Imperial College London

MEng Individual Project Report

Imperial College London

Department of Computing

Mining Semantic Information from Software Development Artifacts

Author:
Joe Rackham

Supervisor:
Dr. Robert Chatley

June 14, 2021

Abstract

In this project we present an investigation into leveraging Natural Language Processing and Machine Learning for the automatic extraction of semantic insights from Software Development Artifacts. We survey existing techniques and propose extensions and modifications. We then construct a workbench for experimenting with these techniques on both data-sets and real projects.

In a Software Development Project both the project repository itself and the corpus of digital communication exchanged between team members are sources of information that could be useful to a developer working on a new task. However as these grow in-size it becomes intractable for a person to manually search through everything. We investigate to what extent we can use Artificial Intelligence to extract this information automatically instead.

Using the workbench we evaluate the extraction techniques to show how our methods are successful in extracting information that would assist a developer.

Acknowledgements

I would like to thank Dr. Robert Chatley for his support throughout this project. His guidance has been instrumental in producing the work that follows.

I would also like to thank my parents Chris and Kerry, and my siblings Josh and Jade for their love and encouragement throughout my education.

Lastly I want to thank Ellie, Roxy, Julie, Vicky and Jack for their friendship, camaraderie and willingness to be used as a rubber duck throughout University.

Contents

1 Introduction 4

2 Background 5
- 2.1 How Software Developers Work . . . 5
  - 2.1.1 How do Software Developers Communicate? . . . 5
  - 2.1.2 How do Software Developers Find Code? . . . 6
- 2.2 Vector Space Based Indexing . . . 7
  - 2.2.1 Basic Indexing . . . 7
  - 2.2.2 What Constitutes a term? . . . 7
  - 2.2.3 Weighting . . . 7
  - 2.2.4 Latent Semantic Indexing . . . 8
- 2.3 Mining Source Code for Semantic Information . . . 8
- 2.4 Mining Digital Communication . . . 8
  - 2.4.1 Social Dynamics in Communication . . . 8
  - 2.4.2 Identifying Individuals . . . 9
  - 2.4.3 Linking Emails to Source Code . . . 9
  - 2.4.4 Disentangling Slack Conversations . . . 10
  - 2.4.5 Clustering Emails . . . 11
- 2.5 Cognitive Load . . . 11
  - 2.5.1 Cognitive Load Theory . . . 11
  - 2.5.2 Cognitive Load in Software Development . . . 11
- 2.6 Existing Tools . . . 12
  - 2.6.1 Mylyn (formerly Mylar) . . . 12
  - 2.6.2 Kite and Kite Copilot . . . 12
  - 2.6.3 Facebook Aroma . . . 13
  - 2.6.4 Guru . . . 13
- 2.7 Discussion . . . 14

3 Survey of Techniques 15
- 3.1 Project Setup . . . 15
  - 3.1.1 Email . . . 16
  - 3.1.2 Slack . . . 16
  - 3.1.3 Git . . . 17
- 3.2 Experiment: Disentangling Slack Conversations . . . 17
- 3.3 Experiment: Source Code Topic Clustering . . . 18
  - 3.3.1 Extracting Vocab from Source-Code . . . 18
  - 3.3.2 Text Analysis for Topic Extraction . . . 18
- 3.4 Clustering . . . 19
- 3.5 Experiment: Individual Grouping . . . 20
- 3.6 Experiment: Linking Source Code to other artifacts . . . 20
- 3.7 Testing & Ensuring Correctness . . . 22

4 Application of Techniques 23
- 4.1 Message Untangling . . . 24
  - 4.1.1 Training and Evaluating . . . 24
  - 4.1.2 Applying to Author's Projects . . . 25
- 4.2 Topic Extraction . . . 26

4.2.1 Capturing Co-Edit Behaviour 26
4.2.2 Aiding Understanding 29
4.3 Individual Grouping 33
4.4 Linking Source Code to other Artifacts 33

5 Summary of Results **36**

6 Prototype Tool **38**

7 Ethical and Legal Issues **40**

8 Conclusion **41**
8.1 Contributions 41
8.2 Future Work 42

Chapter 1

Introduction

In large-scale software development, the information that would be necessary or useful to complete a particular task may be split over a range of platforms. The code-base itself is obviously a source of information but further advice or information might have been given over Slack, Email, Microsoft Teams or any of a number of digital communication tools. The programmer may need to consult the language documentation and, furthermore, completing the problem in a satisfactory way may mean following an internal set of rules disseminated through email or documentation. These problems are only amplified in organisations with heavy use of remote working or international collaboration where asking someone for a piece of information or an elaboration isn't as simple as walking across the office.

This places a strain on a software developer both in terms of their time and their cognitive load. Because there could be valuable information across all these sources it takes a long time before a developer feels abreast of a task they're working on [1]. It's estimated that 60% of software maintenance is spent on comprehension: understanding how code works, finding out who made previous decision and why they made them, and what practices any modifications need to be follow [2]. Furthermore, developers often switch between multiple tasks a day meaning this process has to be repeated frequently [3]. Keeping everything relevant in their head takes up mental effort that could be devoted to implementing a solution. The alternative is not to look at everything and risk missing out on time-saving advice or making a mistake that needs to be fixed later on.

The goal of this project was identifying and evaluating techniques for mining the semantic information contained within these artifacts with the goal of generating insights that would be useful to a developer. The report presents an investigation into several tasks under this banner including identifying communication artifacts about part of the source code and extracting topics useful in understanding a code-base. Techniques are presented to accomplish these tasks which leverage Natural Language Processing and Machine Learning. Furthermore, this report outlines the construction of a 'Workbench' to facilitate experimentation on data-sets and real repositories. The workbench is used to evaluate both established techniques and proposed extensions, modifications and alternatives.

Report Layout

Chapter 2 outlines some background research into how Software Developers work as well as established techniques for mining textual artifacts and communication. Chapter 3 outlines the construction of the 'Workbench' and details the full range of techniques contained within. Chapter 4 then details applying the Workbench to some of the authors past projects and contains an evaluation of the output produced. Chapter 5 summarises the results obtained from all experiments. A prototype version of a real tool built for developers which uses the methods discussed to provide useful insights is described in Chapter 6. Chapter 7 outlines how the work for this project was conducted in an ethical and legal way and Chapter 8 concludes the report with a summary of the work, its contributions, and some potential directions for future work.

Chapter 2

Background

This chapter outlines background research relevant to the project as well a summary of some existing tools available which aim of assisting providing insights automatically to assist software development.

2.1 How Software Developers Work

2.1.1 How do Software Developers Communicate?

Software development is a highly collaborative exercise, but a large proportion of this collaboration is not done face to face. The historical choice of remote, asynchronous communication was email but the rise of social media technologies as well as the feasibility of teleconferencing have shifted this.

The business collaboration platform, Slack is one of the most popular new communication options amongst software development teams. Founded in 2013, at time of writing Slack reports 790,000 software developers active daily [4]. Slack allows for communication within channels, that team members can join, leave or mute as they wish. However, Slacks specific popularity in the Software Development community is likely due to it's easy integration with other tools developers use, GitHub, Jenkins etc. Slack is especially popular in virtual teams where it provide a forum more informal conversations that can't take place face to face. [5, 6]

Coronavirus has lead to a drastic increase in remote working and software development teams have have relied on digital communication technologies to preserve collaboration. The effects of the pandemic will obviously abate with time but many spectators predict that it will lead to a wider acceptance of remote / flexible working in the future. [7, 8]

To help prioritise which platforms to investigate we wanted to know how communication was split up in modern development teams. Previous research we found was several years old or that focused on the adoption of a specific technology so, instead, we conducted my own survey of industry software developers. Data can be explored further via an Interactive Dashboard at: `https://serene-stream-63388.herokuapp.com/` [1]

Figure 2.1 displays the percentage of participants who had used various communication platforms. We can see that although Slack is used by a significant proportion of developers, email remains more widely used. Slack is however, a lot more popular than competitor, MS Teams. Unlike some previous work, teleconferencing and from face to face were treated a separate forms of communication. We see that, although only approximately 60% of participants had worked on a project with face to face communication. This is most likely a result of the COVID-19 Pandemic.

Figure 2.2 displays which of the tools participants felt were used most frequently. Here the shift away from email is easier to see; it accounts for the smallest proportion of votes despite it's wide

[1] The Web-App may take a couple of minutes to load

adoption. Around 45% of those surveyed said that Slack or Teams was where most of the communication took place.

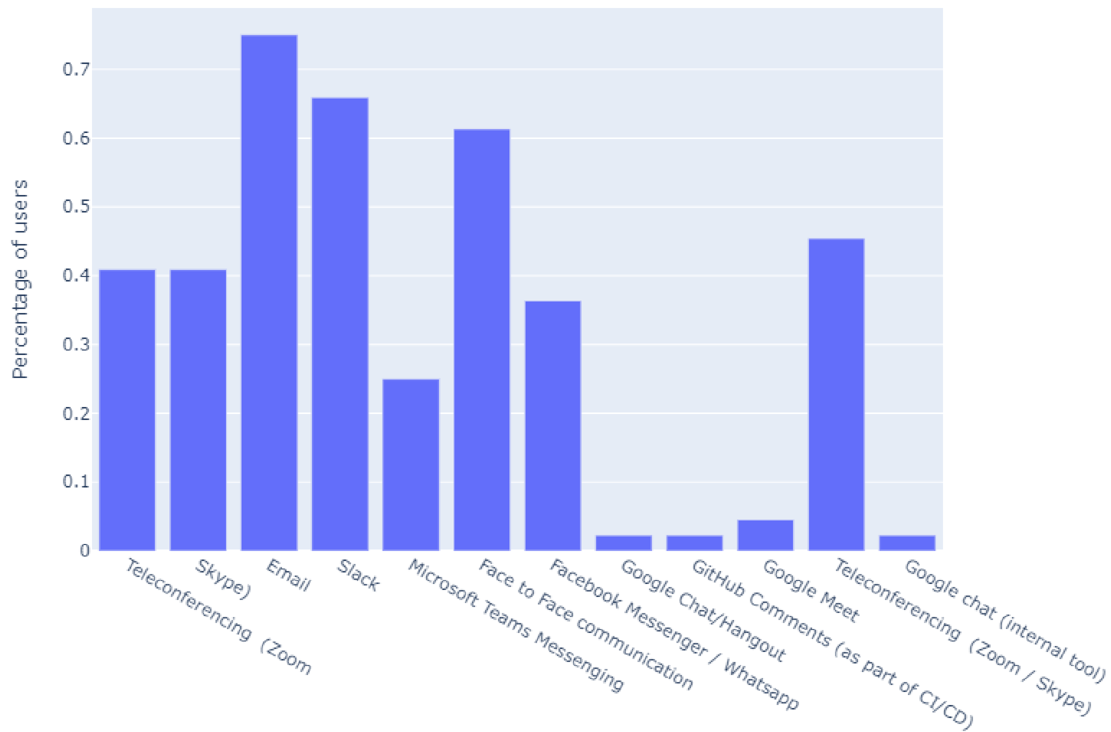

Figure 2.1: Percentage of Surveyed Programmers who had used various Communication Tools

2.1.2 How do Software Developers Find Code?

One of the most frequent developer activities is code search. Sadowski et al. present a case study into how Software Developers find code over a large number of projects at Google. The following insights are presented in the case study: [9]

- Searches were conducted for the expected reasons; finding out why code was behaving the way it was and where it was instantiated. However, users also used code search to answer questions about the meta-data like who was responsible for a certain part of the project.
- The most frequently identified reason for search was to find examples or sample code. Developers would likely benefit more from seeing documentation that displays the simplest use case and some usage tips.
- The developers observed had an average of 5.3 search sessions a day with an average of 12 individual searches. These numbers suggest an increase in the use of search when compared with previous research. A potential explanation was also observed; developers frequently used code search as a method of navigation, preferring it to using a file explorer.

Code Search is simple and accessible; it just requires using the search box built into the IDE. This research suggests it is frequently being used for information gathering and navigation rather than identifying specific code artifacts.

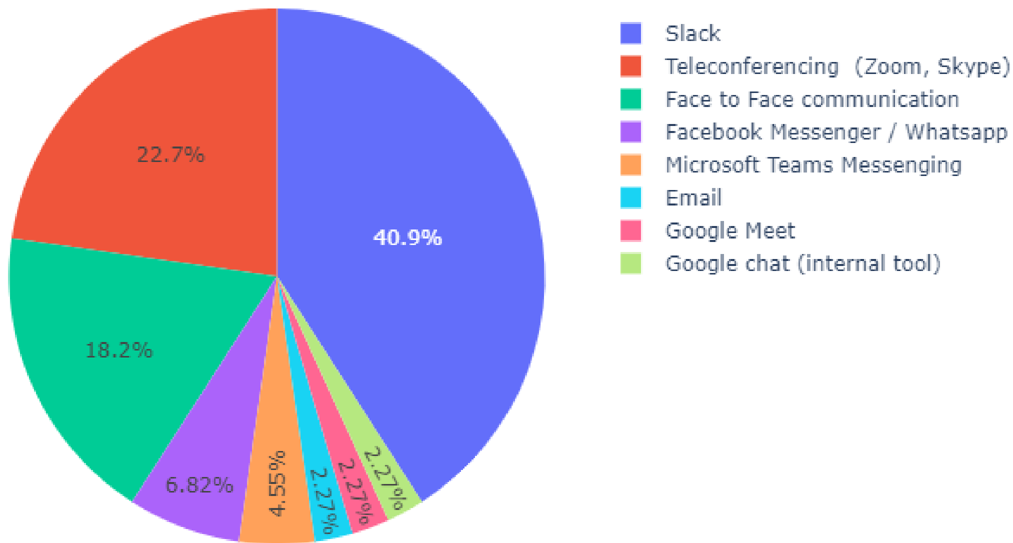

Figure 2.2: Which communication tools was used most frequently by surveyed programmers

2.2 Vector Space Based Indexing

One key task related to this project brief is determining what a document (an email, a slack message, a commit description) is about. Vector Space Based Indexing is a set of Natural Language Processing Methods designed to analyse the relationships between a corpus of documents and terms they use.

2.2.1 Basic Indexing

A vocabulary is built of all the terms used across all the documents and for each document a vector the size of this vocabulary is built. In simple implementations the elements are just the number of times each term occurs in the document.

The similarity of two documents can then be computed as the angle between the two vectors representing the documents. This can be used to directly compare items or to build semantic clusters.

2.2.2 What Constitutes a term?

Standard Natural Language pre-processing techniques can be used to edit the list of terms. For some applications better results might be achieved by being case-insensitive, ignoring stop-words etc. 'Stemming' is a modification to this method presented by Antoniol et al. when linking emails to documentation [10]. Terms that share a root word (ie. model and models) are grouped together.

2.2.3 Weighting

Instead of just pure counts more sophisticated metrics can be used to build the document vectors. One widely used metric is 'Term Frequency / Inverse Document Frequency' (TF-IDF).

$$tf_{c,n} = \frac{d_{c,n}}{\sum_{i=1}^{C} d_{i,n}} \quad idf_c = \log|D| - \log\{d : t_c \in d\} + 1$$

The significance score for an individual cell is then computed as follows:

$$tfidf_{c,n} = tf_{c,n} \cdot idf_c$$

By combining the local 'term-frequency' and global 'inverse document frequency' this metric mitigates the contributions of terms that appear frequently in a document simply because they are used frequently everywhere.

2.2.4 Latent Semantic Indexing

Latent Semantic Indexing is an extension to the basic method where 'Singular Value Decomposition' is applied to the Document-Term matrix. SVD reduces the number of rows whilst preserving the relationship between columns as much as possible. The intuition is that the more compact representation will capture the topics present in documents not the individual terms.

2.3 Mining Source Code for Semantic Information

Code isn't written in Natural Language and programming is often thought of as a human-machine communication task; the programmer is 'telling the computer what to do'. However, developers also communicate with people when coding. Comments and Identifier names are ignored by automated tools but programmers add them to convey to other developers or their future selves what code is doing and why they made certain decisions.

Kuhn et al. show how this text could be used to extract semantic information from Source Code. They outline a method for, using only identifiers and comments, clustering source code into groups that are then named with the terms that best describe them. They use Latent Semantic Indexing to build vectors for each file which can then be clustered. The similarity of each term to each cluster is then computed and the top n are chosen to describe the cluster.

$$\text{Let } t \text{ be a term and } C \text{ be the set of clusters}$$
$$rel(t,c) = sim(t,c) - \frac{1}{|C|} \sum_{c_n \in C} sim(t, c_n)$$
$$sim(t,c) = \frac{1}{|c|} \sum_{c_n \in c} sim(t, c_n)$$

Determining which files belong together is subjective so evaluating the method requires human inspection and therefore the number of case studies outlined is limited. However for these studies Kuhn et al. report sensible clusters being formed. [11]

2.4 Mining Digital Communication

There has already been much work done in the area mining digital communication for the kind of insights useful to this project. As a function of age there is more research into email but work has also been done into Slack and similar platforms.

2.4.1 Social Dynamics in Communication

Bird et al. present an investigation into the email communication within a large, long-running software development project. By examining who sent messages to each other they were able to build a social network for the project (see Figure 2.3); they highlight several dynamics of the developers captured in the graph. Firstly there was a positive correlation between connectedness and productivity and conversely that a high level of source code activity was a good indicator of social-status. From the graph one can identify pairs of people with longstanding good communication that we could infer, work well together. [12]

Maglio et al. investigated whether these kinds of social networks could aid in finding experts on a particular topic, albeit within teams that weren't made up of Software developers. Emails about a particular topic are clustered together and graph is built out of all the senders and recipients. A modified version of the HITS (Hyperlink Induced Topic Search Algorithm) is used to assign every person a 'repute' and 'resourcefulness' score. The repute score gives a ranking of the experts of a particular topic. This approach was compared to a simple content based approach which ranked experts based on how many emails about the topic they had sent. The ground truth ranking was gathered by asking team members to rank their colleagues. The HITs based algorithm consistently matched this ranking better. [13]

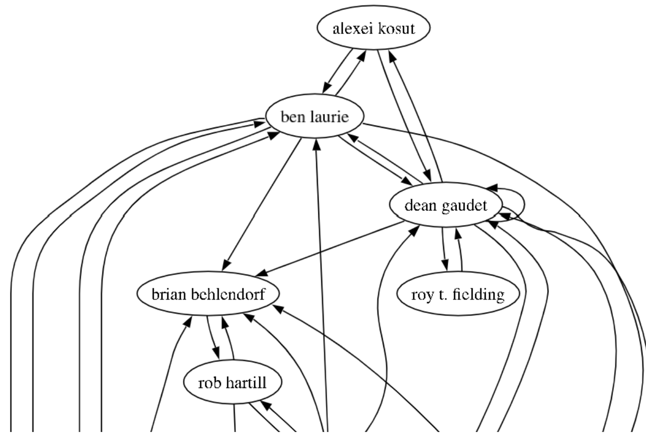

Figure 2.3: Section of social network graph generated in [12]

2.4.2 Identifying Individuals

If within one project multiple different tools and communication platforms are used the same individual may appear under multiple different aliases. Often these will be easy for a human to group (eg. jrackham, joe.rackham and jar17) but non-trivial for a computer to cluster.

Bird et al. presents a clustering algorithm for email addresses belonging to the same person. From each email pair is generated of the email address any name included in the header.

```
<"Joe Rackham", "jar17@ic.ac.uk">
```

For each distinct pair of pairs a measure of similarity is computed using the Levenshtein distance between the various components of both users. This process was applied to the Apache developers mailing list and resulted in few errors. The same approach can also be applied to users in a Version control system. Bird et al. don't discuss the prospect of clustering users across different systems but the method would be simple to apply as long as systems provided an email and username for individuals. [12]

2.4.3 Linking Emails to Source Code

Linking emails, which are primarily written in free-text, to the code artifacts they discuss is a non-trivial problem. With readability in mind software developers are encouraged to name functions and classes using dictionary words, but this makes it difficult to distinguish someone referring to a code artifact from someone simply speaking. Bacchelli et al. present several techniques for establishing a link between an email and a source code class: [14]

Linking Emails to Classes

Several variations of lightweight 'grep-like' methods have been devised where the emails are scanned for the appearance of certain words or elements. Building a link wherever a word or compound word used in the email also appears in the code-base leads to high recall but low precision. To improve precision when dealing with compound words, a link is only built where the compound word isn't itself a dictionary word (for example we would build a link for TreeRoot but not TreeHouse). To improve recognising when single words are referencing a class the best results have been achieved by requiring that some package information is also given:[2]

```
(.*) (\s|\.|\\|/) <packageTail> (\.|\\|/) <EntityName>
((\.(java|class|as|php|h|c))|(\s))+ (.*)
```

This approach performs well for Java and similar languages but C doesn't have packages and in some languages it isn't common to use compound-word class names. Higher recall can be achieved looking for capitalised words surrounded by spaces or punctuation:

[2]Regex reproduced from Bacchelli et al. [14]

```
(.*) (:punct:|\s)+ <EntityName> (:punct:|\s)+ (.*)
```

These approaches are simple to implement and fast. However links aren't ranked and performance degrades as the code bases grows.

Alternatively Vector Space Based Indexing can be used (see Section 2.2). The email corpus is put into a vector space and then the vocabulary is used to build a vector representing the source code. No terms are added at this stage so the vocabulary doesn't get bloated with code keywords. We can then compute the similarity between the source code and each email and get a ranked list of matches.

$$\cos\theta_n = \frac{d_n^T q}{||d||_2||q||_2}$$

The previous methods fail to deal with 'Synonymy'; where the email refers to the same artifact in multiple different ways. Using Latent Semantic Indexing aims to recognise the topics in the email corpus and deal with this problem. Using the components of the Singular Value decomposition from the email-term matrix the source code term vector can be transformed into a topic vector of the right dimension.

Despite being more complex and improving recall, these IR based techniques come at a sharp cost to precision. In the experiments presented by Bacchelli et al. the lightweight methods achieved a better overall F-Measure. The later methods, especially LSI were also expensive to compute over a large corpus.

Linking Paragraphs to methods

Once the link between an email and a class is established, Panichella et al. present a way of linking paragraphs to individual methods.

Paragraphs more than 10% made up of keywords and special characters are disregarded. References to methods need to contain the word 'method' as well as the method name followed by an open parenthesis. If their are multiple methods with the same name, either because the email was matched to multiple classes or due to overloading, the paragraph is assigned to all of them.

Panichella et al. also extends this method specifically to use email communication to find a description for undocumented methods. Firstly they use the English Stanford Parser to disregard paragraphs not in the affirmative form. Paragraphs are then scored based on the presence of words like 'override', 'invoke' and the names of the method's parameters. Paragraphs not meeting a threshold are disregarded. Paragraphs are then ranked by textual similarity with the method they are suspected to be describing. This filtering definitely removes paragraphs that are, in fact, talking about a method, but the goal is to find only those that can also lead to useful descriptions.

This method is limited in that it only identifies existing descriptions instead of synthesising new ones from the best candidates. However, Panichella et al. applied their methods on the Lucene and Eclipse open source systems and found that, where the corpus of communications contained a useful description, they could mine it with up to 79% precision for Eclipse and 87% for Lucene. [15]

2.4.4 Disentangling Slack Conversations

The characteristics of communication in Slack and similar systems pose challenges to being able to effectively mine useful information. People tend to talk more informally and in shorter, incomplete sentences. More-over multiple conversations might be interleaved within the same channel. [16]

A technique for disentangling IRC messages is proposed by Elsner and Charniak. A maximum entropy classifier is used to determine whether a pair of messages are related to the same conversation. The features input includes both analysis of the text (is one a question?, are cue words used?) and chat specific features (how much time elapsed between messages?) [17]. This technique was extended by Chatterjee et al. who were specifically looking at Slack conversations. They extended

the set of features with Slack specific features like use of emoji or code-block and considered pairs of messages longer apart. This method improved upon Elsner and Charniak and achieved a high level of accuracy.

2.4.5 Clustering Emails

Surendran et al. present a method similar to Section 2.3 but aimed at clustering and naming email topics. The method created clusters that users found 'useful' 70% of the time. To create meaningful cluster names it wasn't enough to just filter out normal english-language stop words (the, an, we, etc.). Instead they also needed to filter out domain specific keywords as only allowing noun-phrases that appear in the subject line of emails to be topic names. Surendran et al. comment that the same method could be applied to other types of document; success was also achieved with source code so it would be interesting to also apply it to Slack Messages and other text.

This research raises the question of why users can't build or at-least give the name for topics themselves. Why is a fully automatic approach necessary. Firstly, whilst sorting emails as they come in is do-able these methods can be turned on an existing inbox with a long history. Furthermore Surendran et al. found users were unwilling to pro-actively invest time they could be spending dealing with the emails contents. [18]

2.5 Cognitive Load

2.5.1 Cognitive Load Theory

Cognitive load theory is based on the principal that humans have a limited working memory. It provides an explanation for why developers would struggle to keep on-top of all artifacts relevant to a task and why a tool that automatically generated insights would be useful. For each task, CL Theory identifies three kinds of load [19]:

- **Intrinsic** - The inherent difficulty of task. Regardless of presentation simple arithmetic is less difficult than solving differential equations.
- **Extraneous** - Comes from how information is presented.
- **Germane** - Somewhat akin to learning. Germane cognitive load refers to the metal resources needed to routinize the task being done and commit it to long-term memory.

High cognitive load leads to an increase in errors and other drains on productivity [20]. Extraneous load is mutable and should be minimised so more mental effort can be devoted to the other two classes. Software tools can reduce their extraneous cognitive load by having a clear and consistent interface. Tools should also aim not to unnecessarily distract the user and divert some of their mental resources.

2.5.2 Cognitive Load in Software Development

There has been some previous work exploring the cognitive load placed on software developers and it's impact on them. It however remains true that more effort is devoted to the technical side of the field and corpus of research into the cognitive side is small [21].

Helgesson et al. present an exploratory case study into causes of software development [1]. They identified drivers of cognitive load in three key groups:

- **Tool** - Tools that didn't make it clear how to use them or what was going on. Tools with delays or periods of non-responsiveness could also cause loss of focus.
- **Information** - Participants experienced increased cognitive load when it was difficult to find information or when it wasn't clear the information they has was complete or reliable. This is the primary problem this project is attempting to tackle.

- **Work process** - Cognitive load drivers relating to the teams work processes included things like not knowing who had responsibility for part of the project or a lack of automated processes. It should be possible to work out who is responsible for, or at least knowledgeable on certain parts of the project based on the communication within the team. This could be surfaced by automatic extraction to further ease cognitive load.

2.6 Existing Tools

There are several existing tools that fit under the umbrella of 'AI Development Assistant' or 'Assistant for Software Development' that have been informative look at in researching this project.

2.6.1 Mylyn (formerly Mylar)

Mylyn is a plug-in for Eclipse that provides a 'task-focused interface'. The motivation behind it's creation was reducing the time programmers spend sifting through potentially thousands of code artifacts to find those relevant to their task and then potentially having to do it again if they switch task. Mylyn collect user interactions with files to compute a DOI (Degree of Interest) weightings. This information is then surfaced in the IDE though features like ranking search results by DOI, views where only active parts of the project are shown and active test runs.

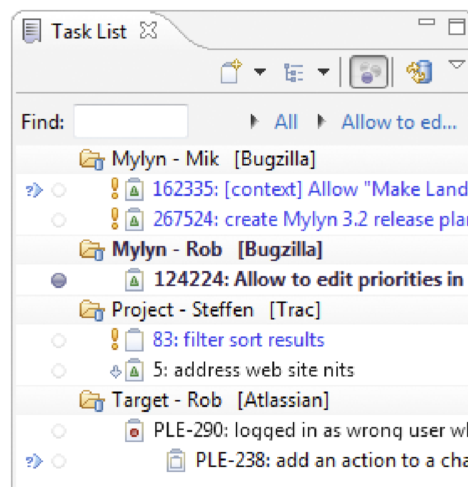

Figure 2.4: Mylyn Task View - The task context for task 124224 is currently being tracked

Mylyn constructs a graph of code elements where each edge and vertex is weighted by DOI. This is done using an interaction history built from both direct interactions (editing a file) and indirect interactions (editing the file this file tests). From this a real number score is assigned to every vertex based on the frequency and recency of interaction. A edge is created when an event in the history takes the user from one element to another, such as navigation. Edges are weighted the same as their target element.

Kersten and GC. Murphy present the results of a field study with industry programmers showing they were more productive using Mylyn. Specifically there was an increase in the amount of time editing code compared with browsing and navigating. The plug-in is also currently used by thousands of programmers. [22, 23]

2.6.2 Kite and Kite Copilot

Kite is a coding assistant for Python. Specifically offering AI powered code completion and identifying frequently used code snippets. Kite performs well and claims to reduce it's users keystrokes by 47%. Kite could clearly improve a programmers working speed but this would only improve their productivity if they were implementing the correct solution. [24]

More interesting, was the sibling product Kite Copilot which follows the cursor and displays relevant and detailed documentation in a separate window (see Figure 2.5). All the information displayed could be easily found via a search engine but there is clearly value in making it more immediately available. An obvious limitation of both tools is their restriction to Python. [25]

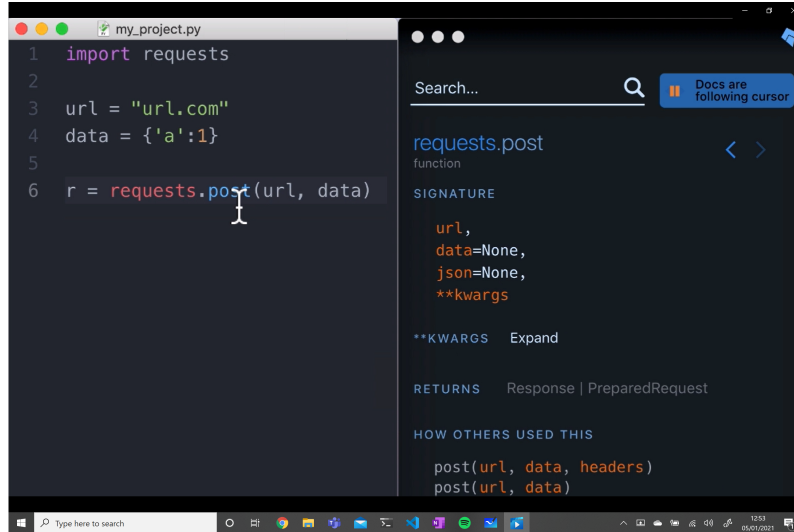

Figure 2.5: Kite Copilot Following Users Cursor in Python Document [25]

2.6.3 Facebook Aroma

Aroma is a 'code-to-code' search tool that uses Machine Learning to find idiomatic usage patterns in a large code base. This can help users find a way to implement a task that's not only correct with respect to the language, but correct with respect to internal rules and best practices. Facebook emphasises the importance of the tool offering suggestions in real-time so use of Aroma can be a part of the flow of programming.[3] [26]

2.6.4 Guru

Guru is a bot for Slack and browser-extension with the goal of 'unifying an organisations collective knowledge' [27]. When a question is answered users can generate a Guru-Card from the response. This knowledge shared is then categorised and assigned to a group or person now responsible for keeping it up to date. Users can easily search and share Guru cards, making it easier to find information and check how long ago it was verified. [28]

An obvious limitation of Guru is that the only tool it integrates with is Slack. Software development teams share information over a wide range of digital tools so it's unlikely Guru would be able to construct a complete picture. Team members must also manually construct and verify cards. In the context of software engineering this would be suitable for capturing some long lasting information but may be too high an overhead to manage information about, typically short-lived, bug-fixes and other small tasks.

[3] Aroma is not publicly available for me to test at time of writing

2.7 Discussion

As well as illuminating existing solutions to some problems related to this projects brief, this background research has confirmed some of our assumptions and provided some insights into the problem domain that will need to be considered going forward.

As anticipated we found that communication amongst modern Software development teams is fragmented across a wide variety of tools. Email remains most researched communication tool despite a decline in it's use in favour for alternatives like Slack. Previous research confirms that the corpus of communication spread across these tools can be mined for useful information. It is possible to extract social networks, find experts on particular topics and find descriptions for undocumented methods. Further research confirms that additional useful insights can be extracted from source code by itself and when it's used in conjunction with communication. Research has been successful in establishing links between email and source code, disentangling slack conversations and discovering semantic topics from documents.

Section 2.6 details an investigation into several existing tools with goals related to this project. These tools primarily just organised or highlighted information the user already had access to; Kite Copilot finds documentation that could be easily Googled, Guru captures answers that were posted in Slack channels. Despite this, these tools (excluding the unreleased Facebook aroma) had each accumulated a large user-base. Furthermore, Sadowski et al. detail how Software Developers use the easily accessible code-search tool in their IDE to find meta-information about the project and guidance on API usage; information for which there is likely a better and more authoritative source. There is clearly value in extracting surfacing this information in the right way at the right time.

Chapter 3

Survey of Techniques

The primary contribution of the work is a 'Workbench' in which multiple different techniques for the extraction of useful information can be applied to Emails, Slack Messages and Repositories of Code. Building the workbench, the aim was to expose all hyper-parameters, options and choices so that the workbench could serve as a useful tool for investigating how best to process these different forms of communication.

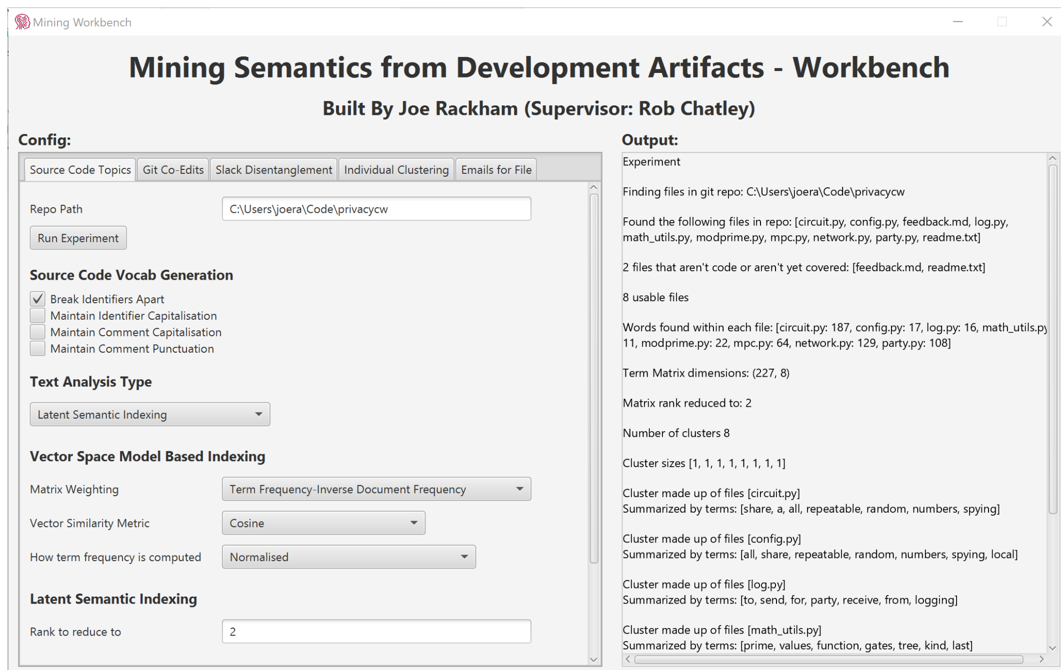

Figure 3.1: The Experiment Workbench

The workbench is built from a series of tabs each corresponding to a different distinct experiment. The experiments are detailed starting Section 3.2.

3.1 Project Setup

The project was built in Kotlin. Using a statically typed, Object-Orientated language helped control the complexity of connecting to multiple different services all relating to communication. All Java libraries can be used in Kotlin, many of which were useful in the construction of the Workbench. Kotlin was chosen over Java because it's more extensive set of functional programming features was usefu in creating pipelines for processing data.

The UI was built using Kotlin Library TornadoFX. Another option was to build the front-end separately to the bulk of the app and creating a web app. The styling and input options in Tor-

nadoFX are limited so this would have potentially led a better looking app final product that was clearer to a new user. However, the goal of the project was to create a platform for scientific experimentation not a product; how it looked wasn't a priority as long as usability wasn't an issue. Furthermore, keeping everything in the same project prevented spending time building code to help the front and back end communicate which wouldn't serve the project brief.

3.1.1 Email

Email is the historic choice for communication in a software development project and it remains popular (see Section 2.1.1). It was crucial to the success of the project that email communication was investigated.

Many open source projects publish an archive of their mailing list online at https://www.mail-archive.com/. We thought large projects with a long history of conversation would be useful in evaluating techniques to extract useful information from email. To this end, we built small Python tool to scrape and process these archives, extracting the useful information such as the content and time sent. We also used the JavaMail library to build an adapter that could read emails from a mail server. So the workbench could be directed at ongoing and non-archived projects.

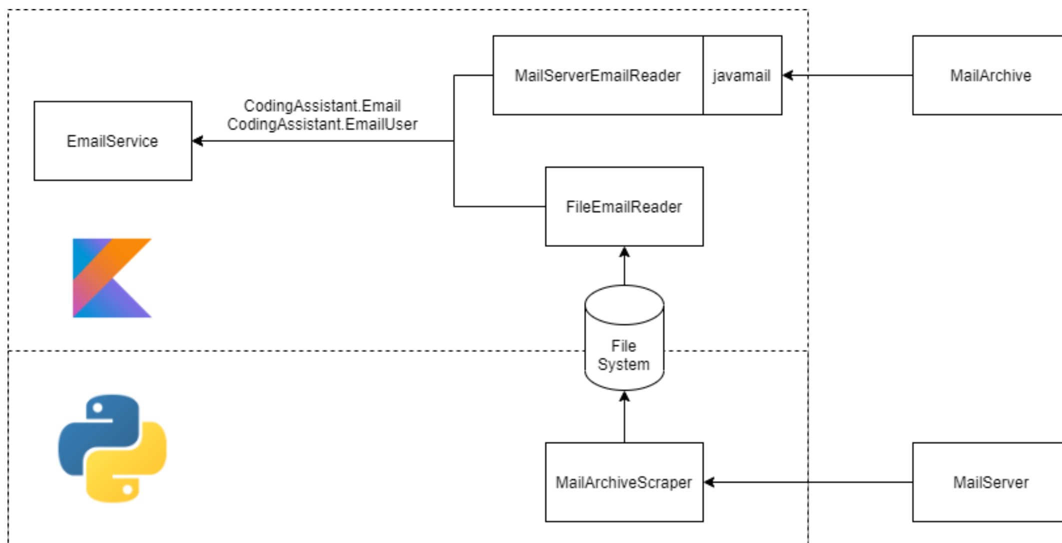

Figure 3.2: How Email communication is retrieved

3.1.2 Slack

Of the programmers surveyed, Slack was the most frequently used communication service (see Section 2.1.1). It was therefore important to investigate the barriers to extracting meaningful semantic information from Slack channels.

We thought it was worth investing the time in building an adapter to the Slack API. There were some data-sets of archived Software Development Slack Channels but maintaining an external online backup of an active Work-space is against Slack's terms of services so these were limited. Building an adapter also enabled the workbench to be aimed at our own projects. The quality of output from topic extraction and clustering is largely subjective; if we were familiar with the source we would be in a better position to evaluate the output. There is a Slack API client available for Java which was used to avoid processing the HTML responses directly. Classes from the Slack API client package are only used within the Slack Adapter to reduce coupling.

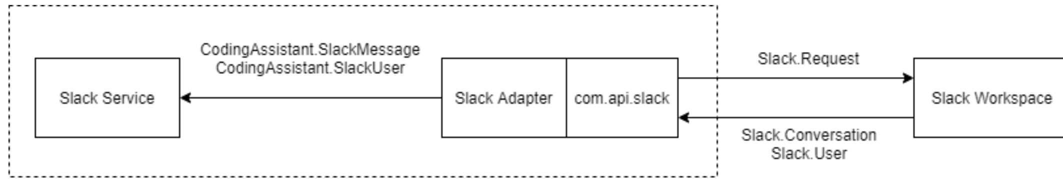

Figure 3.3: Connection via the Slack API to work-space data

3.1.3 Git

Git is a vital source on information about a software engineering project and it's contributors. Git contains information about a repo as it currently stands, how it had evolved in the past, and how a user is changing it. We used the JGit Library to build an adapter to git. We built functionality like retrieving a matrix of co-edits between different files or retrieving a 'bag-of-words' built from commit messages about a particular file that are used throughout the techniques in the workbench.

3.2 Experiment: Disentangling Slack Conversations

As discussed in Section 2.4.4 Slack channels often contain multiple distinct conversations. Although it is normally clear for a human reading the chat to determine where one conversation begins and one ends or to untangle an interleaved chat, this is more difficult for a computer to do automatically.

Chatterjee et al. reports success applying a Maximum entropy classifier to this problem. Using the workbench, one can apply this model to the problem and experiment with it's variations. Listed below are features extracted from pairs of messages. Boolean features are mapped to 1.0 if true or 0.0 otherwise. Some features are the combination of a value calculated for each message separately; there is an option exposed in the workbench to determine if these values are kept separate or combined in the output array.

Single Message Features

Feature	**Definition & Intuition**
MENTION	Does the message mention the other speaker
CUE_WORDS	Does the message contain a cue word to indicate they are interacting with another person. This could be a greeting ("hello, hi" etc.), an answer ("yep", "no can do" etc.) or a thank you ("thanks", "ta!" etc.)
QUESTION	The message contains a question, indicated by a question mark.
LONG	The message is long (>10 words)
TECH_JARGON	Is technical jargon used in the message
CODE_SNIPPET	Presence of a code snippet in the message
EMOJI	How many emoji are used in the message.
MUTUAL_REACTIONS	Does the other speaker react to the message
PRONOUNS	How many pronouns (you, he, she etc.) are there in the message. These should indicate the speaker is talking about or to another person.

Pair Only Features

Feature	**Definition & Intuition**
TIME	The Time between the two messages in seconds bucketed logarithmically. The longer between two messages the less likely they a a part of the same conversation.
SPEAKER	Do they have the same speaker?
SHARED_WORDS	The number of words shared between two messages
TECH_JARGON	Is technical jargon used in either message
MSGS_BETWEEN	How many messages were sent between the two in question. This feature captures a similar relationship to TIME but was intended to capture when a conversation is interrupted.

The lists of cue words and technical jargon were constructed with a combination of publicly available data-sets and collecting suggestions from peers. Because the model is meant to be applicable to any software development chat the technical jargon is made up of more generic words (commit, bug etc.) and not language or project specific keywords.

Maximum entropy classifiers weight each feature independently. This makes the model quick to train and simple to represent. This type of classifier perform supervised learning. To train the mode, a data-set constructed for previous research of Slack Messages grouped by conversation was used `https://zenodo.org/record/3468559#.YJFUB7VKjZt`. The messages were taken from the public Software Development Slacks Clojurians, PythonDev, Elmlang and racket.

3.3 Experiment: Source Code Topic Clustering

As the developers of Mylin discovered the arrangement of files in a projects file system often bears little resemblance to the semantic relationship between them (see Section 2.6.1). Files are frequently edited together with files that are 'far away'. Previous research has been undertaken into extracting semantic topics from source code. We investigated how these methods, as well modifications and extensions of our own extracted relationships between files.

3.3.1 Extracting Vocab from Source-Code

Human-human communication occurs in two places in source code, identifiers and comments. Depending on the project comments might be in natural language or be less structured. A vocabulary for source code is extracted by finding matches for a series of Regex patterns stored in an instance of the SourceCodeLangDetails class for the language. Through the workbench various options are exposed so that the best variant of the process can be determined. These include if identifier names should be split into separate words and whether capitalisation and punctuation should be maintained.

This has the obvious downside of being limited to a finite set of languages that a SourceCodeLangDetails instance is built for. A language agnostic approach is also built into the Workbench for comparison. Only patterns for the generic assignment (`x = ...`) and function call (`f(... )`) are searched for and identifiers are broken apart at any uppercase or non-alphanumeric character.

3.3.2 Text Analysis for Topic Extraction

Encountered in the background research, was the use of Latent Semantic Analysis to extract topics from a corpus of documents. LSI builds a small dimensional vector for both terms and documents in the same vector space. The similarity of these vectors can then be used to cluster documents and determine which terms best describe them. The workbench exposes many options to modify the methodology found in background research. The rank that the matrix is reduced to as well as the vector metric for calculating similarity can adjusted. The term-document matrix can be weighted using just using the pure-counts, the tf-idf metric or alternative 'Term Frequency / Proportional Document Frequency' (TF-PDF).

$$pdf_{c,n} = \frac{d_{c,n}}{\sqrt{\sum_m d_{c,m} * d_{c,m}}} * \exp(\frac{|\{d : t_c \in d\}|}{|D|})$$

Within these methods the 'term frequency' component can be be calculated several different ways (see table). However, amongst other issues, LSI lacks a sound statistical foundation, there are other models which seek to improve upon LSI and correct some issues with the model. Two of these 'refined' approaches were also implemented to compare the quality of output.

Pure Counts	Simply the number of times the term appears in the document
Normalised Counts	The pure count normalised by the number of times the term appears in the corpus. Helps tell words used frequently in a document and genuinely popular words apart.
Boolean	Only 1 or 0
Length Adjusted	The pure count normalised by the length of the document. Longer documents will natural have higer pure counts.
Log Scaled	$ln(1 + x)$
Augmented	$0.5 + 0.5 * x/y$ where y is the biggest frequncy of any term in the document

Probabilistic Latent Semantic Indexing

Instead of modeling the corpus using Linear Algebra, Probabilistic LSI aims to capture the semantic content using probability. The intuition behind this approach is that, with the number of topics is chosen as a hyper-parameter the probability of all co-occurrences can be modeled like so:

$$P(w, d) = P(d) \sum_z P(z|d)P(w|z)$$

The actual values of P(z|d) and P(w|z) are learnt using the iterative Expectation-Maximisation algorithm that finds a local minimum for the likelihood of the corpus. In this model documents and terms aren't matched with single topics, instead they can relate to multiple topics. This allows the model to capture polysemous terms (multiple meanings) which standard LSI can't handle. Whereas LSI is deterministic, PLSI may generate different topic allocations on different iterations. A limitation of the model is that the number of topics must be hand-chosen. However, LSI requires the rank of the rank-reduced term-vector matrix to be chosen which still affects the final output but far more opaquely.

Latent Dirichlet Allocation

Latent Dirichlet Allocation also models the corpus using probability over a fixed number of topics. However the model is more complex, the probability values are tied to actual distributions.

$$\begin{aligned} \rho_{k=1..K} &\sim Dirichlet_V(\beta) \text{ (Distribution of words in topic k)} \\ \theta_{d=1..M} &\sim Dirichlet_K(\alpha) \text{ (Distribution of topics in document d)} \\ z_{d=1..M,w..N_d} &\sim Categorical_K(\theta_d) \text{ (Identity of topic of word w in document d)} \\ w_{d=1..M,w..N_d} &\sim Categorical_K(\rho_{z_{dw}}) \text{ (Identity of word w in document d)} \end{aligned}$$

Learning the various distributions (the associated word probabilities, the topic of words, and the topic mixture of each document) can be achieved via various methods. The workbench uses a library which finds the values using a "Variational Bayes sampling" method.

3.4 Clustering

Several of the experiments require building clusters of objects based on some scoring metric. In much of the literature authors simply refer to 'clustering' but this can be done in multiple ways. The different clustering algorithms implemented and available in the workbench are as follows:

Greedy Clustering - To start, the first item is placed in a cluster by itself. For each subsequent element the scoring metric is computed between it and the already clustered items. The new item is placed into the same cluster as the item that resulted in the best score. This is unless the best score is less than some threshold in which case a new cluster is created.

'No Going Back' Clustering - Similar to greedy clustering but each new item can only be placed in the previously built cluster. More so that greedy clustering the output is dependent of the ordering of the input.

Agglomerative Single-Link - Each element is placed in it's own cluster. The two clusters which score highest are them combined. This is completed until all the clusters are arranged into a dendrogram capturing the hierarchy between objects. 'Single-Link' refers to the fact that the score between two clusters is calculated as the maximum score between any pair of elements.

Agglomerative Complete-Link - Similar to Agglomerative Single-Link. 'Complete Link' refers to the fact the score between two clusters is calculated as the minimum value between any pair.

K-Means - K items are chosen as the original centre of K clusters and every items is assigned to the it's closest centre. The centre is then recomputed based on these assignments and the algorithm iterates until the allocations converge. The K items chosen as the initial centres can have a big impact on the final allocation so the quality of the clustering my vary between different runs.

Mean Shift - A random point is chosen, any items with certain radius are 'covered' by the point. The point is moved toward the centre of the items it covers until the number of covered items stops increasing. This is repeated until there are no points left uncovered. [29]

3.5 Experiment: Individual Grouping

As discussed in Section 2.4.2 when multiple different communication media are used in the same project it's common for the same person to appear under different aliases and emails. Discovering that multiple users are actually the same person is non-trivial to automate. From Email, Git and Slack both email addresses and usernames can be retrieved. Depending on the project and the person the username might be their full name, part of it, or bear no resemblance at all. The following tests are designed to be useful in determining whether two of these users are, infact, the same person.

NAME_DIST	If the Levenshtein distance between the two full names is under some threshold
NAME_SUBSEQUENCE	One of the full-names is a sub-sequence of the other
EMAIL_NAME	One of the full-names is a sub-sequence of the others email
EMAIL_INITIAL	An email contains one of the other persons initials and their other name in full. The other name must be sufficiently long.
EMAIL_DIST	The Levenshtein distance between the two emails is under a certain threshold and they both exceed a certain minimum length
EMAIL_IDENT	Both whole email addresses (account & provider) are identical

All the thresholds and lengths are exposed through the workbench, as is the total number of checks that needs to pass to build a match.

3.6 Experiment: Linking Source Code to other artifacts

Techniques have been proposed for establishing links between a source code artifacts (a class, a method, a package) and the emails which talk about it (See Section 2.4.3). Unlike topic extraction where artifacts under the same semantic umbrella are grouped together, here the goal is to link documents specifically refering to each other. A tab in the workbench is devoted to experimentation with these techniques and how best to apply them. The basis for the methods applied comes from the work of Bacchelli et al. however like with topic extraction this set of methods is expanded with refined NLP models and hyper-parameters are exposed through the workbench. Furthermore, the methods can also be applied to Slack Messages which are grouped together into conversations.

Although the goal is to establish links between specific documents and source code some techniques require establishing a corpus vocabulary. Therefore when a query is launched the whole corpus is indexed to extract useful information, then links are built for source code artifacts one by one. Because techniques are oriented in this way one can search for links to a specific file, a specific subsection of the repository or the whole code-base.

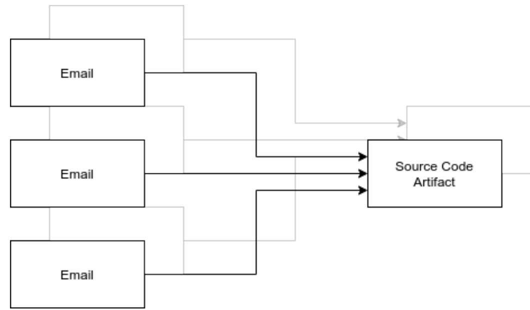

Figure 3.4: Links between emails and source code

Regex - Regex patterns are used to find what might be references to entities within the code base. These are Camel Case or Snake Case compound words or single works surrounded by package, language or path information is extracted. Similarly a list of entities is generated for source code files. Where the same entity is found a link is established.

VSM - Using word frequency information a weighted term-document matrix is built for the corpus. Using the fixed vocabulary of the corpus a vector can be built for source code documents and vector similarity metrics can be used to establish a link with documents. Using the fixed vocabulary found in the communications corpus obviates some of the processing discussed in Section 3.3.1 but there are still choices to be made about whether compound words should be split up etc. Further options include how vector similarity is computed and how the matrix is weighted.

LSI - SVD is used to reduce the rank of the term-document matrix so that every term and documents is represented by a n dimensional vector. Using the components of the decomposition vectors can be created for new documents, the source code artifacts, which can then be used to compute the similarity between documents and source code.

$$\begin{aligned}\text{Let } X \text{ be a term-document vector}\\ \text{SVD Decomposition: } X = U\Sigma V^T\\ \text{Let } v \text{ be a new term vector}\\ \text{Reduced vector: } \tilde{v} = \Sigma_k^{-1} U_k^T v\end{aligned}$$

LDA - Using this probabilistic approach, once a model is built using a corpus one can obtain a topic distribution for a new document. Similarly to the vector space based methods any terms used in the new document not present in the original corpus will be ignored. This acts as a filter, removing most of the language specific keywords and implementation content. The distributions for each document in the corpus and the source code can be compared to build ranked links.

3.7 Testing & Ensuring Correctness

Testing is a pillar of good software development; during the development of the workbench tests were used to ensure its correctness and robustness. The range of matrix, vector and text operations used as a foundation for the NLP techniques as well as useful methods dealing with clusters and http requests that are used throughout the workbench, were abstracted to a utilities package to prevent duplication. A suite of unit tests could be written for each.

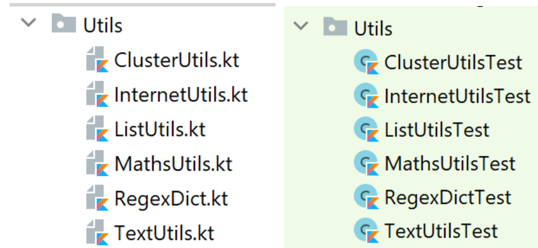

Figure 3.5: Utils packages and corresponding test files

Scripts were also written to test the integration of the external services and ensure data could be read from Git, Slack, and Email. So the rest of the app could be tested independently, mock versions of all the services were created which implemented the same interfaces but read communication from a local file. This had the additional benefit of enabling applying techniques to communication which some specific property not found in the corpus of actual data.

One aspect that complicated testing was the fact that, because of the nature of the output it was difficult to tell if the NLP techniques were being applied correctly, especially when they generate different output on every application. Good quality output was an indication that the implementation was correct but distinguishing between a faulty application of the techniques and them simply failing to capture whatever relationship they were targeting was a challenge. In some cases the methods were simple enough that with a small enough input corpus one could follow along by hand. PLSI and LDA are widely reported to be good topic models for formal prose, the experiments here were to determine their usefulness in indexing source code. We therefore used a data-set of news articles for testing, expecting good results to be generated. Both these techniques were useful in the process of development, but were difficult to automate.

```
AP881218-0003   X   A 16-year-old student at a private Baptist school who allegedly killed one teacher and wounded a
AP880224-0195   X   The Bechtel Group Inc. offered in 1985 to sell oil to Israel at a discount of at least $650 mill
AP881017-0144   X   A gunman took a 74-year-old woman hostage after he was foiled in an attempt to steal $1 million
AP881017-0219   X   Today is Saturday, Oct. 29, the 303rd day of 1988. There are 63 days left in the year. A reminde
AP900117-0022   X   Cupid has a new message for lovers this Valentine's Day and volunteers are lining up to spread t
AP880405-0167   X   The Reagan administration is weighing whether to invoke a law authorizing the seizure of tax pay
AP880825-0239   X   More than 120,000 skins of a protected species of alligator were smuggled into Japan during the
AP880325-0232   X   There will be no organized union boost behind a single candidate in Saturday's Democratic caucus
AP880908-0056   X   Here is a summary of developments in forest and brush fires in Western states:
AP881105-0097   X   Jean-Pierre Stirbois, the No. 2 man in the extreme-right National Front after party leader Jean-
AP880716-0112   X   At least 15 people died and 25,000 residents of Surat town were evacuated after torrential rains
AP880608-0142   X   Actress Betty Buckley sang ``They Can't Take That Away from Me'' at a Shubert Theater service We
AP881110-0228   X   For three years, Charles S. Robb was out of the spotlight that had become so familiar, first as
```

Figure 3.6: News Articles used to test topic models

Chapter 4

Application of Techniques

This chapter outlines results obtained from applying these techniques using the Workbench introduced in the previous chapter. These results are then used to evaluate both the established techniques and the proposed modifications and additions.

Much of the output of the techniques discussed aims to capture semantic information. Whilst some quantitative analysis can be undertaken, one can't but a decimal accuracy value everything. To facilitate a full discussion the workbench was pointed at several projects the author had previously worked on and were in a position to better comment on the output of. These included:

Veracity - A web app with a JavaScript/React front end and a Python/Flask back-end. 6 contributors. Completed over a period of 3 months.

Burst My Bubble - A web app with a JavaScript/React front end and a Python/Flask and Java back-end. 4 contributors. Completed over a period of 2 months.

This also has the benefit of facilitating a discussion on applying the techniques to a closed project as opposed to an open-source repository.

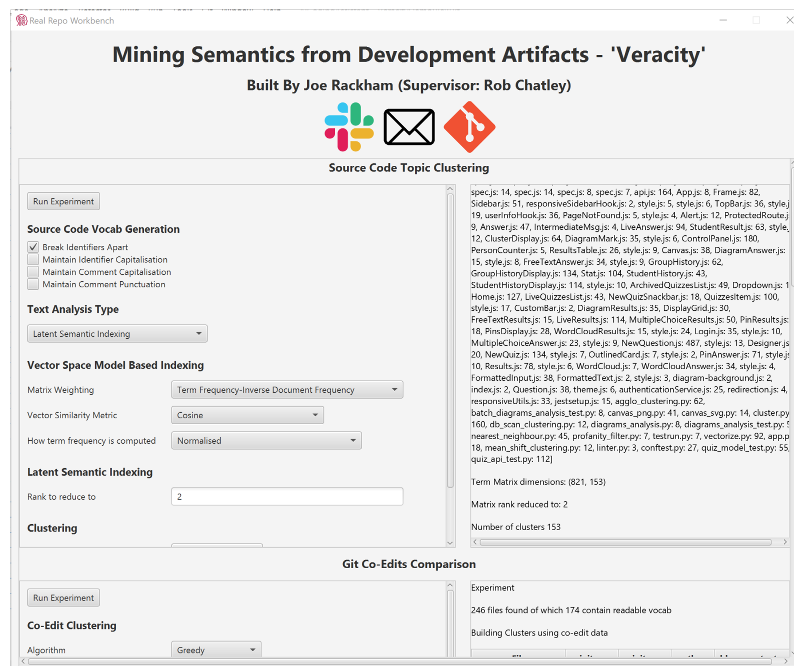

Figure 4.1: The Workbench experimenting on 'Veracity'

4.1 Message Untangling

4.1.1 Training and Evaluating

Using a data set of annotated message pairs from Software Development Slacks, the LR model can be trained and evaluated to see how well it captures the relationship. A high level of accuracy was achieved, with several configurations generating models that correctly paired messages over 90% of the time. The best configuration was using features [TIME, SPEAKER, MENTION, CUE_WORDS, LONG, SHARED_WORDS, CODE_SNIPPED, MSGS_INBETWEEN, REACTIONS] and not splitting up message features. This achieved an accuracy of 91.8%.

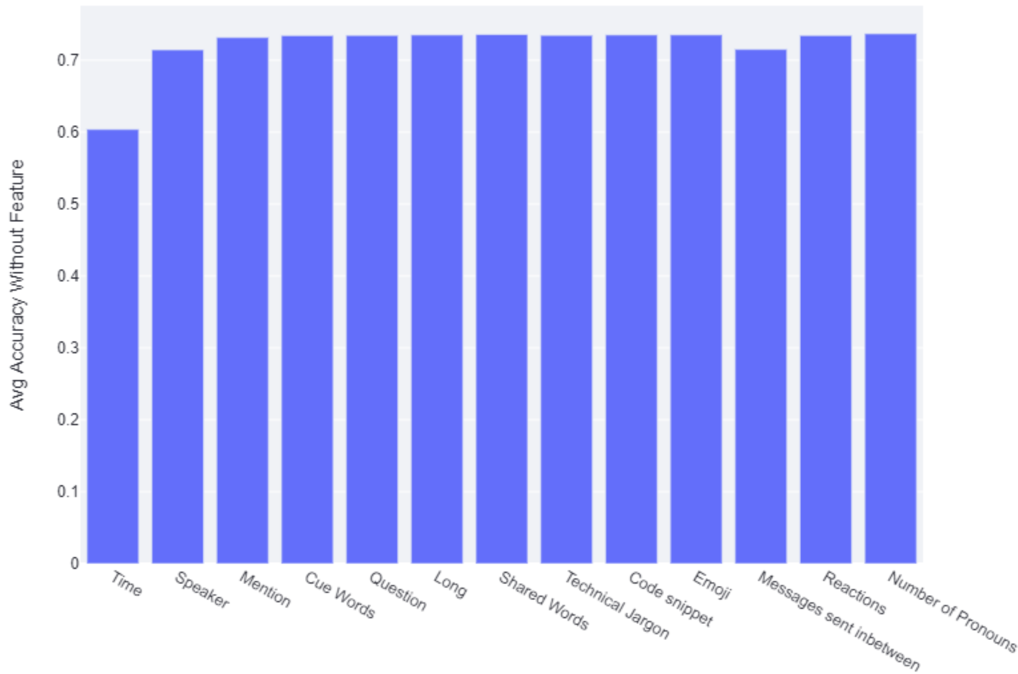

Figure 4.2: The Average Accuracy achieved when a specific feature is turned off

Unsurprisingly the most useful feature by a significant margin was the time between the two messages. However, as discussed by Chatterjee et al. the amount of time between messages varies between different services and different domains. If time was not available as a feature the best accuracy achieved was 71.4% by features [SPEAKER, MENTION, QUESTION, MSGS_INBETWEEN, REACTIONS] also not separated. Neither configuration includes the full set of features which suggests some may have a detrimental impact on performance. However, for every individual features the average accuracy across all sets including it was higher than the average from excluding it.

The goal is that the model generalises to any Slack with a software development focus. To investigate this property experiments were conducted where messages from one Slack were withheld at training time. These models were then evaluated against the unseen Slack. The high level of accuracy was maintained.

Withheld Corpus	Clojurians	Elmlang Beginners	Elmlang General	PythonDev	Racket
Accuracy	89%	87%	88%	92%	88%

An interactive dashboard to explore the data further is available at `https://message-dissentangle.herokuapp.com/` [1]

[1] The Web-App may take a couple of minutes to load

4.1.2 Applying to Author's Projects

The best model found in the previous Section was applied to author's real projects to see if the high accuracy could be recreated. As well as pairing up messages they needed to be grouped into conversations. Two options were experimented with:

- Using the binary class output which would be either 0 or 1
- Using the numerical output of the model which would be some unbounded real number

In both cases the forms of clustering that made sense to test were 'Greedy' and 'No way back' (see Section 3.4). In the process of building conversations not all message pairs were considered. Intuitively, the further two messages are apart the less likely they are relating to each other; potential pairs were built from messages that occurred within a sliding window of 30 messages.

```
---------------------
Ellie Cherrill - so let's wait for everything to pass before setting those in to merge
Ellie Cherrill - Ok looks like Buneme and I both have MRs
1.0
---------------------
Ellie Cherrill - so let's wait for everything to pass before setting those in to merge
Buneme - haha its a match :joy:
1.0
---------------------
Ellie Cherrill - so let's wait for everything to pass before setting those in to merge
Ellie Cherrill - But there is ALOT waiting for the pipeline atm
1.0
---------------------
Ellie Cherrill - <@UP13PGWTY> because you cancelled your more tooltip pipeline and I set
Joe Rackham - 'LaTeX' but yeah that sounds good
0.0
---------------------
```

Figure 4.3: Messages from Veracity Slack alongside class predictions

When a Slack is used more informally the distinction between conversations can be unclear. However, even when taking the strict notion that any change of topic represents a new conversation the best logistic regression configuration predicts the right class 68% of the time (from a sample of 250 messages from both projects).

The binary class output with greedy clustering let to some overly long conversations; one erroneous prediction would fuse together two unrelated conversations. Using 'No way back' clustering and abandoning clusters once a message which didn't fit occurred performed better. There were less overly large clusters and only a few instances where conversations were split into multiple clusters. The best output came from using 'No Way Back' clustering and the real number output of the model, here low value positive predictions could be discounted largely resolving the problem of overly large clusters. There were a few conversations that were split into parts but this was deemed to be preferable to grouping unrelated content.

```
Conversation
Buneme - Sent: 1.5783386600098E9
 oh right, nw I'll just stick to the tooltips then
Buneme - Sent: 1.5783388830108E9
 For the textfield tooltips, is this good enough? “
Joe Rackham - Sent: 1.5783389670114E9
 'LaTeX' but yeah that sounds good
Buneme - Sent: 1.5783389930119E9
 thought something was wrong, thanks :slightly_smil
Buneme - Sent: 1.5783393560122E9
 just submitted a small MR for the tooltips
Buneme - Sent: 1.5783398130128E9
 did we want the question count to just be in live
Julie - Sent: 1.5783398320131E9
 Yeah I'm happy with both!
```

Figure 4.4: Part of a Conversation built using class prediction grouping

One difference between these projects and the open-source projects the model was trained on that may have influenced results is the number of participants. In a the larger Slacks different conversations were almost always were had by different people and overlapped. With a smaller group of participants it was more a question of determining whether two messages from the same person were on distinct topics.

Overall the techniques for building conversations achieved moderate success in capturing the conversations in the private projects. The model achieved a lower, but still high, level of accuracy when applied to the author's projects instead of the training data. When rebuilding these pairs into conversations, groups that were made up of multiple distinct dialogues were largely avoided but some real conversations were split in two.

4.2 Topic Extraction

4.2.1 Capturing Co-Edit Behaviour

One hope in pulling out topics of semantically related artifacts is that this information is useful in suggesting files that a developer, who has modified some files, might modify next. To evaluate success in this aim the topics pulled out of the code-base can be compared with a co-edit count matrix from Git. This data can obviously be impacted by commit pattern of the repository's contributors. Smaller commits lead to lower rates of co-editing, some teams use feature branching whereas others consistently integrate changes.

The commit history of repositories was read to build a co-edit matrix. This provided a metric of similarity between each file which could be used by a clustering algorithm. To determine the similarity between this clustering and one built using the natural language content the following metrics were used:

Let C be a set of classes (true clusters) and K be a set of clusters

$$homogeneity = 1 - \frac{H(C|K)}{H(C)}$$

$$completeness = 1 - \frac{H(K|C)}{H(K)}$$

Let n_{ck} be the number of samples labelled c in cluster k

$$H(C|K) = -\sum_{ck} \frac{n_{ck}}{N} log(\frac{n_{ck}}{n_k})$$

These metrics are combined by the v-score

$$v = \frac{(1+\beta) * homogeneity * completeness}{\beta * homogeneity + completeness}$$

The V-Measure ranges from 1 to 0 with 1 being a perfect result.

There was no meaningful order to files or concept of a mean file so K-Means, Mean-Shift and No-Way-Back weren't valid options to generate the target clusters. Under Greedy Clustering core files that were edited with a high proportion of commits led to massively uneven output (see Figure 4.5). The only usable clusters came from using Agglomerative Complete-Link clustering and partitioning the output dendrogram. Because these were acting as the 'true' clustering labels a specific configuration needed to be chosen. This was somewhat arbitrary, but to prevent overly larger or overly sparse groups the number of edits chosen was the lowest value where the biggest cluster contained no more that 25% of the files. For Veracity this turned out to only be 1.[2] Because this output still contained many clusters of size 1, $\beta = 0.5$ was used in the calculation of the v score so that homogeneity was valued more.

```
133 with sizes [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 42, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Figure 4.5: Greedy Clustering Applied to Veracity Co-Edit Data

[2] BMB was smaller and divided between many repositories so there was insufficient data test against it.

Vector Space Models

As a starting place for the investigation VSM was applied so see what kind of output was produced without applying the more sophisticated techniques below. The term vectors generated needed to be clustered similarly to how the co-edit data was.

However, no matter what algorithm was chosen and how it was configured the output was consistently poor. The output was either trivially homogeneous, because the clusters output were largely singular, or trivially complete because they were very large. The highest V-Measure obtained was ≈ 0.3

Latent Semantic Indexing

Applying LSI to the problem exposed the same range of options as VSM with the addition of the reduced rank. The inital configuration was that described by Kuhn et al..

- Using the tf-idf metric to weight the matrix. Calculating term-frequency using the normalised metric
- Reducing the matrix rank to 10. Values in the range 20-50 were reported to work well but the test projects were small in scope.
- Using cosine-similarity and greedy clustering to build the topic clusters with an initial threshold of 0.5.

Experimenting with the rank and threshold, yielded the following results:

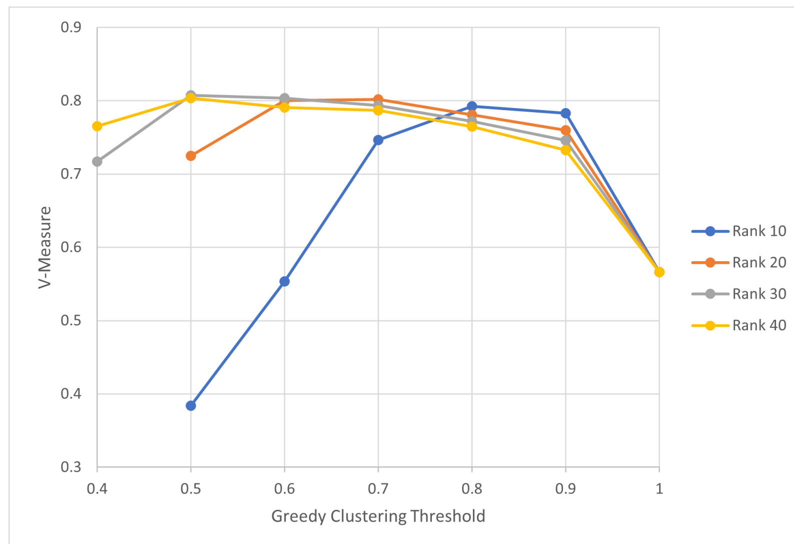

Figure 4.6: How V-Measure is affected by Rank and Greedy Threshold

The best output achieved a V-Measure of 0.8073 with Rank 30 and threshold 0.5. Experiments were also conducted with the other forms of clustering (see Figure 4.7) but this remained the best result.

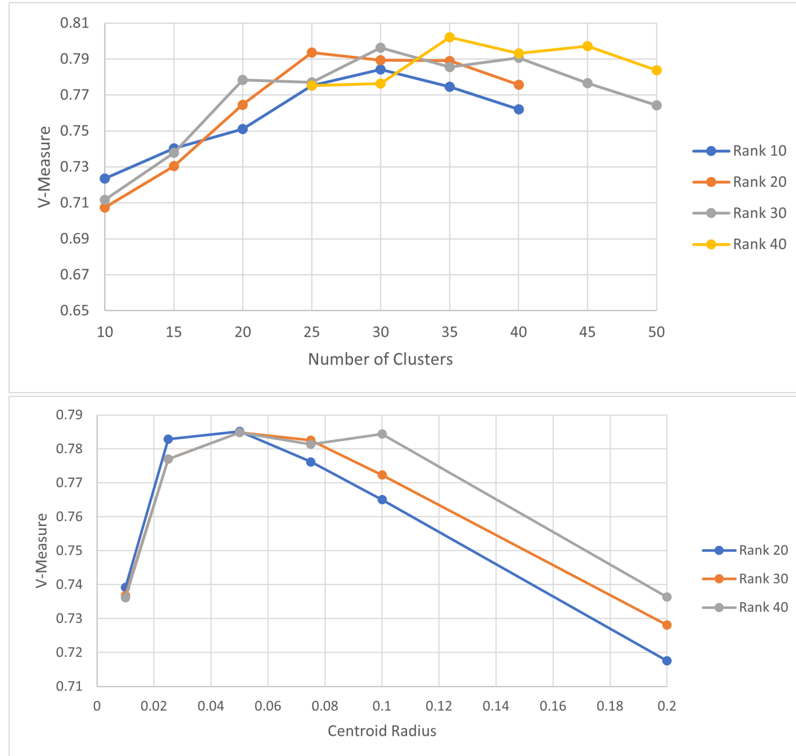

Figure 4.7: V-Measure under K-Means (Top) and Mean-Shift (Bottom) Clustering

Probabilistic Latent Semantic Indexing

A key difference with PLSI influenced how results were obtained. Each document is represented by a mixture of topics not just assigned one. For the purposes of evaluation each file was put in a cluster based on the topic it was most strongly associated with.

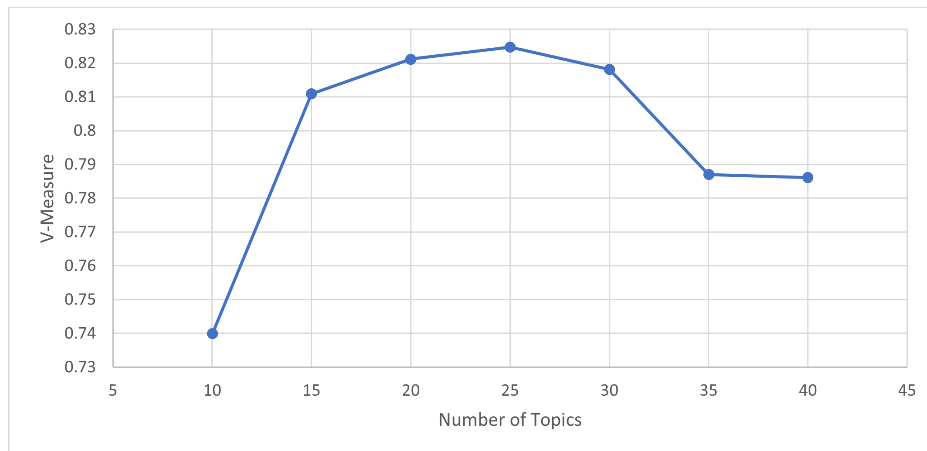

Figure 4.8: V-Measure achieved by PLSI with various topic values

The best result was 0.8248 achieved using 25 topics. This is unsurprising as 25 is also the number of clusters produced from the co-edit data.

Latent Dirichlet Allocation

LDA also outputs a topic distribution instead of a singular assignment. The same steps as with PLSI were taken to evaluate the output. The following displays the V-Measure recorded for as the number of topics is varied.

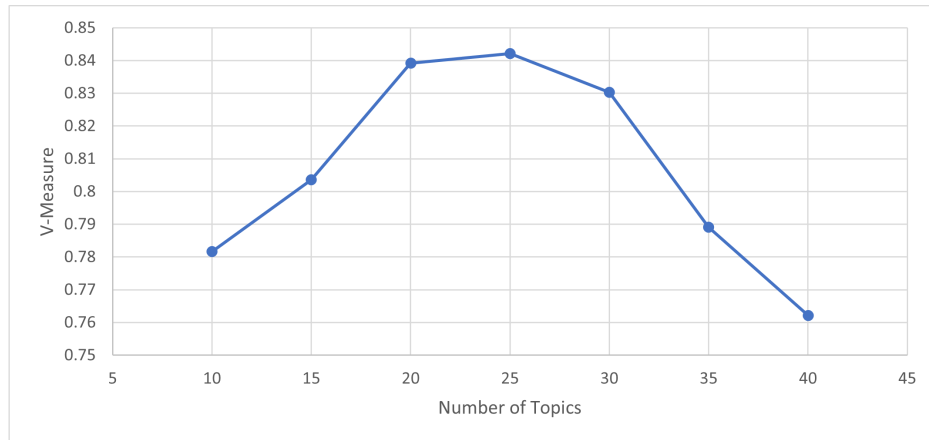

Figure 4.9: V-Measure achieved by LDA with various topic values

The best result was 0.8421 achieved using 25 topics. Again the same value as the number of co-edit clusters. This relationships is problematic; ideally techniques would generalize to any project, being able to generate a good match for the pattern of editing isn't very useful if it can be done when that data is already available. Perhaps an investigation involving a wider range of projects could discover a heuristic based on the size of the project for finding a good configuration.

4.2.2 Aiding Understanding

An alternative use of topic extraction is to aid in understanding a project. Artifacts are meant to be grouped together along semantic, not functional lines. If the topics produced are meaningful and the words chosen to describe them accurate they could serve as a useful tool to a developer.

Throughout the investigation a certain configuration for generating vocab from source code proved to be most effective. Without removing case and punctuation multiple topics would appear described by effectively the same words. A similar division would occur unless identifiers that were compound words were split apart. As one might expect, using the language specific patterns to extract a vocabulary from source code performed better than the language agnostic option.

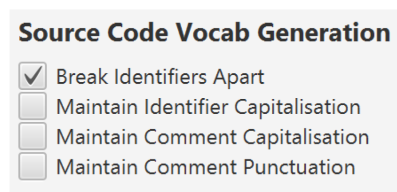

Figure 4.10: Best Configuration of Source Code Vocab Generation Options

One consistent issue was the influence of widespread library use. In both 'Veracity' and 'Burst My Bubble' the React library is used in a large number, but crucially, not all, of the files. In the models where documents are represented by a mixture of topics it was common that UI artifacts would be be significantly made up of a topic represented by React identifiers such as 'Grid' and 'spacing'. Like the language specific set of regex patterns needed to extract vocabulary from source code, the techniques could be updated to also filter out library keywords. Whilst their presence does limit usefulness of output this would massively limit the flexibility of the workbench.

Vector Space Modelling

The starting place of the investigation was seeing what kind of output could be generated without applying the more complex NLP Techniques Discussed below. Unlike LSI term and document vectors were of a different dimension under basic Vector Space modelling, so they couldn't be compared directly. Once clusters were built words to describe them were chosen by finding the term with the highest sum across the vectors.

```
Number of clusters 89

Cluster sizes [1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 3, 1, 2, 14, 5, 1, 1, 1, 3, 7, 1, 7, 2, 2, 1,
1, 1, 1, 1, 14, 1, 1, 1, 2, 1, 1, 2, 1, 1, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 3, 1, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Figure 4.11: Sparse output generated by VSM Topic Extraction

Regardless of clustering algorithm or the value of other parameters this approach consistently generated weak output. Under K-Means clustering the output groups were severely unbalanced and under the remaining algorithms (detailed in Section 3.4) the output was sparse. However, some of the more populated clusters were were partially made up of semantically related files and described by some reasonable terms.

<table>
<tr><td>DiagramAnswer.js, FreeTextAnswer.js, WordCloudAnswer.js, MultipleChoiceAnswer.js, spec.js</td><td>explicit, answered, answer, repeat, choice</td></tr>
<tr><td colspan="2">Veracity - The is a clear link between the various files used to generate UI to output answers and some of the words chosen</td></tr>
<tr><td>FreeTextResults.js, PinResults.js, WordCloudResults.js</td><td>pull, submissions, callback, pins, responses</td></tr>
<tr><td colspan="2">Veracity - Some of the terms relate to all files but pins is just relevant to PinResults.js</td></tr>
<tr><td>ArticleReader.java, Pages.java, PersistentStorage.java, TextAnalyser.java</td><td>document, insert, add, feed</td></tr>
<tr><td colspan="2">BMB - The terms capture how articles are read from rss feeds and added into storage</td></tr>
</table>

Latent Semantic Indexing

Of the three models LSI has both the most parameters to tweak the method and the most opaque relationship between these parameters and the actual output itself. The starting point for the investigation was the configuration described by Kuhn et al. and also used in Section 4.2.1.

Applied to Veracity this initial configuration grouped 97 of the files into one large cluster. As the threshold and rank were adjusted, instead of evenly sized clusters, the output was frequently made up of quite big and quite small topic groups (see Figure 4.12). The best results were with a threshold of 0.6 and the initial value of 10 for the reduced rank, here clusters were a relatively even size and some semantics appeared to have been captured. Some topics were described using semantically related words like quizzes, room, year, new (quizzes are done in rooms and can be created new in the app) or answer, respond, incorrect. There were also related files grouped together. One type of question available in quizzes involves drawing diagrams which are clustered together by ML, much of the code to-do with this part of the app was clustered together despite being split between the back and front end. Similarly from BMB files Pages.java (used to display articles), ArticleReader.java and AzureAdapter.java (used to analyse articles) were all pulled together. However no clusters had both these properties.

All the other clustering algorithms described in 3.4 were applicable to this task with the exception of 'No Going Back'. The Agglomerative algorithms produced useless output when the dendrogram was partitioned into clusters but the dendrogram itself was more useful. Figure 4.13 displays the top portion of one built on Veracity. The tree is unbalanced; one can see clusters relating to small parts of the application like rendering latex or building Word-Clouds that don't branch off. The dendrogram could be useful to a developer in understanding what the atomic parts of a projects and how the fit together.

Cluster sizes [23, 6, 1, 8, 6, 17, 1, 12, 4, 2, 11, 5, 8, 2, 10, 2, 1, 1, 2, 3, 5, 1, 2, 1, 1, 3, 1, 4, 1, 1, 1]

Cluster made up of files [__init__.py, auth.py, ldap_constants.py, spec.js, TopBar.js, Alert.js, Answer.js, IntermediateMsg.js, StudentResult.js, style.js, ClusterDisplay.js, Canvas.js, DisplayGrid.js, FreeTextResults.js, LiveResults.js, MultipleChoiceResults.js, PinResults.js, PinsDisplay.js, WordCloudResults.js, NewQuestion.js, Designer.js, style.js, theme.js]
Summarized by terms: [with, themebreakpointsup, md, marginleft, drawerwidth, 1C drawerwidthpx]

Cluster made up of files [ldap_handler.py, constants.py, spec.js, FreeTextAnswer.js, style.js, CustomBar.js]
Summarized by terms: [flags, dir, enable, graph, def, = tf.compat.v1.graphdef, mayl

Figure 4.12: Example topics and sizes generated by LSI

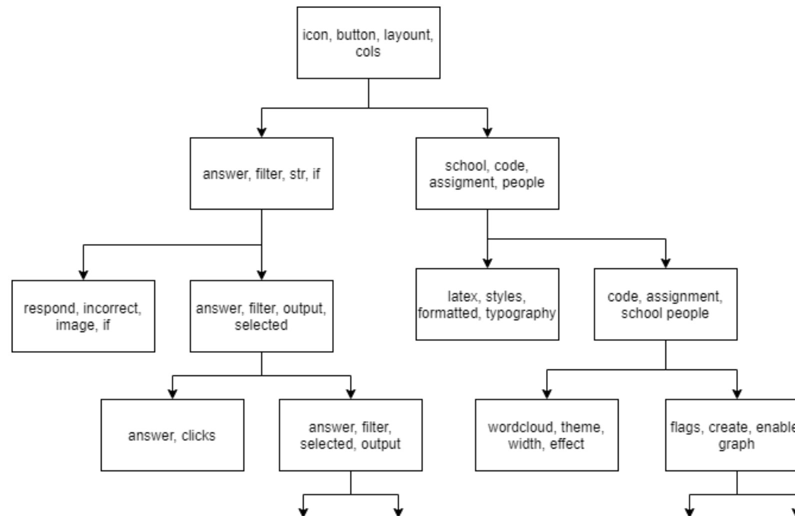

Figure 4.13: Top potion of a dendrogram built on 'Veracity'

Both distance based clustering algorithms produced poor output, consistently grouping the vast majority of files into one cluster irrespective of rank, K-value, radius etc. One explanation for this was that it was the result of all the term vector components being bounded because of the weighting however the same behaviour was observed when using the pure-counts matrix.

Probabilistic Latent Semantic Indexing

The primary difference in applying PLSI instead of standard LSI is that the number of topics must be chosen manually. A further difference is that each file is made up of a mixture of topics, not just assigned to one. Immediately the output appeared to capture some semantic information, Figure 4.14 displays topics described by semantically related words like styles & rgba and quiz, question & responses. As well as the terms the output also included the files most represented by that topic. In some cases these were strongly related files with a clear link to the term, however, sometimes the output seemed more random.

Topic 6: [ldap, app, init, responses, attributes] [ldap_handler.py, config.py, databa
Topic 7: [table, cluster, correct, model, images] [cluster.py, api.js, ControlPanel.js,
Topic 8: [quiz, id, question, responses, methods] [quiz.py, models.py, Question.js
Topic 9: [spacing, styles, make, rgba, updated] [style.js, ClusterDisplay.js, style.js,

Figure 4.14: Example output from PLSI

One topic generated from 'Veracity' was described with terms: test, shallow, expect, describe, display and was most represented by Sidebar.js which renders the Web-Apps sidebar, vectorize.py

which prepares png images for ML analysis and LiveResults.js which displays the responses to a quiz.

Initially 50 iterations of the expectation-maximisation algorithm were used to optimize the model. This was suggested as a reasonable value and increasing it didn't have a noticeable impact on the quality of output [30].

Latent Dirichlet Allocation

From the first application of the technique with it's default configuration the topic descriptions output to the workbench were made up of semantically related words. The topics headings were also things relevant to the project; words like quiz, answer, correct frequently appeared in the output for Veracity and feed, article, category, reader for BMB. Latent Dirichlet Allocation Also requires choosing a fixed number of topics. Intuitively, the ideal number of topics depended on the size of the project. As the number of topics approached the number of files the topic distribution the model captured approached one-topic per document.

Topic 5: [cluster, model, images, answered, answer]
Topic 6: [set, state, grid, icon, alert]
Topic 7: [user, current, login, url, init]
Topic 8: [spacing, styles, make, rgba, dialog]

File	Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
courses.py	0.00	0.00	0.00	0.00	0.00
home.py	0.00	0.00	0.00	0.00	0.00
index.py	0.00	0.00	0.00	0.00	0.00
quiz.py	0.00	0.00	0.00	0.00	0.00

Figure 4.15: Example Topic Descriptions Generated by LDA

Figure 4.16: Sparse Topic Contributions when 100 Topics are used

The right numbers of topics for Veracity, which is made up of around 150 files of code, was about 10-20. The smaller BMB needed only 5-10. One component of Veracity is a set of tools to perform clustering by a variety of methods on hand-drawn images. When the number of topics was set to 10 one was described by the Words (cluster, model, images, image, similarities). Similarly out of 20 topics one was described by (quiz, question, responses, add, response).

Cluster Topic	Top 5 Files by probability of belonging to cluster
Veracity Image Clustering	cluster.py - 0.99, aglo_cluster.py - 0.98, db_scan_clustering.py - 0.93, mean_shift_clustering.py - 0.85, vectorize.py - 0.77
Veracity Question and Response	quiz_model_test.py - 0.79, aglo_cluster.py - 0.98, models.py - 0.67, quiz_api_test.py - 0.43, Question.js - 0.43

4.3 Individual Grouping

The intuition that there would be a need for a method to automatically group together users who were, in fact, the same individual proved correct in these projects. Despite having only 6 direct contributors there were around 40 accounts across Email, Git and Slack for the Veracity project. Similarly there were 15 accounts from the 4 contributors to Burst My Bubble. In several instances the same person appeared multiple times under different aliases in the same service.

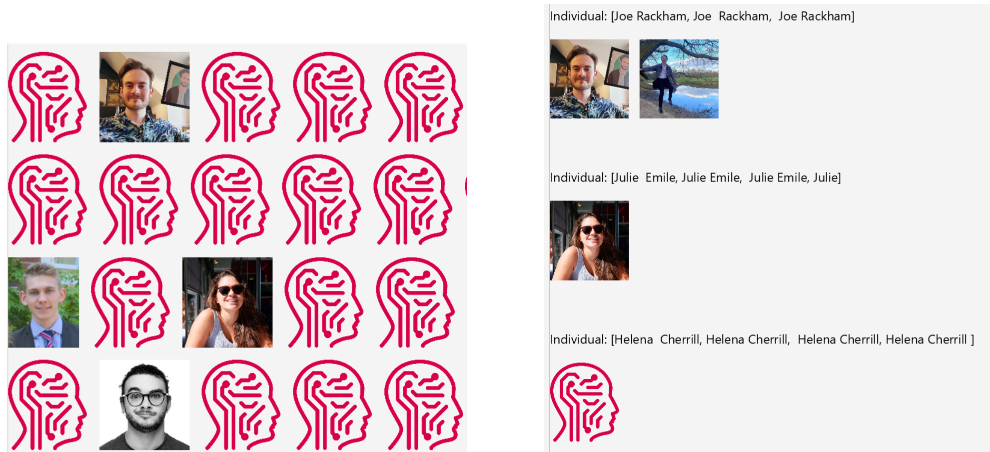

Figure 4.17: Unsorted users and organised individuals

It was considered worse to group two individuals together that aren't the same person than the opposite because this could lead to communication or work belonging to one person being reported as someone else's. Everyone was grouped together correctly from BMB. Applied to the Veracity project the checks achieved a moderate level of success. After some tweaking of thresholds the best results were achieved by requiring two checks to pass. No users were incorrectly grouped together, two individuals were split into two clusters. This was in spite of the fact full names and partial names were used as well as user-ids. One failure that emerged during the application of the process to real users was nicknames. As humans we know 'Richard' and 'Dick' are the same person but these names have limited textual similarity.

4.4 Linking Source Code to other Artifacts

Despite their relatively short duration, a reasonable volume of Email and Slack communication was produced during these projects. Approximately 3000 Slack messages and 300 emails were sent over the duration of Veracity. One might want to search through all this communication to find documents relevant to specific parts of the source code. This could be useful in discovering who had worked on the code, when it was worked on and why certain decisions were made. An investigation was undertaken, applying the various techniques for establishing links introduced in Section 3.6 to the author's projects to evaluate the quality of the links built.

Attempting to establish links with Slack Conversations had the added complication of first building the conversations out of individual messages. Sections 3.3 and 4.2 discuss the best method discovered for doing this.

Regex Based Linking

Following standard practice JavaScript files preset in 'Veracity' and 'BurstMyBubble' used camel case for identifiers. However this also meant that Python files in these repos used snake case. Before this technique was applied the set of Regex patterns was extended to look for uses of snake case.

Despite being the most simple method, Bacchelli et al. reported having the best results applying lightweight regex based techniques. However this wasn't replicated in this investigation. Across the whole corpus of communication this technique consistently generated very few (<10) links.

Looking at the communication there were plenty of references to entities within the code-base but few of them used the proper names. Mostly parts of the app were referred to more informally (see Figure 4.18). This could be a product of the smaller group size working on these projects, the project domain (quizzes and news articles are encountered in everyday life), or the people involved. Either way the success reported by Bacchelli et al. didn't generalize.

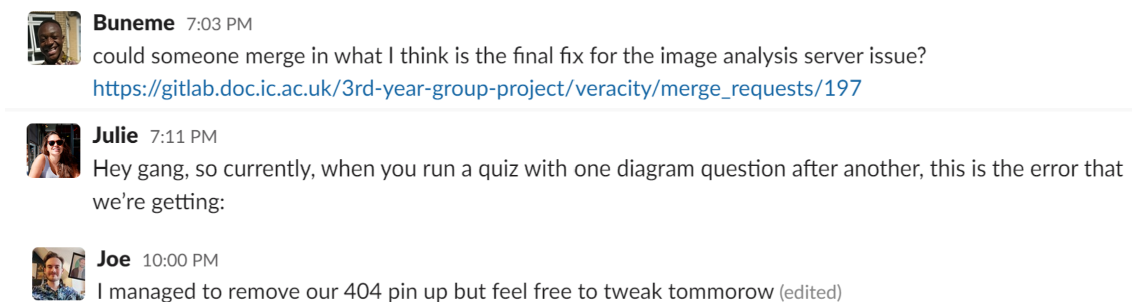

Figure 4.18: Messages from Veracity Slack talking about parts of the app

Vector Space Modelling & Latent Semantic Indexing

Instead of establishing (or not establishing) links between files and documents, under the Vector Space and Latent Semantic Indexing models a way of determining the similarity of every document and source code file is established. Some cut off point can then be chosen to determine artifacts which will be considered related.

After applying these techniques to the communication documents and source code files the output contained lots of files with very high similarity scores. However these top results were populated with links to config files, style files and other parts of the code-base that were either auto-generated or weren't really touched during development. What links were established with other files weren't between artifacts that were actually related. Applying the technique to the Veracity Slack one link with a sim score of 0.9999 was between file ControlPanel.js (which controls UI used by the quiz host) and a message coordinating the time and location of a project meeting.

Experimenting with a range of values for SVD rank, matrix weighting and the other parameters exposed through the workbench failed to have any significant impact on the quality of output.

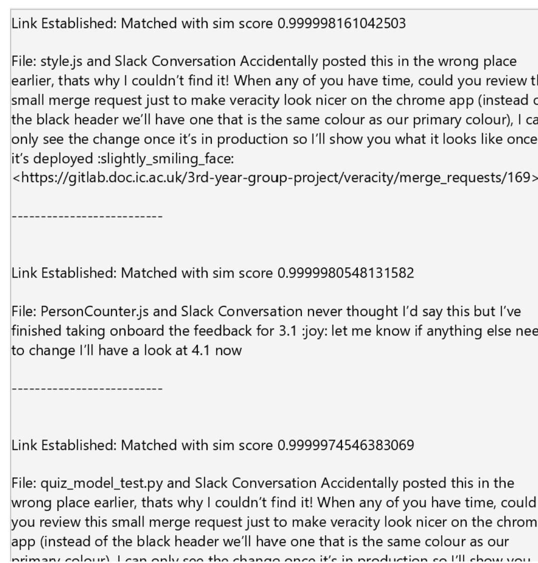

Figure 4.19: High Scoring links established using LSI

Latent Dirichlet Allocation

LDA also produced ranked links between Source Code and communication artifacts. When first applying this technique to the problem the output initially seemed promising. From the Veracity project the highest ranking link was between a related part of the source code and conversation. However the following link related the same conversation and an irrelevant file and the links that followed were similarly poor.

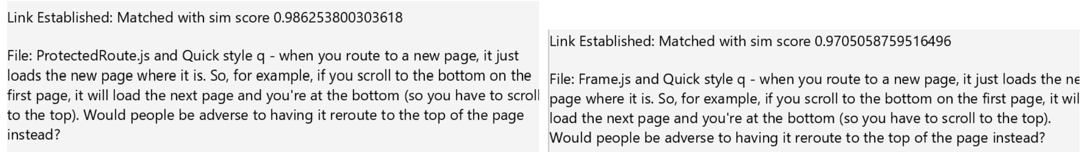

Figure 4.20: High Scoring links with the same conversation established using LDA

Further experimentation with the number of topics and optimisation iterations used by LDA to build it's topic model failed to produce useful links.

The nature of these projects leads to some potential explanations for this output and the lacks of success when compared to the results of Bacchelli et al.. Only terms which appear in the corpus of communication can contribute to the term vector built for source code documents. Whilst the volume of communication isn't insignificant it is smaller than what would be expected for a long running open source project. Only 523 unique terms occur across the entire Veracity conversation history. Figure 4.21 displays part of the sparse tf-idf vectors generated for a portion of the code-base. Additionally, the informality of communication referenced in the previous section my have an impact here. Although using the precise names for entities the techniques wouldn't be able to handle relative references such as "The thing Joe's working on" or "The bug I spoke to you about earlier".

```
[2.103974231809495, 0.0, 0.0, 0.0, 2.7425924249858293, 0.0, 0.0, 0.0, 0.0, 1.887303195000903,
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.887303195000903, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
[2.103974231809495, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
[2.103974231809495, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
[4.20794846361899, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
[75.74307234514183, 14.397049428712647, 0.0, 0.0, 8.227777274957488, 0.0, 0.0, 0.0, 0.0, 0.0,
```

Figure 4.21: Sparsely populated tf-idf vectors

Chapter 5

Summary of Results

The previous Chapters detail how the various techniques for semantic extraction work and some quantitative evaluation as well an investigation applying them to a selection of the author's projects to evaluate the more subjective aspects. What follows is a summary of the insights discovered:

Source Code Topic Extraction

One direction of investigation was grouping files together based on their natural language content and extracting the terms that most represented these groups. This was done by attempting to model the corpus using Vector Space Modelling, Latent Semantic Indexing, Probabilistic LSI and Latent Dirchlet Allocation.

One idea was that these models could produce groups which captured the pattern of co-editing present in Git histories. The techniques described were able to build clusters which strongly mimicked the co-edit data. The groupings were compared using a V-Measure weighted to value homogeneity and a results in excess of 0.82 were achieved. A limitation, however, was that there was no 'one size fits all' configuration to get the best output from the NLP techniques; this was dependent on the project the techniques were applied too.

Another idea was that these techniques would be useful in capturing the semantic topics present in a repository and providing descriptions that could be useful in understanding a code-base. Applied to the author's projects each subsequent model produced a better quality of output, groups that were made up of semantically related files and described by terms indicative of their content. Even under the best model however some output didn't appear sensible.

Disentangling Slack Messages

Extracting each distinct conversation from a Software Development Related Slack is non-trivial to automate. Using a Logistic Regression Model, an annotated data-set of messages and a wide set of features to choose from, the relationship between pairs of messages that form part of the same conversation was modelled. The best configuration achieved an accuracy of 91.8% on a partition of the training data-set held back for evaluation. The features which proved most useful were, understandably, those concerned with the time between the two messages. Without access to time the model's best performance was only 71.4% accuracy.

This success somewhat generalised to the author's projects. The actual pairing together of messages from one project was accurate 68% of the time. When it came to actually building these messages into conversations the best results were achieved using the value output of the model and using a form of clustering in which messages were grouped together in the order they arrived.

Individual Grouping

Across various tools used in development the same individuals can appear under multiple aliases, emails and accounts. It is normally easy for a human to group these together but this is more difficult to do automatically. A series of textual checks are applied onto pairs of identifiers to produce a similarity score which is then clustered to attempt to build groups which reflect real life individuals.

A set of values and thresholds was found that could group group together users without making any false-positive errors. The only false-negatives came emerged from relationships between identifiers that were purely semantic like nicknames.

Linking Source Code to Other Artifacts

Out of a large corpus of communication it is difficult for a person to easily identify artifacts relating to a particular part of the source code. An attempt to automatically build these links was investigated using a variety of methods.

Applied to the authors own projects the linking process proved largely unsuccessful. Lightweight Regex-Based techniques failed because they relied on developers communicating more formally than they actually did. Group members, who were dealing with very 'everyday' problem domains didn't need to use file paths and precise class names to be understood. The more sophisticated natural language based techniques failed to capture the semantics of the artifacts and generated high scoring associations between unrelated documents and files.

Chapter 6

Prototype Tool

One motivation in exploring techniques for the automatic extraction of semantic information from software development artifacts is that these techniques could be used to build useful virtual tools to assist a programmer whilst working. One idea was to build a virtual assistant, somewhat akin to Alexa or Cortana. This would react to the developers interaction with the code-base and surface insights and information deemed to be potentially useful. A prototype version of this assistant was built and is shown being used alongside development work in Figure 6.1.

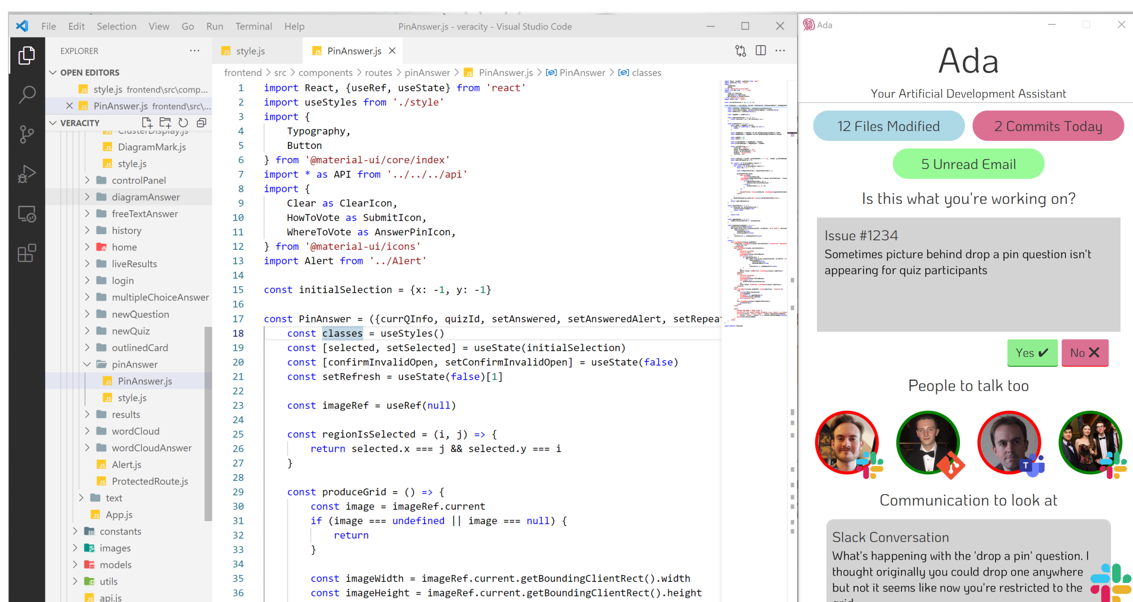

Figure 6.1: Links between emails and source code

The prototype displays insights like recommended communications to look at, which was an area investigated in this project, as well as mocked insights not yet explored, identifying a specific issue number being worked on, as a suggestion of future work. The tool demonstrates how information from the communication services beyond the messages shared within could be surfaced in a useful way. The borders round the portraits symbolise the person's availability; information that could be taken from Slack/Teams and then surfaced. The app also displays information like unread emails and daily commits. The idea is that the app serves as an information hub whilst developing, surfacing value contained within the range of services used and deep-linking to useful artifacts and people.

This prototype is built using the same system as the Workbench but for it to be a real usable application some architectural changes would need to be made (see Figure 6.2). Instead of the users local machine, ML processes would likely be applied externally with results communicated over the internet. This would enable prevent the need to distribute ML models and perform the expensive computation needed for some techniques on user's machines. Furthermore indexes and vocabularies generated from corpora could be stored to avoid costly re-computation.

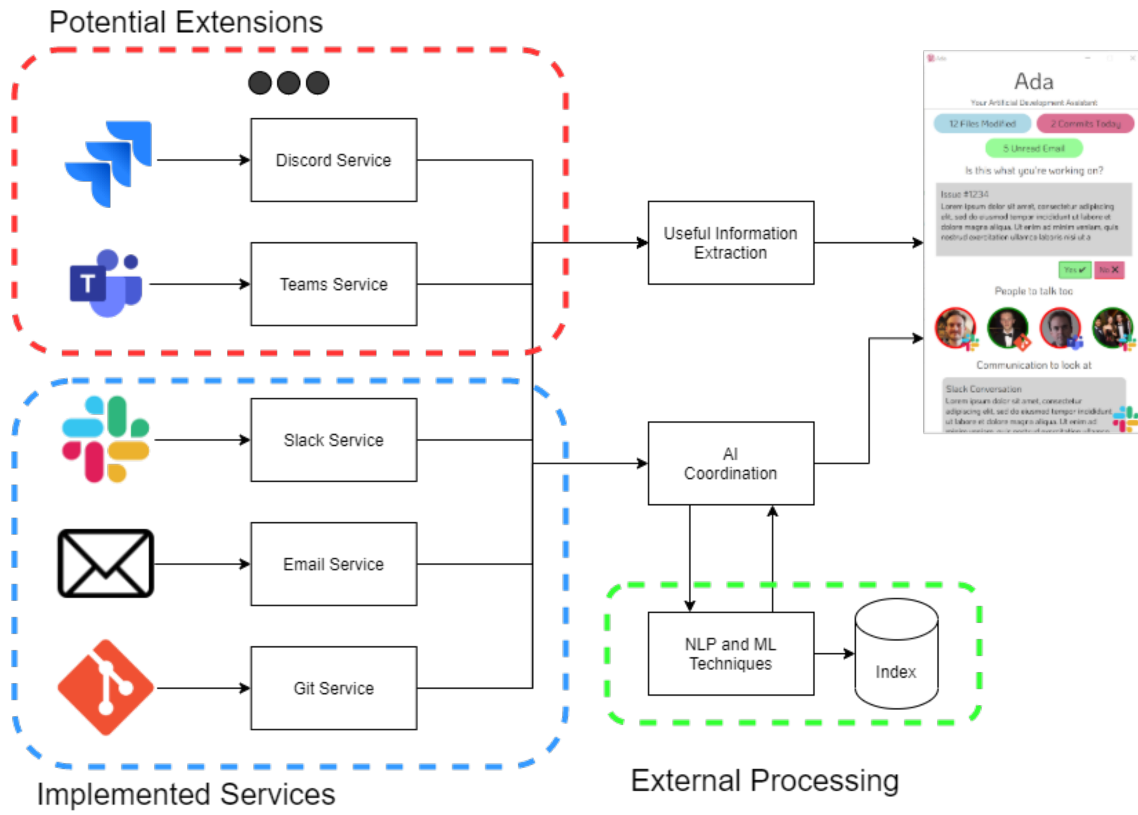

Figure 6.2: Proposed Architecture for Virtual Assistant

Chapter 7

Ethical and Legal Issues

Throughout the project, research and work has been undertaken in an ethical and legal way.

Surveys

During the background research phase of the project, surveys were conducted to determine how software development tools are used in Industry by modern Software Developers. Information was included about how the data was to be used and who would have access to it. No private information or identifiers for the participants were collected. The data is only reported in aggregate both in this report and on the online interactive dashboard (https://serene-stream-63388.herokuapp.com/).

Data-sets

Multiple data-sets are used in the project for model training or the evaluation of techniques. All the data-sets used were publicly available and use for academic work was permitted. Some of these files contained the names and email addresses of real people; care has been taken to not reproduce any of these in the report.

Personal-Projects

The techniques described were applied to some of the author's personal projects the results of which are discussed this report. Permission was obtained from the projects other participants to use their name and image and reproduce any communication sent as a part on the relevant project.

Privacy

The project involved building tools to read messages form various communication tools. Care was taken to not build tools that would allow the Workbench to break privacy. The Slack-App the Workbench is registered under needs to be invited to channels before it can read their history. The mail portion of the app must be configured to read the mail of a specific recipient. There is no potential for private communication using email or Slack direct messaging to be exposed.

Chapter 8

Conclusion

8.1 Contributions

The contributions of this work to the topic of 'Mining Semantic Information from Software Development Artifacts' are as follows:

Survey - A survey of techniques for a variety of tasks under the banner of extracting useful semantic information from artifacts relating to Software Development. The starting point for these methods comes from background research detailed in Chapter 2 but modifications, extensions and alternative methods are presented.

Workbench - A workbench that can be used to investigate the application of these techniques to benchmark data-sets as well as real repositories. Adapters to connect to real email servers and live Slack channels are built in. Full control is given to the user customise how the methods are applied.

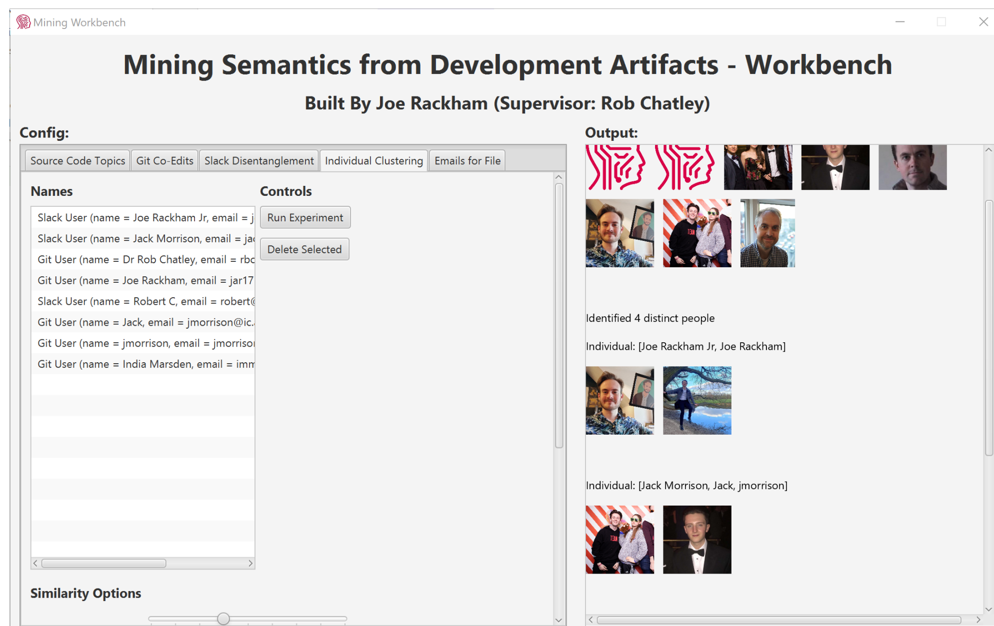

Figure 8.1: Mining Workbench on the grouping users together into individuals

Evaluation - An evaluation of both the starting point of the techniques and the extensions performed using the workbench is presented. Both publicly available data-sets and some of the author's own projects are used to discuss both the quantitative success of various methods and the quality of more subjective output. In several cases the modifications, extensions and alternative methods proposed lead to a high quality of output.

8.2 Future Work

The subject of this project is quite open ended. This section outlines some directions for potential future work building on the discoveries outlined.

More Communication Services - Communication isn't just limited to the services that are explored in this project. Slack and Email were the focus because they were the most popular, but one direction of future research would be to investigate other services and see if they have any unique properties that impact how semantic information is extracted. This could be other text services like Microsoft Teams, but a significant number of survey participants reported using Video Conferencing as a part of developer communication. There would likely be numerous obstacles to extracting useful semantic information from a recorded Zoom meeting, but also lots of value.

Non Communication Artifacts - There are also various types of non-communication based artifacts associated with development that would contain semantic information. An area of future research would be to investigate the likes of issue trackers, documentation, dashboards etc. for techniques of automatic extraction.

Development to the Assistant - One area of future work is to continue developing a digital assistant introduced in Chapter 6 which would use the techniques discussed and information from integrated services to highlight useful information to a developer whilst programming. This could allow for an interesting investigation into how the use of such a tool would impacts the cognitive load placed on a programmer.

Performance - This project has focused on getting the best results from the techniques discussed. However some of these techniques require indexing large corpora of data or applying iterative techniques that need to converge. It can take a long time for any output to be generated or output of a decent quality. One direction of future research would be to investigate methods of speeding up techniques or how the quality of output degrades when time-saving methods are taken. Alternatively, one could look into performing incremental updates to indexed corpora so there isn't a need to recompute everything whenever new Slack Messages or Emails arrive.This would be useful in building tools that could react quickly to a programmer whilst they're coding.

Bibliography

[1] D. Helgesson, E. Engstrom, P. Runeson, and E. Bjarnason. Cognitive load drivers in large scale software development. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 91–94, 2019. doi: 10.1109/CHASE.2019.00030.

[2] SL. Pfleeger and JM. Atlee. *Software engineering - theory and practice (4. ed.)*. Pearson Education, 2009. ISBN 978-0-13-814181-3.

[3] VM. González and G. Mark. "constant, constant, multi-tasking craziness": Managing multiple working spheres. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 113–120, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137028. doi: 10.1145/985692.985707. URL https://doi.org/10.1145/985692.985707.

[4] The value of slack for software developers. [Online]. Available from: https://slack.com/intl/en-gb/resources/why-use-slack/the-value-of-slack-for-software-developers, . [Accessed 11th January 2021].

[5] B. Lin, A. Zagalsky, MA. Storey, and A. Serebrenik. Why developers are slacking off: Understanding how software teams use slack. pages 333–336, 02 2016. doi: 10.1145/2818052.2869117.

[6] V. Stray, N. B. Moe, and M. Noroozi. Slack me if you can! using enterprise social networking tools in virtual agile teams. In *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*, pages 111–121, 2019. doi: 10.1109/ICGSE.2019.00031.

[7] The case for letting people work from home forever. [Online]. Available from: https://www.wired.com/story/the-case-for-letting-people-work-from-home-forever/. [Accessed 12th February 2021].

[8] What's next for remote work: An analysis of 2,000 tasks, 800 jobs, and nine countries. [Online]. Available from: https://www.mckinsey.com/featured-insights/future-of-work/whats-next-for-remote-work-an-analysis-of-2000-tasks-800-jobs-and-nine-countries. [Accessed 12th February 2021].

[9] C. Sadowski, KT. Stolee, and S. Elbaum. How developers search for code: A case study. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1600 Amphitheatre Parkway, 2015.

[10] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10): 970–983, 2002. doi: 10.1109/TSE.2002.1041053.

[11] Adrian Kuhn, Stephane Ducasse, and Tudor Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2006.10.017. URL https://www.sciencedirect.com/science/article/pii/S0950584906001820. 12th Working Conference on Reverse Engineering.

[12] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*,

MSR '06, page 137–143, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933972. doi: 10.1145/1137983.1138016. URL https://doi.org/10.1145/1137983.1138016.

[13] CS. Campbell PP. Maglio, A. Cozzi, and B. Dom. Expertise identification using email communications. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, page 528–531, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137230. doi: 10.1145/956863.956965. URL https://doi.org/10.1145/956863.956965.

[14] A. Bacchelli, M. Lanza, and R. Robbes. *Linking E-Mails and Source Code Artifacts*, page 375–384. Association for Computing Machinery, New York, NY, USA, 2010. ISBN 9781605587196. URL https://doi.org/10.1145/1806799.1806855.

[15] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 63–72, 2012. doi: 10.1109/ICPC.2012.6240510.

[16] P. Chatterjee, K. Damevski, L. Pollock, V. Augustine, and N. A. Kraft. Exploratory study of slack q a chats as a mining source for software engineering tools. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 490–501, 2019. doi: 10.1109/MSR.2019.00075.

[17] M. Elsner and E. Charniak. You talking to me? a corpus and algorithm for conversation disentanglement. In *Proceedings of ACL-08: HLT*, pages 834–842, Columbus, Ohio, June 2008. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/P08-1095.

[18] Arun Surendran, John Platt, and Erin Renshaw. Automatic discovery of personal topics to organize email. 01 2005.

[19] JL. King and AJ. Ehrenberg. The productivity vampires. *Information Systems Frontiers*, 22(1):11–15, Feb 2020. ISSN 1572-9419. doi: 10.1007/s10796-019-09943-9. URL https://doi.org/10.1007/s10796-019-09943-9.

[20] FG. Paas. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of Educational Psychology*, 84(4):429–434, 1992. doi: 10.1037/0022-0663.84.4.429. URL https://doi.org/10.1037/0022-0663.84.4.429.

[21] P. Lenberg, R. Feldt, and LG. Wallgren. Behavioral software engineering: A definition and systematic literature review. *Journal of Systems and Software*, 107:15 – 37, 2015. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.04.084. URL http://www.sciencedirect.com/science/article/pii/S0164121215000989.

[22] M. Kersten and Gail C. GC. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, page 1–11, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934685. doi: 10.1145/1181775.1181777. URL https://doi.org/10.1145/1181775.1181777.

[23] Eclipse mylyn. [Online]. Available from: https://projects.eclipse.org/projects/mylyn. [Accessed 26th December 2020].

[24] Kite for atom. [Online]. Available from: https://www.kite.com/integrations/atom-editor/, . [Accessed 29rd December 2020].

[25] Kite copilot for python. [Online]. Available from: https://www.kite.com/copilot/, . [Accessed 29rd December 2020].

[26] Aroma: Using machine learning for code recommendation. [Online]. Available from: https://ai.facebook.com/blog/aroma-ml-for-code-recommendation/. [Accessed 2nd January 2021].

[27] Guru | slack app dictionary. [Online]. Available from: https://docrevision.slack.com/apps/A0FHVR2R0-guru, . [Accessed 11th January 2021].

[28] Guru. [Online]. Available from: https://www.getguru.com/. [Accessed 11th January 2021].

[29] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, September 1999. ISSN 0360-0300. doi: 10.1145/331499.331504. URL https://doi.org/10.1145/331499.331504.

[30] Java plsa. [Online]. Available from: https://github.com/chen0040/java-plsa. [Accessed 12th May 2021].