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

June 11, 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.

Abstract

In recent years systems that would previously live on different platforms have been integrated under a single umbrella. The proliferation 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 has led to leaner and faster development cycles. This has also reduced the cost of entry and created large, publicly accessible sources of source-code together with related project artefacts.

This shift in tooling has also facilitated a shift in development paradigms, developers now prefer a continuous integration/continuous delivery infrastructure. This has led to more projects adopting a more agile development process. Developers often forgo tasks that may aid project health so that they can instead travel light. However, project health determines project success. Project health encompasses traceability, documentation, adherence to coding conventions, in short tasks that allow for lower maintenance costs and higher accountability.

Simultaneously, this shift has allowed for the proliferation of Natural Language or Natural Language and Formal Language textual artefacts which 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.

Under this umbrella, we 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

The work presented as part of this thesis opens the door to further research in applications of machine learning towards aiding the process of software development and making the profession more accessible. Directly, it helps developers maintain projects in a state of lower technical debt and can, over time, enable more sophisticated analysis by researchers as the quality of the data improves. It also further paves the way towards multi-channel analysis of coding artefacts where signal from both code and natural language are considered. TODO: more stuff here about how this is impactful.

Acknowledgements

Acknowledge all the things!

Contents

List of Figures

List of Tables

Chapter 1

Introductory Material

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 2

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

Paper Authors Profir-Petru Pârţachi, 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 TODO: Refocus abstract as 'we are first to solve this online'. Links between pull-requests and the issues they address accelerate the development of a software project, but are often omitted. We present a new tool, Aide-mémoire, to interactively suggest links to a developer when that developer submits a pull-request. In other words, we designed Aide-mémoire to smoothly integrate into existing workflows in contrast to the previous state of the art approaches that aimed to repair histories to better train bug or estimation predictors. 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 more than 16% of all discovered links were originally missed by developers.

2.1 Introduction

Traceability concerns tracing and studying links among software artefacts; its pillar is links between requirements and their implementation. It is a seminal software engineering concern. In the "move fast and break things" era, adding and maintaining links are too often neglected. Good traceability practices and tooling, however, improve all aspects of software development from requirements elicitation to speeding maintenance.

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. 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 **??**. 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 **??**. Developers use them to prioritise work; reviewers use them to learn the context of the issue. They facilitate fault prediction **?**, bug localisation **???**, 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, sport increased support for linking, developers do not record most links **?**; a trend we confirm 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 (**??**). During PR review, missing links are sometimes discovered manually and the PR is amended. Around 16% of PRs are linked during this process, leaving 38% unlinked.

TODO: First online We present *Aide-mémoire* (A-M) to suggest pertinent PR-

issue links. We call it Aide-mémoire because we hope it will help a project partially retain its "collective memory". A-M suggests a link when a developer submits a PR or closes an issue. A-M does not require invasive instrumentation, but relies instead on content 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-M's principle goal is to make the cost of creating and maintaining PR-Issue links so easy and seamless as to meet Agile's lightweight requirement **?** (**??**).

TODO: Feature selection paragraph here. We then explain our feature selection methodology and how we reduced a larger set of considered features to a manageable one to allow faster development and to further allow a light integration. We start by detailing the feature set from which we set out, providing the intuition of what each feature captures. We then explain how this feature set can be reduced with negligible impact on classification performance and present our resulting feature set. While leaving feature selection to neural architectures is common today, it incurs a higher training cost which we sought to avoid. In machine learning approaches other than deep learning, feature selection is a valuable step that could make or break a model. Within the Software Engineering community, it is often side-stepped. We believe our detailed description of this process is a contribution in its own right.

TODO: Experimental Set-up. **??** details our experimental set-up. We first explain how our longitudinal evaluation mimics the use-case of our tool and how we set it up. We then explain how we can compare against the offline state-of-the-art, RCLinker, and how we modified both the evaluation protocol and the tool, in our reproduction as RCRep, to solve our linking problem. For the completeness of our reproduction, we also include RCRep's results on the original commit-issue linking task both on our Java corpus and the original Apache Commons corpus; it achieves a respectable 0.52 F1 score on the Java corpus and a comparable to the original F1 of 0.74 vs 0.64 of the original on the Apache corpus. Next, we present the metrics

we use to evaluate A-M in **??**. Crucially, we introduce a notion of List accuracy which captures correct negative predictions which allow our model to output and would be missed by the standard measure employed (MAP). We then present our corpora, how they were constructed to ensure representative projects and how we performed the original linking on which we train and validate A-M. Finally, we present the user-study protocol used to assess the utility of A-M as a replacement for the built-in search functionality from GitHub.

The state of the art commit-issue prediction tool is RCLinker **?**. RCLinker is offline and handles commits, not pull-requests, and Java projects, since it requires ChangeScribe **?**. To validate A-M against RCLinker, we therefore had to adapt it (**??**). We call our variant RCRepPR; it suggests a pull-request when its internal RCLinker predictor suggests any commit in the pull-request. ChangeScribe summarises commits for RCLinker which are then used to link them to issues. RCRepPR replaces ChangeScribe descriptions with user-provided PR descriptions, the first post in a PR, semi-structured by convention. Because offline is a degenerate case of online, we run A-M offline when comparing it with RCRepPR; however, we ensure neither tool sees events from the future. We show that A-M is more precise than RCRepPR, achieving achieves a mean Precision of 0.76 and F1-Score of 0.46 compared to RCRepPR's 0.14 and 0.15, with a similar Recall (0.37 vs 0.36) across 47 Java projects.

TODO: Smoothe in first sentence We then evaluate it on a much larger, multi-lingual corpus to show generalisation beyond just Java. 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 (**??**). 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. 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; **??** 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 (**??**). 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 (**??**). 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 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 as the project evolves. Thus, A-M does not require a dedicated maintenance task, substantially enhancing its deployability. Finally, installing A-M only requires installing a server 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 is available at `https://github.com/PPPI/a-m`.

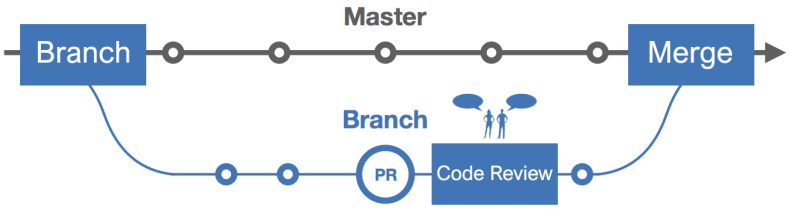


Figure 2.1: Simplified pull-request process. Each small circle represents a commit. A developer opens a new branch, makes code changes and submits a pull-request to code review. Further changes may be made on the branch before being accepted and merged into the Master.

2.2 Motivating Example

?? 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.

?? 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 **??** and **??**. Moreover, there is significant textual similarity between the PR description, the first posting in **??**, and the first three postings in **??**. The particular sub-tokens that overlap are highlighted in the title and postings. There are also common participants (@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.

Figure 2.2: 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.

2.3 Aide-mémoire

Aide-mémoire is the first online PR-issue linking tool. **??** 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. A-M has two modes. In the first mode, it uses the features discovered in **??**. In the second, it replicates the steps from **??** to adapt itself to the project it is being trained on. Next, we use the features

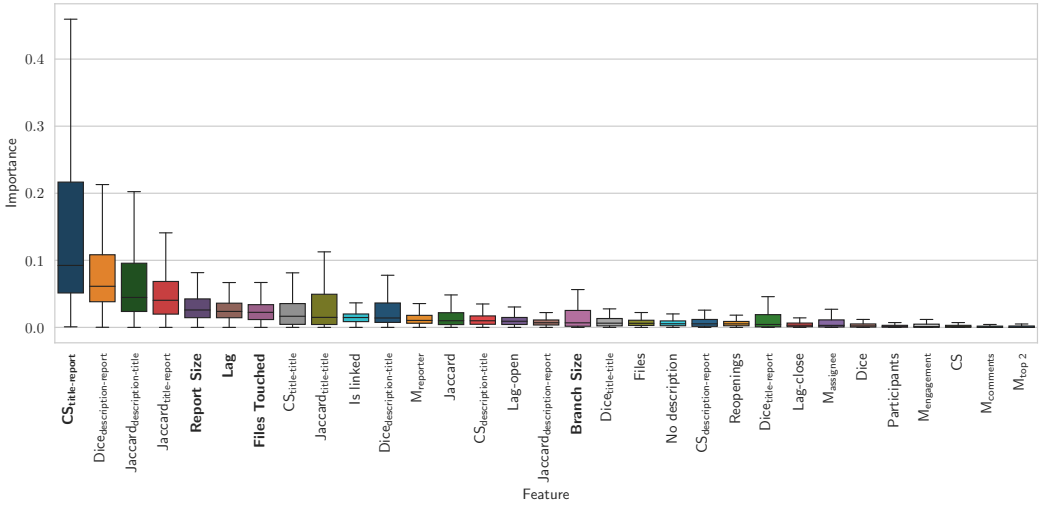


Figure 2.3: Feature Importance as computed using a Random Forests Classifier on the development set. We performed recursive feature elimination to select the final feature set while considering synergistic and antagonistic interactions between the features. The final set of features can be seen in bold. We do not show outlier values for presentation reasons.

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 a 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 **?**.

2.3.1 Model Learning

We train a statistical classifier on PR-issue pairs to learn a probability distribution over possible links. For each candidate link, we calculate the feature vector as detailed in **??** 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 observed empirically on our development set that Mondrian Forests perform well despite this imbalance and hence we do not employ

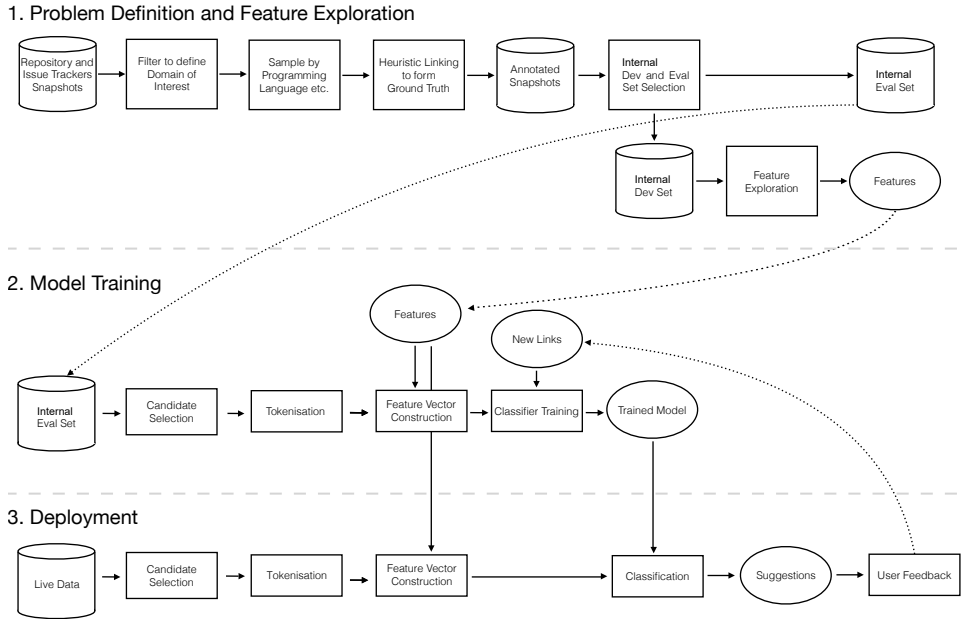


Figure 2.4: Methodology used for instantiating Aide-mémoire. In the first stage, a datastore of repositories is identified, filtered and sampled to define the domain of interest. Heuristic linking generates an initial knowledge to learn and evaluate against. A subset of selected repositories is used 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.

undersampling.

We deviate from the more classical Random Forest used by the state-of-the-art offline tool **?**, and instead employ the online method of Mondrian Forests **?**. 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 **?**. We configure the Mondrian Forest to use 128 estimators, assuming that the recommendation made by Oshiro *et al.* **?** 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 explicitly 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.

2.3.2 Deployment

TODO: make sure this reads A-M has a front-end and a back-end. The back-end learns, maintains, and stores the project's model; the Chrome plug-in front-end parses the issue/PR page for the back-end. Developers interact with Aide-mémoire via a Chrome plug-in that may be used when viewing a PR or issue. The plugin interactively suggests links to a developer closing an issue, submitting or reviewing a PR. A-M suggests links above a user controlled threshold, and displays them in descending likelihood of relevance to the developer's activity. We only suggest links that are classified as more likely than the special 'no_pr' and 'no_issue' entities, silencing our suggestions when our confidence is low.

To learn a model for their project, a developer can install A-M and generate a model locally; entering their GitHub URL instigates the crawling and processing of their repository, or project managers may install a central backend and allow

contributors to interact with it via A-M's Chrome plug-in. The initial training of our system over a large project such as Google's Guava (which at the time of our crawl was a project containing 206 PRs and 2563 issues, and a total of 5392 commits) takes less than two hours on an Intel i7-6820HK@2.70 GHz. Our prototype plug-in subsequently makes suggestions to a developer in under 20 seconds of the completion of their PR or issue closing message.

The related RCLinker tool **?** employed a complex undersampling method, which improved the quality of the Random Forests classifiers generated at some cost — 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 opt to not use any undersampling, relying instead on the empirically observed resilience of Mondrian Forests to class imbalance to tackle the issue. This removes the computational cost associated with the procedure, which is critical for maintaining performance in our online formulation of issue-PR link prediction.

Full source code, deployment tools, and code and scripts required to recreate our evaluation can be found online **?**.

2.4 Exploring the Feature Space of Issue-PR Links

In this section, we first present the full feature space we have considered for A-M. We then discuss how this feature space is constructed. We finish by explaining how we reduce the feature space to both facilitate faster model development, and to better utilise the data we have available. Unusual for this area of research, which opts to understand the contribution of features via ablation post hoc, we employ feature selection. This allows us to rigorously reduce the number of features in the model while still accounting for synergistic relationships and removing only redundant features before evaluating the model on the full dataset.

2.4.1 Feature Space Construction

?? demonstrates that the text artefacts surrounding an issue and a PR can overlap. Previous work on retrospectively repairing *commit-issue* links in an offline context exploited similar overlap **????**. We build a feature space by considering features

employed by the previous state-of-the-art tool **?**, 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 **??**. This link is equivalent to identifying PRs or issues that should be classified as *unlinked*, i.e. lacking an edge in E. The features considered can be grouped into the following categories: *textual*, *social*, *temporal*, and *structural*.

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 **?** relied on commit summarisation tools such as ChangeScribe **?**, 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.

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 **?**, 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 **???** 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:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \cdot log_2\left(\frac{|D|}{|\{d' | t \in d', d' \in D\}|}\right). \tag{2.1}$$

We replace terms that are either too rare (occur a single time in our corpus)

Table 2.1: Features constructed from a document vector representation and metadata. p is a PR object, i is an Issue object, e is a tracebility 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.* **?**.

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

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 **?**, which was designed to handle large corpora efficiently.

We then use this representation to compute cosine similarity:

$$CS(p_j, i_k) = \frac{p_j \cdot i_k}{|p_j| \cdot |i_k|}, \tag{2.2}$$

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

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:

$$\text{bow}(d) = \{(t,\ \text{tf}(t,d)) | t \in d\}. \tag{2.3}$$

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:

$$JS(p_j, i_k) = \frac{|\text{bow}(p_j) \cap \text{bow}(i_k)|}{|\text{bow}(p_j) \cup \text{bow}(i_k)|} \tag{2.4}$$

$$Dice(p_j, i_k) = \frac{|\text{bow}(p_j) \cap \text{bow}(i_k)|}{\min(|\text{bow}(p_j)|, |\text{bow}(i_k)|)} \tag{2.5}$$

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 $\text{Jaccard}_{\text{full-context}}$ and $\text{Dice}_{\text{full-context}}$ make use of all pairs, the other similarity features restrict j and k as

shown in **??**.

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 **??**, 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 **?** 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.

2.4.2 Feature Selection

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. Details on how we constructed this set are provided in **??**. We analyse the features listed in **??**. To reduce the number of features, we first consider the linear correlation of the 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. **??** shows feature importance over the full feature set presented in **??**, 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 **??**). 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 **?**. 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.* **?**,

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.* **?**, leaving other setting to SciPy **?** defaults. We evaluate the features over all (p,i) pairs for each repository; **??** shows the results.

2.5 Aide-mémoire's Experimental Set-up

We perform a preliminary study and two sets of experiments: a comparison with previous related work on a corpus of Java projects, and a standalone evaluation of Aide-mémoire across a larger corpus containing multiple languages. The preliminary study is used to assess the current state of Issue-PR linking on GitHub both as encouraged by contribution guides, usually 'CONTRIBUTING.MD' files located in the root of the repository, as well as practice.

For longitudinal validation, we 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.

To compare A-M with offline work, we first recreate the RCLinker tool from Le *et al.* **?**, which represents the state-of-art of automatic, non-intrusive commit-issue linking. **??** discusses other tools we considered but did not reproduce. We implement the tool within our own experimental framework, and name this replication 'RCRep' to avoid confusion[1]. RCRep is dependent upon commit summaries

[1] We extend our gratitude to the authors of RCLinker for their patience and assistance in recreating their work.

Table 2.2: Performance values on the Java Corpus for RCRep when solving the commit-issue prediction task. This evaluation confirms that our replication 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**

provided by the ChangeScribe tool **?**; we generated the required summaries using a modified version of the ChangeScribe Eclipse plugin that can be run headlessly. We evaluate RCRep on the commit-issue prediction task as a sanity check; the results can be seen in **??**, and demonstrate strong generalisation beyond the Apache Commons corpus used in the original RCLinker paper. We also run our replication on the original corpus. We present these results in **??**. We note that the Random Forest implementations in SciPy **?** and in Weka **?** may use different defaults. We observe our replication to obtain similar results with only a slight penalty to Recall.

To further assess the impact of the problem formulation rather than just the impact of the textual summary of a source code change, we have evaluated RCRep using PR-metadata injected as commit descriptions rather than using ChangeScribe. This configuration assess the quality of ChangeScribe as a source of textual information relative to developer Pull-Request discussions. Results are presented as RCRepPR. All further discussion regarding RCRep refer to RCRepPR as well.

RCRep classifies commit-issue candidate pairs as true links or false links and hence solves the offline commit-issue link prediction task. In contrast, Aide-

Table 2.3: Performance values on the original Apache Corpus for RCRep when solving the commit-issue prediction task. We bias our replication towards higher precision to account for the shift to the perspective use-case and we observe a higher performance of our replication relative to the results reported in Le *et al.* **?** for F1-Score and Precision at the cost of Recall.

Repository	**F1-Score**		**Precision**		**Recall**	
	RCRepl	RCLinker	RCRepl	RCLinker	RCRepl	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

mémoire is an online assistive tool and outputs a variable-length list of suggested links. Hence, the output of RCRep has to be transformed to enable a comparison. If RCRep predicts a link between a commit and an issue, the PR containing the commit will be linked to that issue. We restrict our dataset to projects that have a non-trivial overlap between commit-issue links and PR-issue links so that a translation between the two is at least possible; some projects that use a PR-based workflow do not link commits at all. We allow RCRep to run in an offline manner and collapse all suggestions for the same PR into a single list. We then order RCRep's suggestions by the output probability from its Random Forests classifier, and break 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.

The comparison between A-M and RCRep is limited to Java projects and evaluates the relative effectiveness of A-M's use of PR-level information versus RCRep's reliance on Java commit summaries from the ChangeScribe summarisation tool. The comparison is made over the first corpus described in **??**. The second experiment evaluates the generality of Aide-mémoire across projects of different sizes and programming languages using the second corpus. We rely on comparison to the initial linking knowledge extracted through heuristic linking as detailed in **??**.

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 pa-

per, and 128 for A-M, as discussed in **??**; the separation criteria was entropy for the RCRep, while Mondrian Trees employ a budget to decide when they should refine the partition of the space **?**. 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 in accordance with the original paper.

2.5.1 Measuring Performance

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 accuracy*, 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-s *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).

$$AP(q) = \frac{\sum_{k=1}^{k=\infty} P_q(k)\text{rel}_q(k)}{|\{k \mid k \in \mathbb{N} \wedge \text{rel}_q(k) = 1\}|}. \tag{2.6}$$

Average Precision can be interpreted as the average of the precision at each possible recall for a given query q. To measure the quality of suggestions across all queries in Q, we use Mean Average Precision **?**, which measures the quality of a set of ranked lists of suggestions.

Definition 1. *Mean Average Precision (MAP)*

$$\frac{1}{|Q|}\sum_{q\in Q} AP(q). \tag{2.7}$$

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 our tool during the replay of repository events in the validation suffix.

MAP alone is insufficient for our needs, as providing links that are irrelevant imposes extra cognitive load on the developer, so it is better to have few, high-quality suggestions or no suggestions at all, rather than many low-quality suggestions. MAP does not measure such appropriate empty responses, because $AP(q)$ is undefined for $|r_q| = 0$, which is exacerbated by our use of 'no_issue' and 'no_pr' to create dynamic cut-off points for our suggestions. We denote the set of queries where MAP is not applicable by:

$$Q' = \{q \mid \forall k \in \mathbb{N} \cdot rel_q(k) = 0\}. \tag{2.8}$$

That is the set of all queries where there are no relevant artefacts to predict. Thus, we define list accuracy.

Definition 2. *List accuracy*

$$\frac{1}{|Q|}\Big(\sum_{q\in Q-Q'} AP(q) > 0 + \sum_{q\in Q'} |r_q| = 0\Big). \tag{2.9}$$

We use list accuracy as a proxy for impact on developer workflow. We seek to measure not only when our system correctly predicts any link, but also when it correctly determines that it should not predict anything at all. This penalizes A-M for predicting links when there are none and $AP(q)$ is undefined. This definition represents an uplift of the definition of accuracy from the binary classification scenario to lists where positive examples require at least one match and negative examples

Table 2.4: 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**

require no suggestions for them to count as true positives and true negatives respectively. Hence false positives and false negatives become respectively suggesting a non-empty list when no links exists and suggesting a empty list when at least one link exists.

To obtain Precision, Recall and F1 scores for our tool, due to the list prediction nature 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:

$$p = \frac{\text{tp}}{\text{tp}+\text{fp}}, \tag{2.10}$$

$$r = \frac{\text{tp}}{\text{tp}+\text{fn}}, \tag{2.11}$$

$$\text{f1} = \frac{2pr}{p+r}. \tag{2.12}$$

2.5.2 Corpora

We evaluate across two corpora:

Table 2.5: 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**

1. A Java corpus containing a total of 47 repositories of varying sizes.

2. A corpus containing a total of 213 repositories using a variety of programming languages.

Summary statistics of the corpora are given in **??** and **??**; the SQL queries used to select projects are available online **?**. The two new corpora were designed to evaluate the generality of our approach across a range of repository sizes and different languages. When developing A-M, we also employed a separate 'internal dev' set of repositories, which we exclude from our evaluation. 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 for the purpose of exploratory data analysis and feature selection as detailed in **??**.

Previous work used an Apache Commons dataset first provided by Bachmann *et al.* **?**. This corpus is unsuitable for our evaluation due to the lack of Pull-Request data associated with the commits. Recovering such associations is non-trivial as the corpus is collected from JIRA, and Pull-Request, when used for Apache projects, are kept in GitHub. Further, the corpus is considerably smaller, and due to using only Apache projects, introduces biases in the dataset. We only use this Apache

Commons dataset for sake of completeness when assessing the quality of our reproduction of RCLinker **?**.

The repositories contained in our new corpora are sampled from the GitHub GHTorrent dataset **?**. We exclude projects that have less 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 **?**. 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.

2.5.3 Heuristic Linking

We crawl GitHub to retrieve PRs, issues, commit messages, and diffs for each project. For a notion of initial linking knowledge, we perform heuristic linking by recording explicit links provided by GitHub metadata, as well as checking issue and PR discussions, and commit descriptions that contain either SHA-like (r'[0−9a−f]\{5,40\}') tokens or numerical (r '[\ textbackslash n\ textbackslash r\ textbackslash s tokens that can be disambiguated to a unique artefact in the project. We assume that unlinked PR-issue pairs are negative examples, and linked pairs represent positive examples. This assumption allows us to assess tool performance over large corpora

that are not amenable to manual labelling. As we do not have a manually annotated corpus, we expect some of the negative examples in our corpus to be false negatives, links missed by developers, as well as some true links to be false positives. To mitigate this concern, 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 have removed these links and, after performing the sampling and assessment again, found more than 80% of the filtered links recovered by the heuristics to be correct. We consider this sufficient to train a classifier for most projects, despite the majority of PRs lacking a link as seen in **??** and **??**, and expect a performance improvement over time in practice as the quality of the repository's collective memory improves. We have, however, excluded 72 projects from our corpus after heuristic linking as they were found to have less than 25 recorded links, which would lead to a performance metric computed over at most 5 queries.

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

2.5.4 User Study Protocol

We perform a user study to assess the impact of using our tool on developer workflow. 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 list accuracy of 0.66). We recruited participants by convenience sample from within our Computer Science department, and a total of seven participants chose to participate. They were asked to recover links for the selected examples that we artificially elided. The control group was instructed to use only the GitHub search tool, while the experiment group was asked to perform the task using A-M first and fall back on using GitHub search if they found it necessary. This was intended to mimic how we would expect our tool to be used in the wild.

We measure time taken per linking task in the presence and absence of A-M. Since we cannot 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. As a participant undertakes a task, we count the number of search queries performed and the time taken to perform each linking task. We use the number of search queries, which were usually observed to represent refinements of the original query, as a proxy for the difficulty of the task. The baseline for this is obtained from the control group. The results for time taken are summarised in **??** with a further discussion presented in **??**. After the task, we ask 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.

2.6 Evaluating Aide-mémoire

TODO: Talk about prelim study Our evaluation is composed of two stages: first, we demonstrate that our system outperforms a state-of-the-art offline commit-issue linking tool in solving issue-PR link prediction. Second, we evaluate the ability of A-M to generalise across a wide range of programming languages. Finally, we assess how well our tool integrates into the developer pull-request workflow, in particular as a more efficient way of performing linking compared to the built-in search features of the issue tracker.

2.6.1 The State of Issue-PR Linking on GitHub

To determine the state of linking practice in the wild on GitHub we first queried via GHTorrent **?** 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 have found only around 4.7% of all the projects available via GHTorrent to have 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 have been 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 the linking practice (32 out of 128), and around 43.5% (47 out of 155) of the popularity biased sample. This suggests that the majority of projects on GitHub to not require the maintenance of 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 can bias our presented results. Nevertheless, the difference between he uniform and the popularity biased sample suggest 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, Microsoft among the ones 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 Pull Request reference an already open issue in order for the submission to be accepted.

2.6.2 Comparing A-M and RCRep

?? shows detailed results. Overall, Aide-mémoire clearly outperforms RCRep and RCRepPR across the Java corpus. In terms of precision, RCRep achieves its best result on twitter / distributedlog , which represents a large and well linked project; in terms of F1-Score, it achieves its best result on the similarly well

Table 2.6: Mean performance values of predicting issue-PR links across a sample of the Java Corpus for RCRep, RCRepPR, and A-M. The approach taken by RCRep to solve the commit-issue link prediction 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**			**Mean Recall**			M
	RCRep	RCRepPR	A-M	RCRep	RCRepPR	A-M	RCRep
.../ijkplayer	0.00	0.00	**0.52**	0.00	0.00	**0.06**	0.00
.../AndroidSwipeLayout	0.00	0.00	**0.99**	0.00	0.00	**0.10**	0.00
.../fresco	0.01	0.01	**0.92**	**0.51**	**0.51**	0.37	0.03
.../redex	0.06	0.16	**0.76**	0.47	0.46	**1.00**	0.10
.../libphonenumber	0.01	0.00	**0.83**	**0.50**	0.20	0.26	0.03
.../android-classyshark	0.00	0.33	**0.99**	0.00	**0.72**	0.54	0.00
.../flexbox-layout	0.48	0.29	**0.79**	**0.79**	0.75	0.46	**0.60**
.../gson	0.06	0.07	**0.92**	**0.48**	0.36	0.35	0.10
.../EventBus	0.00	0.13	**0.65**	0.00	0.20	**0.45**	0.00
.../java-design-patterns	0.06	0.17	**0.89**	0.19	**0.22**	0.06	0.09
.../butterknife	0.00	0.08	**0.73**	0.30	**0.64**	0.30	0.00
.../RxJava-Android-Samples	0.41	0.29	**0.71**	0.50	0.43	**0.59**	0.39
.../aUPTR*	0.00	0.00	**0.99**	0.00	0.00	**0.25**	0.00
.../MaterialDrawer	0.04	0.06	**0.93**	0.53	0.58	**0.79**	0.07
.../Hystrix	0.04	0.28	**0.71**	**0.43**	0.32	0.29	0.08
.../AUIL*	0.00	0.00	0.00	0.00	0.00	0.00	0.00
.../RxAndroid	0.03	0.20	**0.86**	0.15	**0.80**	0.50	0.05
.../BottomBar	0.01	0.07	**0.86**	**0.28**	0.04	0.22	0.02
.../leakcanary	0.09	0.19	**0.88**	0.24	0.30	**0.35**	0.13
.../picasso	0.00	0.04	**0.78**	0.27	**0.55**	0.35	0.01
.../distributedlog	0.95	0.64	**1.00**	0.10	0.10	**1.00**	0.18
.../Small	**0.01**	**0.01**	**0.01**	**0.17**	0.16	0.01	0.01
.../zxing	0.15	0.11	**0.72**	**0.99**	0.96	0.20	0.25
Overall	0.10	0.14	**0.76**	0.30	0.36	**0.37**	0.09

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 as an instance of 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 list accuracy 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 RCRep on the original commit-issue link prediction problem; we provide these results in **??**. 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.

2.6.3 Evaluating Aide-mémoire

We evaluate A-M on a multilingual corpus containing 213 projects written in six programming languages. 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:

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

Table 2.7: 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	Acc	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

2. Is our system biased towards a particular programming language or project size?

3. What is the mean quality of our suggestions?

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

5. What is the impact of the Mondrian Classifier?

6. What is the impact of the training data quality on A-M?

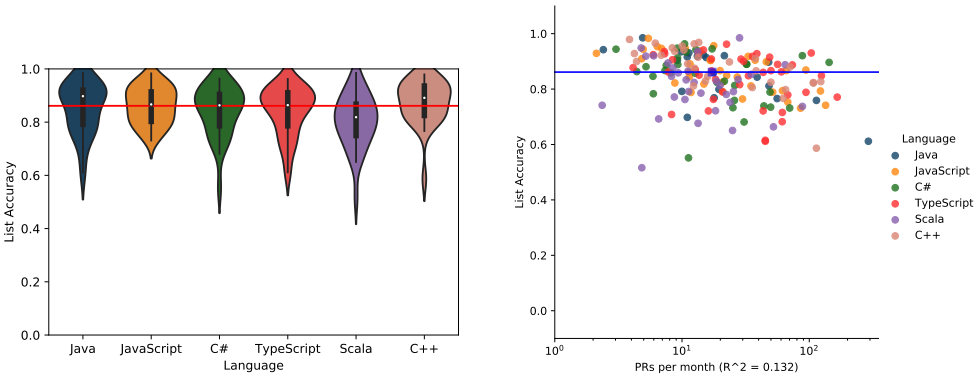
A-M uses 'no_pr' and 'no_issue' entities to truncate its suggestions. If these special entities appear in the k suggestions, we truncate the suggestion list at that

Table 2.8: 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	Acc	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

Table 2.9: Two-sample t-test of per language results against the results for all other languages. Only Scala for our list accuracy metric has a statistically significant difference with $p = 0.001$.

Language	MAP		List Accuracy		Missed by PR Subitter	
	T-value	p-value	T-value	p-value	T-value	p-value
Java	1.50	0.137	0.66	0.507	-0.91	0.367
JavaScript	0.49	0.627	1.52	0.133	-1.67	0.098
TypeScript	-0.64	0.525	1.51	0.133	-0.72	0.475
Scala	-0.73	0.467	-3.38	**0.001**	1.84	0.069
C#	-0.31	0.756	0.175	0.861	0.15	0.882
C++	-0.24	0.808	1.52	0.132	1.36	0.176



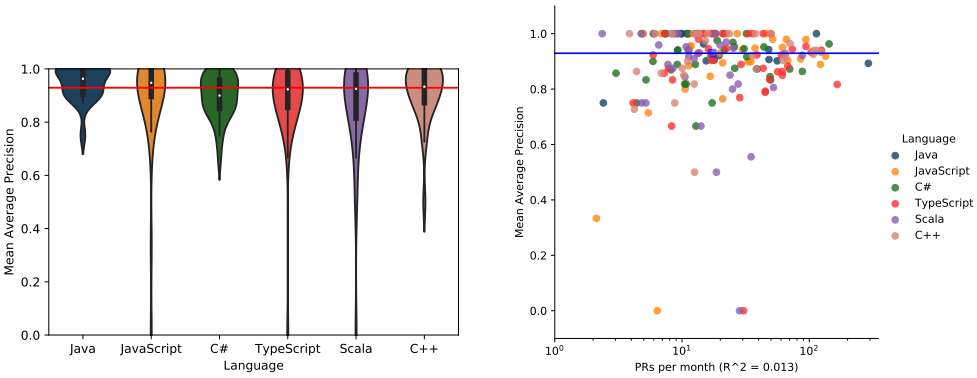
(a) Correct answer in top $k = 5$ or appropriately staying silent; there is no statistically significant difference between languages, except for Scala where we find that there is an effect size of $T = -3.38$ with $p = 0.001$ using a two-sample T-test.

(b) Correct answer in top $k = 5$ or appropriately staying silent as a function of a project's number of PRs per month, median result as a blue square.

Figure 2.5: Performance of A-M as 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 $k = 5$. In **??** we can see that the project language has negligible impact on performance and the median stays consistently close to 0.86 (the red line), while **??** allows us to 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.

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 accuracy*, defined in **??**) . 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 $k = 5$). **??** shows a more detailed view for $k = 5$.

A-M generalises across languages and to large project sizes. **??** and **??** summarise its performance while **??** shows a uniform sample of the results. The MAP is consistently near the median result of 0.95 across languages and in **??** we can indeed see that there is no statistically significant deviation. The results for list accuracy show a similar story: no deviation across languages from our general result



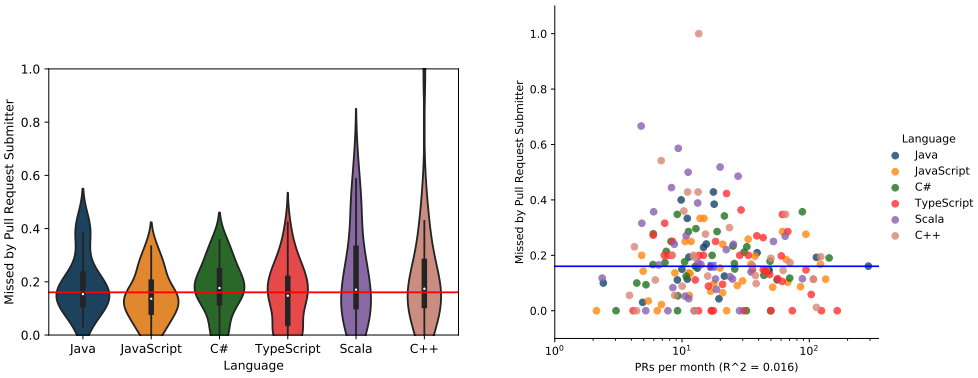
(a) Mean Average Precision across languages. The outlier results present in some languages represent projects where we were unable to learn a prediction model and were not filtered by our number of links requirement.

(b) Mean Average Precision as a function of the number of PRs per month for a project, median result as a blue square.

Figure 2.6: Performance of A-M reported as Mean Average Precision, which we use as a measure of the quality of our suggestions. We observe the median result by language to be near 0.95 in **??**. There is no statistically significant trend with language (as checked using a two-sample T-test) or with scale.

of 0.86 with the exception of Scala, which that has a small but statistically significant deviation. 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.

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 the topic branch, *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



(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 2.7: 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.

suggestions to the submitter. A more detailed breakdown of such cases can be seen in **??**.

A-M augmented to use a Random Forest classifier, which would additionally require batched retraining periodically and cannot benefit from a live feedback loop, 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 list accuracy and recall, showing a recall of only 10%. Detailed results can be seen in **??**. We hypothesise that class imbalance, which we observed Mondrian Forests to deal better with, and the lack of a feedback loop during validation impacted the performance of the Random Forest based implementation.

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

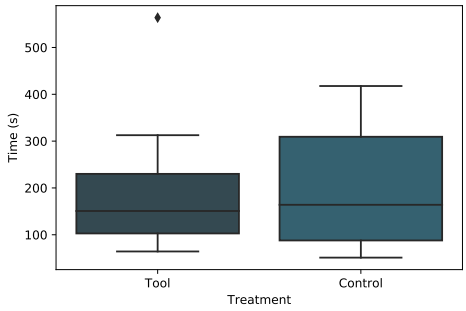


Figure 2.8: Time taken by participants to perform a linking task when using our tool and falling back on GitHub search versus using GitHub only. There is no statistically significant difference in average time taken ($p = 0.41$ according to a Mann-Whitney U test), however, we see that the variance of the time taken per linking task shrinks when using the tool.

from no noise (0%) up to 95% noise. We run this experiment the same internal development corpus as our Feature Selection detailed in **??**. We perform 10-fold cross validation at each noise ratio validating performance using 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 4 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.

2.6.4 User Study

For our user study, the participants that joined had on average more than 5 years of experience programming, if we include programming done as part of their academic degree; however, mostly 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. Regarding Pull Requests, a bit more than half had 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, 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 **??**.

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 keyword from the title and the body of the pull-request. Subsequent criteria differ by participant, but most used the developer and reviewers 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 **??**.

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 have additionally asked for feedback on how we may improve the prototype we presented. A common feedback was to improve the responsiveness of the prototype, one participant suggesting 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 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.".

2.6.5 Threats to Validity

Sampling problem instances from a population always runs the risk of biasing our sample towards our proposed solution, while full uniform sampling may yield low quality samples from a source such as GitHub. We guard against this by sampling instances while biasing on variables not used in the evaluation, and separating the internal dev set used for feature exploration from the repositories used to evaluate the efficacy of Aide-mémoire. By maintaining a separate internal dev set, we also run the risk of underperforming in the final evaluation if it is unrepresentative of the wider corpus, but we accept this potentially suboptimal performance in order to minimize the risk of overfitting. We select our projects from a very large population of 3704251 repositories in total **?**, biasing selection only to focus on more popular projects (as a proxy for quality), exclude extremely trivial repositories under 100 lines of code, and select a range of project sizes by taking a stratified sample. We deliberately avoid the use of key variables such as the number or frequency of PRs when constructing our sample.

While previous studies such as Kalliamvakou *et al.* **?** explain the pitfalls of using GitHub as a data-source for software project, we believe we sufficiently guard against most of them. We require that our considered projects contain code files, and our bias towards popular projects has the side-effect of favouring more active projects as well. We are, however, susceptible to Kalliamvakou *et al.*'s 'Peril VI': 'Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.'. We cannot bias our sample on Pull Request statistics since this might in turn bias our results in a difficult to account for manner. Further, Kalliamvakou *et al.* also detail promises of using GitHub data, chief among them for A-M is 'Promise II: The interlinking of developers, pull requests, issues and commits provides a comprehensive view of software development activities.'.

In order to motivate our approach, we demonstrate that traditional methods that solve the commit-issue linking problem do not transfer well to issue-PR links. Making a comparison is difficult, because it requires the adaptation of commit-issue linking tools to work with PRs. We have made a 'best effort' attempt here,

and RCRep's poor performance on issue-PR links should not be seen as indicative of anything other than its design as a tool for a different task. Indeed, the offline commit-issue linking solution is complementary to the online issue-PR linking tool as it can be used to fix the training data for A-M, which then continues to maintain and improve the quality of linking.

We are facing a seminal collection problem. Because we are interested in operating at scale (for deployability reasons), we want to show that we can bootstrap from existing data. There are two choices: reusing existing, known to be golden, datasets at the risk of overfitting to them or do what we did: synthesis by randomly perturbed existing data in order to work on fresh data from large, real world systems (albeit imperfectly due to dropping links and a lack of a ground truth). This creates a spectrum between a small, curated dataset that might not be representative of the wider systems and raw data from the wild that, if unfiltered, is trivially representative. To slightly tame the wild data, we did perform manual validation of the heuristically obtained knowledge of linking. Which end of the spectrum is desirable is a function of your experimental goal — we care about across-language realism which placed us in the PR and online realm, hence requiring us to obtain our data by synthetic link breaking.

We constructed the initial knowledge 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. If these two types of links have significant statistical differences in the feature space employed, then we risk learning to suggest only those links that a developer is able to construct. However this approach enabled us to automatically evaluate A-M over a large number of projects, protecting against overfitting and demonstrating generalisation across languages. Even if these two types of links are statistically different in the feature space, suggesting the types of links that humans can identify remains a useful aid to maintaining traceability — not only does it reduce the effort required from a PR submitter, it also saves effort within the change review process, where we found

16% of PRs are subsequently linked.

Additionally, 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 have 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.

Perhaps the most critical threat to the (external) validity of this work is whether the results will generalise to other projects beyond our corpus, and even beyond the population we sampled from. We chose a selection of projects of different sizes and languages in our sampling. A-M exploits information in issues, PRs, and a project's source tree to infer missing links, and will not generalise to projects that do not use PRs or change review processes. We make these assumptions explicit and focus our evaluation on projects that make this data available, which was the majority (93%) of projects in our corpus. A more open question is whether closed-source projects will be amenable to our approach: whilst PR-led development is ideal for OSS, we did not evaluate on any closed-source projects. Nevertheless, Kalliamvakou *et al.* **?** remark that companies are adopting workflows more similar to OSS by using GitHub for their commercial projects, specifically workflows based on branching and pull-requests.

TODO: External threat to be merged in As the system evolves, it succumbs to concept drift **??**, that is to say both the number of tokens that the system designates as unknown in the tf-idf model grows and the statistical properties of the links change in arbitrary ways; the system's usefulness decreases. We use tokens marked as unknown as a proxy for concept drift. We are able to quantify this proxy for concept drift by monitoring the rate at which tokens are marked as unknown; this measure can be used to decide when retraining is desirable, such as over a major release cycle. A preliminary study indicates that there is exploitable signal that can be detected by ChangePoint analysis, however, fine tuning the alarm derived from this analysis requires careful consideration of the hyperparameters, which is left as

future work.

2.7 Related Work

TODO: Merge in and soften We also other tools, FRLinker and PULink, but we were unable to reproduce the reported results and our attempts to contact the tool's authors went unanswered. Rath *et al.* **?** requires intrusive instrumentation. BfLink by Prechlelt and Pepper **?** requires a human in the loop for the linking process.

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 **??**, or automatic patch generation tools such as R2Fix **?**. 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, then summarise developer-centric work that solves the related commit-issue linking problem retrospectively.

TODO: Work in the below paragraph or see if we drop. In their work, Kalliamvakou *et al.* **?** detail both the promises of GitHub as well as assumptions made by researcher that may not necessarily hold. A promise of interest to us in particular is that of a unified history of software development that includes the Version Control system as well as the Issue Tracker and PRs. This has the potential to link Natural Language discussions to code changes. We, however, observed that the linking practice is not as high as desirable although tools to support it do already exist. To this end, we wish to provide a tool better integrated into the developer workflow which we realised in A-M.

2.7.1 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 devel-

oper artefacts, such as source code, pull-requests, commits, and issues, but do not focus on them.

TODO: Merge in Traceability links are often not recorded directly and recreated on a need by basis **?**. Indeed, agile practice suggests travelling "light" **?**. When finishing a task, developers have that task in mind, not its administrative details, like PR ids (when closing) or issue ids (when submitting). Bringing these to mind, to inter-link PRs and issues requires a context-switch, so developers prefer start work on the next task. A-M reduces the cost of this context-switch.

Software traceability seeks to infer *traces* (*i.e.* links) between these heterogeneous artefacts **?**. 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 **???**. 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 **?**, 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. An excellent survey detailing the use of such methods within traceability has been done by Borg *et al.* **?**. Further, Mills *et al.* **?** propose to reduce the human effort required by IR based techniques by adding a classification step after the recommender that filters the suggestions.

In contrast, A-M exclusively focuses on the PR-issue link inference problem. Version control and issue tracking are almost ubiquitous in modern software development. PRs and issues are plentiful, well-suited for machine learning which is data hungry, and their format is well-known. Thus, unlike more general traceability tools, A-M does not have to contend with missing artefacts or a profusion of formats. Solutions to issue-PR linking can exploit the structure encoded in PR meta-data; **??** details how A-M uses metadata and infers semantics from source-

code. We designed A-M to seamlessly integrate into modern development practice. As **??** details, A-M suggests links when a developer closes an issue or submits a PR, when this information is pertinent, without intrusive instrumentation of developer tools and the attendant privacy and deployability concerns. In contrast, when general traceability approaches, such as that of Cleland-Huang *et al.* **?** or Neumuller and Grunbacher **?**, are applied to issue-PR linking, they must either rely solely on generic features, employ a issue-PR linking solution as a subsystem, or attempt to heuristically infer the structure of a PR; the latter may involve intrusive instrumentation of a developer's working environment, which A-M does not require.

Asuncion and Taylor **?** use record-replay and hypermedia to tackle software traceability. Their online solution cannot employ offline information retrieval techniques (such as VSM, tf-idf or topic modelling) commonly used in retrospective approaches to traceability **?**. 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 **??**. 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.

Asuncion *et al.* **?** 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. A-M does not rely on LDA and thus scales better because it does not require per query model retraining. A-M's focus on issue-PR links means that it can extract rich feature vectors from PRs and issue inputs, as we detail at the close of the next section.

Ståhl *et al.* **?** propose a light-weight framework on top of continuous integration systems that can provide traceability at different levels of granularity by asking systems to emit specifically formatted JSON messages. This framework, realised

as Eiffel, can serve as a unifying system for different, otherwise heterogeneous systems, to provide an account of the progress made towards deployment. They validate the usefullness of such a system by conducting interviews with developers that use and those that do not use Eiffel to determine if the features provided by it indeed aid them towards delivering better code. The core idea and principle is to make traceability low-cost enough to be viable under agile practices. A-M could integrate with Eiffel by providing messages in the required format indicating new links as developers select them within it.

Additionally, Falessi *et al.* **?** have done work towards quantifying the number of links that are left to be recovered, *i.e.* measuring $\hat{E} - E$ in our formulation. 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. Extending their work to the online setting is a non-trivial matter and beyond the scope of this work.

2.7.2 Commit-Issue Link Inference

TODO: Merge the next two paragraphgs Common practice on GitHub, as observed from "CONTRIBUTING.MD" files, suggest having an Issue for all changes and linking to this issue from the PR or commits within a PR. Enforcing these rules often is costly in the absence of automatic checks.

Further, we also observe the agile practice of spring cleaning a backlog. During a backlog refinement, developers would reconsider outstanding user stories, reprioritise and re-estimate them while removing stale stories. In the extreme case where all stories are stale, all artefacts are lost; not just links, but Issues, Feature requests, user stories, are all discarded in favour of starting the next sprint from a clean slate **?**. This practice hides information that could be beneficial both for developers and researchers. For example, the reduced documentation that is a consequence of this practice increases onboarding costs, while for researchers it hides patterns that could be used to help the development process, such as fault localisation.

Emerging from research into task-aware developer tools **?**, Kersten *et al.* were the first to connect source code to tasks **?**. To this end, they instrumented Eclipse

to infer a developer's task, then recorded the files accessed while the developer worked on that task. Over the resulting data they trained a model to infer file-issue links. This formulation is inherently online and open-world, as developers create tasks and files or finish tasks and remove files. In terminology not used within the paper, Kersten *et al.* were the first to solve the software traceability problem in an online fashion for two specific software artefacts: files and tasks. Their task inference is imprecise and coarse-grained when compared to an issue tracker, and files are much more coarse-grained than PRs or even commits; subsequent work leveraged issue trackers and considered commits to directly address the commit-issue link inference problem. To date, this later work has tackled the commit-issue link inference problem retrospectively, without considering solving the problem at the PR level.

Bachman *et al.* were the first to formulate and quantify the missing link problem as the offline prediction problem of inferring missing commit-issue links given a version history and issue tracker archives **??**. 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. Wo *et al.* measured the similarity of change logs and bug reports with cosine similarity on tf-idf vectors, learning a threshold for true links, and proposed ReLink, the first technique for solving the missing link problem **?**. Nguyen *et al.* exploited commit and issue tracker metadata in MLink **?** to improve recall over ReLink. They evaluated their tool on the Apache Commons corpus, which became a de facto standard benchmark in that area. However, their use of a golden corpus has been argued to be unsound in practice due to the reliance on such a corpus, which negates the need for a solution. In BFLinks, Prechlet and Pepper **?** provide a framework for linking and propose two candidate generators (based on bug and commit IDs respectively) and a series of filters to reduce the candidate set of links, emphasising the importance of looking at the traceability links

bi-directionally. They also argue that previous work, ReLink, required an unsound tuning of filter parameters that require a golden set of links. Li *et al.* further improved recall in RCLinker **?**, the current state of the art. They trained a statistical classifier that uses cosine similarity on tf-idf vectors to undersample the training data. Their tf-idf based cosine similarity measure works better on natural language text than raw source code changes, and they therefore use ChangeScribe **?** as a preprocessing step to textually summarises code changes. PRs are not spared the bad or missing description problem. While the presence of multiple commits can aliviate this issue for A-M, PR summarisation and description generation methods have been proposed. Liu *et al.* **?** propose a tool based on a bi-directional RNN with a copy network to offer descriptions for PRs. They motivate their work from a contributor perspective, however, such work can aid A-M, and potentially other as well. More recently, Sun *et al.* **?** have tackled the issue of labelled data in the context of commit-issue linking by converting the problem to a semi-supervised learning problem and thus allowing the use of unlabelled data in the construction of classifiers.

Separately from the work in commit-issue linking, Rath *et al.* **?** consider traceability between issues and commits (feature implementation and bug fixes) and model the problem from two points of view: process and stakeholders. They deviate from the unigram based VSM of existing work in commit-issue linking opting, for a n-gram VSM, and are also the first paper in this area to perform feature selection. They delegate this task to the Weka implementation for feature auto-selection. They are quite thorough in their evaluation and consider multiple classifiers within the experimental set-up. Rath *et al.*'s work is difficult to compare with previous commit-issue linking research, as their solution uses features obscured or unavailable in an online setting.

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 **??** 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 would 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.

2.8 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 bundles, such as textual descriptions allow 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, does not require customisation of toolchains or invasive monitoring of developers activities; it was designed from the ground up to be deployable.

A future version could interact with Eiffel by Ståhl *et al.* **?** 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.

Chapter 3

Flexeme: Untangling Commits Using Lexical Flows

Paper Authors Profir-Petru Pârţachi, 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.

3.1 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 **??**. Time pressure is one reason. Another is that the boundaries between concerns are often unclear. Murphy-Hill *et al.* **?** 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.* **?** found that tangled changesets (commits) hamper comprehension and that developers need untangling (changeset decomposition) tools. Barnett *et al.* **?** confirmed this need. Herzig *et al.* **??** 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 **?** for multiversion analysis: they constructed a multiversion intraprocedural control flow graph and used it to determine whether a commit fixes all n versions **?**. 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 **?**.

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 **??** 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 **?**.

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

[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.

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`.

3.2 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 **?**. Further, as **?** found, tangled commits have a statistically significant impact on the performance of regressions methods used for defect prediction. **?**, 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. **?** exploited this by using def-use chains, which offer a short-range view of these connections. However, as shown in **??**, 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 **?** compared to **?**, 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 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 **??**. 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 **??** 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. **?** 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 **?**. While they augmented data-flow with lexemes, we augment PDGs and a description of how we achieve this follows.

3.3 Concerns as Lexical Communities

Given consecutive versions of some code, FLEXEME constructs their PDGs and overlays them adding name-flows **?** 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.

?? 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 **??**, **??** and **??**. We discuss the graph clustering approach in **??**.

3.3.1 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 3.3.1. 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 3.3.2. Multi-version Program Dependency Graph (δ-PDG) A δ-PDG for a range of versions $[p,q]$, represented as G^{p-q}, is the disjoin union of all nodes and edges across all versions in $[p,q]$. When $p = q$, G^p represents the δ-PDG at version p.

Definition 3.3.3. Name Flow Graph (NFG) FLEXEME's NFG is a graph $G = (N,E)$, such that the nodes in N are annotated with either a program statement or a conditional expression and contain variable, method or formal names. An edge (n_1,n_2) connects nodes $n_1,n_2 \in N$ if either: (1) the name associated with n_1 is on the RHS of an assign while the name associated with n_2 is on the LHS of the same assign; (2) the name associated with n_1 is the actual name of the formal name associated with n_2.

Definition 3.3.4. Multi-version Name Flow Graphs (δ-NFG) A δ-NFG for versions $[p,q]$ is N^{p-q}. It is the δ-PDG G^{p-q}, augmented with additional edges drawn from the name flows NFG^{p-q} *s.t.* $\forall i \in [p,q], (s,t) \in NFG^i \wedge (s,t) \in G^i \Rightarrow (s,t) \in N^i$.

Inspired by **?**'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 if they first appear in the new version. δ-PDGs retain nodes deleted across

the versions a δ-PDG spans. Deletion becomes a label. For unchanged nodes, we solve a matching problem between the nodes of the δ-PDG and the new version. We use string similarity to filter candidates and we use line-span proximity to rank them. We provide details in **??**. To integrate the new edges into the δ-PDG, we find the existing nodes that matched to the parents and children in the PDG of the new version. For nodes marked as deleted, we extend this label to edges flowing into them as well. We further detail this in **??**. Finally, to obtain a δ-NFG, we endow the δ-PDG with nameflow information for each of the versions considered by matching nodes using their line-spans.

3.3.2 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 3.3.5. *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* **?** 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.

3.3.3 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 **??**. 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.

3.3.4 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 **?** which builds on top of the Weisfeiler-Lehman graph isomorphism test **?**. 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 $l_0 : V(G) \to \mathscr{L}$, where $\mathscr{L}$ 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

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

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 ($\mathscr{G}$) as input, we define the $K(\mathscr{G})_{ij} = k(G_i, G_j), G_i, G_j \in \mathscr{G}$ to compute the matrix of pairwise kernel values.

Let $\mathscr{G}$ be the set of graphs over which we wish to compute graph based similarity, and let $K : \mathscr{G} \to \mathbb{R}^{|\mathscr{G}|} \times \mathbb{R}^{|\mathscr{G}|}$ be a graph kernel. Then the WL Graph Kernel

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

becomes: $K_{WL}(\mathscr{G}) = K(\mathscr{G}_0) + K(\mathscr{G}_1) + \ldots + K(\mathscr{G}_n)$;, where $\langle \mathscr{G}_0, \mathscr{G}_1, \ldots, \mathscr{G}_n \rangle$ is obtained by applying n steps of the isomorphism test to each graph in $\mathscr{G}$

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

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

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

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

For clustering, we opt for agglomerative clustering, like **??**. With node-neighbourhood pairwise similarity information, we can build an affinity, *i.e.* a pairwise distance, matrix for clustering trivially by simply inverting the value, *i.e.* $1 -$ similarity. **??** details the implementation.

3.4 HEDDLE

We discuss constructing a δ-PDG, then the clustering its changed nodes to disentangle patches.

3.4.1 δ-PDG Construction

We implement both nameflow extraction and PDG extraction in Roslyn **?**, 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 nameflows)

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 (**??**) 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 **??**.

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.

3.4.2 Graph Node Clustering

We use GraKeL **?** for the Weisfeiler-Lehman (WL) Graph Kernel **?** 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 **?**. 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.* **?**, *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].

[3] We consider code snippets at line granularity.
[4] `https://github.com/PPPI/Flexeme`

3.4.3 Deployability

HEDDLE takes ten seconds on average to untangle a commit (**??**), 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 **?** 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.

3.5 Experimental Design

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

3.5.1 Corpus Construction

To construct our corpus, we reuse **?** methodology who artificially tangle atomic commits. Therefore, we consider commits that:

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) **?**, 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. **??** 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 **??**. 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

Table 3.1: 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 3.2: 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

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 (**?**, Figure 7).

?? 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 **??**, the subsets on which we compared any two methods when 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.*.

3.5.2 Experimental Set-up

Our experiments assess how well our method recovers the original commits compared to the baseline methods proposed by Barnett *et al.* **?** and Herzig *et al.* **?**. 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:

$$A = \frac{\text{\#Correctly labeled nodes}}{\text{\#Nodes in graph}}. \tag{3.1}$$

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 **?** 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.

3.5.3 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 hypernodes 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 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 construc-

Table 3.3: 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.* ?**			**Herzig *et al.* ?**			**δ-PDG+CV**			**HEDDLE (δ-NFG + WL)**		
	2	3	Overall	2	3	Overall	2	3	Overall	2	3	Overall
Commandline	0.18	0.21	0.19	0.67	0.48	0.64	0.77	0.84	0.80	**0.82**	**0.92**	**0.82**
CommonMark	0.20	*	0.20	0.65	*	0.65	**0.90**	*	**0.90**	0.70	*	0.70
Hangfire	0.16	0.13	0.15	0.70	0.54	0.64	0.84	**0.88**	**0.87**	**0.86**	0.68	0.79
Humanizer	0.18	0.31	0.18	0.64	0.42	0.62	0.69	x	0.69	**0.83**	**0.57**	**0.81**
Lean	0.19	0.12	0.18	0.69	0.62	0.69	**0.84**	0.71	**0.84**	0.77	**0.82**	0.80
Nancy	0.09	0.08	0.09	0.70	0.56	0.67	**0.86**	0.80	**0.86**	0.81	**0.92**	0.84
Newtonsoft.Json	0.15	0.11	0.15	0.71	0.56	0.71	**0.86**	**0.69**	**0.82**	0.71	0.52	0.71
Ninject	0.14	*	0.14	0.57	*	0.57	**0.94**	*	**0.94**	0.80	*	0.80
RestSharp	0.12	0.14	0.12	0.71	0.69	0.70	0.74	0.53	0.70	**0.82**	**0.89**	**0.82**
Overall	0.14	0.11	0.13	0.69	0.62	0.67	**0.83**	**0.84**	**0.83**	0.81	**0.84**	0.81

tion time as we derive the DU-chains, call-graphs and dataflow-graphs from our δ-PDG.

3.6 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.* **?** and Barnett *et al.* **?**. 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.

3.6.1 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.

Table 3.4: Median time taken (s) of 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.* ?**			**Herzig *et al.* ?**			**δ-PDG+CV**			**HEDDLE (δ-NFG + WL)**		
	2	3	Overall	2	3	Overall	2	3	Overall	2	3	Overall
Commandline	**0.10**	**8.51**	**0.12**	10.5	18.56	9.26	0.42	153.5	0.59	0.85	182.6	1.04
CommonMark	**2.56**	*	**2.56**	1.96×10^3	*	1.96×10^3	10.38	*	10.38	14.95	*	14.95
Hangfire	**1.15**	**4.97**	**1.95**	1.30	5.57×10^4	1.72×10^4	0.61	123.9	3.84	8.06	45.29	11.64
Humanizer	**0.44**	**0.23**	**0.41**	40.62	49.53	44.44	9.24	x	9.24	4.86	2.56	4.58
Lean	**1.00**	**1.59**	**1.28**	345.0	573.5	388.3	19.28	17.08	19.28	18.07	24.07	18.23
Nancy	**2.06**	**5.63**	**2.42**	570.5	7.29×10^3	600.1	22.04	20.52	21.96	18.38	85.55	21.78
Newtonsoft.Json	**2.14**	**6.42**	**2.35**	225.6	210.7	230.4	20.04	51.54	20.32	8.01	11.98	8.58
Ninject	**1.25**	*	**1.25**	81.53	*	81.53	4.52	*	4.52	14.99	*	14.99
RestSharp	**0.74**	1.25	**0.78**	46.22	222.9	82.09	7.80	**1.01**	4.99	9.86	26.25	10.11
Overall	**1.02**	**5.02**	**1.41**	81.53	647.9	117.3	7.17	70.35	10.27	7.99	43.29	9.56

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 is 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 (**??**) 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 (**?**, §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. **??** details the threats to HEDDLE's validity this difference in methodology incurs. **??** further details their approach and evaluation and its differences to HEDDLE.

When considering accuracy for an increasing number of concerns (See **??**), 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.

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

3.6.2 Untangling Running Time

?? shows that Barnett *et al.* **?**'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 **??**, 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.

3.7 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 **?**. 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.* **?** 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. **??** 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 **??** and **??** detail. Thus, we were able to directly compare our work with theirs. Barnett *et al.* opted for a different evaluation strategy (**??**), 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.

3.8 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.

3.8.1 Impact of Tangled Commits

Tao *et al.* **?** 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.* **?** 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.* **??** 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 **?**. They find non-atomic commits significantly impact the accuracy of classification and regression tasks such as fault localisation.

3.8.2 Untangling Commits into Atomic Patches

It is natural to think of identification of communities in the δ-NFG as a slicing problem **?**. 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.* **??** 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.* **??** 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.* **?** 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.* **?** 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 nonatomic 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.* **?** take a more developer-centric approach and propose the EpiceaUntangler tool. They instrument the Eclipse IDE and use confidence vot-

ers 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.* **?**'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.

3.8.3 Multiversion Representations of Code

Related work has considered multiversion representations of programs for static analysis. **?** 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 **?** 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.* **?** 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.* (**?**) 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 shar-

ing 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. **?**'s representation is lower level than our δ-NFG, which is more readily usable for static analysis or even untangling tasks as demonstrated in this paper.

3.8.4 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.* **?** 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.* **?** 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.

3.9 Conclusion and Future Work

In conclusion, we propose a novel data structure, the δ-PDG, which represents a multiversion PDG. We show both how this structure can be used to perform commit untangling using agglomerative clustering with graph similarity as well as how it can be used to reproduce previous work in this area. We show that we improve 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.

While we used the δ-NFG to separate concerns in commit histories, these graphs by construction group together concerns. A natural application of our work is to apply the δ-NFG to separate concerns within a version. We believe that the δ-NFG can potentially springboard research into code refactoring. We intend to explore this space in the near future.

```
@@ -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 3.1: 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.

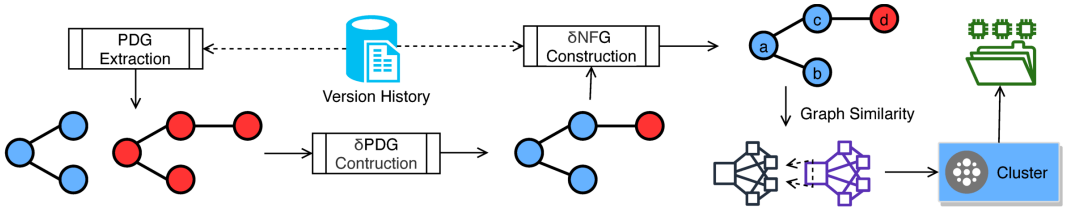


Figure 3.2: Overview of FLEXEME's δ-NFG construction and concern separation.

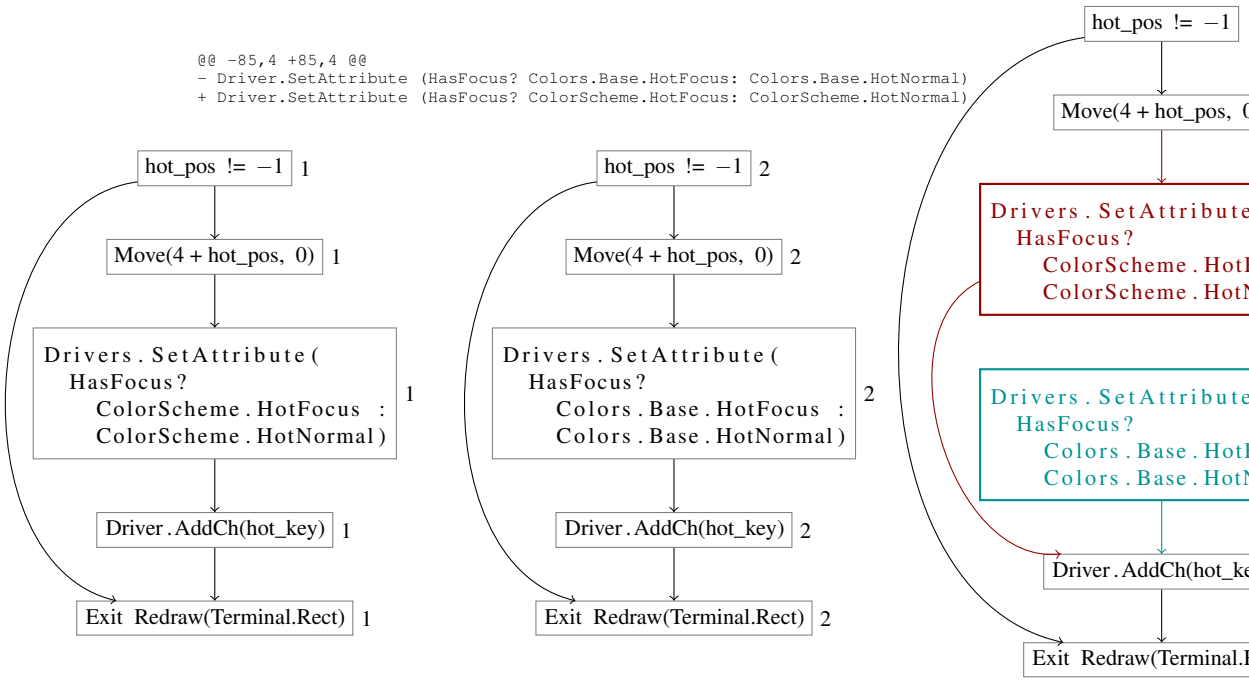
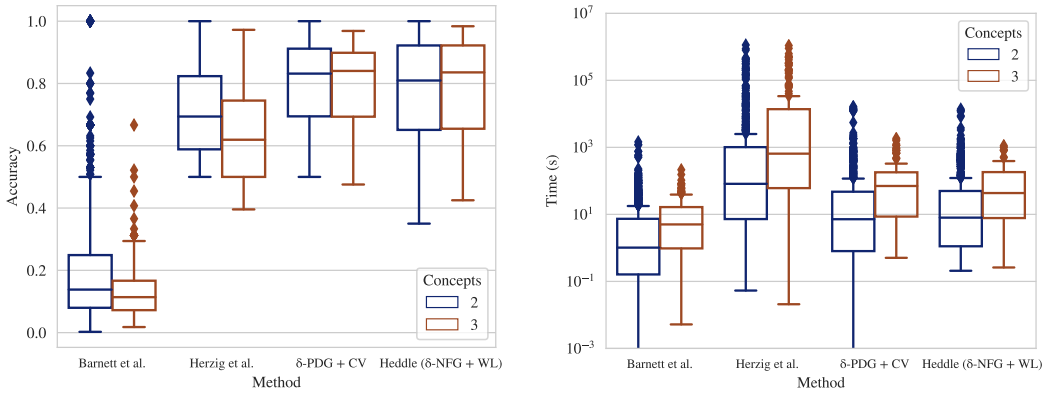


Figure 3.3: Construction of a δ-PDG from two versions — 1 and 2 — of a program. Version 2 is obtained from version 1 by application of the patch shown in the program. Each node is annotated with the version number it is present in.



(a) Accuracy (b) Runtime

Figure 3.4: Boxplot comparing the accuracy of the baseline and HEDDLE (**??**) as well as time taken (s) to segment a commit (**??**) 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).

Chapter 4

POSIT: Simultaneously Tagging Natural and Programming Languages

Paper Authors Profir-Petru Pârţachi, 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.

4.1 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 **?**, 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 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

[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.

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 **?** 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`.

4.2 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 sense of these documents automatically. An example is TaskNav **?**, a tool that supports task-based navigation of software documentation

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

Figure 4.2: 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.

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 **??**, 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 two additional task phrases: (pass in "dest + len") and (pass in "real + len"); we present POSIT's output on this sentence in **??**. 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 **??**.

4.3 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

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

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 **?**.

"*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)" (**?**, §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.

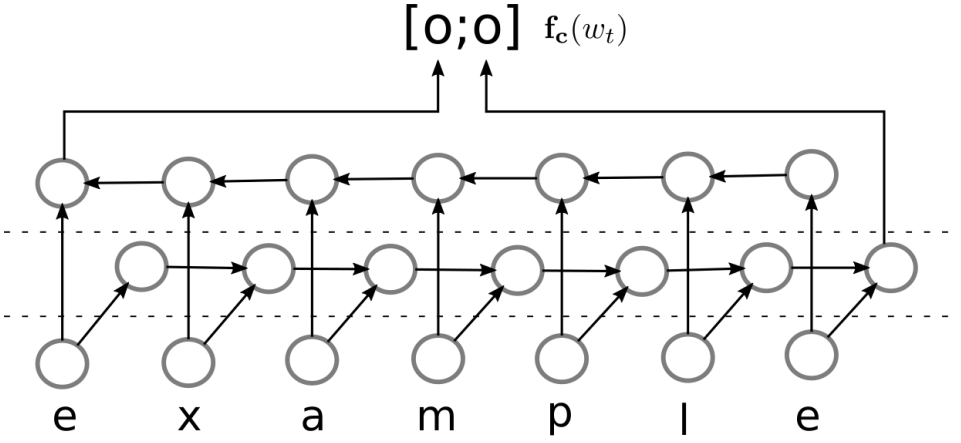
Case I: Morphological Adaptation Consider using affixation to convert foo/ class to foo−ify/verb to denote the action of converting to the class foo. In this case, foo−ify behaves as a bona fide word in L_1. Such examples obey the free-morpheme restriction mentioned above. This enables it to be a separate, 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.

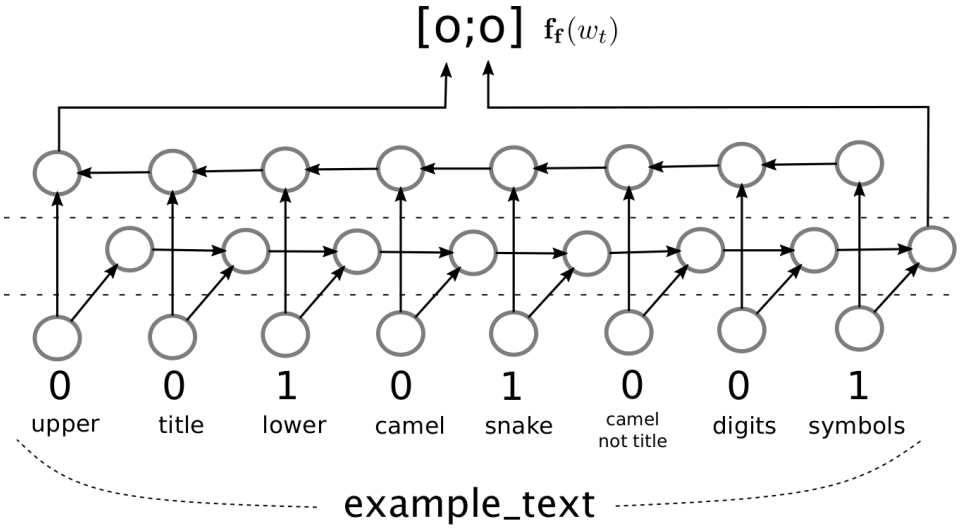
Case III: No Adaptation If no adaptation occurs, then the formal token occurs in L_1. This reduces to the second subcase of the syntactic adaptation case.

4.4 POSIT

POSIT starts from the biLSTM-CRF model presented in Huang *et al.* **?**, augments it to have a character-level encoding as seen in Winata *et al.* **?** and adds two learning targets as in Soto and Hirschberg **?**. **??** 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



(a) Character-level embeddings



(b) Feature vector embeddings

Figure 4.3: 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.

learn token morphology features beyond those that we may expose to it directly as a feature vector.

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

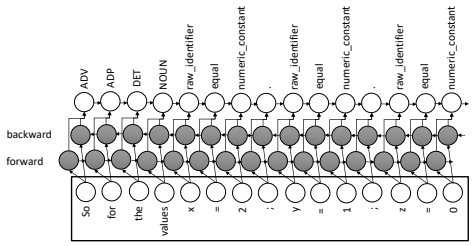


Figure 4.4: A representation of the neural network used for predicting English PoS tags together with compiler derived AST tags. The shaded cells represent LSTM cells, arrows represent the flow of information in the network. The top layer represents a linear Conditional Random Field (CRF) and the transition probabilities are used together with a Viterbi decode to obtain the final output. The first layer is represented by **??** and converts the tokenised sentences into vector representations.

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.* **?**. 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 mor-

phology feature vector did not significantly improve model performance (**??**).

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

$$\mathbf{x}(t) = [\mathbf{f_w}(w_t); \mathbf{f_c}(w_t); \mathbf{f_f}(w_t)], \tag{4.1}$$

$$\mathbf{h}(t) = f(\mathbf{W}\mathbf{x}(t) + \mathbf{U}\mathbf{h}(t-1)), \tag{4.2}$$

$$\mathbf{y}(t) = g(\mathbf{V}\mathbf{h}(t)). \tag{4.3}$$

In **??**, 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 **??**. This figure represents the internals of $\mathbf{f_c}(w_t)$ and $\mathbf{f_f}(w_t)$ from **??** 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 **??**. 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 **??**, 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 **??** and **??**, 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:

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\Sigma_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., K; \tag{4.4}$$

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

$$\mathbf{p}(\text{tag}_t \mid \text{tag}_{t-1}) = \text{softmax}(\mathbf{y}(t)), \tag{4.5}$$

$$\mathbf{p}(\text{l}_{\text{id}_t} \mid \text{l}_{\text{id}_{t-1}}) = \text{softmax}(\text{2LP}(\mathbf{h}(t))), \tag{4.6}$$

?? represents language ID transition probabilities, and **??** — tag transition probabilities. In **??**, 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 **??** and **??** the transition probabilities of two Markov chains. Indeed, we obtain the optimal output sequence by Viterbi-decoding **?**. While **??** 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.

4.5 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. **??** 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 **??**. presents along

Table 4.1: 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.

Corpus Name	Tokens		Sentences		English Only Sentences		Code
	Train&Dev	Eval	Train&Dev	Eval	Train&Dev	Eval	Train&
Stack Overflow	7645103	2612261	214945	195021	55.8%	57.0%	3
CodeComments	132189	176418	21681	8677	11.3%	11.0%	7
Total	7777292	2788679	236626	203698	51.7%	55.1%	3

with the English-code segmentation result.

POSIT implements the network discussed in **??** in TensorFlow **?**. It uses the Adaptive Moment Estimation (Adam) **?** 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. **??** uses this implementation for validation and **??** 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`.

4.5.1 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. **??** presents their

statistics.

The first corpus is the Stack Overflow Data-dump **?** 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 **?**. 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 **??**.

We built the second, CodeComment corpus **?**, 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.

4.5.2 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 (**??**).

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 **??**. For PoS/AST tagging, StORMeD exposes a 'parse' webservice. Given a Stack Overflow post with with correctly labelled code in HTML (**??**), 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

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

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 (**?**, §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

Table 4.2: 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	Language ID Accuracy	Tagging Accuracy	Mean Accuracy	T
Only token embeddings	0.209	0.923	0.568	
Only CRF	**0.970**	**0.926**	**0.948**	
Only feature vector	0.450	0.928	0.689	
Only character embeddings	0.312	0.923	0.617	
No CRF	0.409	0.913	0.661	
No feature vector	**0.966**	**0.924**	**0.945**	
No character embeddings	0.966	0.919	0.943	
All features	0.970	0.917	0.944	

that escape 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 (**??**). 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.

4.5.3 Model Ablation

POSIT depends on three kinds of embeddings — character, token, and token morphology — and CRF layer prior to decoding (**??**). 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.

?? 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 **??** 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.

4.6 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.* **?** start from a parallel corpus of Natural Lan-

guage 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 (**??**).

4.6.1 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 **?**. 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 **?**. 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.

4.6.2 TaskNav++

To demonstrate the usefulness of POSIT's code-aware part-of-speech tagging, we augment Treude *et al.*'s TaskNav **?** 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 **?** tag set; POSIT uses training data labelled with Universal tag set tags **?**. 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 (**??**). 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 **?**. 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 (**??**), 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.

4.7 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.

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

In **??**, we observe 91% tag accuracy for English and 66.7% tag accuracy for

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 4.5: 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. 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.

4.7.2 Threats to Validity

The external threats to POSIT's validity relate mainly to the corpora, including the noisy nature of StackOverflow data **?**, 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 **??**, 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 (**??**). 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 **?** 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 **??**). 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 **??**, 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 **??**. 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++ (**??**), we use a more coarse grained PoS tag set than the original TaskNav potentially reducing its performance.

4.8 Related Work

In software engineering research, part-of-speech tagging has been directly applied for identifier naming **?**, code summarisation **??**, concept localisation **?**, traceability-link recovery **?**, and bug fixing **?**. 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 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 **?**. The advent of social media, has reduced the stigma and provided code-switching data, especially text that mixes English with another language **?**. High quality datasets of code-switched utterances are now under production **?**. For the task of part-of-speech (PoS) tagging code-switching text, Solorio and Liu **?** 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.* **?**, working on an

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

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

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 **?** 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 **?**. POSIT builds upon their model extended with Winata *et al.*'s **?** handling of OOV tokens, as discussed in **??**.

Operating directly on source code (not mixed text), Newman *et al.* **?** 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.* **?** 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 **?**, 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 **?**. StORMeD produces heterogeneous abstract syntax trees (AST), 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 **?**. 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 **??**. 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 (**??**). POSIT advances the state of the art on these two tasks (**??**).

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.

4.9 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 **?** 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`.

4.10 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.

Chapter 5

General Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.