-like source code case. Indeed, it is still an open problem to tackle situations such as those presented by Treude *et al.* [11].

2.4.2 Part of Speech Tagging in Code-Switched Natural Languages

NLP researchers are growing more interested in code-switching text and speech, *i.e.* that mixes multiple natural languages, among other reasons due to the higher availability of corpora. These were made available either by dedicated collections efforts, such as Miami Bangor [70], or due to social media sites, such as twitter, that cause the mixing of English with other languages [71]. Previously, such data was scarce because code-switching was stigmatised [72].

Focusing specifically on part-of-speech tagging, Solorio and Liu [73] presented the first statistical approach to the task of part-of-speech (PoS) tagging code-switching text. On a Spanglish corpus, one that mixes Spanish and English, they heuristically combine PoS taggers trained on larger monolingual corpora and obtain 85% accuracy. We can think of this approach as a mixtures of experts model. Jamatia *et al.* [74], working on an English-Hindi corpus gathered from Facebook and Twitter, recreated Solorio's and Liu's tagger and additionally proposed a tagger built on Conditional Random Fields. The mixtures of experts model performed better at 72% vs 71.6%. In 2018, Soto and Hirschberg [75] proposed a neural network approach, opting to solve two related problems simultaneously: part-of-speech tagging and Language ID tagging. The second task can be thought of as a segmentation task, which in our English-code setting can be rendered using formatting for code-blocks. They combined a biLSTM with a CRF network at both outputs and fused the two learning targets by simply summing the respective losses. This network achieves a test accuracy of 90.25% on the inter-sentential code-switched dataset from Miami Bangor [70].

The languages that previous literature considered for code-switching tasks may have different tag distributions or even tag sets; although not as disjoint as in our case with POSIT. This provided an initial starting point for promising architectures. Section 5.4 details how we combined these approaches to enables us to solve a version of our joint tagging and segmentation task.

2.4.3 Mixed-Text: When Natural Meets Formal

In the context of natural languages mixed with source code and other formal languages (mixed-text), Ponzanelli *et al.* are the first to go beyond using regular expressions to parse such mixed text. When customising LexRank [76], a summarisation tool for mixed text, they employed an island grammar that parses Java and stack-trace islands embedded in natural language, which is relegated to water. They followed up LexRank with StORMeD, a tool that uses an island grammar to parse Java, JSON, and XML islands in mixed text Stack Overflow posts, again relegating natural language to water [77]. StORMeD produces heterogeneous abstract syntax trees (AST), which are ASTs decorated with natural language snippets. They make both the tool and the

resulting corpus available. StORMeD, however, remains reliant on either code-block annotations or heuristics to discern code from English. POSIT instead learns this segmentation from data, and, after a manual evaluation, we indeed observed it to be capable of segmenting Stack Overflow posts where submitters forgotten to include the appropriate formatting (Section 5.6).

3

Aide-mémoire: Improving a Project's Collective Memory via Pull Request–Issue Links

Paper Authors

Profir-Petru Pârțachi, Department of Computer Science, University College London, United Kingdom

David R. White, University of Sheffield, United Kingdom

Earl T. Barr, Department of Computer Science, University College London, United Kingdom

Abstract — Links between pull-requests and the issues they address document and accelerate the development of a software project, but are often omitted. We present a new tool, Aide-mémoire, to suggest such links when a developer submits a pull-request or closes an issue, smoothly integrating into existing workflows. In contrast to previous state of the art approaches that repair related commit histories, Aide-mémoire is designed for continuous, real-time and long-term use, employing Mondrian Forests to adapt over a project's lifetime and continuously improve traceability. Aide-mémoire is tailored for two specific instances of the general traceability problem — namely, commit to issue and pull-request (PR) to issue links, with a focus on the latter — and exploits data inherent to these two problems to outperform tools for general purpose link recovery. Our approach is online, language-agnostic, and scalable. We evaluate over a corpus of 213 projects and six programming languages, achieving a mean average precision of 0.95. Adopting Aide-mémoire is both efficient and effective: a programmer need only evaluate a single suggested link 94% of the time, and 16% of all discovered links were originally missed by developers.

3.1 Introduction

Traceability is the maintenance of relationships between software development artefacts; the most important of these relationships is the link between requirements and their implementation. In the “move fast and break things” era, the addition and maintenance of links are too often neglected. Good traceability practices and tooling can improve all aspects of software development, from requirements elicitation to code maintenance. As such, traceability is a seminal software engineering concern.

In modern development, issues track outstanding work, both reported bugs and feature requests. A Pull-Request (PR) is a sequence of patches submitted for reviewing and merging into a project’s mainline, as illustrated in Figure 3.1. Developers work with issues and PRs, day to day; they are interlinked in a developer’s mind. When these traceability links are recorded, they accelerate software development because developers can use them to restore context [78, 5]. They keep teams informed of progress on feature enhancement and prevent commit reversion and issue reopening by connecting the commits within a PR with the issues they address [79, 80]. Developers use them to prioritise work; reviewers use them to learn the context of an issue. They facilitate fault prediction [81], bug localisation [82, 83, 84], and issue triage.

Despite their importance, these links, like other traceability links, are often not recorded or maintained. Although modern tools, like JIRA and GitHub, provide increased support for linking, developers do not record most links [32]. We confirm this trend in a large-scale analysis of repositories: over half (54%) of PRs are not linked to an issue when submitted, despite the fact that a third of project contribution guidelines recommend linking (Section 3.7.1). During PR review, missing links are sometimes discovered manually and the PR amended. Around 16% of PRs are linked during this process, leaving 38% unlinked.

We present *Aide-mémoire* (A-M), a tool that suggests pertinent PR-issue links to a developer. We call it Aide-mémoire because its aim is to help a project partially retain its “collective memory”. A-M is *online*: it suggests a link when a developer submits a PR or closes an issue. This is critical for uptake. Offering suggestions to developers when they need not context-switch is critical. In his ICSE 2020 Keynote, Peter O’Hearn remarked that an analysis run as a batched process had 0% developer uptake, which jumped to 70% when the same suggestion is made during code-review [85]. A-M does not require invasive instrumentation, but relies instead on the content that developers already produce: commit logs, PRs, and posts on discussion boards. It needs only tokenise its inputs, so it is language-agnostic and this, coupled with the fact that it builds its online classification model incrementally, allows it to scale to large code bases. To train A-M, we mine GitHub projects and their issue trackers. A hurdle all tools that aim to improve developer workflows face is Agile’s “lightweight” requirement [86]: tool adoption and use must quickly pay for itself. A-M’s principal goal is to make the cost of creating and maintaining PR-Issue links so

easy and seamless as to meet this demanding requirement (Section 3.3).

While leaving feature selection to neural architectures is common today, it incurs a higher training cost or requires additional training data. In machine learning approaches other than deep learning, feature selection is a critical step that can make or break a model. With the advent of neural networks in SE, a scan of recent papers will reveal that feature selection has been neglected. We turn to feature selection to speed both A-M's training and, as it is online, its adaptation to new linking regimes. Section 3.4 carefully details this process and can serve as a primer on feature selection for other software engineers who may find it useful in their work.

The state of the art commit-issue prediction tool is RCLinker [36]. RCLinker is offline and handles commits, not pull-requests, and is limited to Java projects, since it requires ChangeScribe [37] to generate natural language descriptions of changesets to Java code. To validate A-M against RCLinker, we therefore had to adapt it (Section 3.6). We call our variant RCRep; to handle PRs, it replaces ChangeScribe summaries with user-provided PR descriptions, the first post in a PR, which is semi-structured by convention. RCRep suggests a pull-request when its internal RCLinker predictor suggests any commit within that pull-request. Because offline is a degenerate case of online, we run A-M offline to compare it with RCRep; we take care to ensure neither tool sees events from the future. We show that A-M is more precise than RCRep, achieving a mean Precision of 0.76 and F1-Score of 0.46 compared to RCRep's 0.14 and 0.15, with a similar Recall (0.37 vs 0.36) across 47 Java projects (Section 3.6.2).

Having established A-M's performance against a baseline, we evaluate it in its native setting: online over a multi-lingual corpus of 213 projects, which contains a range of project sizes and programming languages (Section 3.5.1). We train A-M on project history prefixes of fixed length relative to project size and validate it on the suffix by replaying repository events in chronological order. It achieves high accuracy: a Mean Average Precision (MAP) of 0.95 (Section 3.7.2). We also show that A-M generalises well: there is no statistically significant difference with project size or across languages in performance. A-M maintains performance on projects with over a thousand open issues and hundreds of monthly pull-requests; Section 3.7.2 shows that there is no statistical correlation between any of the performance metrics and project size.

To further validate A-M, we also conducted a modest user study, using convenience sampling (Section 3.5.4). We asked users to find PR–issue links using either A-M or GitHub's built-in search facilities. We compared user performance in terms of time taken per linking task and perceived linking difficulty.

We designed and built A-M to seamlessly integrate with existing developer workflows; A-M must augment, not disrupt them (Section 3.3.2). Therefore, A-M must be highly precise to avoid distracting developers with useless suggestions they discard. Our finding that A-M achieves 0.95 MAP and the results of our user study both suggest that it meets this requirement. An offline approach, like RCLinker, must be periodically retrained, while an online model, like A-M, can learn

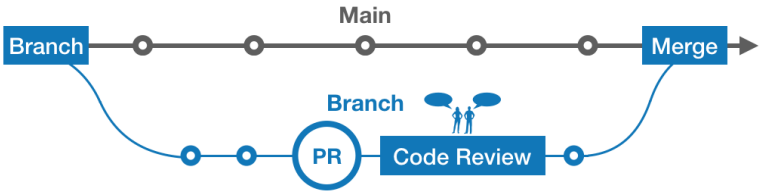


Figure 3.1: Simplified pull-request process. Each small circle represents a commit. A developer opens a new branch, makes code changes and submits a pull-request to code review. Further changes may be made on the branch before being accepted and merged.

as the project evolves. Thus, A-M does not require a dedicated maintenance task, substantially enhancing its deployability. Finally, installing A-M only requires a lightweight backend that can be installed locally or onto a server for sharing, and a Chrome plugin.

Our main contributions follow:

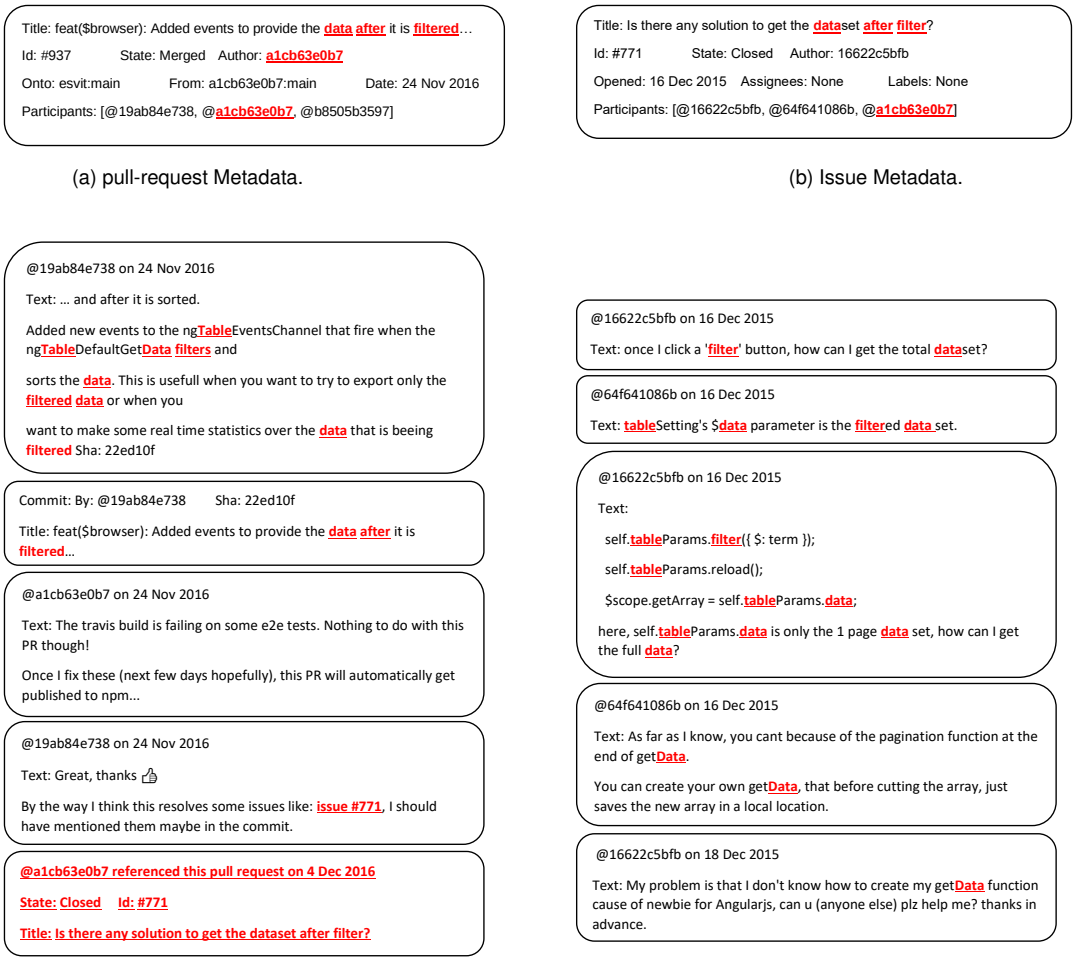
- We present the design and implementation of A-M, a tool that solves the PR-issue link inference problem via online classification, providing pertinent suggestions;
- We evaluate A-M on a large and diverse corpus, and demonstrate that our approach generalises across languages and scales to large projects containing over a thousand open issues and hundreds of PRs per month;
- We show that A-M can exploit information in PRs to outperform related work that solves the traditional offline *commit-issue* linking problem when applied to PRs.

All tools, data and scripts needed to reproduce our work are available at `https://github.com/PPPI/a-m`.

3.2 Motivating Example

Figure 3.1 overviews a typical modern development process: a PR consisting of a set of changes to resolve an issue is submitted for code review. If the link between the PR and issue is not recorded, the issue remains open and the record of why a PR was made is lost.

Figure 3.2 gives an example of an unlinked PR and corresponding issue, where the issue was open but initially missed by the PR submitter. The example is taken from the ng–table project, a table library for AngularJS. On the left of the figure is a pull-request containing code changes that makes it possible for a developer to access the original data of a table after it has been filtered or sorted; this PR addressed the issue in the right of the figure, but was not linked at the time the PR was submitted. Both titles and conversations discuss filtering and mention common terms such as the identifier fragment ‘getData’. This causes a high textual similarity between the titles as can be seen in Figure 3.2a and Figure 3.2b. Moreover, there is significant textual similarity between the PR description, the first posting in Figure 3.2c, and the first three postings in Figure 3.2d. The particular sub-tokens that overlap are highlighted in the title and postings.



(a) pull-request Metadata.

(b) Issue Metadata.

(c) pull-request Discussion.

(d) Issue Discussion (trimmed for typesetting reasons.)

Figure 3.2: Example of a pull-request and a related issue. The PR was not linked to the issue when submitted, but the missing link was subsequently added manually by the submitter after 10 days. The title and discussion share common terminology as well as common participants that can be exploited by Aide-mémoire. Some of the more useful ones are highlighted in red along with the reference message that was created upon closing the issue. Developer names have been pseudo-anonymised.

There are also common participants (identifier @a1cb63e0b7): namely, the PR submitter. Their only interaction is to reference the PR and close the issue, which was done 10 days after the submission of the PR as can be seen in the highlighted message. By exploiting these and other features, Aide-mémoire assists developers by suggesting links at PR submission and issue closure and can reduce such delays.

3.3 Aide-mémoire

Aide-mémoire is the first online PR-issue linking tool. Figure 3.3 gives an overview of the methodology and design of Aide-mémoire. We start by obtaining a list of observed links already recorded within the project issue tracker. We then determine the feature space for A-M, which we

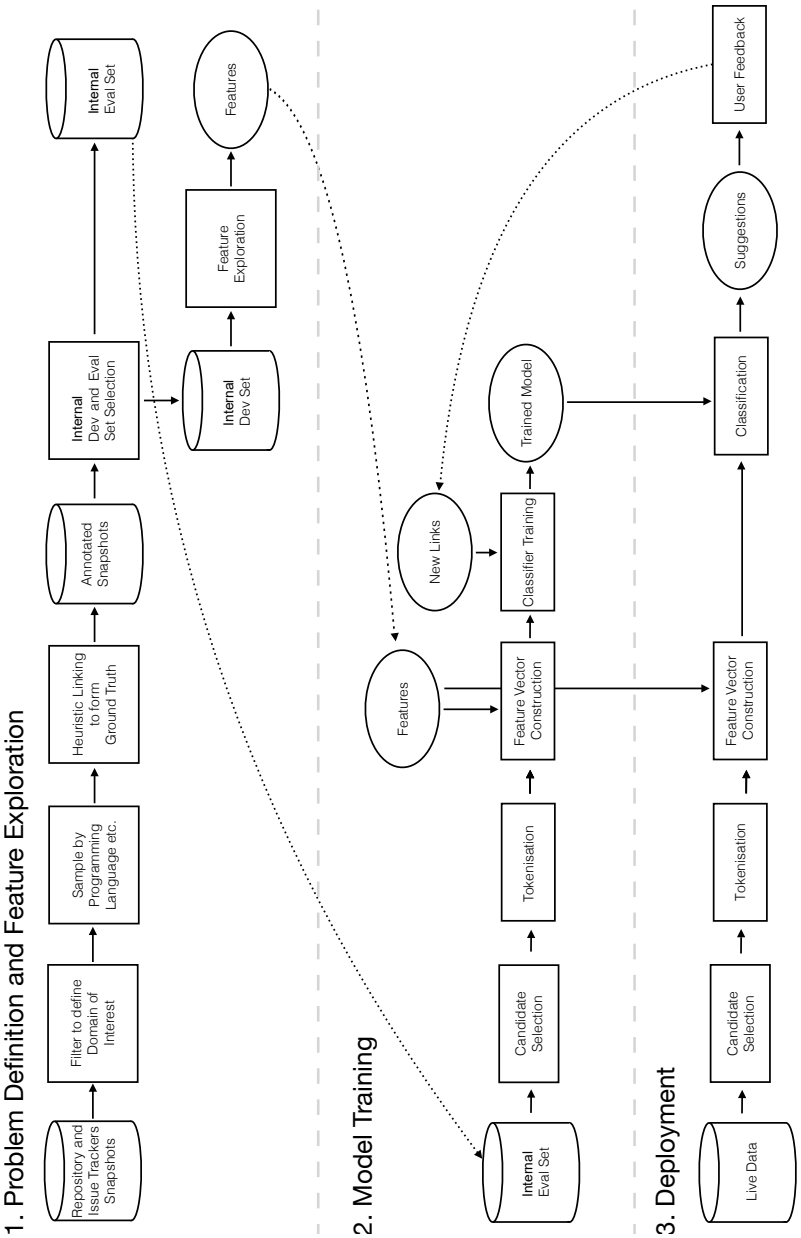


Figure 3.3: Methodology used for instantiating Aide-mémoire. In the first stage, a datastore of repositories is identified, filtered and sampled to define the domain of interest. Heuristic linking generates an initial knowledge to learn and evaluate against. A subset of selected repositories is used to identify powerful features. In Stage 2, the annotated repositories are used to learn a predictive model based on the selected features. In Stage 3, live data from the repositories is used to generate feature vectors and suggest candidate links to a developer using the trained model. Feature exploration is not required for the implementation, in which case the Annotated Snapshots will be used directly in Stage 2 and all features will be considered.

detail in Section 3.4. This process is performed once for our evaluation, but may be performed on a per project basis when deployed. Next, we use the features to learn a model over a set of repository snapshots. We subsequently deploy the learnt model to suggest links when developers submit PRs or close issues.

In solving an online linking problem, we must narrow the set of candidate links we consider to ensure that our system remains responsive. When suggesting issues at PR submission we limit ourselves to *open* issues; one of the main motivations for linking PRs and issues is to ensure the automatic closure of an issue if a PR is merged. For the symmetric problem of suggesting PRs to be linked to a given issue, we use a seven day window of recently submitted PRs, in line with previous work [34].

3.3.1 Model Learning

We train a statistical classifier on PR-issue pairs to learn a probability distribution over possible links. For each candidate PR-issue link, we calculate the feature vector as detailed in Section 3.4 and train the classifier to learn the probability that the link should exist. In prediction mode, we sort links by this probability when presenting suggestions to a developer. As most candidate pairs represent false links, our data is class-imbalanced; however, we observe empirically on our development set that Mondrian Forests perform well despite this imbalance and hence we do not employ undersampling.

We deviate from the more classical Random Forest used by the state-of-the-art offline tool [36], and instead employ the online method of Mondrian Forests [87]. Mondrian Forests represent a class of Random Forests that employ the Mondrian Process to partition the feature space. This process can be interpreted as a stochastic kd-tree. We hypothesise that their resilience to class imbalance arises from their ability to infer tight bounding boxes around positive examples; further investigation which lies outside the scope of this paper is needed to validate this claim.

Mondrian Forests work well in the online case; training them with sequential examples is equivalent to batch training in the limit [87]. We configure the Mondrian Forest to use 128 estimators, assuming that the recommendation made by Oshiro *et al.* [88] for Decision Tree based Random Forests extends to Mondrian Forests as well. Additionally, since probability estimates are obtained by majority voting, this enables us to use the model to obtain finer-grained probability estimates. Once initially trained, we deploy the classifier to provide suggestions to the developers and learn online as further links are created.

We add special ‘no_pr’ and ‘no_issue’ entities that represent the absence of any link, with their structures populated with empty strings and null timestamps. These special cases allow us to explicitly learn when no link should be proposed. We truncate suggestion lists at the index of these special entities, using them as a ‘tidemark’, excluding predictions that are less likely than a link to an empty issue/pr. Learning when these entities apply relies on our use of features that depend on only one side of the link.

Explicitly recording the absence of a link requires the learner to solve two problems at the same time: ‘should there be any link?’ and ‘what should the artefact be linked to?’. Previous

work only focused on the latter, suggesting no link only when no suggestion could be made above some threshold, which itself was learnt post-hoc. Our solution allows the model to learn a per-suggestion threshold as part of it, providing a natural cut-off point when prompting a developer with a list of suggested links.

3.3.2 Deployment

We built A-M to vault Agile's lightweight requirement [86]. A-M's use requires only installing a browser plugin and a backend server. Training requires only the URL of a GitHub repository. A-M can search for links to suggest in parallel to other development tasks, so it seamlessly integrates into existing workflows.

A-M's back-end learns, maintains, and stores the project's model; its Chrome plug-in front-end parses issue and PR pages. Although a developer team would benefit from sharing A-M's backend, individual developers can install both the server and the plugin locally. Developers interact with Aide-mémoire via its plug-in when viewing a PR or issue. The plugin suggests links when a developer closes an issue or submits or reviews a PR. A-M only makes high confidence suggestions (controlled by a user-specified threshold) and displays them in rank order. A-M makes no suggestions that are less likely than the special 'no_pr' and 'no_issue' entities, staying silent when its confidence is low.

To learn a model for their project, a developer can install A-M and generate a model locally. To start training, a developer need only enter their GitHub URL into A-M's command line. This instigates the crawling and processing of their repository. Alternatively, project managers can install a central backend. The initial training of our system over a large project such as Google's Guava, which contained 206 PRs and 2563 issues, and a total of 5392 commits when we crawled it, takes less than two hours on an Intel i7-6820HK@2.70 GHz.

Our prototype plug-in subsequently makes suggestions to a developer in, on average, less than 10s of the completion of their PR or issue closing message. The suggestion itself is fast (subsecond); most of the delay is parsing the issue or PR GitHub page and sending the extracted data to the backend. The expensive HTML wrangling and network roundtrip can be overlapped with, and hidden beyond, a developer's other activities. For instance, a developer need only click on the plug-in to start the process. While it's running in the background, they can continue work on their PR submission or issue triaging. Future engineering work could further mask this cost behind continuous integration, by including it as an additional pass in the build file.

Full source code, deployment tools, and code and scripts required to recreate our evaluation can be found online [89].

3.4 Exploring the Feature Space of Issue-PR Links

In this, the neural era, feature exploration is unfashionable; the networks are left to sort out features on their own at the cost of training time and data. Most projects are small and all start small, and usually lack sufficient labelled data to train a neural network. A-M is not neural because we wanted it to apply to small projects. So, we now describe the traditional feature engineering [90] that underlies A-M. Despite being, or perhaps because it is, out of fashion, software engineering researchers who use ML may find this section useful as a primer. Even neural networks benefit when feature engineering helps architect the network, narrows wide data to the width of a network's input layer, or boosts signal, and hence reduces the amount of training data and training time needed. We first enumerate features drawn from the literature and augment with new features we devised. This collection is comprehensive to the best of our knowledge. Using the full feature set is costly, both in development time and, crucially, when deployed. So, in closing, we show how we reduced our features while retaining synergistic interactions and maintaining A-M's performance by employing Recursive Feature Elimination [91, 90].

3.4.1 Feature Space Construction

Issues and PRs are complicated structures. To classify them, we need textual similarity measures. After defining the measures we use, we detail how we construct the full feature space to capture the structure of issues and PRs.

Similarity Measures: In order to compare the subject of an issue and a PR, accompanying text documents such as commit messages, source code changes, and conversations are transformed into a vector representation using tf-idf, a discriminative model operating at the fine-grained level of *term frequencies*. We preprocess all documents to: remove punctuation, split tokens using whitespace and code conventions (as well as retaining unsplit tokens in the case of identifiers), stem [92], remove stopwords, and exclude single-character tokens.

We use tf-idf for similarity matching between documents; it represents state of the art within offline approaches [36, 12, 33] and can be naturally extended to the open-world setting by maintaining an idf estimate that we update dynamically. We transform documents in our corpus D into term vectors to enable comparison. The value for each term t is the term frequency *tf*, the number of times it occurs in a given document d weighted by that term's inverse relative frequency in the corpus:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \cdot log_2\left(\frac{|D|}{|\{d' | t \in d', d' \in D\}|}\right). \tag{3.1}$$

We replace terms that are either too rare (occur a single time in our corpus) or are too frequent (present in more than 95% of the documents in the corpus) with the unknown, or out-of-vocabulary, token. We make the non-standard choice for tf-idf parameters to ensure a larger vocabulary for small projects as standard practice would induce too many unknown tokens.

We choose to eliminate only those terms that occur a single time to maintain a diverse vocabulary. We use the tf-idf implementation from gensim [93], which was designed to handle large corpora efficiently.

We then use this representation to compute cosine similarity:

$$CS(p_j, i_k) = \frac{p_j \cdot i_k}{|p_j| \cdot |i_k|}, \tag{3.2}$$

where j may take values from {'title', 'description'} and k from {'title', 'report', 'comment'$_1$, ..., 'comment'$_n$}. Only CS$_{\text{full-context}}$ makes use of all pairs, the other cosine similarity features restrict j and k as shown in Table 3.1.

We also considered Jaccard and Dice similarity, common in traceability literature, which work on the bag-of-words representation directly. This representation is computed as follows:

$$\text{bow}(d) = \{(t,\ \text{tf}(t,d)) | t \in d\}. \tag{3.3}$$

Note that both Jaccard and Dice consider multiplicity when computing intersections and unions of bag-of-word representations. For comparisons employing Jaccard or Dice, we use the bag-of-words model directly:

$$JS(p_j, i_k) = \frac{|\text{bow}(p_j) \cap \text{bow}(i_k)|}{|\text{bow}(p_j) \cup \text{bow}(i_k)|} \tag{3.4}$$

$$Dice(p_j, i_k) = \frac{|\text{bow}(p_j) \cap \text{bow}(i_k)|}{\min(|\text{bow}(p_j)|, |\text{bow}(i_k)|)} \tag{3.5}$$

where j may take values from {'title$'$, 'description$'$} and k from {'title$'$, 'report$'$, 'comment$'_1$, ..., 'comment$'_n$} and bow$(\cdot)$ provides the set of pre-processed tokens contained in the artefact to which we apply it. Only Jaccard$_{\text{full-context}}$ and Dice$_{\text{full-context}}$ make use of all pairs, the other similarity features restrict j and k as shown in Table 3.1.

The Feature Space: Developers construct issues and PRs with an inherent structure over time. As Figure 3.2 showed, PRs and issues can, and often do, overlap in their topics and even their text. Previous work on retrospectively repairing *commit-issue* links in an offline context exploited similar overlap [36, 34, 33, 42]. We build a feature space by considering features employed by the previous state-of-the-art tool [36], which was created from first principles informed by human intuition, adapted to the PR context together with three new variants of cosine similarity as well as their Jaccard and Dice equivalents. Further, we consider seven new features that allow the model to learn the new special entities we employ, as discussed in Section 3.3.1. This link is equivalent to identifying PRs or issues that should be classified as *unlinked*. To capture their properties and interrelations, we consider four groups of features: *textual*, *social*, *temporal*, and *structural*.

Table 3.1: Features constructed from a document vector representation and metadata. p is a PR object, i is an Issue object, e is a traceability link, and π^0 projects the first component of a traceability link. CS denotes cosine similarity and M denotes metadata-based features. A direct reference to an object indicates the concatenation of all text from its constituent parts. We propose the bolded features; the rest are due to Le *et al.* [36].

Feature	Description
$\text{CS}_{\text{full-content}}$	$\text{cs}(\text{p}, \text{i})$
$\mathbf{CS_{title\text{-}title}}$	$\mathbf{cs}(\mathbf{p.title}, \mathbf{i.title})$
$\mathbf{CS_{title\text{-}report}}$	$\mathbf{cs}(\mathbf{p.title}, \mathbf{i.report})$
$\mathbf{CS_{description\text{-}title}}$	$\mathbf{cs}(\mathbf{p.description}, \mathbf{i.title})$
$\mathbf{CS_{description\text{-}report}}$	$\mathbf{cs}(\mathbf{p.description}, \mathbf{i.report})$
$\mathbf{Jaccard_{full\text{-}content}}$	$\text{js}(\text{p}, \text{i})$
$\mathbf{Jaccard_{title\text{-}title}}$	$\mathbf{js}(\mathbf{p.title}, \mathbf{i.title})$
$\mathbf{Jaccard_{title\text{-}report}}$	$\mathbf{js}(\mathbf{p.title}, \mathbf{i.report})$
$\mathbf{Jaccard_{description\text{-}title}}$	$\mathbf{js}(\mathbf{p.description}, \mathbf{i.title})$
$\mathbf{Jaccard_{description\text{-}report}}$	$\mathbf{js}(\mathbf{p.description}, \mathbf{i.report})$
$\mathbf{Dice_{full\text{-}content}}$	$\mathbf{ds}(\mathbf{p}, \mathbf{i})$
$\mathbf{Dice_{title\text{-}title}}$	$\mathbf{ds}(\mathbf{p.title}, \mathbf{i.title})$
$\mathbf{Dice_{title\text{-}report}}$	$\mathbf{ds}(\mathbf{p.title}, \mathbf{i.report})$
$\mathbf{Dice_{description\text{-}title}}$	$\mathbf{ds}(\mathbf{p.description}, \mathbf{i.title})$
$\mathbf{Dice_{description\text{-}report}}$	$\mathbf{ds}(\mathbf{p.description}, \mathbf{i.report})$
files	$\lvert\{\text{filename} \mid \text{filename} \in \text{p}, \text{filename} \in \text{i}\}\rvert$
M_{reporter}	1 if the p.submitter = i.reporter, else 0
M_{assignee}	1 if the p.submitter = i.assignee, else 0
M_{comments}	1 if the p.submitter $\in$ i.replies, else 0
$M_{\text{top 2}}$	1 if the p.submitter $\in$ i.top-2, else 0
$M_{\text{engagement}}$	$\frac{\lvert\{c \mid c \in \text{issue.replies, c.author} = \text{p.submitter}\}\rvert}{\lvert\text{issue.replies}\rvert}$
Lag	$\frac{\min(\{\text{abs}(\text{p.timestamp} - e.\text{timestamp}) \mid e \in \text{i.events}\})}{\text{developer-fingerprint(p.submitter)}}$
Lag-open	$\text{p.timestamp} - \text{issue.open.timestamp}$
Lag-close	$\text{issue.close.timestamp} - \text{p.timestamp}$
Lack of description	$\mathbf{1}$ **if the p has no description, else** $\mathbf{0}$
Size of branch	$\lvert\mathbf{p.commits}\rvert$
Number of files touched	$\lvert\{\mathbf{file.name} \mid \mathbf{file} \in \mathbf{p.diff}\}\rvert$
Report size	$\mathbf{len}(\mathbf{i})$
Participants	$\lvert\{\mathbf{c.author} \mid \mathbf{c} \in \mathbf{issue.replies}\rvert\}$
Reopens	$\lvert\{\mathbf{t} \mid \mathbf{t} \in \mathbf{i.transitions} \cdot \mathbf{t.to} = \mathbf{open}\}\rvert$
Existing Link	$\mathbf{1}$ **if** $\exists \mathbf{e} \in \mathbf{E} \cdot \boldsymbol{\pi}^{\mathbf{0}}(\mathbf{e}) = \mathbf{i}$**, else** $\mathbf{0}$

Textual Features: These features employ comparisons between issue and PR artefacts and files modified by changes in a PR. They assume that related issues and PRs will share common terms. We also remark that past offline work [36] relied on commit summarisation tools such as ChangeScribe [37], limiting applicability to programming languages supported by the summariser. We exploit a PR's title and description, enabling us to avoid reliance on code summarisation tools. We compute these features by converting artefacts to tf-idf vectors and computing the described similarity measures.

Social Features: These features are constructed by considering reporters, assignees, and discussion participants. We assume that the same developers that solve an issue are likely to discuss the issue with the reporter, or, under contribution guidelines that require having an

issue open to which a PR refers to, be both the reporter and the PR author. These features, as seen in Table 3.1, are almost all binary. $M_{\text{engagement}}$ is the only exception to this, it measures the proportion of comments that are made by the submitter of a PR. This captures the intuition that a more engaged discussion participant is more likely to contribute via a PR.

Temporal Features: These features capture the properties of issue state transitions. We assume transitions such as issue closures or reopening's are related to the activities of developers and thus may correlate with PR events. Starting from Le *et al.*'s [36] temporal features, we adapt them to the PR setting. Additionally, we model the behaviour of individual developers in terms of the expected time between their most recent interaction with an open issue and their submission of a PR to address that issue. We calculate the mean and variance of a developer's past behaviour, and normalise a given elapsed time in terms of standard deviations from that mean. This allows the model to use a notion of expected time until interaction as a feature that is scale normalised, to allow for individual variation in development time.

Structural Features: These features capture properties relating to the structure of either an issue or PR. They serve to capture signal that can aid classifying them as either linkable or unlinkable. For a PR, the first feature considered, presence of a description, is a check that the PR has non-trivial information provided. The next two represent a proxy for PR size in order to learn what constitutes an unfocused (too large) or trivial (too small) PR. For issues, we consider four features: The size of the report, the number of participants, how many times the issues was reopened and if it is already linked to a PR. We use issue size as a proxy for unfocused issues. We also assume that issues with higher engagement are more likely to be eventually linked to a PR. Issues that are reopened tend to have multiple links to PRs (unless PRs are later merged). Finally, repositories tend to prefer to merge issues and PRs with other issues, resp. PRs, in favour of creating multiple link scenarios, hence we consider the presence of a link to be a signal that additional links are unlikely.

3.4.2 Feature Selection

When selecting features, just using the top k features by importance is tempting. Doing so can, however, miss synergies, redundancies or antagonistic relationships between features. In our case, using the top k by importance indeed would miss redundancies in textual features. We show how we applied the principled approach, Recursive Feature Elimination [91], to enrich the features A-M considers.

To formally analyse the efficacy of matching on text artefacts and other features, we employ a small but representative internal development set of projects separate from our training and testing data. This internal dev set was constructed by bucketing the projects by size into four equal groups and uniformly at random picking a project from each bucket. We analyse the features listed in Table 3.1. To reduce the number of features, we first consider the linear correlation of the

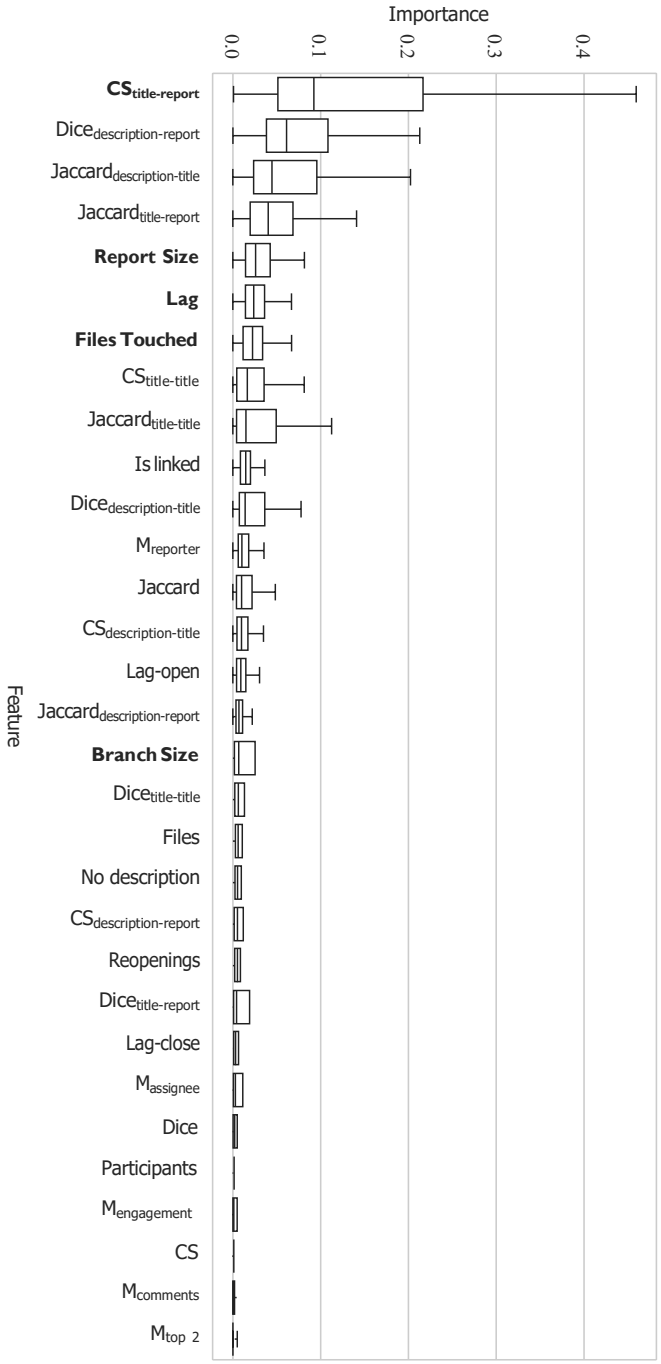


Figure 3.4: Feature Importance as computed using a Random Forests Classifier on the development set. We performed recursive feature elimination to select the final feature set while considering synergistic and antagonistic interactions between the features. The final set of features can be seen in bold. We do not show outlier values for presentation reasons.

described features on our development set and group them if they have a Pearson R^2 above 0.6, *i.e.* we want to consider a single feature from within a cluster of features where they are good linear predictors of each other. Within these groups, we only employ the features with the highest importance according to a Random Forest model.

On this pre-pruned feature set, we then perform recursive feature elimination to further reduce the considered set. The core idea of recursive feature elimination is to ablate features one-by-one as long as the observed performance does not significantly degrade. Figure 3.4 shows feature importance over the full feature set presented in Table 3.1, It suggests that Jaccard should be included in the final set of features. However, recursive feature elimination determines that removing it does not impact model performance since Jaccard and $\text{CS}_{\text{title-title}}$ are linearly correlated, although below our previous R^2 threshold, and the latter has a higher importance. Thus, we reduce the number of features we use for evaluation to $\text{CS}_{\text{title-report}}$, Lag, Report Size, Number of files touched, and Branch Size (bolded in Figure 3.4). A consequence of modelling negative links explicitly is increased importance of features that depend on the size of the artefacts; this is unsurprising as they represent good proxies for determining unlinkable pull-requests and issues.

In all scenarios, we estimate importance using the standard Random Forest implementation provided by SciPy [94]. When training and validating our system, we only consider pull-requests and issues whose last update is within a certain window of relevancy, which we set to seven days as per the recommendation in Wu *et al.* [33], to limit the number of candidate links considered both for feature selection as well as model training. For Random Forest hyperparameters we use Decision Trees and 100 estimators, in line with the recommendation from Oshiro *et al.* [88], leaving other settings to SciPy [94] defaults. We evaluate the features over all (p, i) pairs for each repository; Figure 3.4 shows the results.

3.5 Experimental Set-up for Evaluating Aide-mémoire

Evaluating A-M required substantial logistical effort, which we now describe. We first present our two corpora, a Java corpus, which enables comparison with RCLinker, the previous state of the art, and a multilingual one, on which A-M can spread its wings. We explain how we weakly label these corpora to obtain training data. We introduce performance measures for predicting lists and our adaptation of accuracy to account for our new 'no_pr' and 'no_issue' predictions. We envision Aide-mémoire as an assistant or an alternative to a manual search, so we devised a user study to compare it to GitHub's built-in search facilities; we present the study protocol here.

3.5.1 A Tale of Two Corpora: Java and Multilingual

We use two corpora in this work. Table 3.2 presents the detailed statistics for a subsample of our Java corpus, as well as aggregate statistics across all 47 repositories it contains. We use it to

Table 3.2: Summary Statistics for a uniformly sampled subset of the Java Generalisation Corpus, ordered by the number of existing links in the corpus. A range of projects sizes is included. In the majority of projects, most PRs are not linked to an issue; this can be seen in the last column, the median linking rate being only 0.46 (0.43 for the sample)

Repository	Links	PRs	Commits	Issues	Links/PR
pinterest/teletraan	11	403	774	41	0.03
mikepenz/MaterialDrawer	18	136	1982	1804	0.13
google/android-classyshark	21	92	597	63	0.23
roughike/BottomBar	44	123	789	687	0.36
facebook/fresco	65	241	1839	1567	0.27
googlei18n/libphonenumber	87	583	1476	1255	0.15
square/leakcanary	124	210	376	580	0.59
square/retrofit	275	771	1562	1623	0.36
ampproject/amphtml	3061	6123	6872	4170	0.5
Sample Total	3706	8682	16267	11790	**0.43**
Corpus Total	19785	43101	174456	67720	**0.46**

Table 3.3: Summary Statistics for a uniformly sampled subset of the Non-Java Generalisation Corpus, ordered by the number of existing links in the corpus. A range of projects sizes is included. In the majority of projects, most PRs are not linked to an issue; this can be seen in the last column, the median linking rate being only 0.49 (0.39 for the sample)

Repository	Links	PRs	Commits	Issues	Links/PR
SaschaWillems/Vulkan	18	117	1176	246	0.15
OptiKey/OptiKey	53	147	2170	186	0.36
aseprite/aseprite	53	119	5745	1423	0.45
coryhouse/react-slingshot	75	175	600	277	0.43
akveo/ng2-admin	183	558	1144	669	0.33
angular/zone.js	222	403	633	451	0.55
facebook/draft-js	273	457	679	859	0.6
kadirahq/react-storybook	300	607	4694	964	0.49
hakimel/reveal.js	302	617	2095	1335	0.49
citra-emu/citra	468	1771	4972	1071	0.26
Sample Total	1947	4971	23908	7481	**0.39**
Corpus Total	97637	200414	763002	323667	**0.49**

compare A-M with RCLinker, the previous state of the art. Table 3.3 presents detailed statistics for a subsample of our multilingual corpus, as well as its aggregate statistics. The multilingual corpus comprises 213 repositories and six programming languages. We use it to evaluate A-M in detail in its native setting. The SQL queries used to select projects are available online [89].

The repositories contained in our new corpora are sampled from the GitHub GHTorrent dataset [95], which at the time of our sample contained a total of 3704251 repositories. We exclude projects that have fewer than 100 lines of code across all files, ensuring that there is sufficient code in the repository such that per project vocabularies contain at least 100 terms. We sort by the number of 'watchers' as a proxy for popularity, and select the 50 most popular repositories, a figure limited by the time required to mine relevant data. The popularity bias reduces the inclusion of low quality GitHub projects and increases the number of projects with high issue and PR activity. We exclude projects using natural languages other than English, as

we use the Porter Stemmer built for the English language.

The languages for the multilingual corpus were selected first by uniformly sampling five languages from the most popular 15 as reported by GitHub [96]. Note that the website reported the top 15 languages and the restriction to five additional languages was necessary only to constrain the cost of mining the corpus. The selected languages were Scala, TypeScript, C++, C#, and JavaScript. We again exclude small projects and select the 50 most popular projects for each language. Omitting those we could not successfully crawl from GitHub due to rate limits, we were left with a total of 238 repositories. After additional filtering to remove repositories that have too few examples for the results to be meaningful, the final corpus size is 213.

3.5.2 Weak Labelling

Weak labelling enables the use of large corpora for supervised learning when manual annotation would be prohibitively expensive. While weakly labelled data can be noisy, one need only take care to use it in conjunction with machine learning techniques that can cope with noise. Mondrian Forests are one such technique, as we observe in Section 3.7.2.

As a weak labeller, we heuristically link PRs to issues when a PR-issue pair is explicitly linked in the GitHub metadata or when they both contain a SHA. We identify SHAs as r'[0–9a–f]\{5,40\}' or numerical (r '[\ n\r\s]\#[0–9]+') tokens that can be disambiguated to a unique artefact in the project. We apply this heuristic to PRs, issues, commit messages, and diffs for each project in our corpora. We assume that unlinked PR-issue pairs are negative examples, and linked pairs represent positive examples.

Because our corpora are not manually annotated, some negative examples are false negatives, links missed by developers, and some positive examples are false positives. To mitigate this threat, we manually assessed 30 positive links and 30 negative links sampled uniformly from our corpus. On an initial sample, we found significantly many false links to the issue/pull-request with the ID '#1'. We removed these links and, after resampling and re-assessment, found more than 80% of the filtered links recovered by the heuristic to be correct. This is sufficient for A-M, because A-M, by virtue of its use of Mondrian forests, is robust to noise, as our results show in Section 3.7.4. We have, however, excluded 72 projects from our corpus after heuristic linking, because they have fewer than 25 recorded links, which would lead to a performance metric computed over at most 5 queries due to our 80%/20% train/test split.

These two heuristically-linked corpora are much larger than those previously used; we have made them available to other researchers [97].

3.5.3 Measuring Performance

Suggesting traceability links to a developer naturally requires producing suggestion lists. Thus, to measure A-M's performance, we need performance metrics built for lists. We first present Mean Average Precision, a standard metric from Information Retrieval for list prediction tasks. We

designed A-M to choose its suggestions with care, to avoid burdening developers with low quality suggestions. To measure how well we succeeded, we present hit rate (HR), which counts how often A-M makes a suggestion when it should and how often A-M remains silent when it should. Finally, we explain how we obtain the more usual Precision, Recall and F1-score measures used to assess classifiers.

A-M presents a ranked list of suggestions to a developer containing PRs to be linked (when closing an issue) or else issues to be linked (when submitting a PR). To quantify performance, we use *Mean Average Precision* and *List Hit Rate*, metrics derived from the suggestion lists output by A-M. We describe a single request for a suggestion list as a query q, where Q is the set of all such queries to A-M during the replay of repository events; the list produced in response to an individual query q is denoted r_q. A-M produces suggestion lists of variable length, and we therefore denote the length of the list returned $|r_q|$. For a given query $q \in Q$, $\text{rel}_q(k)$ is an indicator function that returns 1 if the k^{th} item in the response list r_q is relevant and 0 otherwise, including for k values *s.t.* $k > |r_q|$. $P(k)$ is the precision of the first k items returned for a given query.

In order to consider both the precision of individual suggestions, and, crucially, their positions within the suggestion list, we use Average Precision (AP).

$$AP(q) = \frac{\sum_{k=1}^{k=\infty} P_q(k)\text{rel}_q(k)}{|\{k \mid k \in \mathbb{N} \wedge \text{rel}_q(k) = 1\}|}. \tag{3.6}$$

Average Precision can be interpreted as the average of all precision values at each recall value a query q may take as you increase the output list. To measure the quality of suggestions across all queries in Q, we use Mean Average Precision [98], which measures the quality of a set of ranked lists of suggestions.

Definition 3.5.1. Mean Average Precision (MAP)

$$\frac{1}{|Q|} \sum_{q \in Q} AP(q). \tag{3.7}$$

Thus, we treat each PR submission or Issue closure event as an individual query q and measure the quality of our suggestions per project. This considers the order in which suggestions are offered, and amortizes outliers such as PRs that either have a high number of links or are incomplete. In other words, MAP measures the preponderance of true links in the responses offered by A-M during the replay of repository events in the event suffix (Section 3.6.2).

Our task demands a few high-quality suggestions or no suggestions at all, rather than many low-quality suggestions because suggesting irrelevant links wastes a developer's time. For this reason, we built A-M to truncate its suggestions at 'no_issue' or 'no_pr' in its suggestion list. MAP cannot be applied to scenarios where there are no relevant documents to be found. Indeed, it does not apply to empty lists, because $AP(q)$ is undefined for $|r_q| = 0$. A-M's truncation tactic

makes this case occurs fairly often. We denote the set of queries where MAP is not applicable with:

$$Q' = \{q \mid \forall k \in \mathbb{N} \cdot rel_q(k) = 0\}, \tag{3.8}$$

that is, the set of all queries for which there are no relevant artefacts to predict. Then, we define hit rate:

Definition 3.5.2. List Hit Rate

$$\frac{1}{|Q|}\Big(\sum_{q \in Q - Q'} AP(q) > 0 + \sum_{q \in Q'} |r_q| = 0\Big). \tag{3.9}$$

We use hit rate (HR) as a proxy for impact on developer workflow. HR measures not only when A-M correctly predicts a link, but also when it correctly does not. HR penalizes A-M for predicting links when there are none and $AP(q)$ is undefined. HR lifts the notion of accuracy from a binary classification scenario to lists where positive examples require at least one match to be a true positive and negative examples require no suggestions to be a true negative. Hence, a false positive is suggesting a non-empty list when no links exist and a false negative is suggesting an empty list when at least one link exists.

To obtain Precision, Recall and F1 scores for A-M, taking into account the list structure of A-M's output, we perform the following: we truncate all lists to length k or the rank of the no_{issue/pr} entity, whichever occurs first; we concatenate all predictions together; we compute precision, recall and F1-score in the usual way by considering the true positives (tp), false positives (fp), and false negatives (fn) in this setting:

$$p = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fp}}, \tag{3.10}$$

$$r = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fn}}, \tag{3.11}$$

$$\mathsf{f1} = \frac{2pr}{p + r}. \tag{3.12}$$

3.5.4 User Study Protocol

To quantify A-M's potential to help developers link PRs and issues on GitHub, we devised a small scale user study. Here, we describe our user protocol.

Currently, a developer using GitHub must resort to GitHub's generic search facilities to recover missing links, hence we compare A-M against using this in-built search functionality. First, we select a GitHub project from our development corpus and sample uniformly at random to select three positive and two negative examples, reflecting the accuracy of the tool over the project (a hit rate of 0.66). We then artificially omit these links. Participants had to recover these omitted links either: (1) using A-M and falling back to GitHub search should that fail if they are in the experimental group; or (2) using the GitHub search facilities if they are in the control group. We recorded the time taken per task as well as a number of search queries performed.

We recruited participants by convenience sampling from within our Computer Science department, and a total of seven participants chose to participate. Since we could not assume proficiency with the GitHub environment, in particular its search features, both groups were provided with a short tutorial including instruction on how to perform searches on specific fields within a Pull Request or Issue before being asked to perform the linking task. After the task, we asked participants to complete a questionnaire regarding the perceived difficulty of the task, how they determined the relevancy of an issue, and solicited free-form feedback on the tool itself. The results for time taken are summarised in Figure 3.10 with a further discussion presented in Section 3.7.6.

3.5.5 The Longitudinal Evaluation of Aide-mémoire

A-M is the first commit-issue (where we extract commit-issue links from A-M's PR-issue links) predictor that aims to suggest links to a working developer *as they submit commits or issues*; previous work bulk proposes commit-issue links in project histories. It solves an inherently online problem, which restricts any online predictor's training data. This rules out cross-fold validation, which could allow a predictor to incorrectly train on links from the future relative to the time of a given suggestion.

So, to train A-M, we use longitudinal evaluation [99]. We first flatten each repository into a chronological sequence of events; we consider 'PR index', the position of a PR in the chronological sequence of all project PRs, as a proxy for elapsed development time. We then split each project into a prefix and suffix to obtain a $80\%/20\%$ training/test split over the PR indices. We replay the suffix of this event stream, simulating the creation of artefacts such as issues and PRs, and request link predictions from each trained classifier at the time of PR submission or issue closure. In the case of A-M, to simulate developer feedback, if the classifier has a correct prediction in top five, it uses those predictions to update itself. The tf-idf model is continually updated. New tokens are treated as a special unknown token. We look at cross-project performance to assess generalisation, rather than performing multiple longitudinal splits. This method underlies the results we report in Section 3.6.2 and Section 3.7.2.

3.6 Reproducing RCLinker

A-M tackles a new problem: for pull requests (PR), it suggests PR to issue links at submission. This problem is inherently online. To baseline A-M, we turn to the related offline, commit-issue linking task. Tools solving this problem aim to repair histories for use in other productivity enhancing tools that consume histories. Here, we considered two solutions: RCLinker [36] and Rath *et al.*'s link classifier [45]. The two papers use different data sets and different feature spaces. Rath *et al.*'s have not made their classifier publicly available. Another blocker for us is that Rath *et al.*'s link classifier relies on features that would be in the future in our online setting.

Thus, we turned to RCLinker. To effect this comparison (Section 3.6.2), we needed to reproduce and adapt RCLinker. We use the reproduction as an internal classifier in our adaption. Because of its importance, we sanity check our recreation of RCLinker against published results, and then adapt it to our problem in two stages: lifting it from commits to PRs and replacing its commit summariser. Section 3.6.2 details the comparison between A-M and our recreation of RCLinker, demonstrating that an approach tailored to the PR-issue linking problem fairs better on our new task. A-M outperforms our recreation of RCLinker in terms of precision by 0.62, achieving 0.76 at similar recall (0.37 vs 0.36); this is the expected result since A-M was designed for this task while RCRep's core classifier was not. RCRep's similar recall is a testament to RCLinker's solid design.

3.6.1 Constructing RCRep

To adapt RCLinker to A-M's PR-issue linking task, we first had to reproduce it on its original commit-issue linking task. We then modified it to solve the PR-issue linking task with a pair of adaptors: an input that translates PR-issue linking into a commit-issue linking problem and an output adapter that maps RCLinker's solution back to PR-issue links. As a faithful reproduction would discard potentially useful information such as PR descriptions, we also explore using them as a substitute for commit summaries.

RCLinker defines a feature space over commit–issue links. We discussed these features in Section 3.4; Le *et al.* apply these features to commits, not PRs. As usual, RCLinker first extracts these features from an input project. As commit messages are often short (as best practice encourages [100]), Le *et al.* employ ChangeScribe [37] to automatically summarise commits; RCLinker extracts its textual features from these summaries. It is its reliance on ChangeScribe that restricts RCLinker to Java. They train a Random Forest classifier on the derived feature vectors. RCLinker works over the space of all possible links ($I \times C$); to avoid overwhelming the classifier with negative links, they introduce a novel undersampling algorithm. To form negative links, this algorithm replaces issues or commits in a true link with up to five of their nearest neighbours in the feature space. This undersampling does not scale well to the online case — the time taken to undersample grows linearly with both the total number of links *and* the number of artefacts associated with each link, *i.e.* $O(l(i+c))$. We empirically observed Mondrian Forests to be resilient to the class imbalance problem this undersampling mitigates.

We implemented RCLinker's feature set and classifier; this serves as the internal classifier for our adapted variants. It solves the commit-issue linking problem[1]. Our core classifier reproduction depends on ChangeScribe [37] for commit summaries. Our implementation uses a modified version of the ChangeScribe Eclipse plugin that can be run headlessly. As our RCLinker classifier is the lynchpin of our adaptations, we validate it on Bachmann *et al.*'s original commit-issue problem [12] on two corpora: the Apache Commons [12] on which Le *et al.* evaluated RCLinker

[1] Reproduction is often thankless and hard. We thank Le *et al.* for being extremely helpful and responsive to our requests during our reproduction of their work.

Table 3.4: Performance values on the Java Corpus for RCRepCS when solving the commit-issue prediction task. This evaluation confirms that our reproduction of RCLinker is effective; indeed, we find it generalises well beyond the corpus used in the original paper.

Repository	F1-Measure	Precision	Recall
Bilibili/ijkplayer	0.39	0.90	0.27
facebook/fresco	0.59	0.88	0.47
googlei18n/libphonenumber	0.56	0.80	0.48
google/android-classyshark	0.76	0.97	0.65
google/gson	0.45	0.75	0.34
iluwatar/java-design-patterns	0.24	0.46	0.18
JakeWharton/butterknife	0.72	0.72	0.75
mikepenz/MaterialDrawer	0.75	0.76	0.75
nostra13/Android-Universal-Image-Loader	0.79	0.85	0.74
ReactiveX/RxAndroid	0.45	0.75	0.35
roughike/BottomBar	0.55	0.80	0.44
square/leakcanary	0.43	0.53	0.38
square/picasso	0.29	0.35	0.27
wequick/Small	0.49	0.50	0.51
Median	**0.52**	**0.76**	**0.46**

Table 3.5: Performance values on the original Apache Corpus for RCRepCS when solving the commit-issue prediction task. We bias our reproduction towards higher precision to account for the shift to the perspective use-case and we observe a higher performance of our reproduction relative to the results reported in Le *et al.* [36] for F1-Score and Precision at the cost of Recall.

Repository	F1-Score		Precision		Recall	
	RCRepCS	RCLinker	RCRepCS	RCLinker	RCRepCS	RCLinker
CLI	0.76	0.61	0.70	0.45	0.88	0.91
Collections	0.82	0.59	0.72	0.43	0.95	0.92
CSV	0.86	0.54	0.87	0.39	0.85	0.88
IO	0.66	0.70	0.96	0.59	0.50	0.87
Lang	0.82	0.72	0.82	0.58	0.83	0.94
Math	0.55	0.70	0.84	0.61	0.42	0.83
Overall	0.74	0.64	0.82	0.51	0.74	0.89

and our Java corpus from Section 3.5.1. Table 3.5 shows the results on the Apache Commons corpus. Our reproduction obtain similar results with only a slight penalty to Recall. We note that the Random Forest implementations in SciPy [94] and in Weka [101] may use different defaults. Table 3.4 shows the result on our Java corpus. It demonstrates strong generalisation beyond the Apache Commons corpus used in the original RCLinker paper and first introduced by Bachmann *et al.* [12].

We adapt our RCLinker reproduction to the PR-issue linking task in two stages. In the first stage, we wrap its classifier in two translators. The input translator converts PR or issue events into queries over commit-issue links. The output translator converts the commit-issue links predicted by the classifier into the related PR-issue links: it links a PR to an issue if *any* of the commits from that PR are linked to the issue. We name this adaptation “RCRepCS” as it uses ChangeScribe.

In the second stage, we further adapt “RCRepCS” to our setting and replace ChangeScribe

with PR descriptions. This allows us to assess the impact of commit summarisation on the task. We name this variant "RCRep". While commit messages are usually short — developers are encouraged to keep them within 72 characters — PR descriptions tend to be longer. We create a new variant of our reproduction, RCRep, which further replaces ChangeScribe summaries with PR descriptions. This configuration assess the quality of ChangeScribe as a source of textual information relative to developer Pull-Request discussions. Table 3.6 shows the impact of this change; in short, PR summaries outperform ChangeScribe summaries. In the remainder of this section, we use RCRep to refer to both RCRepCS and RCRep.

As Section 3.5.5 details, we longitudinally compare RCRep and A-M. For A-M, we train it on the first 80% of events in a repository and evaluate it on the last 20%. We request a prediction whenever a PR is submitted or issue is closed. For RCRep, we create a repository view that represents the first 80% of the repository. We then provide it the true links and undersample RCRep links using RCLinker's method described above. We then evaluate RCRep by asking it to predict the links that exist within the last 20% of events from a repository.

Aide-mémoire is an online assistive tool that outputs a variable-length ranked list of suggested links. RCLinker outputs unranked commit-issue links. To enable comparison, RCRep's output adaptor collapses all of its core classifier's suggestions for commits in the same PR into a single list. RCRep orders these suggestions by the output probability from its Random Forests classifier; it breaks ties by the distance between PR and issue identifiers, which GitHub assigns jointly and in increasing order to issues and PRs as they are created. Because we compare both variants to A-M, and RCRepCS is Java-only, Section 3.6.2 considers only Java projects and shows that A-M outperforms both variants.

A-M and RCRep use random forest classifiers, which require hyper-parameter settings: we set the class estimates to 10 for RCRep, in line with the original paper, and 128 for A-M, as discussed in Section 3.3.1; the separation criteria was entropy for RCRep, while Mondrian Trees employ a budget to decide when they should refine the partition of the space [87]. Additionally, we use the default random forest probability cut-off of 0.5. Dynamic cut-off due to prediction of 'no_pr' or 'no_issue' occurs on top of the default cut-off when applicable. In tf-idf, the minimum term count was two and the term cut-off set at 95%. For RCRep undersampling, we use the five nearest neighbours, setting this hyperparameter following the RCLinker authors.

3.6.2 Benchmarking Aide-mémoire Performance with RCRep

We now quantify Aide-mémoire performance relative to RCRep's. Unsurprisingly, Aide-mémoire outperforms RCRep on the PR-issue linking task. We further improve RCRep's performance by adapting it to directly to use PR descriptions in place of ChangeScribe's commit summaries.

Table 3.6 shows detailed results. Overall, Aide-mémoire clearly outperforms RCRepCS and RCRep across the Java corpus. In terms of precision, RCRepCS achieves its best result on

Table 3.6: Mean performance values of predicting issue-PR links across a sample of the Java Corpus for RCRepCS, RCRep, and A-M. The approach taken by RCRepCS to solve the commit-issue link prediction problem does not generalise to issue-PR links, even when provided with PR descriptions as summaries, whereas A-M performs well on most projects. Extreme (0.0 or 1.0) results can be observed on small projects where there are few queries in the suffix. The projects with an (*) have had their names shortened for presentation purposes.

Repository	**Mean Precision**		
	RCRepCS	RCRep	A-M
1. Bilibili/ijkplayer	0.00	0.00	**0.52**
2. daimajia/AndroidSwipeLayout	0.00	0.00	**0.99**
3. facebook/fresco	0.01	0.01	**0.92**
4. facebook/redex	0.06	0.16	**0.76**
5. googlei18n/libphonenumber	0.01	0.00	**0.83**
6. google/android-classyshark	0.00	0.33	**0.99**
7. google/flexbox-layout	0.48	0.29	**0.79**
8. google/gson	0.06	0.07	**0.92**
9. greenrobot/EventBus	0.00	0.13	**0.65**
10. iluwatar/java-design-patterns	0.06	0.17	**0.89**
11. JakeWharton/butterknife	0.00	0.08	**0.73**
12. kaushikgopal/RxJava-Android-Samples	0.41	0.29	**0.71**
13. liaohuqiu/android-Ultra-Pull-To-Refresh	0.00	0.00	**0.99**
14. mikepenz/MaterialDrawer	0.04	0.06	**0.93**
15. Netflix/Hystrix	0.04	0.28	**0.71**
16. nostra13/Android-Universal-Image-Loader	0.00	0.00	0.00
17. ReactiveX/RxAndroid	0.03	0.20	**0.86**
18. roughike/BottomBar	0.01	0.07	**0.86**
19. square/leakcanary	0.09	0.19	**0.88**
20. square/picasso	0.00	0.04	**0.78**
21. twitter/distributedlog	0.95	0.64	**1.00**
22. wequick/Small	**0.01**	**0.01**	**0.01**
23. zxing/zxing	0.15	0.11	**0.72**
Overall	0.10	0.14	**0.76**

twitter / distributedlog , which represents a large and well linked project; in terms of F1-Score, it achieves its best result on the similarly well linked google/flexbox–layout. We note that using PRs only as a source of natural language description improves performance across most projects. We conclude that attempting to solve the issue-PR prediction problem as an instance of the commit-issue task is ineffective even when a change summariser is provided. This validates our decision to solve the issue-PR link prediction separately and make use of PR level metadata. In particular, the high false positive rate and the lack of 'no_pr' or 'no_issue' entities impacts the precision of RCRep. Also of note is that both A-M and RCRep fail on projects that have very little linking at the PR level — there is insufficient data to train an accurate classifier; one such project is Android–Universal–Image–Loader, which is just barely above our training example threshold employed as a filter on our corpus.

While RCRep performs poorly on issue-PR link prediction, we emphasise that RCLinker was designed to predict commit-issue links; we include this comparison to motivate the separate handling of issue-PR links. For completeness, we also evaluate the performance of RCRepCS

*

Table 3.6: Mean performance values of predicting issue-PR links across a sample of the Java Corpus for RCRepCS, RCRep, and A-M (cont.).

Repository	Mean Recall			Mean F1		
	RCRepCS	RCRep	A-M	RCRepCS	RCRep	A-M
1.	0.00	0.00	**0.06**	0.00	0.00	**0.11**
2.	0.00	0.00	**0.10**	0.00	0.00	**0.18**
3.	**0.51**	**0.51**	0.37	0.03	0.02	**0.53**
4.	0.47	0.46	**1.00**	0.10	0.20	**0.86**
5.	**0.50**	0.20	0.26	0.03	0.00	**0.39**
6.	0.00	**0.72**	0.54	0.00	0.45	**0.70**
7.	**0.79**	0.75	0.46	**0.60**	0.42	0.58
8.	**0.48**	0.36	0.35	0.10	0.11	**0.51**
9.	0.00	0.20	**0.45**	0.00	0.15	**0.53**
10.	0.19	**0.22**	0.06	0.09	**0.19**	0.12
11.	0.30	**0.64**	0.30	0.00	0.14	**0.42**
12.	0.50	0.43	**0.59**	0.39	0.34	**0.64**
13.	0.00	0.00	**0.25**	0.00	0.00	**0.40**
14.	0.53	0.58	**0.79**	0.07	0.11	**0.86**
15.	**0.43**	0.32	0.29	0.08	0.30	**0.41**
16.	0.00	0.00	0.00	0.00	0.00	0.00
17.	0.15	**0.80**	0.50	0.05	0.31	**0.63**
18.	**0.28**	0.04	0.22	0.02	0.05	**0.35**
19.	0.24	0.30	**0.35**	0.13	0.23	**0.50**
20.	0.27	**0.55**	0.35	0.01	0.08	**0.49**
21.	0.10	0.10	**1.00**	0.18	0.17	**1.00**
22.	**0.17**	0.16	0.01	0.01	**0.03**	0.01
23.	**0.99**	0.96	0.20	0.25	0.20	**0.31**
Overall	0.30	0.36	**0.37**	0.09	0.15	**0.46**

on the original commit-issue link prediction problem; we provide these results in Table 3.4. Additionally, RCLinker is applicable to scenarios where there is no PR information at all, and thus RCLinker and A-M are *complementary*: historical data could be repaired using RCLinker, followed by adopting A-M to improve future linking. In general, offline tools such as RCLinker are hindered by reliance on commit-level information, which makes them less applicable to modern projects that follow a PR-centred development process.

3.7 Evaluating Aide-mémoire on the Online PR-Issue Linking Problem

First, we show that the state of practice in PR linking remains dismal and that tools like A-M are needed. We then evaluate A-M's performance on the PR-issue linking task over a large multilingual corpus. We observe that A-M is indeed language-agnostic and scales to larger projects, benefits from the use of an online classifier and that it shows tolerance to noisy training data. We assess the usefulness of A-M as a developer aid and show that developers could benefit from such a tool.

3.7.1 The Dismal State of Issue-PR Linking on GitHub

To determine the state of linking practice in the wild on GitHub, we first queried GHTorrent [2] for those projects that provide a `CONTRIBUTING.MD` file or similar within the root of the project (which is a GitHub convention for the location of the file). We found only around 4.7% of all the projects available via GHTorrent include such a file (167109 out of 3537142 at the time of query).

Restricting ourselves to these projects, we then considered two ways to obtain subsamples for manual investigation: biasing the sample by popularity and performing a uniform sample over this restricted set of projects. In both scenarios we were looking for explicit statements that links have to be recorded. Example statements are 'It is best practice to have your commit message have a summary line that includes the ticket number [...].', 'This is also the place to reference the GitHub issue that the commit closes.', 'All Pull Requests, [...], need to be attached to a issue on GitHub.'. We also included those projects that referenced an external resource that described a good commit message and that recommended linking to affected ticket numbers.

We sampled 200 projects for each method and were able to obtain the file for 128 projects from the uniform sample, while we obtained 155 for the popularity based sample. Of note here is that we attempted to obtain the most recent version of `CONTRIBUTING.MD` from each project's GitHub page rather than the GHTorrent blob. Our results are as follows: one third of the randomly sampled projects made explicit reference to linking practice (43 out of 128), versus around 43.5% (68 out of 155) of the popularity biased sample. This suggests that the majority of projects on GitHub do not require that the project's collective memory in terms of code-issue traceability links be maintained by Pull Request submitters. It is worth mentioning that we have not considered if the community enforces such a requirement even when it is not codified, or, conversely, if when the practice is codified whether it in enforced. This may bias our presented results. Nevertheless, the difference between the uniform and the popularity biased sample suggests that more popular projects do tend to require such linking. A further observation is that the popular projects that are maintained by a corporate entity — Google, Facebook, and Microsoft in the sample — tend to have a company wide policy regarding contributions from the community that require that issues and Pull Requests be linked and that a Pull Request references an already open issue in order for the submission to be accepted.

3.7.2 The Quality of Aide-mémoire's Suggestions

We evaluate A-M on a multilingual corpus containing 213 projects written in six programming languages. Section 3.5.5 presents the longitudinal evaluation protocol. The corpus contains a variety of project sizes, so we are able to evaluate both the generality and scalability of A-M. Recall that A-M provides a list of suggested issues to be linked to a PR at submission time, and a list of suggested PRs to be linked when an issue is closed. When considering the performance of our system, we sought to answer the following questions:

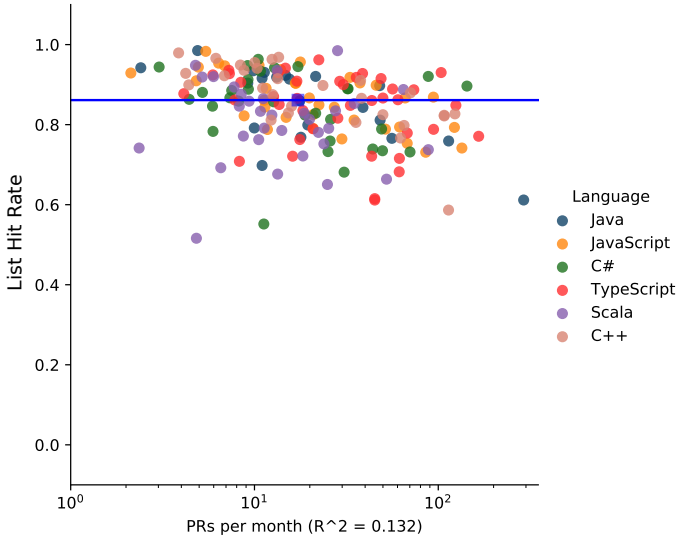


Figure 3.5: Performance of A-M quantified by the percentage of queries where A-M is correctly silent when there is no suggestion, or we report at least one correct link when there is a suggestion to be made for list length $k = 5$, median result presented as a blue square. We see that system performance does not degrade severely with the increase of PRs submitted per month (and by proxy) project size; indeed we find no statistically significant trend associated with the number of PRs per month.

1. What proportion of our suggestions contains at least one true link in a k-length list ($k = 1, 3, 5$)?

2. What is the mean average precision (MAP) of our suggestions?

3. Does our system suggest links that were later caught by PR reviewers?

A-M uses 'no_pr' and 'no_issue' entities to truncate its suggestions. If these special entities appear in the k suggestions, we truncate the suggestion list at that point to avoid suggesting unlikely links. Through these entities, A-M correctly learns to be silent when appropriate. It offers a correct suggestion or remains appropriately silent in 86% of cases for $k = 1$; we will henceforth refer to this desirable behaviour as *List Hit Rate*, as per Definition 3.5.2) . If we consider only predictions of actual links (rather than no prediction), A-M suggests a correct link in 94% of such cases for $k = 1$ (same for $k = 3$ and $k = 5$). Figure 3.5 shows a more detailed view for $k = 5$.

Conclusion (1)

Aide-mémoire suggests the correct true link in a k-length list for $k = 1, 3, 5$ in 94% of the cases.

We now consider the full suggestions lists; we use Mean Average Precision (MAP) to measure their quality. Table 3.7 shows that A-M consistently achieves high MAP; Indeed only seven projects fall below 0.6 MAP, on three of which A-M fails to learn any model. We observe

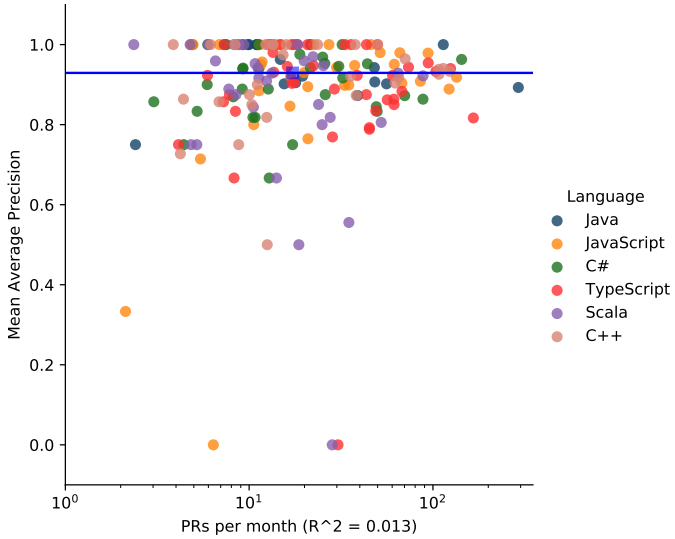


Figure 3.6: Performance of A-M reported as Mean Average Precision as a function of the number of PRs per month for a project, median result as a blue square. We use MAP as a measure of the quality of our suggestions. There is no statistically significant trend with scale.

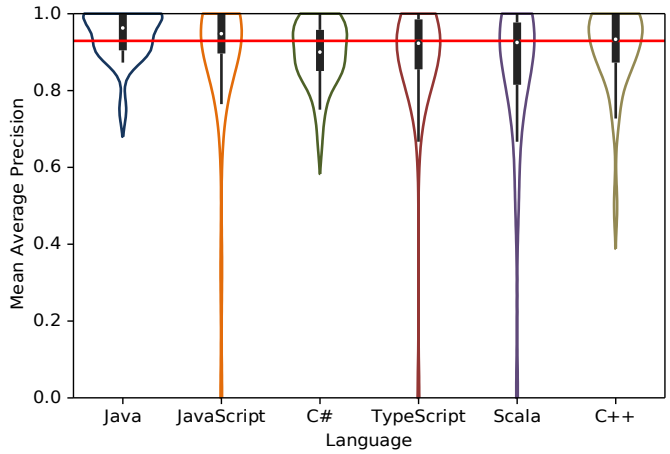


Figure 3.7: Performance of A-M reported as Mean Average Precision across languages. The outlier results present in some languages represent projects where we were unable to learn a prediction model and were not filtered by our number of links requirement. There is no statistically significant trend with language (as checked using a two-sample T-test).

this failure mode when there are insufficient true links in the training data. However, as results for HR are consistently above 0.5, we have succeeded in making A-M learn to be appropriately silent when it is unsure of its predictions. Overall, A-M achieves a median MAP of 0.93 (mean 0.89) and hence produces highly precise suggestions.

Conclusion (2)

Aide-mémoire achieves a median MAP of 0.93 (mean 0.89) over our multilingual corpus.

Table 3.7: Performance of A-M using a Mondrian Forest Classifier across a uniform sample of projects from the second corpus, ordered by PRs per month, after filtering projects that have less than 25 links in total.

Repository	Language	PR/m	MAP	HR	P	R
rtyley/bfg-repo-cleaner	Scala	2.36	**1.00**	0.74	**1.00**	0.12
zealdocs/zeal	C++	4.22	0.73	0.93	0.73	0.26
ecomfe/echarts	JavaScript	5.43	0.71	0.98	0.71	0.08
beto-rodriguez/Live-Charts	C#	8.19	0.87	0.86	0.87	0.47
mobile-shell/mosh	C++	12.31	**1.00**	0.81	**1.00**	0.28
sksamuel/elastic4s	Scala	13.28	0.93	0.94	0.93	0.48
square/picasso	Java	14.76	0.96	0.92	0.96	0.35
square/leakcanary	Java	17.64	**1.00**	0.86	**1.00**	0.33
Microsoft/code-push	TypeScript	18.34	**1.00**	0.84	**1.00**	0.25
kriasoft/react-starter-kit	JavaScript	20.84	0.89	0.82	0.89	0.20
AutoMapper/AutoMapper	C#	21.52	0.94	0.83	0.94	0.47
.../electron-builder	TypeScript	22.39	0.95	0.96	0.95	0.59
NLog/NLog	C#	25.79	0.95	0.76	0.95	0.41
witheve/Eve	TypeScript	29.09	0.89	0.91	0.89	0.44
Microsoft/vscode-react-native	TypeScript	33.26	**1.00**	0.85	**1.00**	0.43
Dogfalo/materialize	JavaScript	37.50	0.95	0.90	0.95	0.36
scala-js/scala-js	Scala	40.63	0.89	0.45	0.89	0.15
spring-projects/spring-boot	Java	48.00	0.94	0.90	0.94	0.23
citra-emu/citra	C++	62.32	0.90	0.77	0.90	0.22
facebook/osquery	C++	65.15	0.92	0.80	0.92	0.43
angular/material2	TypeScript	165.89	0.82	0.77	0.82	0.22
Sample (mean)		31.85	0.92	0.84	0.92	0.32
Sample (median)		21.52	0.94	0.85	0.94	0.33
Overall (mean)		32.13	0.89	0.84	0.89	0.30
Overall (median)		17.64	0.93	0.86	0.93	0.29

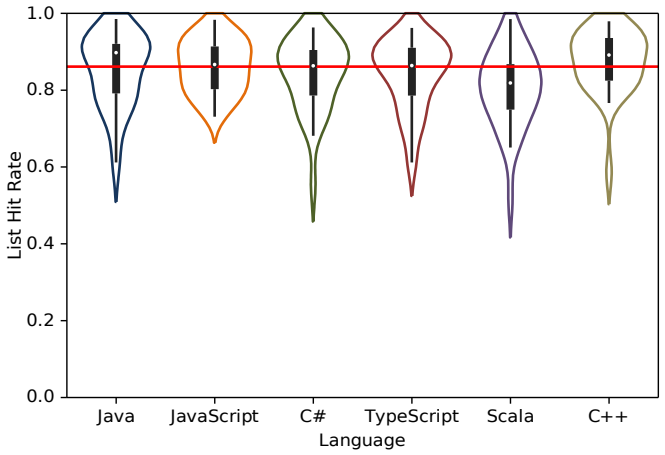
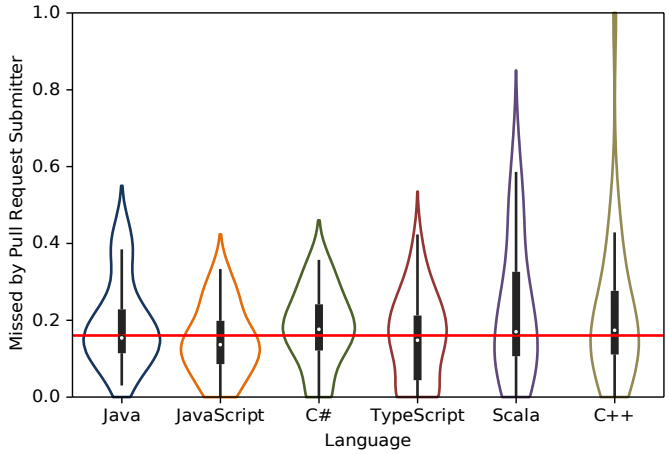
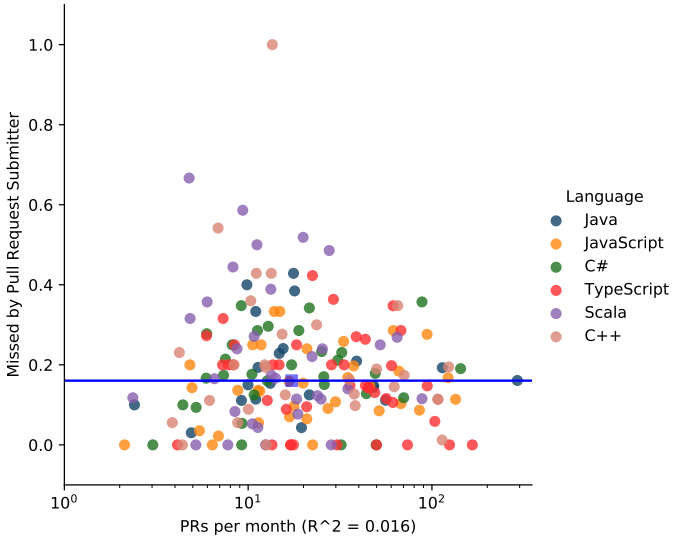


Figure 3.8: Performance of A-M quantified by the percentage of queries where we are correctly silent when there is no suggestion, or we report at least one correct link when there is a suggestion to be made for list length $k = 5$. We can see that the project language has negligible impact on performance and the median stays consistently close to 0.86 (the red line); there is no statistically significant difference between languages, except for Scala where we find that there is an effect size of $T = -3.38$ with $p = 0.001$ using a two-sample T-test.



(a) Percentage of links that were originally missed by the PR submitter and predicted by Aide-mémoire for the programming languages considered.



(b) Percentage of links that were originally missed by the PR submitter and predicted by Aide-mémoire as a function of the project's number of PRs per month, median result shown as a blue square.

Figure 3.9: Percentage of links missed by the PR submitter and predicted by A-M. Here we can see that our system has the potential to save development time during the PR process by suggesting links originally missed by the PR submitter and later discovered by a reviewer. There is no statistically significant deviation among languages.

A-M offers high-quality suggestions, with the correct link frequently appearing amongst the first two suggestions. A-M has low recall, managing to recover only 30% of the links removed in the validation suffixes, and requires some initial data to bootstrap the model. Thus, it cannot be deployed at the start of a new project, rather it must be adopted after the project has completed at least one development cycle. A-M successfully learns to predict links that were missed by the original submitter of PRs, *i.e.* those that were suggested by Pull Request reviewers during the code review process: 28% of the recovered links were predicted using only the PR description and its commits before any discussion took place, *i.e.* 18% out of all originally missed links, and as such the tool can help reduce the burden on change reviewers by offering link suggestions to the submitter. A more detailed breakdown of such cases can be seen in Figure 3.9.

Conclusion (3)

Aide-mémoire found 18% of the links that were initially missed by PR reviewers, suggesting that it can aid the PR review process.

3.7.3 Generalising across Languages and Project Sizes

We hope that A-M will change the state of practice in PR–issue linking. To realise this ambition, it must scale to large projects and language-agnostic, so that large projects and projects using different languages can easily adopt it. We show here that A-M does both.

Figures 3.5, 3.6, 3.8, and 3.7 summarise its performance while Table 3.7 shows a uniform sample of the results. The MAP is consistently near the median result of 0.95 across languages and there is no statistically significant deviation according to a two-sample t-test. The results for HR show a similar story: no deviation across languages from our general result of 0.86 with the exception of Scala, which has a small ($T = -3.38$) but statistically significant deviation ($p = 0.001$). For results along the project scale dimension, we can see that there is no statistically significant trend with all Pearson R^2 values below 0.1, hence we anticipate no degradation of performance for large projects.

Conclusion (4)

Aide-mémoire scales to larger project sizes with no impact on performance (Pearson's $R^2 < 0.05$ for MAP and Pearson's $R^2 = 0.13$ for HR). A-M also performs consistently across languages.

3.7.4 Resistance to Noise

A-M is online and learns from humans as they accept or reject its suggestions. To err is human, so A-M must contend with, and remain robust in the face of, humans giving it incorrect links. Here, we intentionally mislabel training data and observe its noise tolerance.

As weak labelling is inherently noisy, we explore A-M's tolerance to noise by injecting increasing levels of noise and observing the performance profile in terms of P-R Curves. In order to observe and quantify the noise tolerance of A-M, we augment the training data by randomly flipping a proportion of the examples, i.e. adding a false link as true, or marking a true link as false. We control this proportion and explore from no noise (0%) up to 95% noise. We run this experiment over the same internal development corpus as our Feature Selection detailed in Section 3.4. We perform 10-fold cross validation at each noise ratio, validating performance using the unaltered ground truth. Due to performing cross-fold validation, we additionally take care to elide links and references that may leak information regarding a held-out fold from the training folds. Looking in Precision-Recall space, we observe four main performance regimes with a slight improvement at low-noise (5-10%) for both Precision and Recall followed by a step-wise decline in Precision and a linear decline in Recall. The first regime switch is at 15% noise with Precision going from $0.7 - 0.72$ to 0.66 and Recall from $0.12 - 0.13$ to $0.10 - 0.11$. The second regime switch is at 65% noise with a jump down in Precision to 0.45 and Recall at 0.04. The final regime switch is at 90% noise, with performance fully degrading to 0.2 Precision and 0.01 Recall. Provided we are happy with 8% Recall, A-M can tolerate up to 40% noise with Precision staying above or around 0.66.

Conclusion (5)

Aide-mémoire can tolerate noise levels of up to 40% of the labels being incorrect if we require that precision be above 0.66.

We believe this result is strong, because developers using A-M would be familiar with the system they work on and unlikely to mislabel more than 40% of the suggestions. Indeed, we expect mislabellings to be rare, the noise in A-M's data to drop, and A-M's performance to improve over time.

3.7.5 The Importance of Mondrian Forests

A-M uses Mondrian Forests because they are online. They are much less commonly used than Random Forests. One can, of course, adapt an offline technique, like Random Forests, to operate in a quasi-online mode via periodic batched retraining. Further, such retrained models could not quickly benefit from a live developer feedback. Nonetheless, we investigate the feasibility of building such a variant by comparing A-M's default Mondrian Forests implementation to Random Forests trained on the full history in an offline setting.

A-M's Random Forest variant shows worse results on the same validation systems. While both configurations show a high MAP (0.89 for Mondrian Forest and 0.70 for Random Forest), the Random Forest results fall quite short on HR and recall, showing a recall of only 10%. Detailed results can be seen in Table 3.8. We hypothesise that class imbalance, which we observed

Table 3.8: Performance of A-M using a Random Forest Classifier across a uniform sample of projects from the second corpus, ordered by PRs per month, after filtering projects that have less than 25 links in total.

Repository	Language	PR/m	MAP	HR	P	R
greenrobot/EventBus	Java	2.41	**1.00**	0.77	**1.00**	0.17
wkhtmltopdf/wkhtmltopdf	C++	3.86	**0.00**	0.78	**0.00**	0.00
zealdocs/zeal	C++	4.22	**0.00**	0.50	**0.00**	0.00
feathersjs/feathers	JavaScript	4.82	**0.00**	0.70	**0.00**	0.00
hexojs/hexo	JavaScript	6.89	**1.00**	0.55	**1.00**	0.14
ngrx/store	TypeScript	7.31	**1.00**	0.73	**1.00**	0.30
databricks/spark-csv	Scala	8.47	**0.00**	0.56	**0.00**	0.00
roughike/BottomBar	Java	11.00	**0.00**	0.68	**0.00**	0.00
milessabin/shapeless	Scala	11.17	**1.00**	0.81	**1.00**	0.14
mobile-shell/mosh	C++	12.31	**1.00**	0.56	**1.00**	0.04
AutoMapper/AutoMapper	C#	21.52	**1.00**	0.39	**1.00**	0.15
tildeio/glimmer	TypeScript	28.42	**1.00**	0.83	**1.00**	0.11
chartjs/Chart.js	JavaScript	34.82	0.82	0.46	0.83	0.15
square/okhttp	Java	38.91	0.88	0.61	0.88	0.05
rangle/augury	TypeScript	45.12	**1.00**	0.48	**1.00**	0.02
rangle/batarangle	TypeScript	45.12	**1.00**	0.51	**1.00**	0.07
mxstbr/react-boilerplate	JavaScript	51.62	0.67	0.40	0.67	0.07
realm/realm-java	Java	55.92	0.88	0.56	0.90	0.08
apollostack/apollo-client	TypeScript	60.30	**1.00**	0.71	**1.00**	0.08
ng-bootstrap/core	TypeScript	67.82	0.86	0.52	0.86	0.06
mrdoob/three.js	JavaScript	85.45	0.81	0.66	0.81	0.17
Sample (mean)		28.93	0.71	0.61	0.71	0.09
Sample (median)		21.52	0.88	0.56	0.90	0.07
Overall (mean)		37.09	0.70	0.63	0.70	0.10
Overall (median)		17.70	0.90	0.63	0.91	0.08

Mondrian Forests to deal better with, and the lack of a feedback loop during validation impacted the performance of the Random Forest based implementation.

Conclusion (6)

Aide-mémoire's use of the online Mondrian Forest Classifier is crucial to its performance. Using an offline Random Forest Classifier penalises performance: HR falls to 0.63 and recall to 0.10.

As we can see, use of Random Forests, even given the full history, severely degrades Recall, making it doubtful that deploying them in batch mode would be competitive to Mondrian Forests.

3.7.6 Evaluating A-M 's Usability

To validate our intuition that A-M will improve PR-issue linking, we put it in front of users. This is a modest study, based on convenience sampling, but the results are promising. We recruited participants from within our Computer Science department, and a total of seven participants chose to participate. We find that participants prefer A-M to the built-in search facilities of GitHub, and were quicker to perform the linking both in time and number of search queries when using

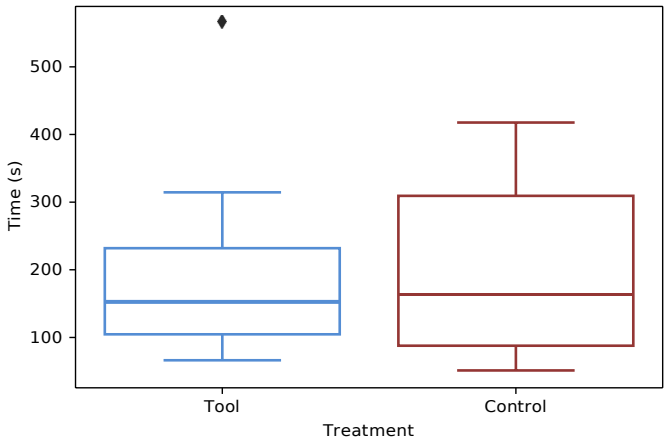


Figure 3.10: Time taken by participants to perform a linking task when using A-M and falling back on GitHub search versus using GitHub only. There is no statistically significant difference in average time taken ($p = 0.41$ according to a Mann-Whitney U test), however, we see that the variance of the time taken per linking task shrinks when using the tool.

A-M.

The participants had on average more than 5 years of experience programming, if we include programming done as part of their academic degree; however, most had less than 1 year of industrial experience. Although all but one used GitHub for personal projects, and all were familiar with the website, less than half used the ticketing system that is present on the website. A little more than half had previous experience submitting PRs and were familiar with the system.

During the user study, participants took on average nine fewer search queries and 21 seconds less per task to find the relevant elided issue when using A-M, however, the time difference is not statistically significant. More importantly, time taken per linking task was more consistently closer to the average for the tool cohort. The distributions of time taken per linking task can be seen in Figure 3.10.

Most participants found the task only a tad on the easy side (3.3 out of 5, where 5 is very easy). The group using the tool found the task easier than the control group (3.5 vs 3.0). When determining the relevancy of an issue to a pull-request, all participants considered keywords from the title and the body of the pull-request. Subsequent criteria differ by participant, but most used the developer and reviewer identifiers as criteria for the search. Two participants further refined their queries using timestamps. These criteria are in line with some of the features considered for our classifier and presented in Section 3.4.

For the tool group, when asked about their preference between the proposed tool and the existing GitHub search facilities, most preferred the tool (4.5 out of 5), and all agreed that they would use such a tool if it were available.

We additionally asked for feedback on how we may improve the prototype we presented. A common suggestion was to improve the responsiveness of the prototype. One participant

proposed that suggestions should be precomputed before they click the plug-in, *i.e.* once the page loads. Two participants suggested that seeing the probability score assigned by the model would help them make quicker decisions as to whether they should investigate the suggestions any further. Overall the reception was positive, one participant stating that "With the usability improvements, it can be of great help for development teams. I believe it can be especially useful for test/QA managers.".

3.7.7 Threats to Validity

A-M faces standard threat to its internal validity, which we mitigate by making our data and experimental harness available for examination and reproduction. Next, we discuss three external threats, of which the last, potential bias from our collection process, is the most important, before turning to the construct validity threat posed by our reliance on weak labelling.

External: Our exclusive use of GitHub and its PRs poses an external threat, because of the extreme class imbalance considering only projects that use PRs, as Kalliamvakou *et al.* explain in their Peril VI [19]. We cannot directly mitigate this threat because we evaluate A-M on PR statistics, like activity. We observe that the bias introduced by our popularity thresholding reduces the class imbalance in PR usage in GitHub projects.

Another external threat is overfitting, which our feature selection could exacerbate. To counteract this threat, we collected a separate internal dev set. We therefore run the risk of underperforming in the final evaluation if the dev set is unrepresentative of the wider corpus, but we accept this potentially suboptimal performance in order to minimise the risk of overfitting.

To collect our project corpus, we first filter by language and size, then use a popularity-threshold over the distribution of projects to avoid low quality or low activity projects (Section 3.5.1). This collection process introduces three distinct threats to the external validity of our results. We consider only Java, JavaScript, C#, TypeScript, Scala, and C++ to make data collection sufficiently fast and cope with GitHub's bandwidth restrictions. To mitigate this threat, we uniformly sample five languages from an existing ranking of popular languages [96], then added Java to allow comparing A-M with RCLinker, the current state-of-the-art. We filter by size only to remove trivial projects, which abound in GitHub. Our threshold of 100 LOC across all project files is conservative, from first principles, and standard practice. We want to bias our corpus toward maintained and consequential projects. Selecting projects by popularity achieves this. To mitigate the external threat this introduces, we note that our popularity-threshold selection does not directly rely on variables that we use in our evaluation.

Construct Validity: The threat of A-M's construct validity rests on our weak labelling of the collected data. Establishing a ground truth over data requires choosing a point on the spectrum between two poles — a golden set of manually labelled and validated data and using unsupervised methods over vast quantities of uniformly sampled raw data occurring in the

wild. The experimental goal determines each pole's desirability. Since we designed A-M to be language-agnostic and to scale, we opted to be closer to the unsupervised pole.

We constructed the weak labels of missing links by removing those that had previously been manually recorded by developers. Thus, our training data is representative only of those links included by developers at PR submission or during change review, rather than those they missed completely. Even if actual links and those recognised by developers are statistically different in the feature space, it is still useful to suggest such links; we found that 16% of PRs are subsequently linked and automating this via A-M would save time.

To mitigate this construct threat, we validated our initial knowledge construction by performing a manual classification of links recorded by our heuristics on a sample of 30 positive and 30 negative examples uniformly selected from all projects. We initially found a significant number of links to the issue/pull request with the ID '#1'. We removed these links and reassessed the correctness of the remaining links to find more than 80% of the links detected by our heuristic as correct.

3.8 Related Work

Good PR-issue linking accelerates development: PR-issue links allow developers to more quickly understand why a pull-request was submitted or how an issue was resolved in code; they also permit the use of productivity enhancing techniques like automatic bug localisation [83, 84], or automatic patch generation tools such as R2Fix [102]. PR-issue links are a developer-centric form of software traceable links, as studied in requirements engineering (RE). We compare and contrast A-M to software traceability, we detail how modern development enables and calls for A-M. Finally, we summarise developer-centric work that solves the related commit-issue linking problem in an offline scenario.

3.8.1 Traceability

Requirements engineering (RE) focuses on stakeholders, decision makers, and their artefacts: requirements, documentation, specification, and design or architectural documents. These artefacts tend to be natural language, text or speech, and often go unrecorded. When they are recorded, they exist in multiple formats, including spreadsheets, email, figures, and printed material. They further encompass developer artefacts, such as source code, pull-requests, commits, and issues, but do not focus on them.

Software traceability seeks to infer *traces* (*i.e.* links) between these heterogeneous artefacts [4]. Missing or hard-to-parse artefacts greatly complicate trace recovery, which is why much work on traceability seeks to provide tooling to decision makers to capture or parse these artefacts [21, 22, 23]. Software artefacts span a multitude of formats. To extract links between them, traceability tools must contend with the heterogeneity of artefacts. As a result, they often

leverage general, abstract features, such as textual features extracted using Information Retrieval techniques [25]. Traceability tools face a deployability challenge. Improving traceability requires capturing developer decisions. To date, researchers have investigated new workflows or invasive instrumentation, which raises privacy concerns, to record these decisions [24].

In contrast, A-M exclusively focuses on a specific software traceability problem — PR-issue link inference. This focus enables A-M to side-step the heterogeneity problem. First, version control and issue tracking are almost ubiquitous in modern software development. PRs and issues are plentiful, well-suited for data hungry machine learning. Second, their format is well-documented. A-M's focus on PR-issue link inference also allows it to exploit the structure in PR meta-data (Section 3.4). Our preliminary study of "CONTRIBUTING.MD" files (Section 3.7.1) also found that 33 out of 43 projects specify a PR template. To address deployability, we designed A-M to seamlessly integrate into modern development practice. As Section 3.3.2 details, A-M suggests links when a developer closes an issue or submits a PR, when this information is pertinent, without intrusive instrumentation and its attendant privacy concerns.

3.8.2 Modern Development

Modern development increasingly relies on tooling that integrates Version Control, Issue Tracking, Wikis, Continuous Integration and Continuous Deployment under a single system. Notable examples are GitHub and Atlassian's JIRA. This new development paradigm poses new problems and opportunities. Kalliamvakou *et al.* [19] elucidates these, using GitHub as their archetypal example. A particular opportunity Kalliamvakou *et al.* identify is this paradigm's integration of Version Control and Issue Tracking. A-M rests on this integration. A-M also takes a step toward realising a promise they identify: interlinking developers, pull requests, issues and commits to offer a comprehensive view of the software development process.

Lack of PR-issue links is an ongoing problem in modern software development (Section 3.7.1). So much so, Agile practice specifies spring cleaning an issue (a user story in Agile terminology) backlog. During backlog refinement, developers remove stale stories and reprioritise and re-estimate remaining stories. When all stories are stale, this practice discards all sprint-related artefacts — issues, feature requests, user stories, as well their links — in favour of starting the next sprint from a clean slate [8]. Projects must resort to this spring cleaning all too often [20]. Clearly, this practice loses valuable information. The loss of documentation it entails it just one example. Adoption of PR-issue inference tooling, like A-M, promises to reduce the need to resort to this drastic measure by enabling automatic issue triaging.

3.8.3 Missing Links

The *missing link problem* is the offline prediction problem of inferring missing commit-issue links given a version history and issue tracker archive. Bachman *et al.* were the first to formulate and quantify this problem [32, 12]. Their work aids developers indirectly by helping researchers and

tool-smiths avoid the bias introduced by missing links that could undermine their techniques or tools. Specifically, they show that by assuming recorded links to be representative of all links, tools are biased to use code from more experienced developers, thus not learning from mistakes or bugs introduced by less experienced contributors.

Bachman *et al.* together with Apache Commons developers manually supplied missing links and published their Apache Commons corpus. Wo *et al.* are the first to attempt to automatically infer them. They propose ReLink [33], which measures the similarity of change logs and bug reports with cosine similarity on tf-idf vectors, learning a threshold for true links. Nguyen *et al.* exploited commit and issue tracker metadata in MLink [34] to improve recall over ReLink. They evaluated MLink on the Apache Commons corpus, making it the de facto standard. ReLink and MLink first consider only commit data to form an initial set of commit-issue links that they then filter. Prechlet and Pepper [35] dispense with the initial commit-only stage, and instead consider both commit and issue data from the start. They argue this bi-directional inference is more sound. Their BFLinks proposes two link predictors (based on bug and commit IDs) and a series of filters to reduce the candidate set of links.

Prior to A-M, Li *et al.* was the state of the art. RCLinker [36] further improves recall. Section 3.6 details RCLinker's realisation. RCLinker relies on ChangeScribe [37] to produce textual descriptions of commits, especially those that lack commit messages. PRs have their own message and aggregate multiple commits and their messages. This fact alleviates the problem of sparse commit descriptions for A-M. Both tools would benefit from better PR summarisation and description. Liu *et al.* [38] propose a tool, based on a bi-directional RNN with a copy network, to tackle this task.

Sun *et al.* [39] used non-source files in commits for commit-issue link inference. They argue that these files are important for capturing developer intent. They use the standard heuristics, such as checking for camelCase or snake_case, to determine the relevancy of a non-source file in a commit. As is conventional, they implement these heuristics as regexes. They use the resulting set of non-source files together with the co-committed source files to compute textual features. They scan a preset and fixed list of the similarity thresholds to find the maximum F1-score where Recall is at least 0.80. The procedure raises two unanswered questions. First, how did the authors determine the threshold granularity? Second, how does training FRLink on F1-score for a task whose performance is measured in terms of F1-score avoid overfitting? They report these choices allow FRLink to improve Recall over previous work while matching or improving F1-score.

Sun *et al.* evaluate FRLink on a new corpus of GitHub projects. This corpus differs from the Apache corpus used in prior work in the conventions governing commit messages: Apache messages tend to be descriptive [40], while FRLink's GitHub sample tends to contain exact matches, because copying issue text is common practice [41]. Sun *et al.* do not investigate the

effect of this differing practice on their results. They specify their corpus in sufficient detail to reconstruct it. FRLink, the tool, however, is not available. When we tried to reproduce FRLink, using its description in Sun *et al.*'s paper, we were unable to reproduce the reported results. We contacted the authors for help explaining and correcting our reproduction without response. More recently, Sun *et al.* [42] treat existing links as labels and reformulate the missing link problem into a semi-supervised problem. As Bachmann *et al.* found, existing links are biased; Sun *et al.* do not discuss how they coped with this bias. They report that their solution, PULink, outperforms FRLink on FRLink's corpus. Like FRLink, PULink is not available.

Ruan *et al.* [41] empirically studied the state of commit-issue linking on GitHub Java projects and found only 42.2% to be linked. They propose DeepLink, a neural approach to the missing links problem. DeepLink trains a text embedding for non-source artefacts and a code embedding for source artefacts using the skip-gram model [43, 44], then passes each of these embeddings separately through an LSTM layer to obtain the final vector representations. They use cosine similarity to compare the vectors, choosing the maximum similarity to represent the score of the commit-issue link. They show an improvement over FRLink in terms of F1-score, and further show that pre-processing heuristics similar to previous work, such as ReLink [33] or MLink [34], help DeepLink achieve a higher F1-score. They also spot that the FRLink corpus had commit logs and issue titles that are exact matches, introducing bias in the dataset which Ruan *et al.* handle, while FRLink does not. Since they did not evaluate DeepLink on Apache Commons or against RCLinker and DeepLink is not available, we do not know its performance relative to RCLinker.

Rath *et al.* [45] also tackle the missing link problem, but from within the requirement engineering community and without referencing the line of work stemming from Bachmann *et al.*. The missing link inference work above uses a vector space model over unigrams for textual features. They opt instead for a n-gram model. They are the first to perform feature selection, using Weka's feature auto-selection. They report promising results but on a different dataset than Apache Commons. As previously noted (Section 3.6), Rath *et al.*'s work is difficult to compare with A-M, because it uses temporal features, such as a predicate that is true when a commit falls between issue's creation and its resolution. In A-M's online setting, this predicate will sometimes be defined in terms of an event that is in the future relative to the present query.

A-M is the first tool that solves issue-PR link prediction in a online fashion. Unlike previous work, we do not rely on change summarisers to produce a natural language description of the source changes; we exploit the PR description instead. As Section 3.3.2 details, using A-M requires only installing a lightweight Chrome plugin or a precommit script. It can integrate into a developer's workflow because it intervenes when a developer submits a commit or closes an issue — when link suggestions are pertinent. Since developers approve our suggestions and silence is merely the status quo, we must avoid distracting them with incorrect suggestions. Thus,

A-M differs from previous work in valuing precision over recall.

3.9 Conclusion and Future Work

Related work has either treated missing links as as an offline problem or has relied on invasive instrumentation across a range of developer activities. We present an alternative in the form of Aide-mémoire, the first tool to solve the problem of missing links in an online setting. Exploiting existing metadata associated with PRs, such as textual descriptions allows us to avoid reliance on more terse commit message or commit summarisation when these are absent. We also generalise across programming languages and project sizes.

We find that Aide-mémoire generalises well across a much larger range of corpora than previously considered, and outperforms a retargeted version of a state-of-the-art offline tool. Crucially, it does not require customisation of toolchains or invasive monitoring of developer activity. As Aide-mémoire incrementally improves linking within a project, it will help to reduce the noise in its own training data; we therefore expect its performance to improve over time as a project uses it.

A future version could interact with Eiffel by Ståhl *et al.* [5] by emitting the appropriately formatted JSON whenever a developer selects to record a link from a PR to an Issue. This would alleviate issues with the current systems, including A-M, where developers enter these manually in a system outside continuous integration frameworks. This manual practice suffers from inconsistent formats, forgetfulness and other human errors that automation can solve.

Acknowledgements — The authors acknowledge the use of the UCL Legion High Performance Computing Facility (Legion@UCL), and associated support services, in the completion of this work. This research is supported by the EPSRC Ref. EP/J017515/1. We would also like to thank Laurie Tratt for sharing his insight into the management of issues and pull requests within Open Source projects.

4

Flexeme: Untangling Commits Using Lexical Flows

Paper Authors

Profir-Petru Pârțachi, Department of Computer Science, University College London, United Kingdom

Santanu Kumar Dash, University of Surrey, United Kingdom

Miltiadis Allamanis, Microsoft Research, Cambridge, United Kingdom

Earl T. Barr, Department of Computer Science, University College London, United Kingdom

Abstract — Today, most developers bundle changes into commits that they submit to a shared code repository. Tangled commits intermix distinct concerns, such as a bug fix and a new feature. They cause issues for developers, reviewers, and researchers alike: they restrict the usability of tools such as git bisect, make patch comprehension more difficult, and force researchers who mine software repositories to contend with noise. We present a novel data structure, the δ-NFG, a multiversion Program Dependency Graph augmented with name flows. A δ-NFG directly and simultaneously encodes different program versions, thereby capturing commits, and annotates data flow edges with the names/lexemes that flow across them. Our technique, FLEXEME, builds a δ-NFG from commits, then applies Agglomerative Clustering using Graph Similarity to that δ-NFG to untangle its commits. At the untangling task on a C# corpus, our implementation, HEDDLE, improves the state-of-the-art on accuracy by 0.14, achieving 0.81, in a fraction of the time: HEDDLE is 32 times faster than the previous state-of-the-art.

4.1 Introduction

Separation of concerns is fundamental to managing complexity. Ideally, a commit to code repositories obeys this principle and focuses on a single concern. However, in practice, many commits tangle concerns [52, 9]. Time pressure is one reason. Another is that the boundaries between concerns are often unclear. Murphy-Hill *et al.* [48] found that refactoring tasks are often committed together with code for other tasks and that even multiple bug fixes are committed together.

Tangled commits introduce multiple problems. They make searching for fault inducing commits imprecise. Tao *et al.* [47] found that tangled changesets (commits) hamper comprehension and that developers need untangling (changeset decomposition) tools. Barnett *et al.* [52] confirmed this need. Herzig *et al.* [10, 9] studied the bias tangled commits introduce to classification and regression tasks that use version histories. They found that up to 15% of bug fixes in Java systems consisted of multiple fixes in a single commit. They also found that using a tangled version history significantly impacts regression model accuracy. In short, tangled commits harm developer productivity two ways: directly, when a developer must search a version history and indirectly by slowing the creation of tools that exploit version histories.

Version histories permit a multiversion view of code, one in which multiple versions of the code co-exist simultaneously. Le *et al.* built on principles described by Kim and Notkin [54] for multiversion analysis: they constructed a multiversion intraprocedural control flow graph and used it to determine whether a commit fixes all n versions [55]. This task is important when multiple versions are active, as in software product lines, and the patch fixes a vulnerability. Inspired by Le *et al.*, we define a δ-PDG, a multiversion program dependence graph (PDG), a graph that combines a program's data and control graphs [103].

We hypothesise that identifiers differentiate concerns. We harvest names that are used together in a program's execution, as in this statement `«takehome:= tax * salary»` to augment our δ-PDG and produce a δ-NFG. A desirable property of our δ-NFG is its modularity. It allows projecting any combination of data, control, or lexeme. Consequently, we could effortlessly reproduce Barnett *et al.*'s and Herzig *et al.*'s methods [52, 9] to explore the design space in tooling for concern separation in commits.

We introduce FLEXEME, a novel approach to concern separation that uses the δ-NFG. We group edits into concerns using the graph similarity of their neighbourhoods. We base this on the intuition that nodes are defined by the company they keep (their neighbourhoods) and cluster them accordingly. Thus, we reduce the concern separation problem to a graph clustering task. For clustering, we start by considering each edit in a commit as an separate concern, then use graph similarity to agglomeratively cluster them. We compute this similarity using the Weisfeiler-Lehman Graph Kernel [104].

We realised FLEXEME in a tool we call HEDDLE[1]. Developers can run HEDDLE to detect tangled commits prior to pushing them or reviewers can use it to ask developers to untangle commits before branch promotion. We show that HEDDLE improves the state-of-the-art accuracy by 0.14, and run-time by 32 times or $3'10''$. We also demonstrate the utility and expressivity of our δ-PDG construct by adapting Herzig *et al.*'s confidence voters (CV) to use the δ-PDG rather than diff-regions; the resulting novel combination, which we call δ-PDG+CV improves the performance and lowers the run-time of Herzig *et al.*'s unmodified approach.

In summary, we present

1. Two novel data-structures, the δ-PDG, a multiversion program dependence graph, and δ-NFG, which augments a δ-PDG with lexemes;

2. The FLEXEME approach for untangling commits that builds a δ-NFG from a version history then uses graph similarity and agglomerative clustering to segment it; and

3. HEDDLE, a tool that realises FLEXEME and advances the state of art in commit untangling in both accuracy and run-time.

All of the tooling and artefacts needed to reproduce this work are available at https://github.com/PPPI/Flexeme.

4.2 Example

Localising a bug with git bisect to determine a bug inducing commit, or reviewing a changeset during code review in presence of tangled commits can make the task unnecessarily difficult or even impossible [47]. Further, as Herzig et al. [9] found, tangled commits have a statistically significant impact on the performance of regressions methods used for defect prediction. Barnett et al. [52], determined that developer productivity benefits from tools that can propose changeset decompositions. In light of this, it is natural to ask how do different code entities co-occur within a concern?

The code entities of a concern tend to be in close proximity with each other in both the control- and data-flow graphs. Barnett et al. [52] exploited this by using def-use chains, which offer a short-range view of these connections. However, as shown in Figure 4.1, connectivity through the data-flow graphs on its own is insufficient to demarcate concerns. Indeed, this observation is reflected in the relatively lower accuracy rates on concern separation reported in Barnett et al. [52] compared to Herzig et al. [9], the concerns 1 and 2 consisting of the hunks `1a-1d` and `2a-2b`, respectively, could be conflated as they are method invocations of the same Driver class. The conflation might occur even though hunks `1b`, `1c` and `1d` are connected via the

[1] A heddle is wire or cords with eyelets that hold warp yarns in a place in a loom. While it does not untangle, a heddle prevents tangles, so we have named our tangle-preventing tool after it.

```
@@ -127,31 +137,32 @@ namespace Terminal {
 {
  this.barItems = barItems;
  this.host = host;

+ ColorScheme = Colors.Menu;                                    (1a)
  CanFocus = true;
 }
 public override void Redraw(Rect region)
 {

- Driver.SetAttribute(Colors.Menu.Normal);                      (1b)
- DrawFrame(region, true);
+ Driver.SetAttribute(ColorScheme.Normal);
+ DrawFrame(region, padding: 0, fill: true);
  for (int i = 0;
       i < barItems.Children.Length;
       i++)
  {
   var item = barItems.Children [i];
   Move(1, i+1);
-  Driver.SetAttribute(

-   item == null ? Colors.Base.Focus :                           (1c)
-   i == current ? Colors.Menu.Focus :
-   Colors.Menu.Normal
-  );
+  Driver.SetAttribute(
+   item == null ? Colors.Base.Focus :
+   i == current ? ColorScheme.Focus :
+   ColorScheme.Normal
+  );
  for (int p = 0; p < Frame.Width-2; p++)
   if (item == null)

-   Driver.AddSpecial(SpecialChar.HLine);                        (2a)
+   Driver.AddRune(Driver.HLine);
   else

-   Driver.AddCh(' ');                                           (2b)
+   Driver.AddRune(' ');
   if (item == null)
    continue;
   Move(2, i + 1);
   DrawHotString(item.Title,

-   i == current? Colors.Menu.HotFocus :                         (1d)
-                 Colors.Menu.HotNormal,
-   i == current ? Colors.Menu.Focus :
-                  Colors.Menu.Normal);
+   i == current? ColorScheme.HotFocus :
+                 ColorScheme.HotNormal,
+   i == current ? ColorScheme.Focus :
+                  ColorScheme.Normal);
```

Figure 4.1: A diff with two tangled concerns: (a) the change of the drawing API (all other changes) and (b) the migration from using chars and special chars to runes (the two changes related to AddRune). Attempting to disentangle this diff with state-of-the-art tools relying on DU-chains fails because the tangled changes are connected in the def-use chain pertaining to Driver and are in close proximity in the file. Using a PDG allows us to additionally exploit control flow information to aid untangling.

use of the Colors.Menu and ColorScheme which provides counterweight to conflating concerns 1 and 2.

Control-flow can help delineate regions or constructs that handle specific types of concerns. For example, a loop could be performing a specialised computation that forms a single concern on its own. We see an example of such a loop in Figure 4.1. The for loop captures the process by which a screen line is generated and is strongly related to how on screen characters are handled (1a and 1b). This suggests that using a Program Dependency Graph (PDG), which encapsulates both control- and data-flow, as a basis for performing commit untangling overcomes some of the shortcomings of using the data-flow alone. The PDG provides evidence that 1a and 1b could be a part of the same concern because of control-flow. Additionally, the PDG also tells us that 1c and 1d could be a part of the same concern by virtue of data-flow through Colors.Menu and ColorScheme. However, it may be observed that the link with 2a–2b is still strong due to flows through Driver.

Lexemes in the two concerns in Figure 4.1 provide strong evidence for their separation. While concern 1 uses Set$_*$ methods in Driver, concern 2 uses Add$_*$ methods. This evidence is missed by PDGs which discard lexical information. Developers tend to use dissimilar names for different tasks. Dash et al. [105] leveraged this observation to augment data-flow with lexical information to successfully identify type refinements. In our work, we take a similar approach and use lexical information to separate concerns. Our approach to introducing lexemes in our PDG representation is similar to the *name-flows* construct of Dash et al. [105]. While they augmented data-flow with lexemes, we augment PDGs and a description of how we achieve this follows.

4.3 Concerns as Lexical Communities

Given consecutive versions of some code, FLEXEME constructs their PDGs and overlays them adding name-flows [105] to build a δ-NFG. We feed the δ-NFG to a graph clustering procedure to reconstruct atomic commits. A δ-NFG naturally captures flows that bind concerns together such as data- and control-flows. Additionally, it also captures natural correlation in names that developers choose when addressing a given concern.

Figure 4.2 overviews how FLEXEME constructs and uses a δ-NFG. For a sequence of contiguous versions, we generate a PDG first with the help of a compiler. We then combine these PDGs to form a multi-version PDG which we call a δ-PDG. We then decorate the δ-PDG with version specific name-flow information to obtain a δ-NFG. We feed the δ-NFG to agglomerative clustering. We use graph similarity to separate concerns across the original set of contiguous versions. We discuss the details of the δ-NFG construction in Section 4.3.1, Section 4.3.2 and Section 4.3.3. We discuss the graph clustering approach in Section 4.3.4.

4.3.1 Multi-Version Name Flow Graphs

We now formally define the PDG, δ-PDG and δ-NFG in order to bootstrap discussion on the δ-NFG construction.

Definition 4.3.1. *Program Dependency Graph (PDG).* FLEXEME's PDG is a directed graph with node set N and edge set E *s.t.* each node $n \in N$ is annotated with either a program statement or a conditional expression; each edge $e \in E$ has an optional annotation representing the name or the data that flows along it, and a kind that describes the relationship type: data or control.

Definition 4.3.2. *Multi-version Program Dependency Graph (δ-PDG).* A δ-PDG ^{p,q} is the disjoint union of all nodes and edges across all versions in $[p,q]$. δ-PDG ^{p,p} is the PDG at version p.

Definition 4.3.3. *Name Flow Graph (NFG).* FLEXEME's NFG is a standard dataflow graph $G = (N,E)$, augmented with name flows, the raw lexemes of the literals and identifiers in a program text that originate at some node flow across edges and collect in downstream nodes. A name flow labels an edge with those lexemes that flow across it and a node with those that either originate at it or flow into it.

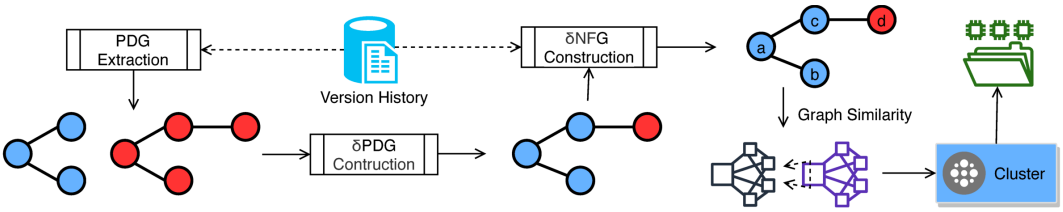


Figure 4.2: Overview of FLEXEME's δ-NFG construction and concern separation.

We re-use NFG from Dash *et al.*'s Refinym [105]. To a first approximation, lexemes flow along def-use chains and collect in the variables on the LHS of assignments.

Definition 4.3.4. *Multi-version Name Flow Graphs (δ-NFG).*
A δ-NFG ^{p,q} is the δ-PDG ^{p,q}, augmented with name flows: If an edge exists in both the PDG and the NFG of a version, we augment the PDG edge with the corresponding name flow.

Inspired by Le and Pattison [55]'s multiversion intraprocedural graphs, we are the first to propose and construct δ-PDGs. To construct a δ-PDG, we start from the initial version considered. For each subsequent version, we make use of line-span information in the PDG and UNIX diff on the source-files to determine changed and unchanged nodes, making FLEXEME language agnostic. Changed nodes are introduced to the δ-PDG as they appear in the new version. δ-PDGs retain nodes deleted across the versions a δ-PDG spans. Deletion becomes a label. We match unchanged nodes between the nodes of the δ-PDG and the new version. To match, we use string similarity to filter candidates and we use line-span proximity to rank them (Section 4.3.2). For nodes the new version deletes, we backpropagate the delete label to edges flowing into them. To add edges, we consider all unmatched edges in the new version, then match their source and target either to existing δ-PDG nodes, or, when a either the source or target does not exist in input δ-PDG, match it to a fresh node (Section 4.3.3). Finally, to obtain a δ-NFG, we endow the δ-PDG with name flow information for each of the versions considered by matching nodes using their line-spans.

4.3.2 Anchoring Nodes Across Versions

To integrate a fresh PDG G^j, we start with the patch P_{ij}. We view each hunk $h \in P_{ij}$ as a pair of snippets (s_i^-, s_j^+) where version i deletes s_i^- and version j adds s_j^+. Snippets s_i^- and s_j^+ do not exist for hunks that only add or only delete; ϕ represents these patches. Accounting for s_i^- is straightforward: we do not update the nodes in the G^{1-i} that fall into s_i^-. Accounting for s_j^+ is non-trivial; much like patching utilities, we need a notion of context.

For all s_j^+, we introduce fresh nodes in the δ-PDG. However, we cannot anchor these nodes until we identify counterparts for nodes in G^j that were untouched by P_{ij}. Identifying untouched nodes in G^i is straightforward; we can simple check locations in P_{ij} against the location of each node in G^j. All touched nodes from G^j are treated as new added nodes and need to be

introduced in the δ-PDG. Identifying the counterpart in G^i of an untouched node in G^j requires a notion of node equivalence.

Definition 4.3.5. *Node Equivalence*. Given a node v_j in a G^j and a candidate node v_i in $G^{1,i}$,

$$v_j \equiv v_i \iff \forall (p,q) \in R(A[v_j], A[v_i]).F(p,q) \geq T$$

Here, $A[k]$ returns all nodes adjacent to the node k in a graph. R is a variant of the *Stable Roommates Problem* [106] where nodes drawn from $A[v_i]$ are matched with nodes drawn from $A[v_j]$. Each node in $A[v_j]$ has an affinity for nodes in $A[v_i]$ proportional to the similarity of lexemes at the nodes. The operator R considers these affinities and tries to match each node with the one it has the highest affinity for. By construction, R always returns a one-to-one match even though two different nodes a and b may have a strong affinity to a third node c. In such a case, either a or b will be matched with c but not both; the one that is not matched to its highest preference (defined in terms of affinity) is matched to the next node from its preference list. We further require that lexical similarity, computed by the function F, is above the threshold T. This lets us control the level of fuzziness while matching nodes. In this work, we have chosen F to be string edit distance and set $T = 1.0$ thus requiring exact matches.

4.3.3 Integrating Nodes Across Versions

Once counterpart nodes are identified using a notion of node equivalence, the next task is to store the location information for the untouched nodes in G^j at theirs counterpart and mark the location with version j. Finally we add all the nodes P_{ij} adds to the δ-PDG and create edges between them and the counterparts of their parents and children.

We demonstrate the δ-PDG construction in Figure 4.3. We show the PDG for two versions of an application — 1 and 2. Since version 1 is our initial version in this example, it is also our δ-PDG G^1. We have omitted the data-flows in the PDG for brevity. Each node in the PDG contains a list of location-version tuple; the location information has been supressed for simplicity. The patch above the two PDGs is the *diff* of the two versions. We have two snippets in the patch: s_1^- for the snippet that is to be deleted in 1 and s_2^+ the snippet that is to be added in 2. Accounting for the deleted line comes for free as we store locations for snippets across versions. All we need to do is to check the location information for s_2^- against the span of the nodes in the G^1. For s_2^+, we need to search for equivalent nodes across the two versions. We perform fuzzy matching on lexemes in nodes shortlisted using location information as detailed above. In the case of s_2^+, we identify the call expressions Move and Driver.AddCh(...) as parents and children, respectively, for s_2^+. Merging s_2^+ into the δ-PDG shown on the right is then a straightforward task of drawing edges between nodes and their parents/children. Once we obtain this δ-PDG representation, we perform the untangling task in a reconstructive manner.

```
@@ -85,4 +85,4 @@
- Driver.SetAttribute (HasFocus? Colors.Base.HotFocus: Colors.Base.HotNormal)
+ Driver.SetAttribute (HasFocus? ColorScheme.HotFocus: ColorScheme.HotNormal)
```

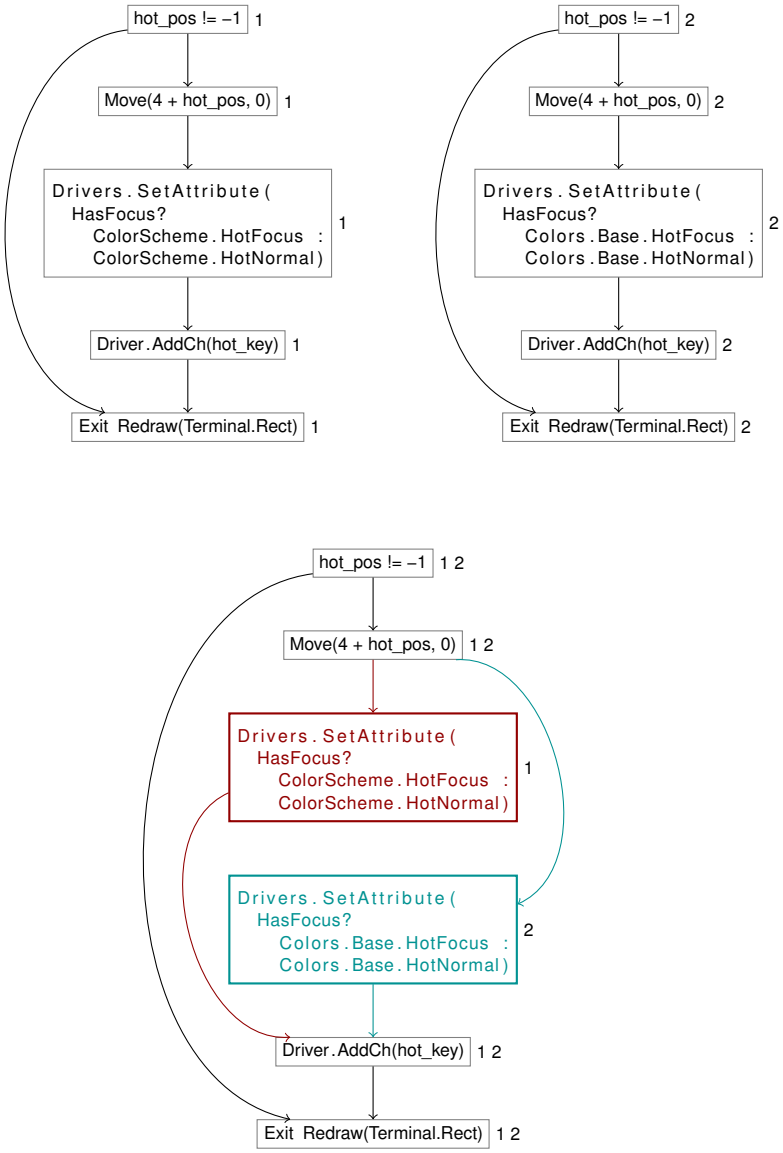


Figure 4.3: Construction of a δ-PDG from two versions — 1 and 2 — of a program. Version 2 is obtained from version 1 by application of the patch shown in the program. Each node is annotated with the version number it is present in.

4.3.4 Identifying Concerns

We start by assuming each change is atomic, and iteratively merge changes that are similar enough. At a high-level, we expect similar nodes to have similar "neighbourhoods". To measure this, we build the k-hop neighbourhood[2], for each node. We then cluster by similarity of these neighbourhoods. To compute graph similarity, we use the Weisfeiler-Lehman graph kernel [104] which builds on top of the Weisfeiler-Lehman graph isomorphism test [107]. For a pair of graphs, the test iteratively generates multi-set labels. When two graphs are isomorphic, then all the sets are identical.

Formally, let the initial vertex labelling function of the graph be $l_0 : V(G) \to \mathcal{L}$, where $\mathcal{L}$ is the space of all node labels. At step i, let $l_i(v) = \{\{l_{i-1}(v')\} \mid v' \in N(v)\}$, where $N(v)$ is the set of neighbours of vertex v. At each iteration i, this process labels the node v with a set comprising the labels of all of v's neighbours. This set becomes the new label of that node. Since isomorphism testing can diverge, we bound it to n iterations and obtain the sequence $\langle G_0, G_1, \ldots, G_n \rangle$.

A positive semi-definite kernel on the non-empty set X is a symmetric function

$$
\begin{aligned}
&k : X \times X \to \mathbb{R} \\
&\textit{s.t.} \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j k(x_i, x_j) \geq 0, \\
&\forall n \in \mathbb{N}, x_1, \ldots, x_n \in X, c_1, \ldots, c_n \in \mathbb{R}.
\end{aligned}
$$

This function can take the arguments in either order and, for any parametrization by real constants, has non-negative weighted sum over all inputs. A graph kernel is a positive semi-definite kernel on a set of graphs. When a kernel takes a set of graphs ($\mathcal{G}$) as input, we define the $K(\mathcal{G})_{ij} = k(G_i, G_j), G_i, G_j \in \mathcal{G}$ to compute the matrix of pairwise kernel values.

Let $\mathcal{G}$ be the set of graphs over which we wish to compute graph based similarity, and let $K : \mathcal{G} \to \mathbb{R}^{|\mathcal{G}|} \times \mathbb{R}^{|\mathcal{G}|}$ be a graph kernel. Then the WL Graph Kernel becomes: $K_{WL}(\mathcal{G}) = K(\mathcal{G}_0) + K(\mathcal{G}_1) + \ldots + K(\mathcal{G}_n)$;, where $\langle \mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n \rangle$ is obtained by applying n steps of the isomorphism test to each graph in $\mathcal{G}$

The WL Graph Kernel K is a meta-kernel that extends an underlying kernel. We want a Subtree WL Graph Kernel that counts the number of identical rooted subtrees for each node in the graph of the same depth as the iteration. In our case, this maps to identical downstream behaviour from each node in terms of each of the flows considered. To achieve this behaviour, we set K to be the Vertex Label Histogram Kernel and encode outgoing flow types in the label function. This kernel is defined as follows: Let Ψ be a function that embeds the graph into a

[2] In our experiments we consider the $k = 1$-hop neighbourhoods.

vector space, often called a feature map in literature, and let $\langle \cdot, \cdot \rangle$ be the inner product, then

$$\Psi(G) = \mathbf{f};$$

$$\mathbf{f}_i = |\{v \mid v \in V(G), l_i \in \mathcal{L}, lv(v) = l_i\}|;$$

$$K(\mathcal{G})_{ij} = \langle \Psi(G_i), \Psi(G_j) \rangle;$$

For clustering, we opt for agglomerative clustering, like Herzig and Zeller [10], Herzig et al. [9]. With node-neighbourhood pairwise similarity information, we can build an affinity, *i.e.* a pair-wise distance, matrix for clustering trivially by simply inverting the value, *i.e.* $1 -$ similarity. Section 4.4.2 details the implementation.

4.4 HEDDLE

HEDDLE realises FLEXEME. HEDDLE first constructs a δ-PDG for each input file, and combines them into a δ-NFG. HEDDLE then decomposes the δ-NFG into a forest, and uses graph kernels to compute distance matrices which we input into agglomerative clustering. We close by describing how a project could adopt HEDDLE.

4.4.1 δ-PDG Construction

We implement both name flow extraction and PDG extraction in Roslyn [108], the open-source compiler for C# and Visual Basic from Microsoft. We store the PDG in GraphViz Dot format. We then implement the PDG merging procedure as described above over the dot files. This allows us to reuse the merging procedure; it is language agnostic. One need only provide PDGs (and optionally name flows) as 'dot' files. To enable the merging process, we store the origin line-span[3] and method membership information in the nodes along with the usual data associated with such graphs, *i.e.* expression information and edge kind. To obtain textual diffs needed for δ-NFG construction, we make use of the UNIX 'diff' tool.

By default, HEDDLE constructs one δ-NFGs per file. To mitigate the problems cross-file dependencies cause and reduce the cost of untangling, HEDDLE merges all graphs associated with a commit, into a single structure. This merge uses node equivalence (Definition 4.3.5) when operating on files that share a namespace. A key difference in the same-version, cross-file setting is that we need to copy over both types of changed nodes and add the edges similar to the added nodes scenario described in Section 4.3.

To simplify our code, our implementation of PDG extraction does not consider `goto` statements in the Control Flow Graph; this does not matter much in practice, as goto statements are very rare in our corpus.

[3] We consider code snippets at line granularity.

4.4.2 Graph Node Clustering

We use GraKeL [109] for the Weisfeiler-Lehman (WL) Graph Kernel [104] implementation and leave the number of iterations of the isomorphism tests at the library's default of 10. We set the underlying kernel to Vertex Histogram Graph Kernel to obtain the same behaviour as the Subtree WL Graph Kernel.

For agglomerative clustering, we use SciPy [94]. We precompute the affinity matrix by using the WL kernel similarity. We call the clustering method with the linkage parameter set to "complete", which mimics the behaviour described in Herzig *et al.* [10], *i.e.* when two groups are merged, the maximal distance from any member of the group to any other group is kept as the new distance. We stop merging groups when they are less than 0.5 similar to any other group instead of providing oracle information to the method. This models the fact that, in practice, developers do not know how many concerns a commit contains.

Code for both δ-PDG construction and node clustering is available online[4].

4.4.3 Deployability

HEDDLE takes ten seconds on average to untangle a commit (Section 4.6.2), and 45s on average to construct and merge the PDG for a commit into a δ-PDG. This is beyond the one second limit suggested by Nielsen [110] for processes that allow a user to feel like operating directly on data, which suggests that our tools should be onboarded into a process that is out-of-band with regards to developer attention. An example of such a process is the build automation within continuous deployment. HEDDLE could be added as an additional pass at the end of the build process providing an untangle report for the code reviewers, ready to be inspected when the review process starts. Further, on-boarding HEDDLE in such a manner makes it independent of the workflow and tooling choices made by the developers; the system would only need to be deployed on the build servers. The report provided would allow reviewers to better focus on the different parts of the patch and aid patch comprehension. There is an initial onboarding cost requiring generating NFGs for all source files in the code-base. However, our construction is incremental and for any fresh patch that needs integration into the δ-NFG, we only need to consider the files touched by the patch.

4.5 Experimental Design

In this section, we discuss how we constructed a dataset, measure untangling performance and our reproduction of two baseline methods.

4.5.1 Corpus Construction

To construct our corpus, we reuse Herzig et al. [9] methodology who artificially tangle atomic commits. Therefore, we consider commits that:

[4] https://github.com/PPPI/Flexeme

1. Have been committed by the same developer within 14 days of each other with no other commit by the same developer in between them.

2. Change namespaces whose names have a large prefix match.

3. Contain files that are frequently changed together.

4. Do not contain certain keywords (such as 'fix', 'bug', 'feature', 'implement') multiple times.

The first criterion mimics the process by which a developer forgets to commit their working directory before picking up a new task. The next criterion is an adaptation of Herzig *et al.*'s 'Change close packages' criterion to the C# environment. The third considers files that are coupled in the version history, thus creating a tangled commit not too dissimilar from commits that naturally occurred. The intuition being that if commit A touches file f_A and commit B touches file f_B, *s.t.* f_A and f_B are frequently changed before(coupling) [111], then A and B should be tangled. The final criterion is a heuristic to ensure that we do not consider tangling commits that we are certain are not atomic. We add this condition to mitigate the problem of tangling actually tangled commits which would cause an issue when computing ground truth. Overall, this artificially created corpus mimics some of the tangled commits we expect developers to make; specifically, it captures the intuition of a developer committing multiple consecutive work units as a single patch. Section 4.7 discusses the threat this poses to HEDDLE's validity.

Following the above procedure, we obtain a shortlist of chains of SHAs of varying length for nine C# systems; we show project statistics in Table 4.1. These SHAs refer to atomic commits. We sanity check that they are atomic by uniformly sampling 30 commits from our corpus and examining each commit for up to five minutes. We found 27/30 to be atomic, 2 of the tangled commits refactor comments (which are invisible and therefore atomic to HEDDLE), and 1 tangled commit due to merging content from a different versioning system (SVN) in a single commit. From this study, we extracted two heuristics that we used to filter out non-atomic commits. Specifically, we excluded all merge commits and those that generate δ-PDGs that have no changed nodes.

We attempt to create tangled commits by selecting the SHAs in the tail of these chains and git cherry-picking them onto the head. We then mark the originating commit in the tangled diff using the individual atomic diffs as not all selections are successful. Some of the successful selections may not have changes from all tangled commits as later commits may shadow them. Therefore, we perform a final pass to learn the actual number of surviving concerns. In the end, we built two sets of tangled commits: those that tangle 2 and those that tangle 3. This models the most common numbers of tangled concerns in the wild [9, Figure 7].

Table 4.2 shows the final statistics for our corpus, where the number of concerns is the count of surviving concerns at the end of the selection process. We report all successfully generated data-points and detail, in Section 4.6, the subsets on which we compared any two methods when

Table 4.1: Project statistics. The last revision indicates the commit at which we performed the 'git clone'.

Project	LOC	# of Commits	Last revision
Commandline	11602	1556	67f77e1
CommonMark	14613	418	f3d5453
Hangfire	40263	2889	175207c
Humanizer	56357	1647	604ebcc
Lean	242974	7086	71bc0fa
Nancy	79192	5497	dbdbe94
Newtonsoft.Json	71704	299	4f8832a
Ninject	13656	784	6a7ed2b
RestSharp	16233	1440	b52b9be

Table 4.2: Successfully tangled commits.

Project	Concerns		
	2	3	Overall
Commandline	308	32	340
CommonMark	52	0	52
Hangfire	229	87	316
Humanizer	85	4	89
Lean	154	24	178
Nancy	284	67	351
Newtonsoft.Json	84	7	91
Ninject	82	0	82
RestSharp	95	18	113
Overall	1373	239	1612

at least one of them did not run on the full corpus due to time-outs. We do not treat time-outs as a zero accuracy result, but drop them from consideration. We remark that the primary source of time-outs is the computational cost of running our reproduction of Herzig *et al.*.

4.5.2 Experimental Setup

Our experiments assess how well our method recovers the original commits compared to the baseline methods proposed by Barnett *et al.* [52] and Herzig *et al.* [9]. Additionally, we measure the runtime cost of the different methods. For this, all methods are run in isolation on the same high-end laptop (i7-8750H @ 3.20 GHz, 16 GB RAM @ 2666 MHz) and we compute accuracy for all methods as follows:

$$A = \frac{\text{\#Correctly labeled nodes}}{\text{\#Nodes in graph}}. \tag{4.1}$$

Both baselines, as well as HEDDLE, may recover an arbitrary permutation of the ground truth labels. To avoid artificially penalising them, we first use the Hungarian Algorithm [112] to find the permutation that maximises accuracy. Consider the ground truth '$[01122]$', should a tool output '$[20011]$', a naïve approach would award it 0.0 accuracy, while a trivial permutation of the labelling function reveals that this is indeed 1.0 accuracy. We report this maximal accuracy for each method.

For the purpose of timing, we perform one burn-in run of commit segmentation followed by 10 repeats that are used to compute the runtime cost. We then obtain the speed-up factor as a non-parametric pairwise comparison between the methods. Notably, we do not include the cost of the static analysis required for each method, rather only the cost of segmentation. This is due to deriving the required program representations for each method as a projection from the δ-NFG.

4.5.3 Reproducing Barnett et al. and Herzig et al.

In order to use Barnett *et al.*'s method as a baseline, we had to re-implement it, because its source is not public. Their method rests on def-use chains. They retain all def-use chains that intersect a diff hunk in a commit. To obtain this, we first recover the dataflow graph and then we separate the flows by 'kill' statements, such as assignments. In a δ-PDG, this becomes a def-use chain projected onto a diff hunk. Two chains are equivalent if (a) they are both changed and are both uses of the same definition or (b) they are a changed use of a changed definition. Under this partition, they divide the parts into trivial and nontrivial. All diff-regions in a trivial part fall within a single method. To avoid overwhelming developers with chains that are highly likely to be atomic, they do not show trivial parts; implicitly, they are assuming that developers can see and avoid method-granular tangling. In contrast, we, like Herzig *et al.*, consider method-granular tangling, so our re-implementation does not distinguish trivial and non-trivial parts.

To reproduce Herzig *et al.*'s method, we reconstruct the call graph by collapsing into hyper-nodes by method membership, we recover the dataflow from the δ-PDG, and we additionally generate an occurrence matrix specifying the files changed by a commit, as well as file sizes in terms of number of lines. Finally, we compute a diff-region granular corpus for all the successful tangles, as the Herzig *et al.* algorithm works at a diff-region granularity. Using this information, we construct a distance matrix for each tangled commit. This distance matrix is populated by the sum of the distances from each individual voter. All confidence voters are identical to the original paper with one exception. We replaced package distance by namespace distance; we do, however, compute it in the same manner. At evaluation time, we also provide the number of concerns to be untangled. This is known by construction, as in the original paper. We perform agglomerative clustering on the resultant matrix using complete linkage, *i.e.* taking the maximum distance over all diff-regions within a cluster.

We also create a version of Herzig *et al.*'s confidence voters method that operates directly on δ-PDGs, which we call δ-PDG+CV. The last stage here is not dissimilar to FLEXEME, with the remark that we still provide Herzig *et al.*'s approach with oracle access to the number of concerns while FLEXEME requires only a similarity threshold. We implemented the voters so that only file distance and change coupling require auxiliary information. This is pre-computed from the git history of the project under analysis. Every other voter — call graph distance, data dependency

Table 4.3: Median performance of untangling commits for each method by project and number of tangled concerns. The performance differences are significant to $p < 0.001$ for all overall results according to a two-tailed Wilcoxon pair-wise test on the common set of data-points, except δ-PDG+CV vs HEDDLE, which is significant to $p < 0.001$. Entries indicated by a '*' signify that there was no relevant data point to report the performance on and those indicated by 'x' indicate time-outs.

Project Name	**Barnett *et al.* [52]**			**Herzig *et al.* [9]**		
	2	3	Overall	2	3	Overall
Commandline	0.18	0.21	0.19	0.67	0.48	0.64
CommonMark	0.20	*	0.20	0.65	*	0.65
Hangfire	0.16	0.13	0.15	0.70	0.54	0.64
Humanizer	0.18	0.31	0.18	0.64	0.42	0.62
Lean	0.19	0.12	0.18	0.69	0.62	0.69
Nancy	0.09	0.08	0.09	0.70	0.56	0.67
Newtonsoft.Json	0.15	0.11	0.15	0.71	0.56	0.71
Ninject	0.14	*	0.14	0.57	*	0.57
RestSharp	0.12	0.14	0.12	0.71	0.69	0.70
Overall	0.14	0.11	0.13	0.69	0.62	0.67

Project Name	**δ-PDG+CV**			**HEDDLE (δ-NFG + WL)**		
	2	3	Overall	2	3	Overall
Commandline	0.77	0.84	0.80	**0.82**	**0.92**	**0.82**
CommonMark	**0.90**	*	**0.90**	0.70	*	0.70
Hangfire	0.84	**0.88**	**0.87**	**0.86**	0.68	0.79
Humanizer	0.69	x	0.69	**0.83**	**0.57**	**0.81**
Lean	**0.84**	0.71	**0.84**	0.77	**0.82**	0.80
Nancy	**0.86**	0.80	**0.86**	0.81	**0.92**	0.84
Newtonsoft.Json	**0.86**	**0.69**	**0.82**	0.71	0.52	0.71
Ninject	**0.94**	*	**0.94**	0.80	*	0.80
RestSharp	0.74	0.53	0.70	**0.82**	**0.89**	**0.82**
Overall	**0.83**	**0.84**	**0.83**	0.81	**0.84**	0.81

and namespace distance — are computed on demand only for the nodes that we consider for merging.

For both baselines, as well as HEDDLE, we measure only the time taken to untangle and not the construction of auxiliary structures. We exclude the construction time as we derive the DU-chains, call-graphs and dataflow-graphs from our δ-PDG.

4.6 Results

In this section, we compare HEDDLE against our two baselines, in terms of accuracy and runtime. To implement our baselines, we reproduced the methodology and tooling from Herzig *et al.* [9] and Barnett *et al.* [52]. We show that HEDDLE outperforms, in both accuracy and run-time, our reproduction of Herzig *et al.*'s method. We report comparisons between tools only on the subset of data-points on which both tools run to completion.

Table 4.4: Median time taken (s) to untangling commits for each method by project and number of tangled concerns up to 3 sig figs. The runtime cost differences are significant to $p < 0.001$ for all overall results according to a two-tailed Wilcoxon pair-wise test, except δ-PDG+CV vs HEDDLE, where the difference is not statistically significant ($p = 0.51$). Entries indicated by a '*' signify that there was no relevant data point to report the performance on and those indicated by 'x' indicate time-outs.

Project Name	**Barnett *et al.* [52]**			**Herzig *et al.* [9]**		
	2	3	Overall	2	3	Overall
Commandline	**0.10**	**8.51**	**0.12**	10.51	8.56	9.26
CommonMark	**2.56**	*	**2.56**	1.96×10^3	*	1.96×10^3
Hangfire	**1.15**	**4.97**	**1.95**	1.30×10^4	5.57×10^4	1.72×10^4
Humanizer	**0.44**	**0.23**	**0.41**	40.62	49.53	44.44
Lean	**1.00**	**1.59**	**1.28**	345.05	173.58	288.35
Nancy	**2.06**	**5.63**	**2.42**	570.57	1.29×10^3	600.16
Newtonsoft.Json	**2.14**	**6.42**	**2.35**	225.62	510.77	230.49
Ninject	**1.25**	*	**1.25**	81.53	*	81.53
RestSharp	**0.74**	1.25	**0.78**	46.22	222.98	72.09
Overall	**1.02**	**5.02**	**1.41**	81.53	647.99	117.35

Project Name	**δ-PDG+CV**			**HEDDLE (δ-NFG + WL)**		
	2	3	Overall	2	3	Overall
Commandline	0.42	153.55	0.59	0.85	182.62	1.04
CommonMark	10.38	*	10.38	14.95	*	14.95
Hangfire	10.61	123.99	13.84	8.06	45.29	11.6
Humanizer	9.24	x	9.24	4.86	2.56	4.58
Lean	19.28	17.08	19.28	18.07	24.07	18.23
Nancy	22.04	20.52	21.96	18.38	85.55	21.78
Newtonsoft.Json	20.04	51.54	20.32	8.01	11.98	8.58
Ninject	4.52	*	4.52	14.99	*	14.99
RestSharp	7.80	**1.01**	4.99	9.86	26.25	10.11
Overall	7.17	70.35	10.27	7.99	43.29	9.56

4.6.1 Untangling Accuracy

When recovering the original partition of the δ-NFG from our artificial tangle of code concerns, HEDDLE achieves a median accuracy of 0.81 and a high of 0.84 on the project Nancy; it outperforms Herzig *et al.* by 0.14 and trails δ-PDG+CV only by 0.02 while scaling better to big patches. Unlike Herzig *et al.*, HEDDLE achieves this result without resorting to heuristics or manual feature construction.

HEDDLE outperforms both baselines in accuracy, the difference being statistically significant at $p < 0.001$ (Wilcoxon pair-wise test), and matches the performance of δ-PDG+CV, the new technique we have built from grafting Herzig *et al.*'s confidence voters on top of our δ-NFG ($p = 0.76$, Wilcoxon pair-wise test). Unlike HEDDLE, the other approaches consider file-granular features. Specifically, they compute a probability that two files are changed together, and, by proxy, tackle the same concern, from the version history. This allows them to better cluster related changes that span multiple files. HEDDLE, in such cases, relies only on the existence of call

edges between the different files when projected onto a δ-NFG. Further, Herzig *et al.* has oracle access to the number of concerns while HEDDLE has not. Despite the lack of explicit file-level relationships or oracle access, HEDDLE's accuracy matches confidence voters when applied to δ-NFGs, and outperforms them when they are applied to diff-regions.

Of the four methods we consider, our re-implementation of Barnett *et al.*'s method (Section 4.5.3) produces the lowest median accuracy — 0.13. We believe that two reasons account for this accuracy. First, we evaluate our re-implementation on trivial parts. Second, Barnett *et al.* speculate that their high FN rate is due to relations, like method calls, that def-use chains miss [52, §VI.A]. We emphasise that Barnett *et al.* performed much better in its native setting. They built their approach for Microsoft developers and the commits they handle on a daily basis. Section 4.7 details the threats to HEDDLE's validity this difference in methodology incurs. Section 4.8.2 further details their approach and evaluation and its differences to HEDDLE.

When considering accuracy for an increasing number of concerns (See Figure 4.4), only Barnett *et al.* and Herzig *et al.* report statistically significant performance drops ($p < 0.01$ and $p < 0.001$ according to a Mann-Whitney U test). Barnett *et al.*'s drop is, however, not observable at two decimal points, while Herzig *et al.*'s drop is by 0.07. Both δ-NFGs-based tools report statistically indistinguishable results as the number of concerns increases.

As HEDDLE is not privy to the number of concerns, its behaviour on atomic commits is interesting. When we apply HEDDLE to atomic commits, they are correctly identified as atomic with 0.63 accuracy. This results is when we ask the the yes/no question 'Is this commit atomic?'. We also want to determine how wrong HEDDLE is when creating spurious partitions. For this, we consider the node-level accuracy of HEDDLE. The result is 0.93, suggesting that HEDDLE often mislabels a small number of changed nodes.

Table 4.3 shows detailed per project results broken down by project and number of concerns for each of the four untangling techniques.

4.6.2 Untangling Running Time

Figure 4.4b shows that Barnett *et al.* [52]'s def-use chain technique is by far the fastest. This result is expected because the algorithm is, at its core, strongly connected components detection over a sparse graph, and is therefore linear in the number of nodes in def-use chains that contain at least one addition. However, as we have previously seen in Table 4.3, its accuracy is considerably worse.

HEDDLE is 32 times faster than Herzig *et al.* in a pair-wise ratio test. Where n is the number of diff-regions, the Herzig *et al.* technique requires n^2 shortest path computations, each requiring the solution of n^2 reachability queries over the dataflow graph. Consider the sparse occurrence matrix that encodes which commit touched which file; its dimensions are the number of commits by number of files that ever existed in the repository. Herzig *et al.*'s technique also sums each

row of this matrix. Although their technique needs these steps only to populate the distance matrix before agglomerative clustering, these operations are expensive and must be computed for all diff-regions within a patch. The fact that their technique is heavy weight is unsurprising.

When compared to δ-PDG+CV, the performance on graphs tangling only two concerns is comparable; however, HEDDLE scales better as the number of concerns, and the number of changed nodes increases. We estimate the runtime of both HEDDLE and δ-PDG+CV using a robust linear model regression and fitting a second order polynomial in the number of changed nodes (n). We find HEDDLE to scale with $t = 0.3371 - 0.0041n + 0.0015n^2$, $R^2 = 1.00$ and δ-PDG+CV with $t = 0.8794 - 0.0528n + 0.0019n^2$, $R^2 = 0.99$. At 500 nodes changed, which is common in our dataset, this would account for a difference of 68 seconds.

Finally, we compute the pair-wise ratio of runtimes and find that HEDDLE is, over the median of these ratios, 9 times slower than Barnett *et al.* and 32 times faster than Herzig *et al.* at untangling commits, taking, on average, ten seconds per commit.

4.7 Threats to Validity

HEDDLE faces the usual threat to its external validity: the degree to which its corpus of commits across a set of projects is representative. The fact that we construct tangled commits exacerbates this threat and introduces the construct validity threat that commits that we assume are atomic, are not, in fact, atomic. To address the latter threat, we validated the atomicity of the commits, from which we built tangled commits, on a small, uniform sample of 30 commits across our corpus. As is conventional, we choose 30 because this is typically when the central limit theorem starts to apply [113]. We did not validate the representativeness of our corpus against a real-world sample of tangled commits. Ground truth in real-world samples can be hard to identify, so we opted to use the methodology from Herzig *et al.* [9] to create an artificial corpus that mimics some tangled commits we expect developers to make; it captures the intuition of a developer committing multiple consecutive work units as a single patch. This decision restricts our results only to the type of tangled commits we mimic, which generalise only in so far as our algorithmically tangled commits generalise. Further, like Barnett *et al.*, we evaluated HEDDLE only on C# files, so, despite FLEXEME's language-agnosticism, HEDDLE's result may not generalise to other languages.

Our re-implementations of Herzig *et al.*'s and of Barnett *et al.* may contain errors. Section 4.5.3 details these re-implementations and where they differ from their authors' descriptions of the original implementations. Finally, we published these re-implementations at `https://github.com/PPPI/Flexeme`, so other researchers can vet our work.

We borrowed Herzig *et al.*'s commit untangling evaluation strategy wholesale, as Section 4.5.1 and Section 4.5.2 detail. Thus, we were able to directly compare our work with theirs. Barnett *et al.* opted for a different evaluation strategy (Section 4.8.2), because obtaining a ground truth for their evaluation is too time-consuming in their setting. Thus, we can neither directly

compare HEDDLE against their approach, nor assess our re-implementation relative to their tool. They also conducted a user study showing both a developer need for such tooling and that their suggestions are useful. Because we did not conduct a user study, our results lack the sanction of developer approval.

4.8 Related Work

We first discuss the impact of tangled commits both on developers and researchers. We then discuss approaches to untangling such commits followed by a discussion of multiversion representations. We conclude with a discussion of graph kernels.

4.8.1 Impact of Tangled Commits

Tao *et al.* [47] were amongst the first to highlight the problem of change decomposition in their study on code comprehension; they highlight the need for decomposition when many files are touched, multiple features implemented, or multiple bug fixes committed. The latter is diagnosed by Murphy-Hill *et al.* [48] as a deliberate practice to improve programmer productivity. Tao *et al.* conclude that decomposition is required to aid developer understanding of code changes.

Independently, Herzig *et al.* [10, 9] investigate the impact of tangled commits on classification and regression tasks within software engineering research. The authors manually classify a corpora of real-world changesets as atomic, tangled or unknown, and find that the fraction of tangled commits in a series of version histories ranges from 7% to 20%; they also find that most projects contain a maximum of four tangled concerns per commit, which is consistent with previous findings by Kawrykow and Robillard [46]. They find non-atomic commits significantly impact the accuracy of classification and regression tasks such as fault localisation.

4.8.2 Untangling Commits into Atomic Patches

It is natural to think of identification of communities in the δ-NFG as a slicing problem [114]. However, boundaries across concerns do not naturally map to a slicing criterion; it is unclear how to seed a slicing algorithm and when to terminate it. This is because concerns are linked with multiple edges which makes their separation difficult to specify with a slicing criterion. In the rest of this section, we discuss the literature around the problem of tangled commits and the theoretical foundations of FLEXEME.

Research on the impact of both tangled commits and non-essential code changes prompted an investigation into changeset decomposition. Herzig *et al.* [10, 9] apply confidence voters in concert with agglomerative clustering to decompose changesets with promising results, achieving an accuracy of 0.58-0.80 on an artificially constructed dataset that mimics common causes of tangled commits. In contrast, Kirinuki *et al.* [49, 50] compile a database of atomic patterns to aid the identification of tangled commits; they manually classify the resulting decompositions as True, False, or Unclear, and find more than half of the commits are correctly identified as

tangled. The authors recognise that employing a database introduces bias into the system and may necessitate moderation via heuristics, such as ignoring changes that are too fine-grained or add dependencies.

Other approaches rely on dependency graphs and use-define chains: Roover *et al.* [51] use a slicing approach to segment commits across a Program Dependency Graph, and correctly classify commits as (un)tangled in over 90% of the cases for the systems studied, excluding some projects where they are hampered by toolchain limitations. They propose, but do not implement, the use of System Dependency Graphs to reduce some of the limitations of their approach, such as being solely intraprocedural. FLEXEME tackles interprocedural and cross-file dependencies by merging the δ-PDGs of the files touched by a commit.

Barnett *et al.* [52] implement and evaluate a commit-untangling prototype. This prototype projects commits onto def-use chains, clusters the results, then classifies the clusters as trivial or non-trivial. A cluster is trivial if its def-use chains all fall into the same method. Barnett *et al.* employ a mixed approach to evaluate their prototype. They manually investigated results with few non-trivial clusters (0-1), finding that their approach correctly separated 4 of 6 non-atomic commits, or many non-trivial clusters (> 5), finding that, in all cases, their prototype's sole reliance on def-use chains lead to excessive clustering. For results containing 2–5 clusters, they conducted a user-study. They found that 16 out of the 20 developers surveyed agreed that the presented clusters were correct and complete. This result is strong evidence that their lightweight and elegant approach is useful, especially to the tangled commits that Microsoft developers encounter day-to-day. During the interviews, multiple developers agreed that the changeset analysed did indeed tangle two different tasks, sometimes even confirming that developers had themselves separated the commit in question after review. In addition to validating their prototype, their interviews also found evidence for the need for commit decomposition tools. Because they use def-use chains and ignore trivial clusters, Barnett *et al.*'s approach can miss tangled concerns that FLEXEME can discern. Barnett *et al.*'s user study itself shows that this can matter: it reports that some developers disagreed with the classification of some changesets as trivial.

Dias *et al.* [53] take a more developer-centric approach and propose the EpiceaUntangler tool. They instrument the Eclipse IDE and use confidence voters over fine-grained IDE events that are later converted into a similarity score via a Random Forest Regressor. This score is used similarly to Herzig *et al.* [9]'s metrics, *i.e.* to perform agglomerative clustering. They take an instrumentation-based approach to harvest information that would otherwise be lost, such as changes that override earlier ones. This approach also avoids relying on static analysis. They report a high median success rate of 91% when used by developers during a two-week study. While Dias *et al.* sidestep static analysis, they require developers to use an instrumented IDE. HEDDLE is complementary to EpiceaUntangler: it allows reviewers to propose untanglings of code that may originate from development contexts where instrumentation might not be possible.

4.8.3 Multiversion Representations of Code

Related work has considered multiversion representations of programs for static analysis. Kim and Notkin [54] investigate the applicability of different techniques for matching elements between different versions of a program. They examine different program representations, such as String, AST, CFG, Binary or a combination of these as well as the tools that work on them on two hypothetical scenarios. They only consider the ability of the tools to match elements across versions and leave the compact representation of a multiversion structure as future work. Some of the conclusions from the matching challenges presented by Kim and Notkin [54] are echoed in FLEXEME as well, we make use of the UNIX diff as it is stored within version histories; however, we also make use of line-span hints from the compilers for each version of the application to better facilitate matching nodes within a NFG.

Le *et al.* [55] propose a Multiversion Interprocedural Control Graph (MVICFG) for efficient and scalable multiversion patch verification over systems such as the PuTTY SSH client. Our δ-PDG is a generalisation of this approach to a more expressive data structure, with applications beyond traditional static analysis.

Alexandru *et al.* [56] generalise the Le *et al.* MVICFG construction to arbitrary software artefacts by constructing a framework that creates a multiversion representation of concrete syntax trees for a git project. They adopt a generic ANTLR parser, allowing them to be language agnostic, and achieve scalability by state sharing and storing the multi-revision graph structure in a sparse data structure. They show the usefulness of such a framework by means of 'McCabe's Complexity', which they implement in this framework such that it is language agnostic, does not repeat computations unnecessarily and reuses the data stores in the sparse graph by propagating from child to parent node. Sebastian and Harald [57] propose a compact, multiversion AST that cleverly shares state across versions. FLEXEME, in contrast, rests on PDGs and is well-suited for the untangling tasks, as our evaluation demonstrates.

4.8.4 Semantic Slicing of Version Histories

Features in a system often co-evolve, which tangles the changes made for a one high-level feature with others in a version history. To resurface feature-specific changes, they dynamically slice a target version, then walk backwards in history while they can reverse the intra-version patch without conflict; at each version they reach, they add any commit that contains a hunk that touches the current slice to it. The goal of this semantic slicing of version histories is to find a minimal slice of a version history that captures the evolution of a feature. Li *et al.* [58, 59] first formulated and introduced this problem. Semantic slicing is a form of commit untangling backwards through history. This retrospective framing is why they treat the history as immutable. In this initial solution, Li *et al.* treat commits as atomic so their slices may contain noise introduced by tangled commits. To reduce this noise, Li *et al.*, in more recent work [60], unpack commits

into single-file commits into a private, local history. FLEXEME, in contrast, is static and online: built from the ground up to rewrite commits as developer make history. As such FLEXEME and semantic slicing are complementary: FLEXEME would improve the signal to noise ratio of semantic slicing. An interesting direction for future work would be to use FLEXEME to preprocess version histories prior to semantically slicing them as with Definer [60].

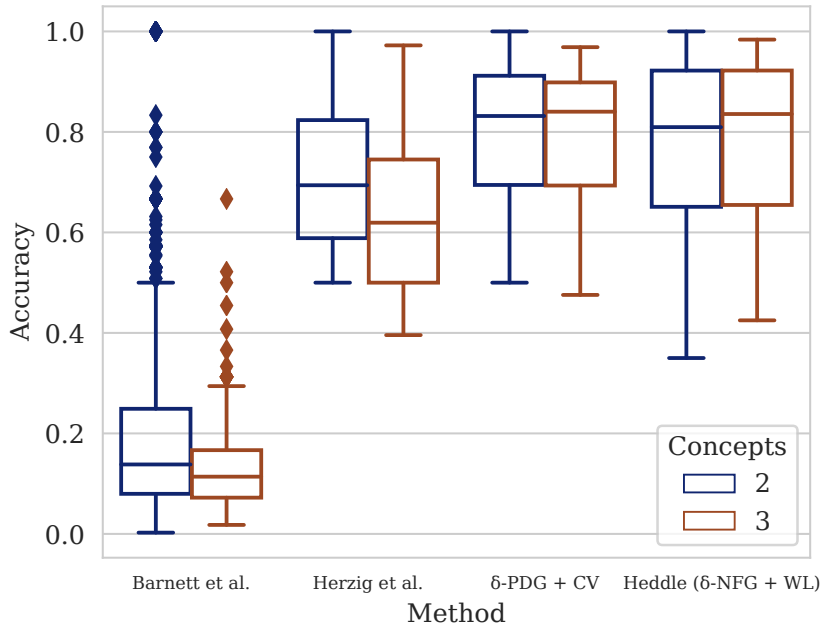
4.8.5 Graph Kernels

Real world data is often structured, from social networks, to protein interactions and even source code. Knowing if a graph instance is similar to another is useful if we wish to make predictions on such data by means that employ either similarity or distance. Vishwanathan *et al.* [115] provide a unified framework to study graph kernels, bringing previously defined kernels under a common umbrella and offering a new method to compute graph kernels on unlabelled graphs in a fast manner, reducing the asymptotic cost from $O(n^6)$ to $O(n^3)$. They mainly study the construction of the different graphs and demonstrate the run-time improvement without applying it to a downstream prediction task. Shervashidze *et al.* [104] introduce the Weisfeiler-Lehman Graph kernel, which they evaluate with three underlying kernels — subtree, edge histogram, and shortest path — on several chemical and protein graph datasets. Although code is often represented as a graph structure and the methods presented here are also used by us to compute graph similarity, this literature primarily concerns itself with chemical and social network datasets that have become standardised benchmarks.

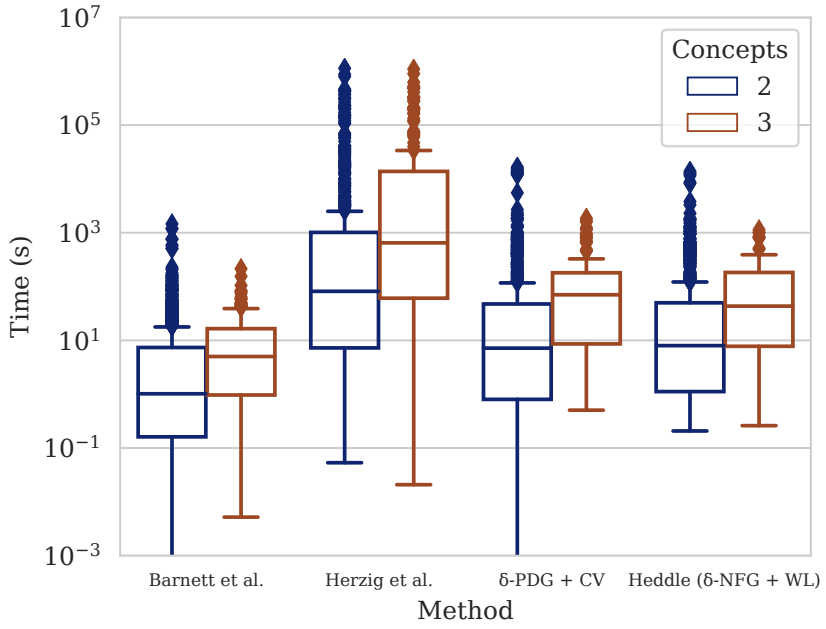
4.9 Conclusion

We have presented FLEXEME— a new approach to commit untangling. FLEXEME's realisation in HEDDLE advances the state-of-the-art: it is 0.14 more accurate (achieving 0.81) and 32 times faster than the previous state-of-the-art. This result rests on a novel data structure, δ-NFG, which augments a multiversion program dependence graph (also introduced in this paper) with name flows. δ-NFG facilitate dual-channel reasoning across versions. Thus, we believe that δ-NFG will be useful on tasks other than commit untangling, such as code refactoring, notably renaming, or code summarisation, as when suggesting docstrings.

Acknowledgements — This research was supported by the EPSRC Ref. EP/J017515/1 and EPSRC Ref. EP/P005314/1.



(a) Accuracy



(b) Runtime

Figure 4.4: Boxplot comparing the accuracy of the baseline and HEDDLE (Figure 4.4a) as well as time taken (s) to segment a commit (Figure 4.4b) for all projects. The drop in accuracy for Herzig *et al.*'s approach as the number of concerns increases is significant to $p < 0.001$ and Barnett *et al.*'s to $p < 0.01$ according to a Mann-Whitney U test. The results of the same test for δ-PDG+CV and HEDDLE indicate that there is no statistically significant difference ($p = 0.28$ and $p = 0.76$ respectively). All increases in time taken to segment are statistically significant ($p < 0.001$, Mann-Whitney U test).

5

POSIT: Simultaneously Tagging Natural and Programming Languages

Paper Authors

Profir-Petru Pârțachi, Department of Computer Science, University College London, United Kingdom

Santanu Kumar Dash, University of Surrey, United Kingdom

Christoph Treude, University of Adelaide, South Australia, Australia

Earl T. Barr, Department of Computer Science, University College London, United Kingdom

Abstract — Software developers use a mix of source code and natural language text to communicate with each other: Stack Overflow and Developer mailing lists abound with this mixed text. Tagging this mixed text is essential for making progress on two seminal software engineering problems — traceability, and reuse via precise extraction of code snippets from mixed text. In this paper, we borrow code-switching techniques from Natural Language Processing and adapt them to apply to mixed text to solve two problems: language identification and token tagging. Our technique, POSIT, simultaneously provides abstract syntax tree tags for source code tokens, part-of-speech tags for natural language words, and predicts the source language of a token in mixed text. To realize POSIT, we trained a biLSTM network with a Conditional Random Field output layer using abstract syntax tree tags from the CLANG compiler and part-of-speech tags from the Standard Stanford part-of-speech tagger. POSIT improves the state-of-the-art on language identification by 10.6% and PoS/AST tagging by 23.7% in accuracy.

5.1 Introduction

Programmers often mix natural language and code when talking about the source code. Such mixed text is commonly found in mailing lists, documentation, bug discussions, and online fora such as Stack Overflow. Searching and mining mixed text is inevitable when tackling seminal software engineering problems, like traceability and code reuse. Most development tools are monolingual or work at a level of abstraction that does not exploit language-specific information. Few tools directly handle mixed text because the differences between natural languages and formal languages call for different techniques and tools. Disentangling the languages in mixed text, while simultaneously accounting for cross language interactions, is key to exploiting mixed text: it will lay the foundation for new tools that directly handle mixed text and enable the use of existing monolingual tools on pure text snippets extracted from mixed text. Mixed-text-aware tooling will help bind specifications to their implementation or help link bug reports to code.

The *mixed text tagging* problem is the task of tagging each token in a text that mixes at least one natural language with several formal languages. It has two subproblems: identifying a token's origin language (language tagging) and identifying the token's part of speech (PoS) or its abstract syntax tree (AST) tag (PoS/AST tagging). A token may have multiple PoS/AST tags. In the sentence "I foo-ed the String 'Bar'.", 'foo' is a verb in English and a method name (of an object of type String). Therefore, PoS/AST tagging involves building a map that pairs a language to the token's PoS/AST node in that language, for each language operative over that token.

We present POSIT to solve the 1+1 mixed text tagging problem: POSIT distinguishes a Natural language (English) from programming language snippets and tags each text or code snippet under its language's grammar. To this end, POSIT jointly solves both the language segmentation and tagging subproblems. POSIT employs techniques from Natural Language Processing (NLP) for code-switched[1] text. Code-switching occurs when multilingual individuals simultaneously use two (or more) languages. This happens when they want to use the semantics of the embedded language in the host language. Within the NLP space, such mixed text data tends to be bi- and rarely tri-lingual. Unique to our setting is, as our data taught us, the mixing of more than three languages, one natural and often many formal ones — in our corpus, many posts combine a programming language, file paths, diffs, JSON, and URLs.

To validate POSIT, we compare it to Ponzanelli *et al.*'s pioneering work StORMeD [77], the first context-free approach for mixed text. They use an island grammar to parse Java, JSON and XML snippets embedded within English. As English is relegated to water, StORMeD neglects natural language, builds ASTs for its islands, then augments them with natural language snippets to create heterogenous ASTs. POSIT tags both natural languages and formal languages, but does not build trees. Both techniques identify language shifts and both tools label code

[1] The fact that the NLP literature uses the word "code" in their name for the problem of handling text that mixes multiple natural languages is unfortunate in our context. They mean code in the sense of coding theory.

snippets with their AST labels. POSIT is designed from the ground up to handle *untagged* mixed text after training. StORMeD looks for tags and resorts to heuristics in their absence. On the language identification task, StORMeD achieves an accuracy of 71%; on the same dataset, POSIT achieves 81.6%. To compare StORMeD and POSIT on the PoS/AST tagging task, we extracted AST tags from the StORMeD output. Despite not being designed for this task, StORMeD establishes the existing state of the art and achieves 61.9% against POSIT's 85.6%. POSIT outperforms StORMeD here, in part, because it finds many more small code snippets in mixed text. In short, POSIT advances the state-of-the-art on mixed text tagging.

POSIT is not restricted to Java. On the entire Stack Overflow corpus (Java and non-Java posts), POSIT achieves an accuracy of 98.7% for language identification and 96.4% for PoS or AST tagging. A manual examination of POSIT's output on Stack Overflow posts containing 3,233 tokens showed performance consistent with POSIT's results on the evaluation set: 95.1% accuracy on language tagging and 93.7% on PoS/AST tagging. To assess whether POSIT generalises beyond its two training corpora, we manually validated it on e-mails from the Linux Kernel mailing list. Here, POSIT achieved 76.6% accuracy on language tagging and 76.5% on PoS/AST tagging.

POSIT is directly applicable to downstream applications. First, its language identification achieves 95% balanced accuracy when predicting missed code labels and could be the basis of a tool that automatically validates posts before submission. Second, TaskNav [13] is a tool that extracts mixed text for development tasks. POSIT's language identification and PoS/AST tagging enables TaskNav to extract more than two new, reasonable tasks per document: on a corpus of 30 LKML e-mails, it extracts 97 new tasks, 65 of which are reasonable.

Our main contributions follow:

- We have built the first corpus for mixed text that is tagged at token granularity for English and C/C++.
- We present POSIT, an NLP-based code-switching approach for the mixed text tagging problem;
- POSIT can directly improve downstream applications: it can improve the code tagging of Stack Overflow posts and it improves TaskNav, a task extractor.

We make our implementation and the code-comment corpus used for evaluation available at https://github.com/PPPI/POSIT.

5.2 Motivating Example

The mix of source code and natural language in the various documents produced and consumed by software developers presents many challenges to tools that aim to help developers make

On Fri, 24 Aug 2018 02:16:12 +0900 XXX <xxx@xxx.xxx> wrote:
[...]
Looking at the change that broke this we have:
`<-diff removed for brevity->`
Where "`real`" was added as a parameter to `__copy_instruction`. *Note that we pass in "*`dest + len`*" but not "*`real + len`*" as you patch fixes.* `__copy_instruction` was changed by the bad commit with:
`<-diff removed for brevity->`
[...]

Figure 5.1: Example e-mail snippet from the Linux Kernel mailing list. It discusses a patch that fixes a kernel freeze. Here the fix is performed by updating the RIP address by adding `len` to the `real` value during the copying loop. Code tokens are labelled by the authors using the patches as context and rendered using `monospace`.

Where^{ADV} "`real`"$^{string\_literal}_{*}$ was^{VERB} added^{VERB} as^{ADP} a^{DET} parameter^{NOUN} to^{ADP} `__copy_instruction`$^{method\_name}_{*}$.$^{\cdot}$ Note^{NOUN} that^{ADP} we^{PRON} pass^{VERB} in^{ADP} "`dest + len`"$^{string\_literal}_{*}$ but^{CONJ} not^{ADV} "`real + len`"$^{string\_literal}_{*}$ as^{ADP} you^{PRON} patch^{VERB} fixes^{NOUN} .$^{\cdot}$ `__copy_instruction`$^{method\_name}_{*}$ was^{VERB} changed^{VERB} by^{ADP} the^{DET} bad^{ADJ} commit^{NOUN} with^{ADP} :$^{:}$

Figure 5.2: POSIT's output from which TaskNav++ extracts the tasks (pass in "dest + len") and (pass in "real + len"). We show the PoS/AST tags as superscript and mark tokens with $_{*}$ if they are identified as code. POSIT spots the two mention-roles of code tokens as 'string_literal's.

sense of these documents automatically. An example is TaskNav [13], a tool that supports task-based navigation of software documentation by automatically extracting task phrases from a documentation corpus and by surfacing these task phrases in an interactive auto-complete interface. For the extraction of task phrases, TaskNav relies on grammatical dependencies between tokens in software documentation that, in turn, relies on correct parsing of the underlying text. To handle the unique characteristics of software documentation caused by the mix of code and natural language, the TaskNav developers hand-crafted a number of regular expressions to detect code tokens as well as a number of heuristics for sentence completion, such as adding "This method" at the beginning of sentences with missing subject. These heuristics are specific to a programming language (Python in TaskNav's case) and a particular kind of document, such as API documentation dominated by method descriptions.

POSIT has the potential to augment tools such as TaskNav to reliably extract task phrases from any document that mixes code and natural language. As an example, in Figure 5.1, we can see an e-mail excerpt from the LKML[2]. TaskNav only manages to extract trivial task phrases from this excerpt (e.g., "patch fixes") and misses task phrases related to the code tokens of `dest`, `real`, and `len` due to incorrect parsing of the sentence beginning with "Note that ...". After augmenting TaskNav with POSIT, the new version, which we call TaskNav++, manages to extract

[2] https://lkml.org/lkml/2018/8/24/19

two additional task phrases: (pass in “dest + len”) and (pass in “real + len”); we present POSIT’s output on this sentence in Figure 5.2. These additional task phrases extracted with the help of POSIT will help developers find resources relevant to the tasks they are working on, e.g., when they are searching for resources explaining which parameters to use in which scenario. We discuss the performance of TaskNav++ in more detail in Section 5.6.2.

5.3 Mixed Text Tagging

Tags are the non-terminals that produce terminals in a language’s grammar. Given mixed text with k natural languages and l formal languages, let a token’s tag map bind the token to a tag for each of the $k+l$ languages. We consider a formal language to be one which, to a first approximation, has a context-free grammar. The *mixed text tagging problem* is then the problem of building a token’s tag map. For example, in the sentence, “’lieben’ means love in German”, ’lieben’ is a subject in the frame language English and a verb in German. Moving to a coding example, in a sentence such as “I foo-ed the String ’Bar’.”, we observe ’foo’ to be a verb in English and a method name (of an object of type String).

A general solution produces a list of pairs: part-of-speech tags for each of the k natural languages together with the natural language for which we have the tag, and AST tags for each of the l formal languages together with the language within which we have the AST tag. We also consider two special tags Ω and ε that are fresh relative to the set of all tags within all natural and formal languages. We use ε to indicate that a particular language has no candidate tag, while Ω is paired with the origin language, answering the first task of our problem. In the first example above, ’lieben”s tag map is [(Ω, De), (Verb, De), (Noun, En), (ε, C)], if we consider English, German and C. In the code example, ’foo’s’ tag map is [(Ω, C), (ε, De), (Verb, En), (method_name, C)]. In multilingual scenarios, a token might have a tag candidate for every language.

The mixed text tagging problem is context-sensitive. We argue below that determining the token’s origin language is context-sensitive for a single token code-switch. The proof rests on a series of definitions from linguistics which we state next. To bootstrap, a *morpheme* is an atomic unit of meaning in a language. Morphemes differ from words in that they may or may not be free, or stand alone. We source these definitions from Poplack [72].

“*Code-switching* is the alternation of two languages in a single discourse, sentence or constituent. . . . [deletia] . . . [It] was characterised according the degree of integration of items from one language (L_1) to the phonological, morphological, and syntactic patterns of the other (L_2)” [72, §2 ¶2]. We use L_1 to refer to the frame language and L_2 to the embedded one. Further, context-switching has two restrictions on when it may occur. It can only occur after free morphemes. The second restriction is that code-switching occurs at points where juxtapositions between L_1 and L_2 do not violate the syntactic rules of either language. Code-switching allows

integrating items from L_2 into L_1 along any one of phonological, morphological, or syntactic axis, but not all three simultaneously. This last case is considered to be mono-lingual L_1.

Adaptation occurs when an item from L_2 changes when used in L_1 to obey L_1's rules. Adaptation has three forms: morphological, phonological, and syntactical. *Morphological adaptation* represents modifying the spelling of L_2 items to fit L_1 patterns. *Phonological adaptation* represents changing the pronunciation of an L_2 item in an L_1 context. *Syntactic adaptation* represents modifying L_2 items embedded in a discourse, sentence, or constituent in L_1 to obey L_1's syntax. Finally, L_2 items can be used in L_1 *without adaptation*. In this case, these items often reference the code-entity by name and are used as a 'noun' in L_1.

We now consider three cases: (I) L_2 items are morphologically adapted to L_1, (II) L_2 items are syntactically adapted to L_1, and (III) no adaptation of L_2 items occurs before their use in L_1. We do not consider phonological adaptation of L_2 items into L_1 as that is not observable in text.

Case I: Morphological Adaptation Consider using affixation to convert foo/class to foo−ify/verb to denote the action of converting to the class foo. In this case, foo−ify behaves as a bona fide word in L_1. Such examples obey the free-morpheme restriction mentioned above. This enables it to be a separate, stand-alone morpheme/item within L_1. The juxtaposition restriction, further ensures that this parses within L_1. Lacking a context to indicate foo's origin, a parsers would need to assume that it is from L_1.

Case II: Syntactic Adaptation This case manifests similarly to morphological adaptation, such as tense agreement, or, potentially, as word order restrictions. If spelling changes do occur, this case reprises the morphological adaptation case. If the only adaptation is word order, then the task becomes spotting a L_2 token that has stayed unchanged in a L_1 sentence or constituent. This is impossible in general if the two language's vocabularies overlap.

Case III: No Adaptation If no adaptation occurs, then the formal token occurs in L_1. This reduces to the second subcase of the syntactic adaptation case.

5.4 POSIT

POSIT starts from the biLSTM-CRF model presented in Huang *et al.* [116], augments it to have a character-level encoding as seen in Winata *et al.* [117] and adds two learning targets as in Soto and Hirschberg [75]. Figure 5.4 presents the resulting network. The network architecture employed by POSIT is capable of learning to provide a language tag for any $k+l$ languages considered. This model is capable of considering the context in the input using the LSTMs, it can bias its subsequence choices as it predicts tags based on the predictions made thus far, and the character-level encoding allows it to learn token morphology features beyond those that we may expose to it directly as a feature vector.

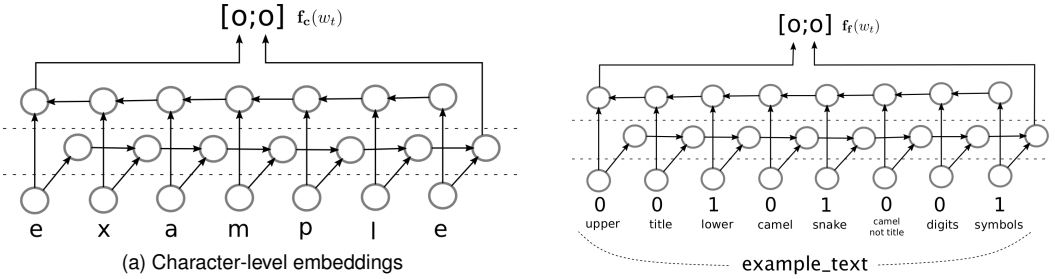


Figure 5.3: Computation of embeddings at the character level and from coding naming and spelling convention features. In the bottom most layer, the circles represent an embedding operation on characters or features to a high-dimensional space. The middle layer represents the forward LSTM and the top most layer — the backward LSTM. At the word level, character and feature vector embeddings are represented by the concatenation of the final states of the forward and backward LSTMs represented by $[\cdot;\cdot]$ in the diagrams above.

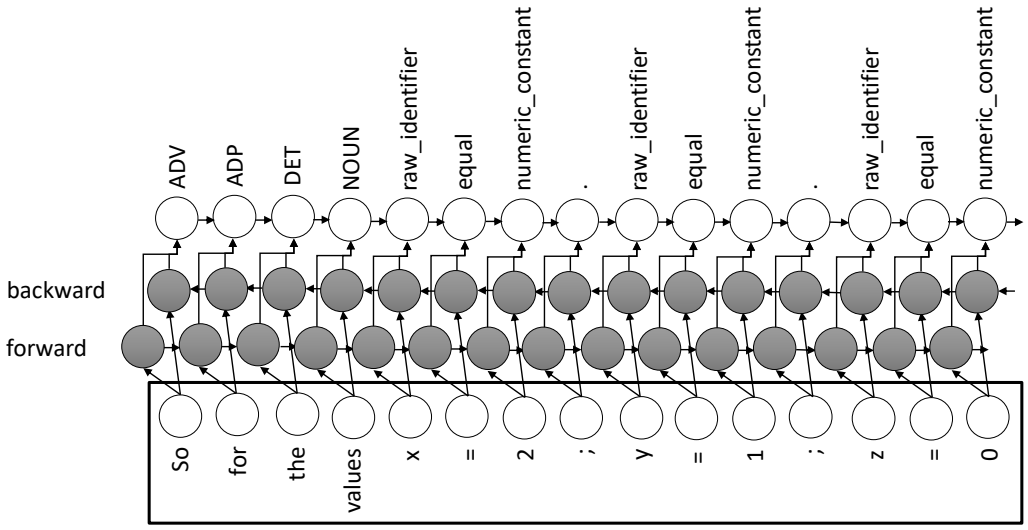


Figure 5.4: A representation of the neural network used for predicting English PoS tags together with compiler derived AST tags. The shaded cells represent LSTM cells, arrows represent the flow of information in the network. The top layer represents a linear Conditional Random Field (CRF) and the transition probabilities are used together with a Viterbi decode to obtain the final output. The first layer is represented by Equation (5.1) and converts the tokenised sentences into vector representations.

Feature Space. We rely on source code attributes to separate code from natural language while tagging both simultaneously. We derive vector embeddings for individual characters to model subtle variations in how natural language is used within source code snippets. Examples of such variations are numbered variables such as i1 or i2 that often index axes during multi-dimensional array operations. Another such variation arises in the naming of loop control variables where the iterator could be referred to in diverse, but related ways, as `i`, `it` or `iter`. These variations create out-of-vocabulary (OOV) words which inhibit modelling of the mixed text. The confounding effects of spelling mistakes and inconsistencies in the NLP literature have been independently observed

by Winata *et al.* [117]. They proposed a bilingual character bidirectional RNN to model OOV words. POSIT uses this approach to capture character level information and address diversity in identifier names.

Additionally, we consider the structural morphology of the tokens. Code tokens are represented differently to natural language tokens. This is due to coding conventions in naming variables. We utilise these norms in developing a representation for the token. Specifically, we encode common conventions and spelling features into a feature vector. We record if the token is: (1) UPPER CASE, (2) Title Case, (3) lower case, (4) CamelCase, (5) snake_case; or if any character: (6) other than the first one is upper case, (7) is a digit, or (8) is a symbol. It may surprise you that font, while often used by humans to segment mixed text, is not in our token morphology feature vector. We did not use it as it is not available in our datasets. For the purposes of code reuse, we use a sequential model over this vector as well, similar to the character level vector, although there is no inherent sequentiality to this data. By ablating the high-level model features, we found that this token morphology feature vector did not significantly improve model performance (Section 5.5.3).

Encoding and Architecture. At a glance, our network, which we present diagrammatically in Figure 5.4, works as follows:

$$\mathbf{x}(t) = [\mathbf{f_w}(w_t); \mathbf{f_c}(w_t); \mathbf{f_f}(w_t)], \tag{5.1}$$

$$\mathbf{h}(t) = f(\mathbf{W}\mathbf{x}(t) + \mathbf{U}\mathbf{h}(t-1)), \tag{5.2}$$

$$\mathbf{y}(t) = g(\mathbf{V}\mathbf{h}(t)). \tag{5.3}$$

In Equation (5.1), we have three sources of information: character-level encodings ($\mathbf{f_c}(w_t)$), token-level encodings ($\mathbf{f_w}(w_t)$) and a feature vector over token morphology ($\mathbf{f_f}(w_t)$). Each captures properties at a different level of granularity. To preserve information, we embed each source independently into a vector space, represented by the three f functions. For both the feature vector and the characters within a word, we compute a representation by passing them as sequences through the biLSTM network in Figure 5.3. This figure represents the internals of $\mathbf{f_c}(w_t)$ and $\mathbf{f_f}(w_t)$ from Equation (5.1) and allows the model to learn patterns within sequences of characters as well as coding naming or spelling conventions cooccurrence patterns. The results of these two biLSTMs together with a word embedding function $\mathbf{f_w}$ are concatenated to become the input to the main biLSTM, $\mathbf{x}(t)$ in Equation (5.1). This enables the network to learn, based on a corpus, semantics for each token. This vector represents the input cells in our full network overview in Figure 5.4, which is enclosed in the box.

We pass the input vector $\mathbf{x}(t)$ through a biLSTM. The biLSTM considers both left and right tokens when predicting tags. Each cell performs the actions of Equation (5.2) and Equation (5.3), with the remark that the backwards-LSTM has the index reversed. This allows the network to consider context up to sentence boundaries. We then make use of the standard softmax function:

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\Sigma_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., K; \tag{5.4}$$

which allows us to generate output probabilities over our learning targets as such:

$$\mathbf{p}(\text{tag}_t \mid \text{tag}_{t-1}) = \text{softmax}(\mathbf{y}(t)), \tag{5.5}$$

$$\mathbf{p}(\text{l}_{\text{id}_t} \mid \text{l}_{\text{id}_{t-1}}) = \text{softmax}(\text{2LP}(\mathbf{h}(t))), \tag{5.6}$$

Equation (5.6) represents language ID transition probabilities, and Equation (5.5) — tag transition probabilities. In Equation (5.6), 2LP represents a 2-layer Multi Layer Perceptron. We make use of these transition probabilities in the CRF layer to output Language IDs and tags for each token while considering predictions made thus far. The trained eye may recognise in Equation (5.5) and Equation (5.6) the transition probabilities of two Markov chains. Indeed, we obtain the optimal output sequence by Viterbi-decoding [118]. While Equation (5.5) may seem to indicate that only single tags can be output by this architecture, this is not true. Given enough data, we can map tuples of tags to new fresh tags and decode at output time. This may not be as efficient as performing multi-tag output directly.

To train the network, we use the negative log-likelihood of the actual sequence being decoded from the CRF network and we backpropagate this through the network. Since we have two training goals, we combine them in the loss function by performing a weighted sum of the negative log-likelihood losses for each individual task, then train the network to perform both tasks jointly. When deployed, POSIT makes use of the CLANG lexer python port to generate the token input required by f_w, f_c, and f_f.

5.5 Evaluation

For each token, POSIT makes two predictions: language IDs and PoS/AST tags. The former task represents correctly identifying where to add </code>-tags. This measures how well POSIT segments English and code. Section 5.5.2 reports POSIT's performance on this task on the evaluation set. For PoS/AST tag prediction, we focus on POSIT's ability to provide tags describing the function of tokens for both modalities reliably. To measure POSIT's performance here, we consider how well the model predicts the tags for a withheld evaluation dataset, which Section 5.5.2. presents along with the English-code segmentation result.

Table 5.1: Corpus statistics for the two corpora considered together with the Training and Development and Evaluation splits. We performed majority class (English) undersampling only for the Stack Overflow training corpus.

Category	**Fold**	Stack Overflow	CodeComments	**Total**
Tokens	Train&Dev	7645103	132189	7777292
	Eval	2612261	176418	2788679
Sentences	Train&Dev	214945	21681	236626
	Eval	195021	8677	203698
English Only	Train&Dev	55.8%	11.3%	51.7%
	Eval	57.0%	11.0%	55.1%
Code Only	Train&Dev	32.6%	79.4%	36.9%
	Eval	38.0%	79.6%	39.7%
Mixed	Train&Dev	11.6%	9.4%	11.4%
	Eval	4.9%	9.3%	5.1%

POSIT implements the network discussed in Section 5.4 in TensorFlow [119]. It uses the Adaptive Moment Estimation (Adam) [120] optimiser to assign the weights in the network. We trained it up to 30 epochs or until we did not observe improvement in three consecutive epochs. We used micro-batches of 64, a learning rate of 10^{-2}, and learning decay rate of 0.95. We use a 100 dimensional word embedding space and a 50 dimensional embedding space for characters. The LSTM hidden state is 96 dimensional for the word representation, 48 dimensional for characters and 4 for the token morphology feature vector. The output of the tag CRF is the concatenation of all final biLSTM states. We use a 2 layer perceptron with 64 and 8 dimensional hidden layers for language ID prediction. We apply a dropout of 0.5. Section 5.5.2 uses this implementation for validation and Section 5.5.3 uses it for ablation. The model's source code is available at `https://github.com/PPPI/POSIT`.

All POSIT runs, training and evaluation, were performed on a high-end laptop using an Intel i7-8750H CPU clocked at 3.9GHz, 24.0 GB of RAM and a Nvidia 1070 GPU with 8 GB of VRAM.

The state-of-the-art tool StORMeD, which we use for comparison, is available as a webservice, which we use by augmenting the demo files made available at `https://stormed.inf.usi.ch/#service`.

5.5.1 Corpus Construction

For our evaluation, we make use of two corpora. We use both to train POSIT, and we evaluate on each to see the performance in two important use-cases, a natural language frame language with embedded code and the reverse. Table 5.1 presents their statistics.

The first corpus is the Stack Overflow Data-dump [121] that Stack Overflow makes available online as an XML-file. It contains the HTML of Stack Overflow posts with code tokens marked using `</code>`- as well as `</pre class="code">`-tags. These tags enable us to construct a ground-truth for the English-code segmentation task. To obtain the PoS tags for English tokens, we use the tokeniser and Standard Stanford part-of-speech tagger present in NLTK [122]. For

AST tags, we use a python port of the CLANG lexer and label tokens using a frequency table built from the second, CodeComment corpus. This additionally ensures that both corpora have the same set of AST tags. We allow matches up to a Levenstein distance of three for them; we choose three from spot-checking the results of various distances: after three, the lists were long and noisy. We address the internal threat introduced by our corpus labelling in Section 5.7.2.

We built the second, CodeComment corpus [123], from the CLANG compilation of 11 native libraries from the Android Open Source Project (AOSP): boringssl, libcxx, libjpeg-turbo, libmpeg2, libpcap, libpng, netcat, netperf, opencv, tcpdump and zlib. We chose these libraries in a manner that diversifies across application areas, such as codecs, network utilities, productivity, and graphics. We wrote a CLANG compiler plugin to harvest all comments and the snippets in the source code around those comments. Our compiler pass further harvests token AST tags for individual tokens in the source code snippets. In-line comments are often pure English; however, documentation strings, before the snippets with which they are associated, contain references to code tokens in the snippet. We further process the output of the plugin offline where we parse doc-strings to decompose intra-sentential mixed text and add part-of-speech tags to the pure English text. Thus, by construction, we have both tag and language ID ground-truth data. We allow matches up to 3 edits away to account for misspellings that may exist in doc-strings. The former ground-truth is obtained from CLANG during the compilation of the projects, while English comments are tokenised and labelled using NLTK as above. For code tokens in comments, we override their language ID and tag using information about them from the snippet associated with the comment.

A consequence of using CLANG to source our AST tags is that we are limited to the mixed text tagging problem with a single natural language ($k = 1$) and a single formal language ($l = 1$). This limitation is a property of the data and not the model.

5.5.2 Predicting Tags

Here, we explore POSIT's accuracy on the language identification and PoS/AST tagging subtasks of the mixed text tagging problem. We adapt StORMeD for use as our baseline and compare its performance against that of POSIT. We note, even after adaption for the AST tagging task for which it was not designed, StORMeD is the existing state-of-the-art. We close by reporting POSIT's performance on non-Java posts.

To compare with the existing tool StORMeD, we restrict our Stack Overflow corpus to Java posts, because Ponzanelli *et al.* designed StORMeD to handle Java, JSON, and XML. Further, StORMeD and POSIT do not solve the same problems. StORMeD parses mixed posts into HAST trees; POSIT tags sequences. Thus, we flatten StORMeD's HASTs and use the AST label of the parent of terminal nodes as the tag. Because StORMeD builds HASTs for Java, JSON or XML while POSIT uses CLANG to tag code tokens, we built a map from StORMeD's AST

tag set to ours[3]. As this mapping may be imperfect, StORMeD's observed performance on the PoS/AST tagging task is a lowerbound (Section 5.7.2).

For the language identification task, StORMeD exposes a 'tagger' webservice. Given mixed text in HTML, it replies with a string that has </code> HTML-tags added. We parse this reply to obtain our token-level language tags as in Section 5.5.1. For PoS/AST tagging, StORMeD exposes a 'parse' webservice. Given a Stack Overflow post with with correctly labelled code in HTML (Section 5.5.1), this service generates HASTs. We flatten and translate these HASTs as described above. To use these services, we break our evaluation corpus up into 2000 calls to StORMeD's webservices, 1000 for the language identification task, and the other 1000 for HAST generation. This allows us to comply with its terms of service.

Language Tagging Here, we compare how well StORMeD and POSIT segment English and code in the Java Stack Overflow corpus. Unlike StORMeD's original setting, we elide user-provided code token labels, both from StORMeD and POSIT to avoid data leakage. Predicting them is the very task we are measuring. The authors of StORMeD account for this scenario [77, §II.A]. Although StORMeD must initially treat the input as a text fragment node, StORMeD still runs an island grammar parser to find code snippets embedded within it. Despite being asked to perform on a task for which it was not designed, due to the elision of user-provided code labels, StORMeD performs very well on our evaluation set and, indeed, as pioneering, post-regex work, defined the previous state of the art on this task. In this setting, StORMeD obtains 71% accuracy, POSIT achieves 81.6%.

PoS/AST Tagging Here, we use StORMeD as a baseline to benchmark POSIT's performance on predicting PoS/AST tags for each token. Granted, on the text fragment nodes, we are actually measuring the performance of the NLTK PoS tagger. Unlike the first task, we allow StORMeD to use user-provided code-labels for this subtask. POSIT, however, solves the two subtasks jointly, so giving it these labels as input remains a data leak. Therefore, we do not provide them to POSIT. After flattening and mapping HAST labels to our label universe, as described above, StORMeD achieves a more than respectable accuracy of 61.9%, while POSIT achieves 85.6%.

On a uniform sample set of 30 posts from queries to StORMeD, we observed StORMeD to struggle with single word tokens or other short code snippets embedded within a sentence, especially when these, like foo, bar, do not match peculiar-to-code naming conventions. While this is also a more difficult task for POSIT as well, it fares better. Consider the sentence 'Class A has a one-to-many relationship to B. Hence, A has an attribute collectionOfB.'. Here, StORMeD spots Class A and collectionOfB, the uses of A and B as stand-alone tokens slips passed the heuristics. POSIT manages to spot all four code tokens. POSIT's use of word embeddings, allows it to learn typical one word variable names and find unmarked code tokens that escape

[3]The mapping can be found online at https://github.com/PPPI/POSIT/blob/92ef801e5183e3f304da423ad50f58fdd7369090/src/baseline/StORMeD/stormed_evaluate.py#L33.

Table 5.2: The result on the evaluation set for the different ablation configurations. All configurations use the token embedding as it is our core embedding. Observe that using only the CRF layer performs best on the language identification and the POSIT's tagging tasks.

High-level features	LID Acc	Tag Acc	Mean Acc	Epoch (hh:mm)
Only token embeddings	0.209	0.923	0.568	00:37
Only CRF	**0.970**	**0.926**	**0.948**	**01:03**
Only feature vector	0.450	0.928	0.689	00:45
Only char embeddings	0.312	0.923	0.617	02:32
No CRF	0.409	0.913	0.661	02:59
No feature vector	**0.966**	**0.924**	**0.945**	**03:19**
No char embeddings	0.966	0.919	0.943	01:39
All features	0.970	0.917	0.944	03:30

StORMeD's heuristics, such as all lowercase function names that are defined in a larger snippet within the post. For its part, StORMeD handled documentation strings well, identifying when code tokens are referenced within them. POSIT preferred to treat the doc-string as being fully in a natural language, missing code references that existed within them even when they contained special mark-up, such as @.

Beyond Java, JSON, and XML POSIT is not restricted to Java, so we report its performance on the entire Stack Overflow corpus and on the CodeComment corpus. The former measures the performance on mixed text which has English as a frame language; the latter measures the performance on mixed text with source code as the frame language. On the complete Stack Overflow corpus, POSIT achieves an accuracy of 97.7% when asked to identify the language of the token and an accuracy of 93.8% when predicting PoS/AST tags. We calculated the first accuracy against user-provided code tags and the second against our constructed tags (Section 5.5.1). On the CodeComment corpus, we tweak POSIT's training. As examples within this corpus tend to be longer, we reduce the number of micro-batches to 16. After training on CodeComment, POSIT achieves an accuracy of 99.7% for language identification and an accuracy of 98.9% for PoS/AST tag predictions.

5.5.3 Model Ablation

POSIT depends on three kinds of embeddings — character, token, and token morphology — and CRF layer prior to decoding (Equation (5.1)). We can ablate all except the token embeddings, our bedrock embedding. We used the same experimental set-up described at the beginning of this section, with one exception: When ablating the CRF layer, we replaced it with a 2-Layer Perceptron whose output we then apply softmax to.

Table 5.2 shows the results. Keeping only the CRF-layer reduces the time per epoch from 3:30 hours to 1:03 hours (the first bolded row). On average, POSIT's model requires 6 to 7 epochs until it stops improving on the development set, so we stop. This configuration reduces training time by $\sim$14 hours. Further, it slightly increases performance. Only using the CRF, however, manual spot-checking reveals that POSIT incorrectly assigns token that obey

common coding conventions and method call tokens as English. This is due to English to code class imbalance, and inspecting Table 5.1 makes this clear. The best performing model under human assessment of uniformly sampled token (the second bolded row) removes only the token morphology feature vector. Essentially, this model drops precisely those heuristics that we anecdotally know humans use when performing this task. Since dropping either the character or the token morphology embeddings yields almost identical performance, we hypothesise that POSIT learns these human heuristics, and perhaps others, in the character embeddings. We choose to keep character embeddings, despite training cost, for this reason.

5.6 POSIT Applied

POSIT can improve downstream tasks. First, we show how POSIT accurately suggests code tags to separate code from natural language, such as Stack Overflow's backticks. POSIT achieves 95% balanced accuracy on this task. Developers could use these accurate suggestions to improve their posts before submitting them; researchers could use them to preprocess and clean data before using it in downstream applications. For instance, Yin *et al.* [124] start from a parallel corpus of Natural Language and Snippet pairs and seek to align it. POSIT could help them extend their initial corpus beyond StackOverflow by segmenting mixed text into pairs. Second, we show how POSIT's language identification and PoS tagging predictions enable TaskNav — a tool that supports task-based navigation of software documentation by automatically extracting task phrases from a documentation corpus and by surfacing these task phrases in an interactive auto-complete interface — to extract new and more detailed tasks. We conduct these demonstrations using POSIT's best performing configuration, which ignores token morphology (Section 5.5.3).

5.6.1 Predicting Code Tags

Modern developer fora, notably Stack Overflow, provide tags for separating code and NL text. These tags are an unusual form of punctuation, so it is, perhaps, not surprising that developers often neglect to add them. Whatever the reason, these tags are often missing [61]. POSIT can help improve post quality by serving as the basis of a post linter that suggests code tags. A developer could use such a linter before submitting their post or the server could use this linter to reject posts.

Our Stack Overflow corpus contains posts that have been edited solely to add missing code tags. To show POSIT accuracy at suggesting missing code tags, we extracted these posts using the SOTorrent dataset [125]. First, we selected all posts that contain a revision with the message "code formatting". We uniformly, and without replacement, sampled this set for 30 candidates. We kept only those posts that made whitespace edits and introduced single or triple backticks. By construction, this corpus has a user-defined ground truth for code tags. We use the post before

the revision as input and compare against the post after the revision to validate. POSIT manages to achieve a balanced accuracy of 95% on the code label prediction task on this corpus.

5.6.2 TaskNav++

To demonstrate the usefulness of POSIT's code-aware part-of-speech tagging, we augment Treude *et al.*'s TaskNav [13] to use POSIT's language identification and its PoS/AST tags.

To construct TaskNav++, we replaced TaskNav's Stanford NLP PoS tagger with POSIT. Like TaskNav, TaskNav++ maps AST tags to "NN". TaskNav uses the Penn Treebank [126] tag set; POSIT uses training data labelled with Universal tag set tags [127]. These tags sets differ; notably, the Penn Treebank tags are more granular. To expose POSIT's tags to TaskNav's rules to use those rules in TaskNav++, we converted our tags to the Penn Treebank tag set. This conversion harms TaskNav++'s performance, because it uses the Java Stanford Standard NLP library which expects more granular tags, although it can handle the coarser tags POSIT gives it.

To compare TaskNav and TaskNav++, we asked both systems to extract tasks from the same Linux Kernel Mailing List corpus that we manually analysed (Section 5.7.1). TaskNav++ finds 97 new tasks in the 30 threads or 3.2 new tasks per thread. Of these, 65 (67.0%) are reasonable tasks for the e-mail they were extracted from. Two of the authors performed the labelling of these tasks, we achieved a Cohen Kappa of 0.21, indicating fair agreement. Treude *et al.* also report low agreement regarding what is a relevant task [13]. To resolve disagreements, we consider a task reasonable if either author labelled it as such. The ratio of reasonable tasks is in the same range as that reported in the TaskNav work, *viz.* 71% of the tasks TaskNav extracted from two documentation corpora were considered meaningful by at least one of two developers of the respective systems. TaskNav prioritises recall over precision to enable developers to use the extracted tasks as navigation cues. POSIT's ability to identify more than two additional reasonable tasks per email thread contributes towards this goal.

Inspecting the tasks extracted, we find that some tasks benefit from POSIT's tokenisation. For example in 'remove excessive untagging in gup' vs 'remove excessive untagging in gup.c' the standard tokeniser assumed that the use of '.' in 'gup.c' indicates the end of a sentence. Our tokenisation also helps correctly preserve mention uses of code tokens: 'pass in "real + len"' and 'pass in "dest + len"', and even English-only mention uses: 'call writeback bits "tags"', 'split trampoline buffer into "temporary" destination buffer'. In all these cases, either TaskNav finds an incorrect version of the task ('add len to real rip') or simply loses the double-quotes indicating a mention use (for the English-only mention cases).

POSIT's restriction to a single formal language proved to be a double-edged sword. It helped separate patches that are in-lined with e-mails in our manual analysis of POSIT on the LKML (Section 5.7.1), while here we can see that it is problematic. By training only on a single programming language, POSIT misidentifies change-log and file-path lines as code. This

propagates to TaskNav++, which in turn incorrectly adds these as tasks since POSIT stashes the path or change-log into a single code element. At times, this behaviour was also beneficial, such as annotating the code in the task: ‘read <tt>extent [i]</tt>’, this comes at the cost of generating incorrect tasks such as: ‘change android <tt>/ ion / ion.c | 60 +++ +++ +++ +++ +++ +++ +++ +</tt>’. We hypothesise that a solution to the general mixed text tagging problem would avoid this problem by explicitly training to identify file paths.

5.7 Discussion

In this section, we first perform a deep dive into POSIT’s output and performance. Then we address threats to POSIT’s model, its training, and methodology.

5.7.1 POSIT Deep Dive

POSIT is unlikely to be the last tool to tackle the mixed text tagging problem. To better understand what POSIT does well and where it can be improved, we manually assessed its output on two corpora: a random uniform sample of 10 Stack Overflow posts from our evaluation set and a random uniform sample of 10 e-mails from the Linux Kernel Mailing List sent during August 2018. The Stack Overflow sample contains 3,233 tokens while the LKML — 17,451. We finish by showing POSIT’s output on a small Stack Overflow post. Broadly, POSIT’s failures are largely due to tokenisation problems, class imbalance, and lack of labels. Concerning the label problem, our data actually consists of a single natural language and several formal languages, one of which is a programming language, the others include diffs, URLs, mail headers, and file paths. This negatively impacted TaskNav++ by exposing diff headers and file paths as code elements, inducing incorrect tasks to be extracted. Our deep dive also revealed that POSIT accurately PoS-tags English, accurately AST-tags lone code tokens, and learned to identify diffs as formal, despite lack of labels.

To pre-process training data, POSIT uses two tokenisers: the standard NLTK tokeniser and a Python port of the CLANG lexer. POSIT uses labels (Stack Overflow’s code tags) in the training to switch between them. In the data, we observed that POSIT had tagged some double-quotation marks as Nouns. Since the user-provided code labels are noisy [61], we hypothesise the application of the code tokeniser to English caused this misprediction. Designed to dispense with code-labels, POSIT exclusively relies on the CLANG lexer port during evaluation. Unsurprisingly, then, we observed POSIT incorrectly tagging punctuation as code-unknown as multiple punctuation tokens are grouped into single tokens that do not normally exist in English. We suspect this to due to applying English tokenisation to code snippets. Clearly, POSIT would benefit from tokenisation tailored for mixed text.

Within code segments, we also observed that POSIT had a proclivity to tag tokens as ‘raw_identifier’. This indicates that context did not always propagate the ‘method_name’, or

'variable' tags across sentence boundaries. As the 'raw_identifier' tag was the go-to AST label for code, it suggests a class imbalance in our training data with regards to this label. Indeed, we observed POSIT to only tag a token as 'method_name' if it was followed by tokens that signify calling syntax — argument lists, including the empty argument list ().

This deep dive revealed a double-edged sword. Our sample contained snippets that represent diffs, URLs or file paths. POSIT's training data does not labels these formal languages nor did tokenisation always preserve URLs and file paths. Nonetheless, POSIT managed to correctly segment diffs by marking them as code, performing this task exceptionally well on the LKML sample. URLs and file paths were seen as English unless the resource names matched a naming convention for code. For URLs, POSIT tagged key-argument pairs (post_id=42) as ('variable', 'operation', 'raw_identifier'). Later in Section 5.6.2, POSIT's tendency to segment diffs as code was detrimental, since it stashed diff headers into a single code token, causing TaskNav++ to produce incorrect tasks.

An additional observation during our manual investigation is the incorrect type of tag relative to the language of the token. Consider the following:

	English	Code
PoS output	33.4%	1.5%
AST output	5.6%	59.5%

We obtain these numbers by considering the agreement between the language identification task and the type of tag output for the Stack Overflow posts and LKML mails used in this deep dive. We can see that for 7.1% of the tokens (908) in our manual investigation POSIT outputs the wrong type of label given the language prediction. This was also observed by the authors for cases where one of the predictions was wrong while the other was correct, such as tagging a Noun as such while marking it as code. This is because we separated the two tasks after the biLSTM and trained them independently. We hypothesise that adding an additional loss term that penalises desynchronising these tasks would solve this problem. Alternatively, one could consider a more hierarchical approach, for example first predicting the language id, then predicating the tag output conditioned on this language id prediction.

For monolingual sentences, either English or code, POSIT correctly PoS- or AST-tagged the sequences. Spare the occasional hiccup at switching from English to code a single token too late, POSIT correctly detected the larger contiguous snippets. As code snippets ended, POSIT was almost always immediate to switch back to identifying tokens as English. For smaller embedded code snippets, POSIT correctly identified almost all method calls that were followed by argument lists, including '()'. POSIT almost always correctly identified operators and keywords even when used on their own in a mention role in the host language. Further, single token mentions of typical example function names, like foo or bar, code elements that followed naming conventions, or

There are two ways of fixing the problem. The first is to use a comma to sequence statements within the macro without robbing it of its ability to act like an expression.

```
#define BAR(X) f(X), g(X)
```

The above version of bar `BAR` expands the above code into what follows, which is syntactically correct.

```
if (corge)
    f(corge), g(corge);
else
    gralt();
```

This does not work if instead of `f(X)` you have a more complicated body of code that needs to go in its own block, say for example to declare local variables. In the most general case the solution is to use something like `do ... while` to cause the macro to be a single statement that takes a semicolon without confusion.

Figure 5.5: Example sentence taken from Stack Overflow which freely mixes English and very short code snippets, here rendered using monospaced font. We can see both inter-sentential code-switching, such as the macro definition and the short example if statement snippet, as well as intra-sentential code-switching, the mention of the code token `f(X)` and the code construct '`do ... while`'.

code tokens that were used in larger snippets within the same post were correctly identified as code.

In Figure 5.5, we observe 91% tag accuracy for English and 66.7% tag accuracy for code. The language segmentation is 76.7% accurate. POSIT correctly identifies the two larger code snippets as code except for the first token in each: '#define' and 'if'. It fails to spot do ... while as code, perhaps due to do and while being used within English sufficiently often to obscure the mention-role of the construct. On the other hand, it correctly spots f(X) as code since f and X are rarely used on their own in English.

5.7.2 Threats to Validity

The external threats to POSIT's validity relate mainly to the corpora, including the noisy nature of StackOverflow data [128], and the potential of the model to overfit. POSIT generalises to the extent to which its training data is representative. To avoid overfitting, we use a development set and an early stopping criterion (three epochs without improvement), as is conventional.

In Section 5.6.1, we show that despite the noisy training labels, POSIT is capable of predicating code-tags/spans that users originally forgot to provide. We also explore POSIT's performance on a corpus that is likely to differ from both training corpora, the Linux Kernel Mailing List (LKML) which was used during the deep dive (Section 5.7.1). This validation was performed manually due to lack of a ground truth; automatically generating a ground truth for this data would not escape the internal threats presented below. On this corpus, POSIT achieves a language identification accuracy of 76.6% and a PoS/AST tagging accuracy of 76.5%. Two of the authors

have performed the labelling of this task and the Cohen kappa agreement [129] for the manual classification is 0.783, which indicates substantial agreement. We resolved disagreements by considering an output correct if both authors labelled it as such.

Neural networks are a form of supervised learning and require labels. We labelled our training in two ways, using one procedure for language labels and another for PoS/AST tags. Both procedures are subject to a threat to their construct validity. The language labels are user-provided and thus subject to noise, PoS tags are derived from an imperfect PoS tagger, and AST tags are added heuristically. For language labels, we both manually labelled data and exploited a human oracle. We manually labelled a uniformly sampled subset of 10 posts with 3,233 tokens from our Stack Overflow evaluation data, then manually assessed POSIT's performance on this subset. Two authors performed the manual labelling and achieved a Cohen kappa of 0.711 indicating substantial agreement. A similar procedure was applied to the LKML labelling task. On this validation, POSIT achieved 93.8 accuracy. Our Stack Overflow corpus contains revision histories. We searched this history for versions whose edit comment is "code formatting". We then manually filtered the resulting versions to those that only add code token labels (defined in Section 5.6.1). POSIT achieved 95% balanced accuracy on this validation. For the PoS/AST tagging task, we manually added the PoS/AST tags on the same 10 Stack Overflow posts, we used above. Here, POSIT achieved 93.7%.

In Section 5.5.2, we used StORMeD as a baseline for POSIT. As previously discussed, StORMeD was not designed for our task and handles Java, JSON, and XML. Adapting to our setting introduces an internal threat. To address this threat, we evaluated both StORMeD and POSIT only on those Stack Overflow posts tagged as Java. These tags are noisy [130, 128]. When evaluating StORMeD, its authors, Ponzanelli *et al.* used the same filter. We also map the StORMeD AST tag set to ours[4]. If the true mapping is a relation, not a function, then this would understate StORMeD's performance. This is unlikely because Java ASTs and CLANG ASTs are not that dissimilar. Further, POSIT must also contend with this noise. When building TaskNav++ (Section 5.6.2), we use a more coarse grained PoS tag set than the original TaskNav potentially reducing its performance.

5.8 Related Work

In software engineering research, part-of-speech tagging has been directly applied for identifier naming [62], code summarisation [63, 64], concept localisation [65], traceability-link recovery [66], and bug fixing [67]. We first review natural language processing (NLP) research on code-switching, the natural language analogue of the mixed text problem. This is work on which we based POSIT. Then we discuss initial efforts to establish analogues for parts of speech

[4]The mapping can be found online at https://github.com/PPPI/POSIT/blob/92ef801e5183e3f304da423ad50f58fdd7369090/src/baseline/StORMeD/stormed_evaluate.py#L33.

categories for code and use them to tag code tokens. We close with the pioneering work on StORMeD, the first context-free work to automatically tackle the mixed text tagging problem.

NLP researchers are growing more interested in code-switching text and speech[5]. The main roadblock had been the lack of high-quality labelled corpora. Previously, such data was scarce because code-switching was stigmatised [72]. The advent of social media, has reduced the stigma and provided code-switching data, especially text that mixes English with another language [71]. High quality datasets of code-switched utterances are now under production [70]. For the task of part-of-speech (PoS) tagging code-switching text, Solorio and Liu [73] presented the first statistical approach to the task of part-of-speech (PoS) tagging code-switching text. On a Spanglish corpus, they heuristically combine PoS taggers trained on larger monolingual corpora and obtain 85% accuracy. Jamatia *et al.* [74], working on an English-Hindi corpus gathered from Facebook and Twitter, recreated Solorio's and Liu's tagger and they proposed a tagger using Conditional Random Fields. The former performed better at 72% vs 71.6%. In 2018, Soto and Hirschberg [75] proposed a neural network approach, opting to solve two related problems simultaneously: part-of-speech tagging and Language ID tagging. They combined a biLSTM with a CRF network at both outputs and fused the two learning targets by simply summing the respective losses. This network achieves a test accuracy of 90.25% on the inter-sentential code-switched dataset from Miami Bangor [70]. POSIT builds upon their model extended with Winata *et al.*'s [117] handling of OOV tokens, as discussed in Section 5.4.

Operating directly on source code (not mixed text), Newman *et al.* [69] sought to discover categories for source code identifiers analogous to PoS tags. Specifically, they looked for source code equivalents to Proper Nouns, Nouns, Pronouns, Adjectives, and Verbs. They derive their categories from 1) Abstract syntax trees, 2) how the tokens impact memory, 3) where they are declared, and 4) what type they have. They report the prevalence of these categories in source-code. Their goal was to map these code categories to PoS tags, thereby building a bridge for applying NLP techniques to code for tasks such as program comprehension. Treude *et al.* [11] described the challenges of analysing software documentation written in Portuguese which commonly mixes two natural languages (Portuguese and English) as well as code. They suggested the introduction of a new part-of-speech tag called Lexical Item to capture cases where the "correct" tag cannot be determined easily due to language switching.

Ponzanelli *et al.* are the first to go beyond using regular expressions to parse mixed text. When customising LexRank [76], a summarisation tool for mixed text, they employed an island grammar that parses Java and stack-trace islands embedded in natural language, which is relegated to water. They followed up LexRank with StORMeD, a tool that uses an island grammar to parse Java, JSON, and XML islands in mixed text Stack Overflow posts, again relegating natural language to water [77]. StORMeD produces heterogeneous abstract syntax trees (AST),

[5]The NLP term for text and speech that mixed multiple natural languages.

which are ASTs decorated with natural language snippets.

StORMeD relies on Stack Overflow's code tags; when these tags are present, island grammars are a natural choice for parsing mixed text. Mixed text is noisy and Stack Overflow posts are no exception [61]. To handle this noise, StORMeD resorts to heuristics (anti-patterns in the nomenclature of island grammars), which they build into their island grammar's recognition of islands. For instance, if whitespace separates a method identifier from its '(', they toss that method identifier into water. To identify class names that appear in isolation, they use three heuristics: the name is a class if it is a fully qualified name with no internal spaces, contains two instances of CamelCase, or syntactically matches a Java generic type annotation over builtins. They use similar rules to handle Java annotation because Stack Overflow also uses '@' to mention users in posts. Heuristics, by definition, do solve a problem in general. For example, the generic method names often used in examples — foo, bar, or buzz — slip past their heuristics when appearing alone in the host language. This is true even when the post defines the method. Indeed, we show that no island grammar, which, by definition, extend a context-free grammar, can solve this Sisyphean task for mixed text, because we show this task to be context-sensitive in Section 5.3. Island grammar's anti-patterns do not make island grammars context-sensitive.

StORMeD and POSIT solve related but different mixed text problems. StORMeD recovers natural language, unprocessed, from the water, builds ASTs for its islands, then decorates those ASTs with natural language snippets to build its HAST. In contrast, POSIT tags both natural languages and formal languages, but does not build trees. StORMeD and POSIT do overlap on two subtasks of mixed text tagging: language identification and AST-tagging code. To compare them on these tasks, we had to adapt StORMeD. Essentially, we traverse the HASTs and consider the first parent of a terminal node to be the AST tag. We map these from the StORMeD tag set to ours (Section 5.5.2). POSIT advances the state of the art on these two tasks (Section 5.5.2).

As a notable service to the community, Ponzanelli *et al.* both provided a corpus of HASTs and StORMeD as an excellent, well-maintained web service. Their HAST corpus is a structured dataset that allows researchers a quick start on mining mixed text: it spares them from tedious pre-processing and permits the quick extraction and processing of code-snippets, for tasks like summarisation. We have published our corpus at `https://github.com/PPPI/POSIT` to complement theirs. Our project, which culminated in POSIT, would not have been possible without these contributions.

5.9 Conclusion

We have defined the problem of tagging mixed text. We present POSIT, implemented using a biLSTM-CRF Neural Network and compared it to Ponzanelli *et al.*'s pioneering work, StORMeD [77] on Java posts in Stack Overflow. We show that POSIT accurately identifies English and code tokens (81.6%), then accurately tags those tokens with their part-of-speech

tag for English or their AST tag for code (85.6%). We show that POSIT can help developers by improving two downstream tasks: suggesting missing code labels in mixed text (with 95% accuracy) and extracting tasks from mixed text through TaskNav++, which exploits POSIT's output to find more than two new reasonable tasks per document.

POSIT and our CodeComment corpus are available at https://github.com/PPPI/POSIT.

Acknowledgements — We thank Ponzanelli *et al.* for developing and maintaining StORMeD, a powerful and easy-to-use tool, and for their prompt technical assistance with the StORMeD webservice. This research is supported by the EPSRC Ref. EP/J017515/1.

6

Conclusions

As projects continue, they accumulate bit-rot, technical debt, and generally suffer an attrition to their health. Maintaining them in shape is indeed an active process, oft forgone for implementing new features. One can hardly blame the developer when the cost is typically a considerable time investment and the reward is considerably delayed. To fix the situation, we need to fix either the cost or the incentives. In this thesis, I consider reducing the cost of maintaining project health. I present three tools and approaches that can aid developers, either directly or via tool smiths, actively maintain project health. I present Aide-mémoire, a tool for PR-Issue link prediction, Flexeme, a commit untangling tool, and POSIT, a mixed-text segmentation and tagging tool.

6.1 Summary of Contributions

In Aide-mémoire, we proposed the first online Pull-request-Issue linker. It aims to help developers with suggestions when they are least likely to require a context-switch, during Issue triaging and PR submission. This is key to obtaining developer support, as the context-switch cost may make or break tools that offer suggestions [85].

Flexeme untangles version histories for developers as they make history. When committing a local branch to upstream, a developer is likely to still have the context of their work in mind. This enables them to assess the correctness of the suggestion with minimal cost. Further, atomic histories can benefit from a wider range of tools: semantic version history slicing is expected to suffer from less noise, while also making git bisect and other delta debugging approaches over commits more attractive.

Through POSIT, we hope to have opened a new line of research. We formulated the mixed-text tagging problem and demonstrated the benefits of considering both natural and formal language channels when considering sources that mix both. Although POSIT is closer to a meta-tool, it assists tool smiths make their tools aware to the two channels intertwined in a

mixed-text, POSIT can directly help user of software fora such as Stack Overflow by detecting missed code annotations.

Overall, this thesis demonstrates how the developer process can be augmented to benefit both researchers as well as practitioners: a healthy project holds promises for both. The areas in which we improve the state-of-the-art all serve to provide cleaner development and project histories, or improve the tool accessibility to developer communication over mixed-channels.

6.2 Summary of Future Work

The work presented in this thesis identifies directions for further research. I discuss some of these directions in this section.

Mixed-text comprehension POSIT is a step along the path to mixed-text comprehension. It offers the first building blocks, an equivalent for Part of Speech tags for mixed-text. More sophisticated models can build upon this to aid comprehension and parsing of mixed text such that meaning is carried over from the Natural Language channel to the algorithmic channel. Further, POSIT solves a reduced case of the mixed-text parsing problem. Indeed, it assumes a single formal and a single natural language.

Graph Neural Network Approaches for Graph Similarity In Flexeme, we make use of graph kernels for graph similarity computation; however, recent research in Graph Neural Networks show promising results, and future work may focus on replacing our graph kernels for a neural network implementation, if a suitable training corpus is available.

Multi-version PDG Static Analysis As we use the δ-PDG for pairs of versions, we do not exploit the structure to its full potential. Static analysis that relies on PDGs could be made version aware such that the analysis can be carried out over multiple versions by simply considering the δ-PDG.

Online Traceability In Aide-mémoire, we considered online traceability of PR-Issue links. This idea could be extended to other forms of traceability, for example, the commit-issue linking problem from which we started. While the research area of Just-In-Time Traceability indeed exists and is active [131, 132, 133, 134], it often uses offline Information Retrieval techniques thus making the developer feedback delayed. An online approach could serve developers better by being able to adapt faster to changing requirements.

Appendix A

Graph Kernels

In this Appendix, I will first define a kernel and a graph kernel. I will then show some of the earlier examples of kernels: from vertex/edge label histogram kernels through to random-walk and graphlet kernels. I will then detail the general Weisfeiler-Lehman graph kernel framework and show how to instantiate it with a subtree graph kernel. Kernels in general, and graph kernels being no exception, are used to allow machine learning approaches, such as clustering, that make use of similarity or (pseudo-)metrics to learn. We use the Weisfeiler-Lehman subtree graph kernel in Chapter 4 to allow re-clustering δ-NFG graphs into atomic patches.

A kernel is a symmetric function $k : X \times X \to \mathbb{R}$ on non-empty sets X. A kernel is positive semi-definite if:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j k(x_i, x_j) \geq 0,$$

$$\forall n \in \mathbb{N}, x_1, \ldots, x_n \in X, c_1, \ldots, c_n \in \mathbb{R}.$$

The function k, for any parametrization by real constants, has a non-negative weighted sum over all inputs. A *graph kernel* is a positive semi-definite kernel on a set of graphs. As we use kernels as a measure of similarity, we are interested in those kernels that have an equivalent inner product definition. As inner products are positive semi-definite, we are only interested in positive semi-definite kernels.

Now, let us consider a collection of graphs $\mathcal{G}$, where each graph $G \in \mathcal{G}$ has vertexes from an arbitrary vertex set $\mathcal{V}$. Further, let $\mathcal{L}_v$ be the set of node labels, and let $lv : \mathcal{V} \to \mathcal{L}_v$ be the graph labelling function. Similarly, for edges, let $\mathcal{L}_e$ be the set of edge labels, with a similarly defined le.

We will now consider a kernel to be a function $k : \mathcal{G} \to \mathbb{R}^{|\mathcal{G}|} \times \mathbb{R}^{|\mathcal{G}|}$. In particular:

$$k(\mathcal{G})_{ij} = \langle \psi(G_i), \psi(G_j) \rangle, \tag{A.1}$$

where ψ embeds a graph into a vector space and $\langle \cdot, \cdot \rangle$ represents the inner product. By doing so, we avoid needing to compute an explicit feature space, which, depending on our set X, can be costly. Instead, we can operate on the images of our data points, here graphs, in a vector space. This is often referred to as a kernel trick [135]. This trick may not be possible for kernels that are not positive semi-definite.

A.1 Vertex and Edge Label Histogram Kernels

To build your intuition on how to construct a graph kernel, we start with two simple, but useful, examples that aggregate a graph's labels to build a histogram summary of a graph: vertex/edge label histogram kernels. For the vertex label histogram kernel, let $n = |\mathcal{L}_v|$, and let ψ return length n vectors $\mathbf{f}$, such that the i-th value represents the count of the i-th vertex label in the graph, *i.e.*:

$$\psi(G) = \mathbf{f}, \tag{A.2}$$

$$\mathbf{f}_i = |\{v \mid v \in V, lv(v) = \mathcal{L}_{vi}\}|. \tag{A.3}$$

The matrix generated by our kernel function is then populated by the inner product of these vertex label count vectors. We similarly define such a kernel for edge label histograms by substituting $\mathcal{L}_e$ for $\mathcal{L}_v$ above, and le for lv. Let the count vectors generated by edge histograms be $\mathbf{g}$.

We can define a third kernel simply by considering:

$$k(\mathcal{G})_{ij} = \langle [\mathbf{f}, \mathbf{g}]_i, [\mathbf{f}, \mathbf{g}]_j \rangle, \tag{A.4}$$

where $[\mathbf{f}, \mathbf{g}]$ represents vector concatenation. That is, we consider the inner product of both vertex and edge label histograms when computing the inner product. This defines the vertex-edge label histogram kernel.

For each of the three presented kernels, vertex label histogram, edge label histogram, and vertex-edge label histogram, we can obtain the Gaussian radial basis function (RBF) [136] version of the kernel as follows. Let $\mathbf{h} = \mathbf{f}$, $\mathbf{h} = \mathbf{g}$, or $\mathbf{h} = [\mathbf{f}, \mathbf{g}]$ as appropriate for the respective kernel. The Gaussian version of the kernels then becomes:

$$k(\mathcal{G})_{ij} = \exp\left(-\frac{\|\mathbf{h}_i - \mathbf{h}_j\|}{2\sigma^2}\right), \tag{A.5}$$

where σ^2 is the variance of our Gaussian RBF kernel.

If we were to apply these kernels to our commit untangling task in Chapter 4, we would make use of only the immediate neighbourhoods of changed nodes. We would lose considerable topological information: that of multi-step neighbours.

A.2 Random-Walk Kernel

Let us now consider a more complicated kernel that better preserves topological information: random-walk based kernels. For the κ-step random walk kernel, we consider as input a vector of weights $(\lambda_0, \lambda_1, \ldots, \lambda_\kappa)$ where each $\lambda_i \in \mathbb{R}^+$ and define the kernel as:

$$k(\mathcal{G})_{ij} = \sum_{i',j'=1}^{|V_\times|} \left[\sum_{t=0}^{\kappa} \lambda_t A_\times^t \right]_{i'j'}, \tag{A.6}$$

where $A_\times$ represents the adjacency matrix of the direct tensor product $G_\times = (V_\times, E_\times, \mathsf{lv}_\times, \mathsf{le}_\times)$ of $G_i = (V_i, E_i, \mathsf{lv}_i, \mathsf{le}_i)$ and $G_j = (V_j, E_j, \mathsf{lv}_j, \mathsf{le}_j)$ *s.t.*:

$$V_\times = \{(v_i, v_j) \in V_i \times V_j \mid \mathsf{lv}_i(v_i) = \mathsf{lv}_j(v_j)\},$$

$$E_\times = \{((u_i, u_j), (v_i, v_j)) \in V_\times \times V_\times \mid \mathsf{le}_i((u_i, v_i)) = \mathsf{le}_j((u_j, v_j))\},$$

and labels are propagated *i.e.*:

$$\mathsf{lv}_\times((v_i, v_j)) = \mathsf{lv}_i(v_i),$$

$$\mathsf{le}_\times(((u_i, u_j), (v_i, v_j))) = \mathsf{le}_i((u_i, v_i)).$$

Random-walk based kernels, at an intuitive level, find sub-graphs that are shared between the graphs whose similarity we wish to computing. Instead of obtaining these sub-graphs through random walking, we can consider a collection of sub-graph patterns: graphlets. For the graphlet kernel, for each graph G_i consider the vector $\mathbf{l_i} = (l_i^1, l_i^2, \ldots, l_i^s)$ such that l_i^k records the number of embeddings of graphlet L_k into G_i. Each $L_k, \forall k \in [1..s]$ is a graph with n nodes. We say that $L = (V_L, E_L)$ can be embedded into G if there exists an injective $\alpha : V_L \to V$ *s.t.* $(v,w) \in E_L \iff (\alpha(v), \alpha(w)) \in E$. Similar to previous kernel formulations, we define $\psi(G_i) = \mathbf{l_i}$ and thus our kernel is:

$$k(\mathcal{G})_{ij} = \langle \mathbf{l_i}, \mathbf{l_j} \rangle \tag{A.7}$$

Due to high computational costs, implementations of this kernel ignore Vertex and Edge labels. For more details, see Shervashidze *et al.* [137]. If applied to Flexeme, this would discard code snippet and other valuable information that can help solve the untangling problem. Indeed, using only topological information from data- and controlflows, it is difficult to identify concerns in a commit.

A.3 Weisfeiler-Lehman Kernel

Let us now first consider the Weisfeiler-Lehman (WL) kernel framework [104], which arose from a graph isomorphism test [107]. The kernel equivalent to the isomorphism test was first introduced in the Biomedical community as a method of computing the similarity between graph structures such as proteins as a proxy for determining the properties of new protein structures [138]. When combined with a subtree kernel, this allows us to consider graphs that can have similar walks through the graph given arbitrary starts within them; each node is treated as a root of a tree while the trees themselves record walks of bounded lengths. Two graphs are more similar the larger the intersection of their constituent node trees when considering both topology and labels. In Flexeme's setting, this means that the two graphs are similar if they have similar statements following each other: a proxy for similar executions.

For the WL kernel, consider the following: the labelling function lv associated with a graph $G = (V,E)$ becomes the labelling function lv_0, the labelling function of 0 steps of Weisfeiler-Lehman (the base case). To perform a step, we replace the current label of a node v with the ordered multi-set that contains the current label unioned with the sorted multi-set of labels of the neighbours of v. This multi-set of labels is then compressed to a short label. We obtain lv_{i+1} from lv_i as follows:

$$lv_{i+1}(v) = \text{compress}(\{lv_i(v)\} \cup \{lv_i(v') \mid v' \in N(v)\}) \tag{A.8}$$

This process is repeated for h iterations. In particular, the compress function must ensure that nodes obtain identical new labels if and only if they have identical multi-sets of labels. Due to obtaining a sequence of labelling functions, we also generate a sequence of length h of graphs $(G_0, G_1, \ldots, G_h)$ which differ only in their associated labelling functions. We now define the WL kernel itself, which is, in effect, a meta kernel, as:

$$k_{WL}(\mathcal{G}) = k(\mathcal{G}_0) + k(\mathcal{G}_1) + \ldots + k(\mathcal{G}_h) \tag{A.9}$$

where k is an inner kernel. k_{WL} will be a graph kernel as linear combinations of kernels preserve the properties of being semi-definite positive and symmetric.

In Chapter 4, we combine the WL kernel formulation above with the subtree kernel as our inner k. This instantiation of WL is defined as follows. Let $\Sigma_i \subseteq \Sigma$ be the alphabet of all letters that occur in the labelling of nodes in graphs up the the i-th iteration of WL. We assume all Σ_i are pair-wise disjoint. Further, and without loss of generality, let $\Sigma_i = \{\sigma_{i0}, \sigma_{i|\Sigma_i|}\}$ be ordered sets. Let $c_i : \mathcal{G} \times \Sigma_i \to \mathbb{N}$ be a count function, *s.t.* $c(G, \sigma_{ij})$ returns the number of occurrence of σ_{ij} in G as

a node label. We will now build our vector as:

$$\psi(G) = (c_0(G, \sigma_{00}), \ldots, c_0(G, \sigma_{0|\Sigma_0|}), \ldots, c_h(G, \sigma_{h|\Sigma_h|})) \tag{A.10}$$

By defining the projection function ψ, we side-step the process of computing k directly for each graph. The inner product of the vector space onto which ψ projects represents the number of common rooted subtrees shared by the respective projected graphs. Finally, let $\psi(G_i) = s_i$, then we obtain the similarity matrix as:

$$k(\mathcal{G})_{ij} = \langle \mathbf{s_i}, \mathbf{s_j} \rangle \tag{A.11}$$

This enables Flexeme to then perform agglomerative clustering; the intuition is that common rooted subtrees represent a good proxy for similar execution paths.

Appendix B

Colophon

This document was set in the Helvetica typeface for text and the `Inconsolata typeface` for code using the MiKTEX distribution of LATEX and BibTEX, composed with VS Code. Plots were created using Seaborn in Python.

Bibliography

[1] GitHub. GitHub: About continuous integration. https://docs.github.com/en/actions/guides/about-continuous-integration#about-continuous-integration, 2020. Accessed: 2020-09-17. 19

[2] Georgios Gousios and D. Spinellis. GHTorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, June 2012. ISBN 978-1-4673-1761-0. doi: 10.1109/MSR.2012.6224294. 19, 63

[3] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. 2019. URL http://arxiv.org/abs/1909.09436. 19

[4] Jane Cleland-Huang, Olly Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software Traceability: Trends and Future Directions. *FOSE 2014: Proceedings of the on Future of Software Engineering (36th ICSE 2014)*, pages 55–69, 2014. doi: 10.1145/2593882.2593891. 20, 28, 73

[5] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework. *Empir. Softw. Eng.*, 22(3):967–995, 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9457-1. 20, 28, 29, 40, 77

[6] Patrick Mader, Orlena Gotel, and Ilka Philippow. Motivation matters in the traceability trenches. In *2009 17th IEEE International Requirements Engineering Conference*, pages 143–148. IEEE, 2009. 20

[7] Ralf Dömges and Klaus Pohl. Adapting traceability environments to project-specific needs. *Communications of the ACM*, 41(12):54–62, 1998. 20

[8] Agile Alliance. Agile Alliance: Backlog refinement. https://www.agilealliance.org/glossary/backlog-grooming/, 2019. Accessed: 2019-11-26. 20, 27, 74

[9] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empir. Softw. Eng.*, 21(2):303–336, 2016. ISSN 1573-7616. doi: 10.1007/s10664-015-9376-6. 21, 24, 32, 33, 80, 81, 88, 89, 90, 91, 93, 94, 96, 97, 98

[10] Kim Herzig and Andreas Zeller. The impact of tangled code changes. *IEEE Int. Work. Conf. Min. Softw. Repos.*, pages 121–130, 2013. ISSN 2160-1852. doi: 10.1109/MSR.2013.6624018. 21, 32, 80, 88, 89, 97

[11] Christoph Treude, Carlos A Prolo, and Fernando Figueira Filho. Challenges in analyzing software documentation in Portuguese. In *Proceedings of the 29th Brazilian Symposium on Software Engineering*, pages 179–184. IEEE, 2015. 21, 35, 36, 122

[12] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links. *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, page 97, 2010. doi: 10.1145/1882291.1882308. 22, 24, 27, 29, 47, 58, 59, 74

[13] Christoph Treude, Mathieu Sicard, Marc Klocke, and Martin Robillard. TaskNav: Task-based Navigation of Software Documentation. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 649–652, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2819009.2819128. 24, 105, 106, 117

[14] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992. 25

[15] Edith Tom, Aybuke Aurum, and Richard Vidgen. A consolidated understanding of technical debt. 2012. 25

[16] Edith Tom, AybüKe Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013. 25, 26

[17] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T Barr. Game-theoretic analysis of development practices: Challenges and opportunities. *Journal of Systems and Software*, 159:110424, 2020. 26

[18] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T Barr. The assessor's dilemma: Improving bug repair via empirical game theory. *IEEE Transactions on Software Engineering*, 2019. 26

[19] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014*, pages 92–101, New York, New York, USA, 2014. ACM Press. ISBN 9781450328630. doi: 10.1145/2597073.2597074. URL http://dl.acm.org/citation.cfm?doid=2597073.2597074. 27, 72, 74

[20] Laurie Tratt. Personal communication with laurie tratt. Online, 2018. 27, 74

[21] C. Neumuller and P. Grunbacher. Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 145–156, September 2006. doi: 10.1109/ASE.2006.25. 28, 73

[22] JIRA. JIRA: Link JIRA issues to Confluence pages automatically. https://www.atlassian.com/blog/confluence/link-jira-issues-to-confluence-pages-automatically, 2017. Accessed: 2017-08-20. 28, 73

[23] GitHub. GitHub: Autolinked references and URLs. https://help.github.com/articles/autolinked-references-and-urls/, 2017. Accessed: 2017-08-20. 28, 73

[24] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. Software traceability with topic modeling. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:95–104, 2010. ISSN 0270-5257. doi: 10.1145/1806799.1806817. 28, 29, 74

[25] Markus Borg, Per Runeson, and Anders Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, December 2014. ISSN 1573-7616. doi: 10.1007/s10664-013-9255-y. URL https://doi.org/10.1007/s10664-013-9255-y. 28, 74

[26] Chris Mills. Automating Traceability Link Recovery Through Classification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 1068–1070, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3121280. URL http://doi.acm.org/10.1145/3106237.3121280. 28

[27] Hazeline U. Asuncion and Richard N. Taylor. Capturing Custom Link Semantics Among Heterogeneous Artifacts and Tools. In *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '09, pages 1–5, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3741-2. doi: 10.1109/TEFSE.2009.5069574. 28, 29

[28] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best Practices for Automated Traceability. *Computer*, 40(6):27–35, June 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.195. 28

[29] K. Pohl. PRO-ART: enabling requirements pre-traceability. In *Proceedings of the Second International Conference on Requirements Engineering*, pages 76–84, April 1996. doi: 10.1109/ICRE.1996.491432. 28

[30] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, March 1996. ISSN 0740-7459. doi: 10.1109/52.506462. 28

[31] Davide Falessi, Massimiliano Di Penta, Gerardo Canfora, and Giovanni Cantone. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering*, 22(3):996–1027, June 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9460-6. URL https://doi.org/10.1007/s10664-016-9460-6. 29

[32] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced?: Bias in Bug-Fix Datasets. *Proc. ESEC/FSE*, pages 121–130, 2009. ISSN 0168-1656. doi: 10.1145/1595696.1595716. 29, 40, 74

[33] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. ReLink: Recovering Links Between Bugs and Changes. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 15–25, 2011. doi: 10.1145/2025113.2025120. URL http://doi.acm.org/10.1145/2025113.2025120. 29, 31, 47, 48, 52, 75, 76

[34] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, 2012. doi: 10.1145/2393596.2393671. 29, 31, 45, 48, 75, 76

[35] Lutz Prechelt and Alexander Pepper. Bflinks: Reliable Bugfix Links via Bidirectional References and Tuned Heuristics. *International scholarly research notices*, 2014, 2014. 29, 75

[36] Tien Duy B Le, Mario Linares-Vásquez, David Lo, and Denys Poshyvanyk. RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information. In *IEEE International Conference on Program Comprehension*, volume 2015-Augus, pages 36–47. IEEE, May 2015. ISBN 9781467381598. doi: 10.1109/ICPC.2015.13. 30, 41, 45, 47, 48, 49, 50, 57, 59, 75

[37] Mario Linares-Vasquez, Luis Fernando Cortes-Coy, Jairo Aponte, and Denys Poshyvanyk. ChangeScribe: A Tool for Automatically Generating Commit Messages. *Proceedings - International Conference on Software Engineering*, 2:709–712, 2015. ISSN 0270-5257. doi: 10.1109/ICSE.2015.229. 30, 41, 49, 58, 75

[38] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic Generation of Pull Request Descriptions. *arXiv preprint arXiv:1909.06987*, 2019. 30, 75

[39] Yan Sun, Qing Wang, and Ye Yang. FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance. *Information and Software Technology*, 84:33–47, 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.11.010. 30, 75

[40] Apache. Coding and Commit Conventions. https://subversion.apache.org/docs/community-guide/conventions.html, 2020. Accessed: 2020-07-09. 30, 75

[41] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. Deeplink: Recovering issue-commit links based on deep learning. *Journal of Systems and Software*, 158:110406, 2019. 30, 75, 76

[42] Y. Sun, C. Chen, Q. Wang, and B. Boehm. Improving missing issue-commit link recovery using positive and unlabeled data. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 147–152, October 2017. doi: 10.1109/ASE.2017.8115627. 30, 48, 76

[43] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. 30, 76

[44] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 30, 76

[45] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Maeder. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th International Conference on Software Engineering*, 2018. doi: 10.1145/3180155.3180207. 31, 57, 76

[46] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. *Proceeding 33rd Int. Conf. Softw. Eng. - ICSE '11*, page 351, 2011. ISSN 0270-5257. doi: 10.1145/1985793.1985842. URL http://portal.acm.org/citation.cfm?doid=1985793.1985842. 31, 32, 97

[47] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes? *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. - FSE '12*, page 1, 2012. doi: 10.1145/2393596.2393656. URL http://dl.acm.org/citation.cfm?doid=2393596.2393656. 32, 80, 81, 97

[48] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.41. 32, 80, 97

[49] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proc. 22nd Int. Conf. Progr. Compr. (ICPC 2014)*, pages 262–265, New York, New York, USA, 2014. ACM Press. ISBN 9781450328791. doi: 10.1145/2597008.2597798. URL http://dl.acm.org/citation.cfm?doid=2597008.2597798. 32, 97

[50] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Splitting commits via past code changes. *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pages 129–136, 2017. ISSN 1530-1362. doi: 10.1109/APSEC.2016.028. 32, 97

[51] De Roover and Ward Muylaert. Untangling Source Code Changes Using Program Slicing. 2017. 32, 98

[52] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. *Proc. - Int. Conf. Softw. Eng.*, 1(August 2014):134–144, 2015. ISSN 0270-5257. doi: 10.1109/ICSE.2015.35. 32, 33, 80, 81, 91, 93, 94, 95, 98

[53] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stephane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2015 - Proc.*, pages 341–350. IEEE, March 2015. ISBN 9781479984695. doi: 10.1109/SANER.2015.7081844. URL http://ieeexplore.ieee.org/document/7081844/. 33, 98

[54] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, 2006. doi: 10.1145/1137983.1137999. 33, 34, 80, 99

[55] Wei Le and Shannon D. Pattison. Patch verification via multiversion interprocedural control flow graphs. *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014*, pages 1047–1058, 2014. ISSN 0270-5257. doi: 10.1145/2568225.2568304. URL http://dl.acm.org/citation.cfm?doid=2568225.2568304. 34, 80, 84, 99

[56] Carol V Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019. 34, 99

[57] Panichella Sebastian and Proksch Harald. Redundancy-free Analysis of Multi-revision Software Artifacts Redundancy-free Analysis of Multi-revision Software Artifacts. (May), 2018. 34, 99

[58] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic Slicing of Software Version Histories. *IEEE Trans. Softw. Eng.*, 44(2):182–201, 2018. doi: 10.1109/TSE.2017.2664824. 34, 99

[59] Yi Li, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 686–696, 2015. 34, 99

[60] Yi Li, Chenguang Zhu, Milos Gligoric, Julia Rubin, and Marsha Chechik. Precise semantic history slicing through dynamic delta refinement. *Autom. Softw. Eng.*, 26(4):757–793, 2019. ISSN 15737535. doi: 10.1007/s10515-019-00260-8. 34, 35, 99, 100

[61] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving low quality stack overflow post detection. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 541–544. IEEE, 2014. 35, 116, 118, 123

[62] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceeding 8th Work. Conf. Min. Softw. Repos. - MSR '11*, page 203, New York, New York, USA, 2011. ACM Press. ISBN 9781450305747. doi: 10.1145/1985441.1985471. URL http://portal.acm.org/citation.cfm?doid=1985441.1985471. 35, 121

[63] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting Program Comprehension with Source Code Summarization. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 223–226, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810335. URL http://doi.acm.org/10.1145/1810295.1810335. 35, 121

[64] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, October 2010. doi: 10.1109/WCRE.2010.13. 35, 121

[65] Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 156–159. IEEE, 2010. doi: 10.1109/icpc.2010.29. 35, 121

[66] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013. 35, 121

[67] Yuan Tian and David Lo. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 570–574. IEEE, 2015. 35, 121

[68] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012. 35

[69] Christian D Newman, Reem S Alsuhaibani, Michael L Collard, and Jonathan I Maletic. Lexical Categories for Source Code Identifiers. *SANER'17*, 2017. 35, 122

[70] Margaret Deuchar. BilingBank Spanish-English Miami Corpus. http://www.bangortalk.org.uk/speakers.php?c=miami, 2010. 36, 122

[71] Yogarshi Vyas, Spandana Gella, Jatin Sharma, Kalika Bali, and Monojit Choudhury. Pos tagging of english-hindi code-mixed social media content. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 974–979, 2014. 36, 122

[72] Shana Poplack. Sometimes I'll start a sentence in Spanish Y TERMINO EN ESPAÑOL: toward a typology of code-switching. *Linguistics*, 18:581–618, January 1980. doi: 10.1515/ling.1980.18.7-8.581. 36, 107, 122

[73] Thamar Solorio and Yang Liu. part-of-speech tagging for English-Spanish code-switched text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1051–1060. Association for Computational Linguistics, 2008. 36, 122

[74] Anupam Jamatia, Björn Gambäck, and Amitava Das. part-of-speech tagging for code-mixed english-hindi twitter and facebook chat messages. In *Proceedings of the International Conference Recent Advances in Natural Language Processing*, pages 239–248, 2015. 36, 122

[75] Victor Soto and Julia Hirschberg. Joint part-of-speech and Language ID Tagging for Code-Switched Data. pages 1–10, 2018. 36, 108, 122

[76] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing complex development artifacts by mining heterogeneous data. *IEEE Int. Work. Conf. Min. Softw. Repos.*, 2015-Augus:401–405, 2015. ISSN 2160-1860. doi: 10.1109/MSR.2015.49. 36, 122

[77] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. StORMeD: Stack overflow ready made data. *IEEE Int. Work. Conf. Min. Softw. Repos.*, 2015-Augus:474–477, 2015. ISSN 2160-1860. doi: 10.1109/MSR.2015.67. 36, 104, 114, 122, 123

[78] Walid Maalej and Hans-Jörg Happel. Can Development Work Describe Itself? *7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 191–200, 2010. ISSN 0270-5257. doi: 10.1109/MSR.2010.5463344. 40

[79] Qing Mi and Jacky Keung. An empirical analysis of reopened bugs based on open source projects. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, pages 1–10, 2016. doi: 10.1145/2915970.2915986. 40

[80] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken Ichi Matsumoto. *Studying re-opened bugs in open source software*, volume 18. 2013. ISBN 1066401292. doi: 10.1007/s10664-012-9228-6. 40

[81] Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process*, 26(12):1172–1192, August 2016. ISSN 2047-7473. doi: 10.1002/smr.1797. 40

[82] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 43, 2011. ISBN 9781450305747. doi: 10.1145/1985441.1985451. 40

[83] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.11.002. 40, 73

[84] Ming Wen, Rongxin Wu, and Shing-chi Cheung. Locus : Locating Bugs from Software Changes. *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pages 262–273, 2016. doi: 10.1145/2970276.2970359. 40, 73

[85] Peter O'Hearn. Icse 2020 keynote: Formal reasoning and the hacker way. [Online: https://youtu.be/bb8BnqhY3Ss?t=2599], 2020. 40, 125

[86] Sridhar Nerur, RadhaKanta Mahapatra, and George Mangalaraj. Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5):72–78, 2005. 40, 46

[87] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. Mondrian forests: Efficient online random forests. In *Advances in neural information processing systems*, pages 3140–3148, 2014. 45, 60

[88] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Baranauskas. How Many Trees in a Random Forest? *Lecture notes in computer science*, 7376, July 2012. doi: 10.1007/978-3-642-31537-4_13. 45, 52

[89] Profir-Petru Pârțachi, Earl T. Barr, and David R. White. Aide-mémoire: Accurate Issue Links at Pull Request submission. https://github.com/PPPI/a-m/, 2017. Accessed: 2017-08-14. 46, 53

[90] Max Kuhn and Johnson Kjell. *Feature Engineering and Selection: a Practical Approach for Predictive Models*. CRC Press, 2019. 47

[91] Scikit-learn. Recursive Feature Elimination: SciKit Implementation. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html, 2020. Accessed: 2020-06-17. 47, 50

[92] M.F. Porter. An algorithm for suffix stripping, 1980. ISSN 0033-0337. 47

[93] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. URL http://is.muni.cz/publication/884893/en. 48

[94] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed 17.08.2018]. 52, 59, 89

[95] Georgios Gousios and D. Spinellis. Google Cloud Public Table of GitHub Projects. https://bigquery.cloud.google.com/dataset/ghtorrent-bq:ght, 2017. Contents from 2.9M public, open source licensed repositories on GitHub; Accessed: 2017-08-10. 53

[96] GitHub. GitHub Octoverse 2016. https://octoverse.github.com/, 2016. Accessed: 2017-08-07. 54, 72

[97] Profir-Petru Pârțachi, Earl T. Barr, and David R. White. Datasets as pickled python objects. https://figshare.com/s/83c448eb518b3d04651f, 2017. Accessed: 2020-02-25. 54

[98] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715. 55

[99] Donald R. Hedeker and Robert D. Gibbons. *Longitudinal Data Analysis*. WileyInterscience, 2006. ISBN 9780471420279. 57

[100] Linux Kernel. Linux Kernel Commit Message Practice. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/submitting-patches.rst?id=bc7938deaca7f474918c41a0372a410049bd4e13#n664, 2020. Accessed: 2020-06-19. 58

[101] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278. 59

[102] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. *R2Fix: Automatically generating bug fixes from bug reports*. PhD thesis, University of Waterloo, 2013. 73

[103] Jeanne Ferrante, Karl J. Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 167 LNCS, pages 125–132, 1984. ISBN 9783540129257. doi: 10.1007/3-540-12925-1_33. URL https://www.cs.utexas.edu/{~}pingali/CS395T/2009fa/papers/ferrante87.pdf. 80

[104] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011. 80, 87, 89, 100, 130

[105] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. RefiNym: using names to refine types. In *Proc. 2018 26th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. - ESEC/FSE 2018*, pages 107–117, 2018. ISBN 9781450355735. doi: 10.1145/3236024.3236042. 83, 84

[106] Robert W. Irving and David F. Manlove. The Stable Roommates Problem with Ties. *J. Algorithms*, 43(1):85–105, April 2002. ISSN 0196-6774. doi: 10.1006/jagm.2002.1219. URL http://dx.doi.org/10.1006/jagm.2002.1219. 85

[107] Boris Weisfeiler and Andrei A Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968. 87, 130

[108] Microsoft. Microsoft rosyln. https://github.com/dotnet/roslyn. Accessed: 2018-05-31. 88

[109] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. GraKeL: A Graph Kernel Library in Python. *arXiv preprint arXiv:1806.02193*, 2018. 89

[110] Jakob Nielsen. Response times: the three important limits. *Usability Engineering*, 1993. 89

[111] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005. 90

[112] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97. doi: 10.1002/nav.3800020109. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109. 91

[113] Robert V. Hogg, Elliot A. Tanis, and Dale L. Zimmerman. *Probability and statistical inference*. Pearson, 2020. 96

[114] Frank Tip. A Survey of Program Slicing Techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994. 97

[115] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010. 100

[116] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF Models for Sequence Tagging. 2015. ISSN 0270-6474. doi: 10.1061/(ASCE)CO.1943-7862.0000274. URL http://arxiv.org/abs/1508.01991. 108

[117] Genta Indra Winata, Chien-Sheng Wu, Andrea Madotto, and Pascale Fung. Bilingual Character Representation for Efficiently Addressing Out-of-Vocabulary Words in Code-Switching Named Entity Recognition. In *Proceedings of the Third Workshop on Computational Approaches to Linguistic Code-Switching*, pages 110–114, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-3214. URL https://www.aclweb.org/anthology/W18-3214. 108, 110, 122

[118] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967. ISSN 0018-9448. doi: 10.1109/TIT.1967.1054010. 111

[119] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org. 112

[120] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014. URL http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14. 112

[121] Stack Exchange. Stack Exchange Data Dump. https://archive.org/details/stackexchange, 2018. [Online; accessed 05-Sep-2018]. 112

[122] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009. 112

[123] Profir-Petru Pârțachi, Santanu K Dash, Christoph Treude, and Earl T Barr. Code Comment Corpus. https://github.com/PPPI/POSIT/blob/master/data/corpora/lucid.zip, 2019. [Online; accessed 24-Jan-2020]. 113

[124] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. *Proc. - Int. Conf. Softw. Eng.*, pages 476–486, 2018. ISSN 0270-5257. doi: 10.1145/3196398.3196408. 116

[125] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: Reconstructing and analyzing the evolution of Stack Overflow posts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 319–330. ACM, 2018. doi: 10.1145/3196398.3196430. 116

[126] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. 1993. 117

[127] Slav Petrov, Dipanjan Das, and Ryan McDonald. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086*, 2011. 117

[128] Luca Ponzanelli. Holistic recommender systems for software engineering. In *Companion Proc. 36th Int. Conf. Soft. Eng.*, pages 686–689, 2014. 120, 121

[129] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi: 10.1177/001316446002000104. 121

[130] Jens Dietrich, Markus Luczak-Roesch, and Elroy Dalefield. Man vs machine: a study into language identification of stack overflow code snippets. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 205–209. IEEE Press, 2019. doi: 10.1109/msr.2019.00041. 121

[131] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.*, pages 138–147. IEEE, 2003. 126

[132] Jun Lin, Chan Chou Lin, Jane Cleland-Huang, Raffaella Settimi, Joseph Amaya, Grace Bedford, Brian Berenbach, Oussama Ben Khadra, Chuan Duan, and Xuchang Zou. Poirot: A distributed tool supporting enterprise-wide automated traceability. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 363–364. IEEE, 2006. 126

[133] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4): 13–es, 2007. 126

[134] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007. 126

[135] Sergios Theodoridis. *Pattern Recognition*. Elsevier, 2008. ISBN 9780080949123. 128

[136] M. D. Buhmann. Radial basis functions. *Acta Numerica*, 9:1–38, 2000. doi: 10.1017/S0962492900000015. 128

[137] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495, 2009. 129

[138] M. Sugiyama, E. Ghisu, F. Llinares-López, and K. M. Borgwardt. graphkernels: R and python packages for graph comparison. *Bioinformatics*, 34(3):530–532, 2018. 130