Article:
Majumdar, S., Bansal, A., Das, P.P. et al. (2022) Automated evaluation of comments to aid software maintenance. Journal of Software: Evolution and Process, 34 (7). e2463. ISSN: 2047-7473

https://doi.org/10.1002/smr.2463

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Automated Evaluation of Comments to aid Software Maintenance

Srijoni Majumdar*[1,6] | Ayush Bansal[2] | Partha Pratim Das[2] | Paul D Clough[3,4] | Kausik Datta[5] | Soumya Kanti Ghosh[2]

[1]Advanced Technology Development Centre, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India
[2]Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India
[3]Peak Indicators, Chesterfield, UK
[4]Information School, University of Sheffield, Sheffield, UK
[5]Mentor, A Siemens Business, Kolkata, West Bengal, India
[6]TCG Crest, Institute for Advancing Intelligence, Centres for Research and Education, Kolkata, West Bengal, India

Correspondence
*Corresponding author name, Email: srijoni.majumdar@iitkgp.ac.in

Present Address
TCG Crest, Institute for Advancing Intelligence, Centres for Research and Education, Kolkata, 700091, India

Summary

Approaches to evaluate comments based on whether they increase code comprehensibility for software maintenance tasks are important, but largely missing. We propose `Comment`*Probe* for automated classification and quality evaluation of code comments of C codebases based on how they can help to understand existing code. We conduct surveys and document developers' perceptions on the type of comments that prove useful to maintaining software in the form of comment categories. A total of 20,206 comments have been collected from open source `Github` projects and annotated with assistance from industry experts. We develop features to semantically analyse comments to locate concepts related to categories of usefulness. Additionally, features based on code and comment correlation are designed to infer whether the comment is also consistent and not superfluous. Using Neural Networks, comments are classified as Useful, Partially Useful and Not Useful with Precision and Recall scores of 86.27% and 86.42% respectively. The proposed framework for comment quality evaluation incorporates industry practices and adds significant value to companies wanting to formulate better code commenting strategies. Furthermore, large codebases can be de-cluttered by removing comments not helpful in maintaining code.

KEYWORDS:
Comment Quality, Machine Learning, Code Comprehension, Knowledge Graph, Ontology

1 | INTRODUCTION

To solve any maintenance task, developers spend the majority of their allocated time reading and understanding the source code before performing any modifications or enhancements.[1] Even though this process is tedious and worsens in the case of unreadable code, developers often prefer it over consulting documents and trackers which are often inconsistent.[2]

Reading comments along with associated source code can significantly help to comprehend the design of the code and subsequently locate relevant dependencies or change propagations.[3,4] Although comments can be noisy, inconsistent, and may not evolve with the source code,[5] they are still semantically rich and easier to follow and hence the second most-used documentation for software maintenance tasks.[3] Comment analysis approaches have mainly focussed on detecting inconsistent comments[6,7], but not appreciably on the quality and relevance of the information contained in a comment. A poorly written or a superfluous comment duplicating the information evident from source code identifiers can hinder the readability of code, even though it may be consistent.[4] Furthermore, comment quality assessment can also help to develop guidelines for the do's and don'ts of writing comments.

Assessing quality in terms of the 'usefulness' of the information contained in comments can be relative and perceived differently based on the context.[8] Several metrics and features have been proposed to classify comments based on *explicit* syntactic information, such as the presence of specific tags (e.g., @param, @deprecated, etc.), words and symbols; or *implicit* details, such as the type of associated code construct, comment

length, parts-of-speech of comment words or the cosine similarity of words in code-comment pairs.[9,10,11] However, these approaches do not assess the quality of the information contained in the comment in terms of its importance in facilitating code comprehensibility. Bosu et al.[12] attempted to extract such parameters for assessing code review comments only (logged in a separate tool) in terms of how it helped developers write better code through a detailed survey at Microsoft. They also proposed textual features to classify the review comments as *Useful* and *Not Useful* based on the parameters. A similar quality assessment framework is essential to understand the type of source code comments that can help for standard maintenance tasks, but is largely missing and thus forms the main scope of this paper.

In an attempt to understand an unknown segment of code whilst maintaining it, developers attempt to interrelate the programming-related concepts (algorithms, data structures, exceptions, etc.) with software (application) specific entities and operations to build a mental model of the software.[13,14,15] For example, to understand a change request to add a new payment method in a banking module, developers first attempt to locate code that implements the functionalities related to payment (software specific) and then understand *how* it has been implemented, e.g., the algorithm used, or the data-structures updated. These concepts are characterised and grouped as *Knowledge Domains*. Prior authors[15,13,16] have proposed that the software development / general technical (software design, programming, algorithms, processes, etc.) and the software / application specific knowledge domains are relevant for comprehending the application during software maintenance. Many of the concepts related to the knowledge domains and their interrelations often manifest in comments. A comment may contain information related to the temporal state of events, indicators for possible defects, such as overflow, type cast errors, application specific entities, algorithm specific details, and data-structure descriptions. In this paper, we study comments of open source projects from `Github` and explore the software development processes followed in companies and extend the proposed domains[15,13,16] with software evolution and project management related knowledge domains (representative code comments from `Github` in Table 1). A single comment can contain concepts from multiple knowledge domains which can help the developers to understand the design of the surrounding code. Hence, we conduct semi-structured and structured surveys with developers of software companies to understand the nature of the comments in terms of comment categories based on knowledge domains. Furthermore, we investigate which of the categories are *useful* for the execution of present-day maintenance tasks in the software industry. The comment categories form the basis for comment classification and quality assessment.

We introduce `Comment`$Probe$, an approach for the automated detection of quality of code comments based on the comment categories relevant for code comprehension during software maintenance. Textual features have been developed to identify concepts pertaining to relevant categories in comments as obtained from the surveys with developers. To devise features to estimate the repetitiveness, and inconsistencies of the identified concepts, they have been semantically correlated to the code constructs and their structure and attributes in the form of a *knowledge graph*. This is a representation of code construct names (functions, variables, macros, etc.), concepts (from comments and construct), and their interrelationships (using a set of semantic relations). For example, a function (node) and a variable (node) linked with the relation *arguments / parameters* (edge), or an application-specific concept (node) linked with a function (node) with edges *modelled in*. We use a corpus of 20,206 comments collected from 5 open source `Github` projects and annotated using a semi-supervised setup coupled with reviews from developers from software companies. We combine Long Short Term Memory[17] and Neural Networks[18] to train a supervised model and achieve Precision, Recall and F_1 scores of 86.27%, 86.42%, and 86.34% respectively to predict comments in a three-grade normative model for quality - as *Useful*, *Partially Useful* or *Not Useful*.

We focus mostly on C codebases as these languages are mainly used for system-level programming targeted towards efficiency,[19] have lower abstractions[1], and more freedom of semantic expression compared to languages, such as Java and Python. Comprehension tasks in C are more difficult and have a deeper dependence on how well they are commented due to weaker self code readability.[20] Therefore, being able to provide better support through comment quality assessment is relevant.

TABLE 1 Example comments from `Github`[21]

No.	Code Snippet with Comments	Knowledge Domains
1	*/* Array contains* **protease patterns** *for* **amino acid analysis** **/* static prot_struct_p patrn[LEN];	**Genome**, etc. are Application (Molecular Biology) domain
2	*/* uses 2D data matrix (size 8) for light rider bot module @Input: position, visited - matrix @Working: minimum spanning tree @Output: broadcast_matrix */* void flood_fill(ptr, position, visited){ } *// Check overflow issues on count for caller on png_malloc_array with boundary calculations*	Software Development – data structure description, algorithm summary, data dependency, possible exceptions
3	*//* **Barriers** *by* **Sandra Suarez** *for* **CR#147**, *production live in* **CVS# commit#103-9/02** int calculatePOS(_OBJ){ }	version (software evolution), developer details (project management)

[1]As an example of abstraction level, Thread is a class in Java but a function call in C.

The overall aim of this work is to develop an approach to estimate the quality of code comments for software maintenance purposes. In summary, the main contributions of this paper are:

- An exploratory study with the developer community working in companies of various product areas to analyse and formulate a set of *comment categories* that are useful for understanding new and previously unseen C code.
- A *semantic analysis framework* for comments using textual features and code-comment correlation features using a *knowledge graph*
- An automated robust *quality assessment* model for code comments based on a machine learning approach.

Furthermore, we have provided a set of comment categories based on knowledge domains, an exhaustive enumeration of the knowledge domains using ontologies, a ground truth generation approach, and an annotated collection of 20,206 comments extracted from `Github`. Each of these can be used more widely to help investigate other tasks involving comments, beyond maintenance. The source code for feature generation, final feature sheet, saved models and the pre-trained word embeddings are available in Github ([22], [23]).

This paper is organized as follows. In Sections 2 and 3 we discuss related work and important terminologies and concepts. The survey conducted with developers is outlined in Section 4. Feature Extraction, semi-supervised ground truth generation and the automated inference process using the `Comment`$Probe$ machine learning approach is discussed in Sections 5, 6 and 7 respectively. The strengths and weaknesses of `Comment`$Probe$ are analysed in Section 8 and the paper concludes in Section 9 with directions for future work.

2 | RELATED WORK

We discuss some of the approaches that have been proposed to analyse and assess the quality of comments through detecting inconsistencies and classifying comments.

Detecting inconsistencies within source code: Tan et al. [6,24] use the occurrence and sequence of words (from an enumerated list) in a comment and associated code to develop rules for detecting inconsistent comments related to memory access and synchronisation. For example, checking for the occurrence of `lock` before function `foo()` in tokenised code if the associated comment says that lock is used. Similarly, Ratol et al. [7] perform lexical matching between code and comment to detect redundancy of information. The above approaches analyse code and comment using a set of indicator words or the structure to detect irregularities, and do not specifically focus on the various semantic interpretations of the words.

Comment classification: Comments have been studied to identify categories based on the placement of the comment in code or related to the presence of a set of words, a pattern, or special symbols. Ying et al. [25] conduct an empirical study to derive the attributes of the various categories of task comments used by developers working with Java. The authors presented a series of keywords (like todo, fixme) and their likely structure (mention of the bug id, developer details, url, date, etc.) that are used to write comments related to subtasks, short term tasks, problem indicators, concern tags, bookmarks, communication or edge cases. Storey et al. [26] extended the work by Ying et al. [25] with a detailed study to understand how the task comments are interpreted across projects and in the different phases of the software lifecycle. Padioleau et al. [27] manually analyse 1,050 comments that are randomly sampled from 3 open source C projects – Linux, FreeBSD, and Open Solaris to study their general characteristics and categorise them based on memory, locks, data-structure related, errors, control flow, todo or fixme and the like. Aman et al. [28] analyse documentation and inner comments of Java open source projects and enumerate a set of words that mostly manifest in these comments.

Although these approaches attempt to categorise comments, they do not extract attributes to assess the importance of these categories in terms of their use and relevance in the software development process. They target specific types of comments and mostly use manual annotation to categorise comments. Also, small corpora of comments have been used, which may not be representative of all aspects of commenting behaviours and not suited to setting up a robust automated quality assessment framework.

Comment quality evaluation: Steidl et al. [11] use textual features, comment length, line distances between code-comment pair for automated classification of comments (using statistical learning algorithms of Weka [29]) into seven categories – application task, inline, member, header, copyright, code comment, and section / block. Furthermore, they also provide a quality model based on comparing the similarity of words in code-comment pairs using the Levenshtein distance and length of comments to filter out trivial and non-informative comments. Haouari et al. [10] analysed 30 Java code fragments including comments with a survey involving 49 developers where they provided an example comment type or explanation and a comment quality score, such as fair or poor. Rahman et al. [30] propose a framework to detect useful and non-useful code review comments (logged in review portals) based on attributes identified from a survey conducted with developers of Microsoft [12]. They use textual features (Table 2) over an annotated dataset of 1,200 review comments for automated quality assessment using classification and regression tree algorithms. Recent work in the Declutter Challenge of DocGen2 by Liu et al. [31] identifies 'useless' comments using textual and structural features (Table 2) in a machine learning framework.

These past approaches are mostly relevant for comments in Java programs where commenting and source code naming follow a standard protocol. The proposed features in these approaches interpret comments in terms of how the various programming level concepts are encoded and

their significance to developers. However, the dataset used for learning is limited and may not detect all outliers. Even though Rahman et al.[30] assess comments based on attributes derived from industry practices, they are limited to code review comments only. Finally, none of these approaches semantically correlate code with comments to analyse the consistency and redundancy of concepts in a comment, which is also important before deducing the overall quality of the comment. The approaches mostly use the next line to fix scope or check if there is a method nearby (see Table 2). In `Comment`$Probe$ we attempt to address these concerns and provide a taxonomy of comments useful for maintenance based on the various concepts which developers inherently rely upon when coding in C. We study a large dataset of 20,206 comments from open source C projects and question developers working in various product areas to develop a comprehensive set of categories. We propose textual and structural features to detect comment categories and semantically correlate code with comments, based on a vector space augmented code knowledge graph and a robust assessment of comment quality.

TABLE 2 Characteristics of the important quality assessment approaches

Parameters	Steidl et al.[11]	Rahman et al.[30]	Liu et al.[31]
Features for quality detection	1. *c_coeff*: Similar words between code and comment based on levenshtein distance < 2 2. Length of comments	1. Numerical estimation of reading ease 2. Stop word ratio 3. Proportion of interrogative sentences 4. Percentage of code constructs in review comments 5. Lexical similarity between words of code and review comments using cosine distances 6. Number of commits on a file authored by a developer 7. Number of commits on a file reviewed by a developer 8. Percentage of external libraries in review comments	1. Similarity between words of code and review comments using cosine distances 2. Percentage of exact match of words between code and comment 3. Number of lines between comment and nearest associated code 4. Comment Length 5. Type of Comment - Document level, Block and Line
Code in Scope	comment before or after method $\rightarrow$ method name; inline comment $\rightarrow$ constructs in the same line,	code constructs mentioned in review comments	code constructs in next line of comment
Dataset	1,330 comments (C++ and Java)	1,200 review comments logged in Codeflow	1,194 annotated comments from JabRef project[32]
Quality classes	Not Useful	Useful and Not–Useful	Informative and Non-Informative

3 | DESIGN AND ARCHITECTURE OF COMMENTPROBE

`Comment`$Probe$ attempts to learn a robust quality model for C code comments based on attributes that can aid in software maintenance as validated by developers. The major components of `Comment`$Probe$ (Figure 1) are C1: a survey with developers to identify categories based on *knowledge domains*, C2: feature design and extraction, C3: ground truth generation, and C4: machine learning architecture. The inputs for feature design and extraction (C2) are i) comment categories from the developer study, ii) enumerated concepts and ontologies, iii) pre-trained embeddings, iv) the extracted set of comments, and v) corresponding source code. Based on the inputs and using various natural language methods and front-end language parser tools (clang[33]), we formulate textual and code comment correlation features (details in Section 5). The pre-trained embeddings are based on software development texts (component C2.a) to generate vector representations for concepts in comments and code. The ground truth generation has been done through manual annotation and expert review to generate the labelled data (C3) from 20,206 comments. Finally, the neural network architecture trains on the labelled data set to learn a model to classify comments as *useful*, *partially useful*, *and not useful* (C4). Features are extracted from test data (comments and code) and using the saved learned model, `Comment`$Probe$ predicts comment quality as *useful*, *partially useful* or *not useful* (refer Figure 1).

3.1 | Knowledge Domains

Previous authors[15,13,16] have proposed that concepts required to comprehend code during software maintenance can be grouped into three knowledge domains – a) General technical / Software Development concepts about software design (algorithms, control flow, heuristics, patterns, strategies, metrics), programming (languages, data structures, memory, exceptions), software processes (life cycle, testing, review, debugging), b) Application / Software specific knowledge related to the real-world task the software aims to solve and its operations and entities, and c) Business knowledge related to client requirements, client practices, etc. A comprehension process requires programmers to map and interrelate these concepts to construct a mental model of *how* the software works.[16,34]

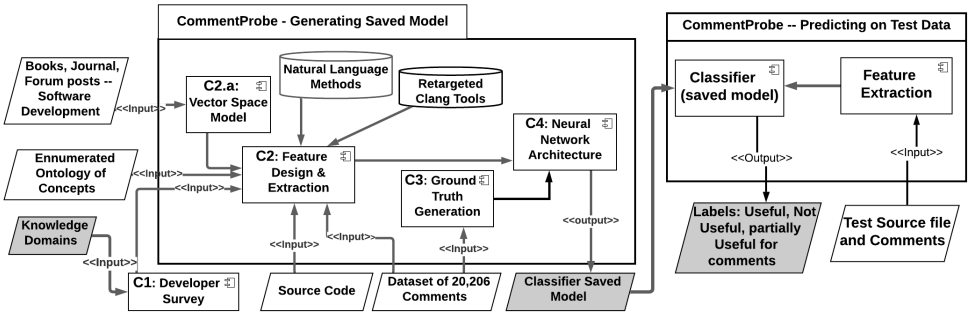


FIGURE 1 Architecture – Comment$Probe$

From developer requirements (analysed in the surveys reported by authors in [35,36,2]) we learn that often software evolution related knowledge (e.g., commits, issues and changelogs) is relevant for maintenance tasks, as they track changes from the inception of software to its present-day state. Also, developer details help to track *who* modified the software. Hence, we extend the knowledge domain taxonomy presented in existing approaches [15,13,16] with software evolution and project management related knowledge domains. The concepts from the domains are mostly found embedded in source code and various metadata, such as comments and issue trackers (Table 3). It is observed that concepts from most of the domains manifest in comments (examples in Table 3).

TABLE 3 Knowledge domains: manifestation in various metadata of a software repository [16,34]

Knowledge Domains	Project Sources	Code Examples
Domains related to application Design and Development		
Software Development (SD) or Program Domain [37] : domain of programming like algorithm, memory, data structure, concurrency, software processes and design, debugging, testing, builds and installations	Source Code, Runtime Traces, **Code Comments**, Bug trackers (Bugzilla, ClearQuest, JIRA), Version Trackers (github, SVN), KT Sessions, Documents	AD concepts (cookies, incognito window) with SD concepts (Reverse-delete) Source: Chromium Project [38] *// Ensure* **cookies** *gets wiped after last* **incognito window** *closes; uses* **Reverse-Delete** *algorithm on node* `CloseBrowser(chr_ptr){}`
Application Specific / Domain (AD) or Problem Domain [37]: entities and operations of the specific application	**Code Comments**, Bug & Version trackers, KT Sessions, Documents	
Domains Related to Application Evolution		
Version Changes: changes and commits for an application	Version Trackers, **Code Comments**, KT sessions	*// fixed* **Bugzilla:7456** $->$ *Bugzilla::Web::Uitimestamp, in commit* **CVS#82_15/07/2017** `thisFrame = Timing::get().frameNumber;`
Bug Fixes: reported bugs, fix summaries along with root cause analysis	Bug trackers, **Code Comments**, KT sessions	
Domains Related to Project Management		
Developer Information: mapping of developer details to bugs, commits, source code elements	**Code Comments**, Bug trackers (Bugzilla, ClearQuest, JIRA), Version Trackers (github, SVN), Documents	Source: cURL [39] *// Copyright (C) 1998 - 2019*, **Daniel Stenberg,** <**xyz@abc.com**>, *et al.* `#include "tool_cfgable.h"`
Business Specs.: company and client details	Induction manuals, emails, KT sessions	
Tacit: personal experiences, like *Client A is tough to handle, System B frequently crashes*	KT sessions & personal interaction	

4 | SURVEYING CODE COMMENTING PRACTICES

We carried out surveys with participants (developers) from the software industry. These had the main objective of formulating a set of relevant comment categories frequently used by developers during maintenance tasks. We manually analysed comments and conducted a pilot study with industry experts to arrive at an initial set of categories. These were subsequently validated through 1-1 interviews with developers and directed examples based on the survey (workflow in Figure 2). Finally, we attempt to define comment quality using survey outcomes.

4.1 | Pilot Study with Key Informants

We adopt the process by Robert Weiss in [40] to first conduct an exploratory pilot survey with experts to design a structured questionnaire that would generate less subjective responses. Companies known to the authors, working in various product areas and using C for software development,

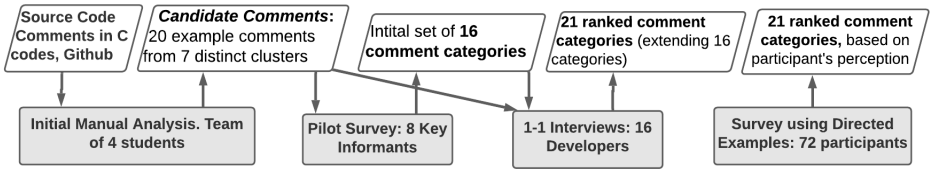


FIGURE 2 Survey Workflow

were approached and invited to participate in the survey. We created a team of 8 *Key Informants* (experts), consisting of technical and module leads who had worked and managed C projects in industry for more than 15 years and were well acquainted with the source code of the products.

Initial Manual Analysis: To use example comments during the interviews (inputs to pilot survey, Figure 2) and gain greater insight, we use Purposive sampling[41] to collect comments (samples) from a diverse range of open source `Github` C projects (our population) to gain maximum variation with a smaller set of samples[41]. For this, we conduct – a) web search for C projects in *Github* belonging to various application domain areas, such as image processing, command line tools and scientific computation; b) search projects based on C compilation strategies, such as `Cmake` and `GNU make` (e.g., Wikipedia history - pages of `Cmake` contain a list of Github projects), and c) C *Github* projects used in various research papers we studied. The effort of manual analysis was subdivided into two teams (each team – 2 graduate students) for execution and corresponding peer review. In the process we manually examine around 1,600 comments and identify around 7 distinct clusters related to data structure operations, working summary, description of function parameters, possible exceptions, mapping to application interface details, macro descriptions and library includes. We select 20 comments from each cluster as *candidate comments* (CandidateComments.csv and 1600_comments.xlsx in[42]), to provide an approximate representation of the various types in a cluster (exception type comments may contain only boundary conditions, or handlers called on detection, etc.). We follow the survey process proposed by Kvale et al.[43] based on interviewing, thematising, and validating.

TABLE 4 Question sets for Interviews in Pilot Study

General commenting behaviours
Do you comment code? Do you update comments when you change code?
Type of comments developers refer to whilst reading and comprehending existing code
Which type of comments do you refer while reading existing code? Which comments you refer to understand the design of existing code? or to fix bugs in existing code?
Concepts from which Knowledge Domains developers typically record in comments. How are they grouped? How are comments placed in code?
Do you write comments for specific constructs? Any specific information, which you record in comments? Do you write comments globally for a file? or inline comments? at the start of every block, method or class? What type of information do you record? Do you use comments to communicate with other developers? Do you write any project management details in comments? Do you write your personal experiences while fixing a bug or enhancing code in the comments?
Perceptions of noisy and inconsistent comments
Do you update comments written by others? Do you remove junk or irrelevant comments? Do you mention your contact details in comments associated with code developed by you?

- *Step 1: Interviews and study of comment examples*: We ask queries related to the use of comments in code comprehension, redundant comments based on themes emerging from the available literature related to commenting practices, enumerated in Table 4. To reduce the subjectivity of their responses, the *Key Informants* showed us sample comments they refer to or write from company source code which we noted using the relevant words and structure (this could not be shared due to corporate confidentiality). They also helped us to map responses to the *candidate* comments wherever applicable. Each interview lasted for around 45 minutes.

- *Step 2: Thematic Analysis*: We aim to deduce comment categories based on the combination of the concepts (belonging to the various *knowledge domains*) from the qualitative responses (interview notes and marked *candidate* comments) of the *Key Informants*. Thematic Analysis helps to locate themes or patterns in qualitative data[44], which we employ to decide on the list of patterns / codes to differentiate comment categories. Concepts related to software development or software evolution knowledge domains are generic in nature, and hence we use an enumerated list to decide on the codes (deductive coding). Codes such as `'array'`, `'matrix'`, `'mega'`, `'hex'` are mostly used in comments to describe dataset or data-structure. However, for application-specific and project management domains, we analyse interview responses for themes (inductive coding[45]).

  According to the inputs from the survey, we observe that a certain type of comment can occur at the block level, as well as inline. Hence we analyse for three placement positions – a) *Inline*: comments added beside code in the same line, b) *Function / Block*: comments placed before the start of a function, class, if-else or loop block, user defined namespaces or any bounded scope (code inside '{', '}') and c) *File*:

start of the file before any `includes` statement. Comments before the start of any class are classified as block level, even though they may be the first comment (block level has higher precedence than file). Codes varied according to comment placements. For example, the codes 'size' and 'bit' were mostly suited for inline comments related to type – dataset / data-structure description. Hence, we employ a hierarchical coding frame[45] for file level, method / block, and inline level and try to infer and collect comment clusters (categories) at each level using the extracted set of codes. Within the collected set, if a cluster (identified by a specific set of codes) is present in the responses of at least 5 informants, we tag it as a valid category; otherwise as `'others'`. Comments related to the summary of functions or dataset description were found in the responses of most informants. However, codes, such as `'gpu number'`, `'hyperthreading'` or `'policies'`, `'sigma'`, `'service'` probably signify hardware characteristics or business policies of clients (these could only be found in the responses of 2 informants). In some cases, the code list of a type of comment was extended, such as terms related to algorithmic complexities were added for the category – working summary / working principle (outline of how the function is designed, input / outputs, etc.).

We conduct three rounds of interviews and thematic analysis for deducing, or merging or deleting categories, until we do not find any distinct categories. We perform a group interview only after the end of the rounds, to reduce the influence of informant opinions.

Validation - Group interview: To validate the categories (themes) and subcategories, we conduct the group interview as a brainstorming session and share the coding frames, *candidate* comments and the sample comments provided by the informants for finalising categories. The categories and their mapping to the knowledge domains as analysed from the pilot survey are enumerated in Table 7.

4.2 | Study I: 1-1 Developer Interviews

1-1 developer interviews were conducted to refine the initial categories as obtained from the pilot study and also to identify any additional ones.

Sampling Procedure: Deciding on suitable developers to interview was crucial to our study. The *Key Informants* supervised the process and we use *heterogeneous purposive sampling*[41] on the following aspects to gain maximum insights and reduce bias: a) developers working in software companies of different product areas and working with C, b) developers with various roles – maintenance, development, management, and c) developers working with different code bases. The employers of the developers participating in the study are characterised in Table 5. The participants (developers) and their roles and years of experience are shown in Table 6.

TABLE 5 Details of software companies involved in the study

Company	Product Area	Programming Languages	Developer Tools	# Participants
Mentor Graphics (I) Pvt. Ltd., Kolkata, India (MG)	EDA	C, C++	vim	6
Tata Consultancy Services, UK (TCS)	IT and Consulting	C, SQL, Perl	Sublime, SAP BI	3
Peak Indicators, Chesterfield, UK (PK)	Analytics	Java, SQL, C	Eclipse, JDeveloper	4
National Digital Library, India (NDL)	Content Mgmt.	Python, C	vim, Sublime	2
HappyWired, Chesterfield, UK (HW)	Office 365, SharePoint	Java, C, C#	Eclipse, JDeveloper	1

TABLE 6 Survey participants (developers)

#Part.	Role	Company	Experience (Years)
P1	Software Programmer	MG	3
P2	Software Programmer	MG	9
P3	Software Programmer Lead	MG	12
P4	Technical Lead & Programmer	MG	15
P5	Architect, Manager	MG	16
P6	Architect, Manager	MG	16
P7	SQL Developer	TCS	10
P8	SQL Developer	TCS	8
P9	Technical Lead	TCS	8
P10	Front end web developer	PK	1.5
P11	Analytics Specialist	PK	6.5
P12	Analytics Specialist	PK	1
P13	Analytics Specialist	PK	3.2
P14	Project Manager	NDL	13
P15	Research Manager	NDL	15
P16	Office 365 Developer	HW	1.5

Survey Design : We conduct two rounds of interviewing and thematising[43] (similar to the Pilot Study, Section 4.1) based on the initial categories extracted from the pilot study and the *candidate comments* from the initial manual analysis (inputs to the developer survey, Figure 2).

a) Interviewing: The objective was to record their perceptions about each category, to analyse and refine the categories further with additions, deletions and modifications. Apart from these, the developers were asked to show which type of comments they use to understand code and solve maintenance tasks (any two change requests or bug fix) using the actual source code and other resources. The comment structure and concepts were noted for further analysis. We interviewed 16 developers at their workplace for around 45 minutes per interview.

b) Thematising: We extend the codes created as part of the pilot survey in Section 4.1 and use the similar hierarchical coding frame[45] to cluster the comments for further interviewing.

We arrive at a set of 21 comment categories (Table 8) and the developers were then asked to select the comment categories they mostly refer to whilst understanding code for bug fixing, or adding or modifying the code (marked as $\times$ in Table 9). Some categories are rarely used or written by developers. We rank the categories based on the count of developers who find it relevant (count of $\times$ in Table 9) and further segregate the count based on the maintenance tasks (bug fixing, adding, modifying code) in Figure 4.

TABLE 7 Comment categories – identified from Pilot Study

File Comments	Function / Block Comments	Inline Comments
a - Summary of the functions defined (relevant mostly for header files) b - Mapping to developer details c - Control flow for the file d - Installation requirements e - Build instructions	a - Working summary (algorithm outline) b - Mapping to application specific entities c - Description of the data being used d - Links to project management details e - Descriptions of external library functions used f - Possible exceptions g - Start and End Marker comments for namespace (user defined), macros, class, function	a - Mapping to application specific entities b - Usage of every import c - Working summary (role in the algorithm) d - Allowed values, possible exceptions

TABLE 8 Comment categories – identified from 1-1 Developer Interviews. Mappings to Knowledge Domains - Software Development (SD), Application Domain (AD), Software Evolution (SE), Project Management (PM)

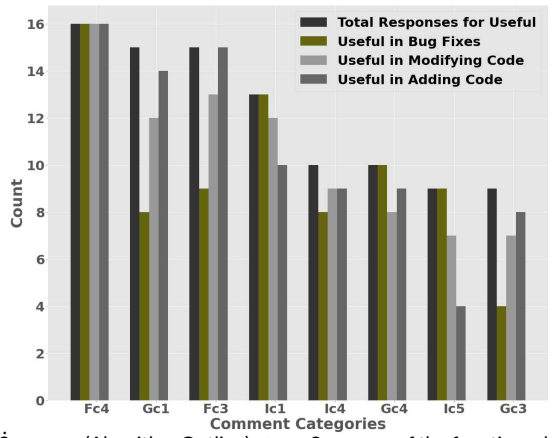
File Comments	Function / Block Comments	Inline Comments
G_{c1} - Summary of the functions defined (relevant mostly for header files) [SD] G_{c2} - External links to documents [AD, SD, PM] G_{c3} - Mapping to developer details [PM, SD] G_{c4} - Mapping to Application Specific Entities [AD, SD] G_{c5} - Basic summary / interaction summary / control flow for the file [SD] G_{c6} - Installation requirements / build instructions / system requirements [SD]	F_{c1} - Data dependency [SD, AD] F_{c2} - Call order of routines [SD] F_{c3} - Working Summary (Algorithm Outline) [SD] F_{c4} - Mapping to Application Specific Entities [SD, AD] F_{c5} - Description of the dataset or global data stores being used [SD, AD, SE] F_{c6} - Links to Commits / Issues [SD, SE] F_{c7} - Descriptions of parameters / return type [SD, AD] F_{c8} - Possible exceptions [SD, SE] F_{c9} - Description of external libraries used [SD, AD] F_{c10} - Markers - namespace, macros, class, function [SD]	I_{c1} - Mapping to Application Specific Entities [SD, AD] I_{c2} - Usage of every import [SD, AD] I_{c3} - Description of external libraries used [SD, AD] I_{c4} - Working Summary (role in the algorithm) [SD] I_{c5} - Allowed values, possible exceptions [SD, AD, SE]
Examples for some categories, more examples in [42]		
G_{c1}: - *file contains stub functions for memory allocation of libpng – raster-graphics file-format*	F_{c5}: - */* works on a two dimensional data matrix (each of size 8) generated from light rider bot module */* void flood_fill(ptr, position, visited) { }	I_{c1}: - static const char *_rl_term_kD; */* Delete key in the webpage - Terminal */*

TABLE 9 Comment categories (Refer Table 8) used by participants (developers) for Software Maintenance

	Survey Participants																
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	#
G_{c1}	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		15
G_{c2}	×	×	×	×	×	×				×		×					8
G_{c3}	×	×	×	×	×	×				×		×				×	9
G_{c4}			×	×	×	×				×	×	×	×	×	×		10
G_{c5}										×		×					2
G_{c6}															×		1
F_{c1}									×			×			×		3
F_{c2}												×					1
F_{c3}	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		15
F_{c4}	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	16
F_{c5}	×	×	×				×	×				×					6
F_{c6}	×	×		×												×	4
F_{c7}								×				×		×	×	×	5
F_{c8}				×						×			×	×	×		5
F_{c9}												×					1
F_{c10}	×			×			×	×	×			×					6
I_{c1}	×	×	×	×	×	×	×		×	×	×	×	×			×	13
I_{c2}			×					×				×	×				4
I_{c3}				×	×			×									3
I_{c4}	×	×	×	×	×	×	×	×	×							×	10
I_{c5}	×	×	×	×	×	×	×	×	×								9
#	11	11	11	13	10	9	8	10	8	10	5	16	7	6	8	5	

× – Participants use this category, # – denotes count

FIGURE 3 Top 8 relevant categories and how they are used in Software Maintenance



Top 8 Categories: F_{c4} - Mapping to Application Specific Entities, F_{c3} - Working Summary (Algorithm Outline), G_{c1} - Summary of the functions defined in the file, I_{c1} - Mapping to Application Specific Entities, G_{c4} - Mapping to Application Specific Entities, I_{c4} - Working Summary (Role in the Algorithm), I_{c5} - Allowed values, possible exceptions, G_{c3} - Mapping to developer details

4.3 | Study II: Structured Survey based on Directed Examples

To validate the dominant categories identified from the developer interviews (input of 21 categories, Figure 2), we record the perception of a larger group of developers[40] for code comment snippets representing comment categories.

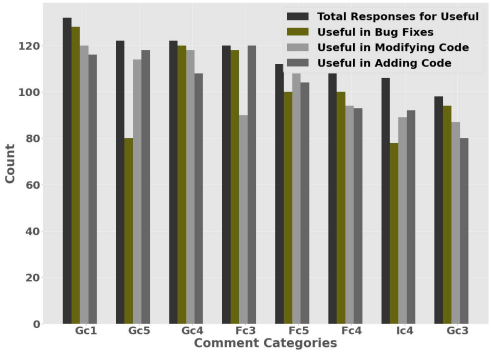
Survey Design: We formulate *Directed Examples* to represent comment categories using the sample comments shared by *Key Informants* and developers during the interviews. As the sample examples could not be shared, similar examples were picked from open-source `Github` C projects, such as `cURL`[39] and `LibPng`[46]. Initially 74 code comment examples were collected. However, after removal of duplicates and irrelevant examples we ended up with 42 examples. The set of 42 comments had 2 examples from each category. The comments and associated source code, together with the query boxes, were set up using the `typeform`[47] crowdsourcing platform and `github.io`[21]. For every code and comment pair, the following queries were asked – a) *Do you think the comment is Useful, Not Useful or Partially Useful?*, response via a *radio button*; b) *Why do you think so?*, subjective response through a textbox; and c) *How would you have written a comment to describe the code?* subjective response through a textbox (screenshot of an example comment in Figure 15 in Appendix). For three comments (in the set of 42), the concepts were repeated in associated code, or could be easily derived from structure or metadata to depict the various types of redundancy (similar to the ones in Table 11), to understand the developer's perception of *not useful* comments.

Analysis of Survey Results: A total of 72 participants responded to the survey for all the 42 code examples (for 21 categories). Every category has 2 examples, hence a total of 144 responses (72×2) were obtained for the same (refer Table 10). The participants were (a) software developers working in companies, such as WANdisco, Synopses and Interra Ltd (32 responses), (b) graduate students (23 responses), and (c) contributors to developer forums (17 responses). Based on the count of the label – Useful provided by the participants, the categories are ranked (top 15 shown in Table 10) and analysed further based on their use in bug fixing, or adding or modifying code from responses for better approximation (Figure 4). Comparing with the top ranked categories derived from the developer survey (Tables 9), there are around 6 to 7 common categories – G_{c1}, G_{c4}, F_{c3}, F_{c4}, F_{c5}, F_{c8}, I_{c4}, I_{c5} (Tables 9 and 10) in the top 8, which are considered as *dominant categories*. Comments that outline the algorithmic details of a function (working summary), map to application specific entities or indicates possible exceptions are perceived most relevant. Furthermore, these categories were relevant for all maintenance tasks (similar counts for bug fixing, adding code or modifying code (refer Figures 3 and 4). The redundant comments were labelled as *Not Useful* in 96% of the responses and *Partially Useful* in the remaining responses. This validates the characteristics of a *Not Useful* comment, which we use to define the parameters of comment quality and design features.

TABLE 10 The categories from Study II (ranked on count of *useful* responses)

Categories	Responses citing usefulness for code understanding
G_{c1} - Summary of the functions defined	132/144 (91.6 %)
G_{c5} - basic summary / interaction summary / algorithm outline / data-flow	122/144 (84.7 %)
G_{c4} - Mapping to Application Specific concepts	122/144 (84.7 %)
F_{c3} - Algorithm outline / working summary	120/144 (83.3 %)
F_{c5} - Description of the dataset or global data stores being used	112/144 (77.7 %)
F_{c4} - Mapping to Application Specific Entities	108/144 (75 %)
I_{c4} - Basic working / role in the design	106/144 (73.6 %)
G_{c3} - Mapping to developer details	99/144 (68.7 %)
I_{c1} - Mapping to Application Specific Entities	98/144 (68 %)
I_{c5} - Allowed values, exceptions	97/144 (67.3 %)
F_{c8} - Possible exceptions	91/144 (63.1 %)
I_{c2} - Usage of every import	84/144 (58.3 %)
F_{c6} - Links to project management details	75/144 (52 %)
I_{c3} - Description of external libraries used	63/144 (43.7 %)

FIGURE 4 Top 8 categories - a) bug fixing, b) modifying, c) adding code



Parameters for Quality: Our aim is to develop an overall quality model. Even though a comment may belong to a dominant category, it may be inconsistent or redundant when correlated and analysed with the associated code. Hence, we introduce comments representing redundancies or inconsistencies as part of the structured survey in Study II (examples in Table 11) to understand developers' perceptions.

TABLE 11 Redundant and Inconsistent Comments (part of Study II)

Comment	Explanation
// PHP Shutdown method to destroy the global php hash map, using zend hash api's PHP_MSHUTDOWN_FUNCTION(hash) { ... zend_hash_destroy(&php_hashtable); . }	**Redundant**: concepts (of *knowledge domains*) in comments already encoded in code constructs
T funct(T x, T y) { *// find maximum of two numbers* if (x > y) return x; else return y;}	**Redundant**: Concepts of comments, easily derivable by reading code (structure)
int i; . . . . (2 to 3 lines)... i = len; *// i is integer*	**Redundant**: Concepts match the datatype of construct
/* serial bus is locked before use static int bus_reset (. . .) { .. update_serial_bus (bus * busR); }	**Inconsistent**: No use of lock in associated code

From the survey we gather that developers perceive a comment as *useful* if it contains concepts: a) related to the dominant categories, b) that do not manifest in associated code constructs, c) cannot be easily derived from reading the associated code, and d) that are consistent. A comment is *Partially useful* if they are consistent, contain concepts from less referred categories or some of the concepts manifest in code also. *Not useful* comments are completely redundant or inconsistent. These parameters provide the basis for feature design in `Comment`$Probe$.

5 | FEATURE DESIGN

The main objective for feature design was to identify the concepts related to comment categories and evaluate whether these were already encoded in associated code (redundant), or are inconsistent with the code (examples of redundant and inconsistent comments in Table 11).We develop textual features to locate comment categories and code correlation features to detect redundancy or inconsistency with the associated code based on syntactic and semantic techniques.

5.1 | Feature Set I – Textual Features for Comment Categories

Textual features have been proposed to capture comment structure and locate concepts related to comment categories.

Features based on comment structure: Features, such as comment length or stop word ratio, have been proposed in comment analysis approaches [11,30] to capture comment structure. We extend the idea and use natural language parsing (part-of-speech tagging, universal dependencies using a standard parser like Stanford [48]) to calculate the total number of significant words (not prepositions or conjunctions), and the number of chunks related to verbs or nouns (complete details in Table 12).

TABLE 12 Feature Set I - Features based on Syntax

No.	Feature	Definition
1	*Comment Length* - Count of comment tokens	$CountofWords(C)$
2	*Significant Words Ratio* - % of words $\neq$ stop words, conjunctions, prepositions etc.	$\frac{\sum_{i=1}^{n}(W_{pos}(C_i))}{CountofWords(C)}$ where `n` $\rightarrow$ total number of comment tokens, $C_i \rightarrow$ `ith` word of comment C. W_{pos} reduced or set to null for POS tags for conjunctions, prepositions, etc.
3	*Operational / Conditional* - Checks whether a comment signifies an operation or condition	$\frac{(n*W_{pos==VB})}{CountofWords(C)} + \frac{(k*W_{pos==ADVB})}{CountofWords(C)}$, 'n', 'k' $\rightarrow$ total no. of verb or adverb chunks using NLTK chunkers [49].
4	*Descriptive* - Checks whether a comment signifies a description or state	$\frac{(n*W_{pos==NN})}{CountofWords(C)} + \frac{(k*W_{pos==ADJ})}{CountofWords(C)}$, n', 'k' $\rightarrow$ total no. of noun or adjective chunks using NLTK chunkers [49].

POS: parts of speech tagger [48], W_{pos}: weight for a POS, C: a comment

Features related to concepts belonging to comment categories: Features have been developed to identify dominant comment categories (based on Tables 9 and 10) by retrieving constituent concepts from comments (these can belong to one or more knowledge domains, Table 8). For retrieval, we need to enumerate the constituent knowledge domains in terms of a list of indicator words or phrases. This is similar to the approaches adopted by Aman et al. [28] and Ying et al. [25] to identify documentation and task comments respectively. We employ syntactic (direct, stemmed, lemmatised) matches or conceptual similarities (synonym table, top 10 similar words constructed from pre-trained embeddings) with the enumerated lists. Matches with concepts for a certain category are multiplied with the weight assigned to the category (based on rank), to get the feature value for that category. This process is applied to concepts from all categories, that are retrieved from a comment. The enumerated list of indicator words and phrases for categories, that are used to calculate the corresponding feature values are outlined in Table 16.

Enumeration of Knowledge Domains: As the concepts belonging to the software development knowledge domain are generic and finite [50], we enumerate them and develop an ontology (`SD Ontology`) using the methodology discussed by Noy et al. [51]. We explore a variety of open-source projects and the standard curricula related to software engineering to enumerate the concepts relevant to developing software in C. To formulate the various clusters/classes, we start with the major concepts of C programming language and then try to find sub categories [52]. For example, *operation* is a superclass with *comparison*, *decision-making*, *arithmetic*, *aggregate* or *data structure related operations* as subclasses (Table 13). Obviously there are concepts which belong to multiple parent classes like *array* is a *built-in* and *linear data-structure* (Table 13). The plural forms, short forms, and tense representation for every word have also been enumerated. We construct a reasonably sized and logical ontology of 4756 instances grouped into a set of 20 classes (representative examples in Table 13, complete list and logical relations between classes (uploaded in [53]). The instances have been distributed amongst the 16 developers (participants of 1-1 interviews, Section 4) for a round of manual review and feedback. Application domain concepts are unstructured and open-ended, and hence are enumerated using business requirement documents for an application. Software evolution details are based on tracker systems used in projects (e.g. bug and version trackers), which we also represent in the form of an ontology.

TABLE 13 Software Development concepts – `SD Ontology` (examples)

Some of the concepts related to algorithm				
Sorting / Searching	**Backtracking**	**Divide & Conquer**	**Dynamic Programming**	**Greedy**
Selection Sort	N Queen	Binary Search	knapsack	Map colouring
MergeSort	M colouring	Quick Sort	Subset Sum	Minimal Spanning
Some of the concepts related to datastructure				
Abstract Data Type	**Built-in Type**	**Linear**	**Non Linear**	**Non Homogeneous**
List	Array	Array	Tree	Structure
Stack	Structure	List	Graph	Struct
Some of the concepts related to operations				
Datastructure related	**Comparison**	**Decision making / loop**	**Aggregate**	**Arithmetic**
Iterate	Greater	If	Delete	Maximum
pop	Greater than	Switch	Power	Summation
Other Concepts				
Scope	**State**	**Output**	**Exception**	**Memory**
Global	Sorted	Output	Bad file number	Contiguous
Static	Encrypted	Outcome	Catch	Buffer
Relationships between classes: Operations based on datastructure, decision and loop making – specialised version of base class operations (is-A), Data-structure (HAS-A) state, Algorithm (USE) Data-structure				

Vector Representations of Concepts: Finding similar words helps us to locate concepts in semantically related comments (examples from survey in Table 15) that cannot be captured through enumeration alone.[54] Pre-trained word embeddings for only software engineering related texts extracted from `Stack Overflow` posts have been reported in work by Efstathiou et al.[55]. We train our own embeddings (`SWVec`) using data from multiple sources related to software development: (i) 19GB of posts from Stack Overflow; and (ii) 11GB of computing and books and papers from our institutional library, related to programming, algorithms, architecture, memory, metrics, etc. We experiment with CBOW[56] (static, context independent) and ELMo (dynamic, context-dependant)[57] algorithms. CBOW[56] uses a log-linear classifier to predict the central word given the prefix and suffix words using a sliding window. ELMo is based on two layer bi-directional language models (BiLM) coupled with character convolutions. Out of the models trained (optimal ones are available in[23]), ELMo performs better for ambiguous words which have dual meaning in English and computer science (results in Table 14). A total of 11 models have been trained using CBOW and ELMo by changing hyperparameters related to dimension sizes, iterations and activation functions. The hyperparameters corresponding to the best model trained using ELMo are `dimension=200, minimum frequency of words=50, activation=relu, model=bilstm, char_cnn, iteration=20, 68 hours on two Tesla P100-PCIE-16GB GPU, 128GB RAM`. The best model for CBOW had hyperparamaters – `dimension=200, sub sampling rate=5, minimum frequency of words=50, activation=relu, and runtime of 21.5 hours.`

TABLE 14 Analysis of pre-trained embeddings

word1:word2	SD2Vec$_{cbow}$	SD2Vec$_{elmo}$	**Work**[55]
Ambiguous Words			
Cookies:Java	0.1	0.44	-0.06
Dirty:Concurrency	0.17	0.67	0.19
Code:Smell	0.35	0.63	-0.03
Smell:Debt	0.29	0.67	0.21
Non-ambiguous Words			
Knapsack:Polynomial	0.35	0.49	0.49
Neighbour:Search	0.23	0.4	0.24

TABLE 15 Semantically Equivalent Comments

C1: bitonic sort to generate a digitalized image in form of raster graphics. Multi-threaded with busy waiting
C2: merge sort to generate a digitalized image in the form of bitmap image. Multi-threaded with busy looping
The concepts bitonic sort merge sort; raster graphics and bitmap image; busy looping and busy waiting all have a similar semantics

5.2 | Feature Set II – Code Correlation Features

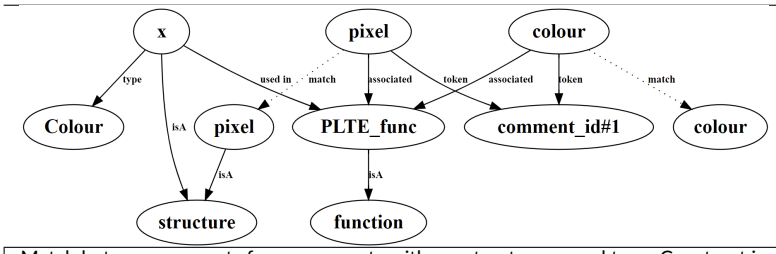
Existing approaches mostly follow syntactic techniques for code and comment correlation based on comment placement (e.g., before or after the method) or comment construct distance[9,10,11]. Hence, the code constructs in scope for a comment are mostly the method name or variable name. To develop features to detect redundancies and inconsistencies of concepts mined from comments, we correlate them to the code knowledge graph corresponding to the code constructs in the comment scope. The code knowledge graph is representative of structural aspects of the constructs (e.g., variable type, storage class, and linkage; or variables defined in a function) and the concepts mined from construct names. In Figure 5, concepts from the comments match the semantic type (`Colour`) of the construct used in the in-scope function, thus rendering the comment as redundant. Linking the comment concepts with only the method name or variable names would not detect this redundancy.

TABLE 16 Feature Set I: Features based *Knowledge Domains* for relevant comment categories (enumerated `SD Ontology` full version in

7 Dominant categories, junk and copyright comment		
No.	**Feature**	**Enumerated List**
1	*Working Summary*	concepts under classes – 'Algorithms', 'Operations as part of Algorithms', 'Operations as part of Data structure', 'Properties of Datastructure', 'Time Complexity / Space Complexity', 'Memory operations', 'Exceptions', 'Threads' and the like, of the `SD Ontology` (Table 13)
2	*Dataset Description*	a) 'Operations as part of Algorithms', 'Operations as part of Data structure' of the `SD Ontology` (Table 13), b) Data type and alloc keywords such as - ["string", "list", "array", "matrix", "memory", "alloc", "malloc", "static", "calloc", "dynamic", "pointer", "binary", "hex", "logs", "buffer", "static", "space", "disk"], c) Dimensions - ["size", "shape", "dimension", "byte", "kilo", "mega", "giga", "tera", "kb", "mb", "gb", "tb"]
3	*Allowed Values, Possible Exceptions*	a) 'Time Complexity / Space Complexity / Memory / Exception' of the `SD Ontology` (Table 13), b) standard error list of C language
4	*Mapping to Application Specific Entities*	*Enumerated list* based on project non-functional requirements and user interface details
5	*Build Instructions / System Requirements / External Libraries*	["gcc", "g++", "make", "config", "build", "install", "mkdir", "cd", "cmake", "git", ".tar", ".gz", ".zip", "cxx", "clang",".dll"], [".h", ".cc", ".hxx", ".hpp", ".cxx",".so"]
6	*Project Management (issues and commits)*	["issue", "commit", "svn", "bug", "jira", "git"], Regular expression based on the issue and version management portal used for a specific issue / commit
7	*Developer Details*	weighted count of matches with developer names using standard libraries for detecting named entities belonging to the cluster *person* (named entity recognition technique [58]), matching with regular expression for contact numbers and emails
8	*Junk, Copyright*	["copyright", "license", "rights", "reserved"], comments with no text

Miscellaneous		
No.	**Feature**	**Value**
9	*Number of Software Development Concepts*	Total count of matches with concepts of `SD Ontology`
10	*Number of Application Specific Entities*	Total Count of matches with the enumerated application specific concepts

weighted counts normalised using mean (μ) and standard deviation (σ). The formula used is $(datapoint - \mu)/\sigma$, It is then transformed to a range of [-1,1] using a hyperbolic tangent function $tanh$




```
Colour x; . . .
//defines type colour for pixel
void PLTE_func(ptr) {
   .....
   x->rgb = PALETTE;
   pixel.def = x->id;

 }
```

Match between concepts from comments with construct name and type. Construct is global and used in function `PLTE_func`

FIGURE 5 Detecting redundant comments using the Knowledge Graph

```
**File: png.h$
1.  // reference colour types for PNG (chunks)
....
14. typedef struct png_color_struct {
15.     png_byte red; png_byte blue;
16.     ...
17.    } png_color;
```

```
**File: pngwutil.c$**
1. #include 'png.h' . . .
6. png_ptr->num_palette = PNG_PAL;....
9. /* palette chunk is used to write .. uses lossless data compression..
14. * Output: png_sPLT_struct[bit_depth]*/
15.   void png_write_PLTE(png_struct p, uint n) {
16.      png_color pal_ptr; ......
21.      png_uint_16 num_pal = png_ptr->num_palette;....
30. // pass color pointer
```

FIGURE 6 $\mathcal{P}_S$: Edited code snippet based on Libpng [46]

Building the Knowledge Graph: The knowledge graph comprises three major components: (a) Primitive Extraction, (b) Concept Analysis and Match, and (c) Scoping and Correlation. We discuss each of the components below with the aid of a code sample $\mathcal{P}_S$ (Figure 6) – a modified code snippet for writing images in the form of chunks from `Libpng` [46] – open source support for Portable Network Graphics (PNG).

a) Primitive Extraction: *Primitives* refer to the comment tokens, or code constructs and their attributes. We re-target the static instrumentation framework of *Low Level Virtual Machine (LLVM)* [33] to create a customised and traversable Abstract Syntax Tree (AST) representation of the code, from which we extract code constructs and their attributes. For example, in code $\mathcal{P}_S$ (Figure 6) the information extracted for $num\_palette$- are – $Attribute - Value$: $\langle def\_line((definition\ line)) = pngwutil.c\#6\rangle$, $\langle acc\_line(use\ line = pngwutil.c\#21\rangle$, $\langle datatype = png\_uint\_16\ \rangle$, $\langle semanticparent = png\_ptr\rangle$, $\langle storageclass = static\rangle$, $\langle linkage = Internal\rangle$. The structural attributes and relationships of some constructs

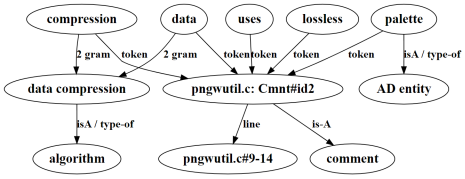


FIGURE 7 Comment Knowledge Graph

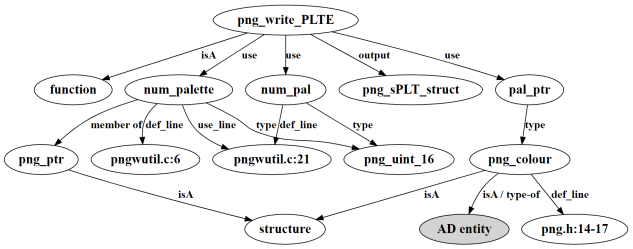


FIGURE 8 Code Knowledge Graph

of the function png_write_PLTE of file pngwutil.c ($\mathcal{P}_S$) are represented in Figure 8. Similarly, Natural Language (NL) parsing is used to pre-process and tokenise comments and extract attributes like line number and POS tags (comment id#2 of pngwutil.c, lines#9-14 with attributes in Figure 7).

b) Concept Analysis and Match: We preprocess comment text, tokenised code and attribute value strings (metadata) of the constructs by removing punctuation and converting to lower case. Concepts from the enumerated ontologies are matched with the pre-processed artefacts using various NLP-based retrieval techniques[59], such as direct or stemmed match, 2 gram matching, synonym table based lookup, etc. Any specific bug or commit number are matched using regular expressions, such as $(\#[0-9a-f]+)|(([0-9a-zA-Z]+:)+[0-9a-zA-Z]+)|(([0-9].)+[0-9])$. Some of the matched concepts in comment and code shown with shaded nodes are shown in Figures 7, 8.

c) Scoping and Correlation: The scope of a comment is determined using a correlation algorithm based on the nearest available character – }, { as they signify the start and end of any bounded scope), the distance between consecutive comments, and the presence of source code identifiers in the comments. For example, a block / method level comment before the start of any scope {, will have in its scope all the constructs till the next } or the next comment is encountered (whichever is nearest). Similarly, for inline comments, the scope is set to all constructs present in the same line. The concepts retrieved from comment tokens are mapped to the code constructs, attributes, and concepts mined from constructs using a set of semantic relations. The comment id#2 of file pngwutil.c (lines#9-14, $\mathcal{P}_S$) occurs just before the start of the method, the scope is set till the next comment. Semantic correlations (eg. associated, type of) of constructs and comment concepts for some constructs within function (`png_write_PLTE`) shown in Figure 9 (details of the visualisation tool, source codes, corresponding readme documents and examples for customisable visualization in[60]).

Code Correlation Features: These are designed by traversing the graph based on semantic relations to find concept matches and inconsistencies (schematic logic in Table 18).

FIGURE 9 Comment id#2 of pngwutil.c (lines#9-14, $\mathcal{P}_S$) correlated with constructs (not all shown). *implements*, *executes*, *associated with* – semantic correlations

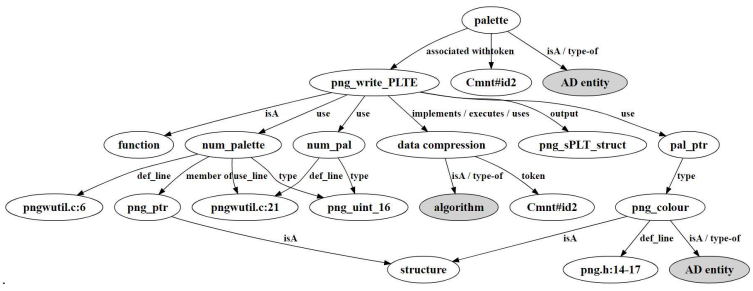


TABLE 17 p-values for features using Mann Whitney Wilcoxon Test[61]

Feature	Useful – Not Useful	Useful – Partially Useful	Partially Useful - Not useful
Comment Length	0.02	0.01	0.00
Significant Words Ratio	0.01	0.02	0.00
Operational / Conditional	0.08	0.02	0.02
Descriptive	0.00	0.02	0.00
Working Summary	0.00	0.02	0.00
Dataset Description	0.00	0.00	0.01
Allowed Values, Possible Exceptions	0.04	0.01	0.00
Mapping to Application Specific Entities	0.00	0.00	0.00
Build Instructions / System Requirements / External Libraries	0.01	0.00	0.00
Project Management (issues and commits)	0.07	0.02	0.00
Developer Details	0.05	0.02	0.02
No. of Software Development Concepts	0.02	0.05	0.02
No. of Application Specific Entities	0.01	0.03	0.00
Junk, Copyright	0.06	0.00	0.01
Code Construct Ratio	0.06	0.00	0.00
Comment Placements	0.03	0.00	0.00
Scope Score	0.00	0.00	0.00
Coherence - Redundant	0.00	0.00	0.00
Coherence - Unrelated	0.00	0.00	0.00
Coherence – Structure	0.03	0.02	0.02

Analysis of Features: We conduct Mann Whitney Wilcoxon Test[61] over the 20 pre-computed features to calculate p-values (refer Table 17) for analysing the distribution of the values to predict between any two classes, such as *useful - not useful*, *useful - partially useful* and *partially useful-not useful* classes. This test has been carried out on the feature values of the complete dataset of 20,206 labelled comments.

Some features, like the ones related to code correlation (Coherence – redundant and Coherence – unrelated) and dominant categories (Mapping to Application Specific Entities, Data Set Description, Working Summary), have p-values less than 0.05 and significant effect sizes (Cohen's d[62]), hence are able to distinguish effectively between all comment class groups (*useful - not useful*, *useful - partially useful* and *partially useful-not useful*). We show the distribution for the feature – *Mapping to Application Specific Entities* in Table 19.

TABLE 18 Feature Set II - Code Correlation Features: Structural and Textual

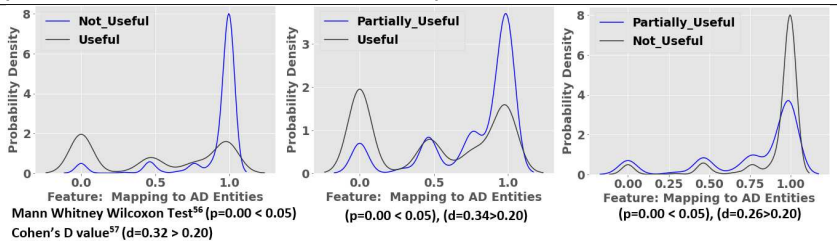
No.	Feature	Extraction Logic
1	*Construct Names in Comment*	$\frac{\sum_{i=1}^{n}(IsConstruct(C_i)==true)}{CountofWords(C)}$, where `n` is the total number of words in comment `C` and C_i is a comment token which matches a construct name
2	*Comment Placements*	Traverse edges related line number to find the `nearest construct`, if same line number and variable type construct returned then inline (a weighted value), if at least a method or a `for, if, while` or '{' returned then method else file level.
3	*Scope Score*	$\frac{1}{1+log(\sum_{i=1}^{k} x_i * distance(x_i, c_{id}))}$, where `k` is the total number of constructs in scope for a comment, x_i is the `ith` construct, distance signifies line number distance (extracted using line relations in graph)
4 & 5	*Coherence - Redundant & Coherence - Unrelated*	$\forall commenttokens$, syntactic and semantic matches with relations (edges) for specific attributes for constructs returned from $Scope(C)$. High score signifies redundancy, very low score signifies unrelated comment
6	*Coherence - Structure*	$\forall commenttokens$, check for syntactic and semantic matches with the data-type, type of the constructs (variable, parameter, methods, etc), type of operators, etc. for constructs returned from $Scope(C)$. Traverses graph using semantic relations like "type", "data type".

KG $\rightarrow$ knowledge graph, Scope(C): returns a list of code constructs in scope for a comment C

TABLE 19 Analysis of Features (Representative Examples)

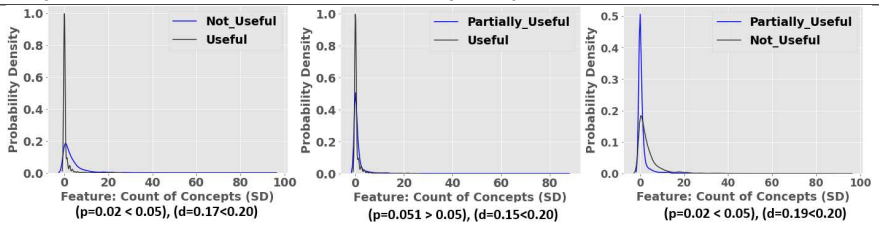
Distribution of feature values[61] $\rightarrow$ distinguishes useful, not useful and partially useful comments, smaller effect sizes

Application Specific Entities if manifested in comments, mostly renders the comment useful



Distribution of feature values does not distinguish between partially useful and useful, effect sizes significantly small

Count of concepts does not differ much in case of useful and partially useful comments



Some features have almost similar distribution of values whilst predicting any of the class groups, e.g., the textual feature - *count of software development concepts* has `p` = (0.051>0.05) to predict classes - *useful - partially useful* (Table 19). However the distributions of the values are different and hence can distinguish between classes – *useful - not useful* and *partially useful-not useful*. In `Comment`$Probe$, we did not retain any feature for training, which has similar distributions of values for predicting all classes - *useful - not useful*, *useful - partially useful* and *partially useful-not useful*.

6 | GROUND TRUTH GENERATION AND VALIDATION

As we could not access the source code of companies involved in our study, we used open source projects from `Github` which supports industry, such as *Big Code*[63] architecture.

Dataset Selection: The mean comment density for C projects in `Github` is 0.18, which would represent a wide range of comments.[64] The comment density signifies the number of lines containing comments divided by the total lines of code for a source file. Selecting the top 10 trending repositories in C was not an option as we wanted specific attributes - a) varied application domains, such as command line tools, visualisation, cloud supported databases, images, b) large number of recent commits, high downloads and active contributors (Github features), c) projects with at least 30% of comments (C code parser (clang tools)[33] to find comments ratio), and d) reasonably well maintained codebases. To characterise a well maintained project, comments part of commit logs are scraped (added comments in `Github` commit denoted by '+', deleted ones by '-') and projects where at least 80% of commits include comment edits are selected. For example, `MariaDB`[65] has 9,524 commits containing comment changes out of the total 10,041 commits).

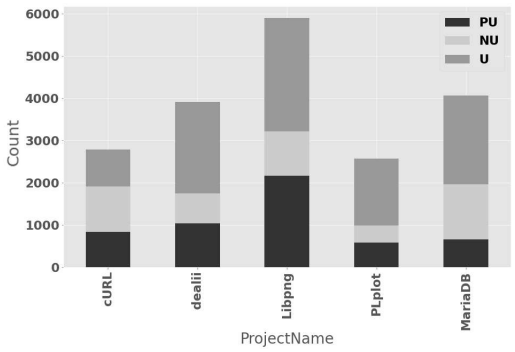


FIGURE 10 Distribution for overall quality scores

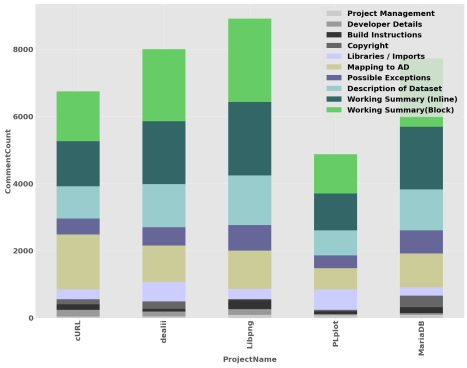


FIGURE 11 Distribution for important comment categories

Based on the above factors, we select 12 projects and sample the comments to check if they contain different concept types. From these we further reduce to 5 projects as characterised in Table 20. From each project, we selected source files using the *modified random sampling approach* of Cochran[66]: $n = \frac{N.pq(Z\frac{\alpha}{2})^2}{(N-1)E^2+N.pq(Z\frac{\alpha}{2})^2}$, where N is the total number of C / C++ files present in an open source project (Lines of Code (LOC) range: 350 - 5000). The random modified (stratified) sampling approach selects source files from the selected 5 projects with equal probability and hence provides an unbiased representation of the population (C code files with comments). As the comment density is reasonable and comment styles are followed uniformly across files (based on manual inspection), a high confidence of around 97% to 98% (Z) is used with a margin of sampling error between 6% and 8%. The parameters were changed for every project and the samples manually re-inspected. We gather a total of 318 files with 20,206 comments.

TABLE 20 Characterisation of DataSet

Project Description	Application Domain	# Files (.c,.cc,.cpp,.h)	# Cmt	% of Cmt
cURL[39]: Command line tool for data transfer with URL	Data Transfer	92	3,781	52.51%
Github Features: Releases - 191, Commits - 25,199, Contributors - 570				
Libpng[46]: Portable Network Graphics, official repository	Image	32	5,875	73.4%
Github Features: Releases - 1615, Commits - 4074, Contributors - 34				
PLplot[67]: Cross-platform, scientific graphics plotting library	Scientific Drawing	27	2559	46%
Github Features: Releases - 147, Commits - 14,011, Contributors - 10				
MariaDB[65]: licensed open SQL server	Database	44	4,060	59.40%
Github Features: Releases - 891, Commits - 189,101 Contributors - 226				
Dealii[68]: Computational solution of partial differential equations	Computing	123	3,901	37%
Github Features: Releases - 31, Commits - 47,959, Contributors - 179				
# → count. cmt → comments. Projects are multi-threaded except Libpng				

Ideally we would have engaged industry professionals to annotate the labels in their entirety; however, this was not possible. Therefore, we designed a three-step approach for generating and validating the ground truth by extending the approach by Pascarella et al.[9] with industry validation: a) annotation by students with high levels of coding skill, b) manual analysis and review, and c) validation by experienced industry developers.

a) *Annotation*: We advertised the annotation task with remuneration at the Indian Institute of Technology, Kharagpur, India. Out of 37 applications, we shortlisted 14 students based on their previous participation in top coding competitions, such as `ICPC`[69] (one of the students was ICPC 2020 world finalist), and various hackathons, including `CodeChef` and `TopCoder`.[70] We design a set of 27 annotation labels (Table 30 in Appendix) which helped to characterise a comment based on the parameters of comment quality. We record the human interpretations of attributes for comment quality detection through the labels, such as comment categories (annotation labels L3: L15), information density (L1:L2), comment structure and scope (L16:L19), and redundancy or inconsistency characteristics (L20:L24). Students were asked to annotate 27 labels (1 or 0) along with a quality score (Useful, Partially Useful, or Not Useful) to reduce subjectivity and also to analyse the human-interpreted relationship between the parameters of comment quality and the assessed labels. Labels, such as information density (L1:L2), can be computed using the ontologies, but in some cases might not be able to interpret an acronym or a concept understandable through human interpretation and hence lead to sound ground truth generation. Each student was asked to annotate 2,887 comments, extracted evenly from the 5 datasets. Every comment was annotated by two students. A total of 156 man-hours were required to complete the annotation process.

TABLE 21 Inspecting Annotation Bias

# Case	Pattern	(%)
Case 1	Overall quality score similar, annotation labels mostly differ for code-comment coherence	18.21
Case 2	Overall quality score similar, annotation labels almost similar	43.77
Case 3	Overall quality score different, annotation labels almost similar	31.03
Case 4	Overall quality score different, annotation labels differ	7.23
Interrater Agreement (κ [71]) between 2 annotators:- 3 quality, 27 annotation labels: 0.734, Interrater Agreement (κ [71]) between 2 annotators:- 27 annotation labels: 0.89		

TABLE 22 Analysis of few of the scenarios of **Case 3** and **Case 4** (Table 21)

Comment File	Comment Text	Annotated Labels	Quality Score	Manual Analysis Score	Manual Analysis
Manual Analysis matches annotated labels					
curl/tests/server/sws.c/sws.c	var which if set indicates that the program should finish execution	L2, L11, L16, L19, L20	U	NU (labels deduce NU)	completely redundant; short; low no. of concepts
curl/lib/curl_path.c	Ignore leading whitespace	L2, L16, L19, L20	PU	NU (labels deduce NU)	completely redundant; short; low no. of concepts
Manual Analysis does not match annotated labels but score					
repos/libpng-code/pngpread.c	/* And check for the end of the stream. */	L3, L9, L11, L14, L17, L18, L21	NU	NU (labels deduce PU)	labels **incorrectly** recorded as partially redundant; high no. of concepts;

U: Useful, NU: Not Useful, PU: Partially Useful, L2: Low Software development concepts, L3: High number of Application Specific Entities, L9: Concepts$\in$ Dataset Description, L11: Concepts$\in$ Working Summary, L14: Concepts$\in$, L16 and L17: Contains more (high) and less (low) source code identifiers in scope, L18 and L19: Long and Short Comment, L20 and L21: Complete and Partial conceptual matches in code-comment pair (details of 27 annotation labels in Table 30 in Appendix), instance from one annotator shown

b) *Manual Analysis and Review*: A team of 2 research scholars (including the first author) randomly selected and analysed 1,000 annotated comments to discuss any disagreements in a weekly meeting with the annotators to help improve annotation consistency. The weekly meetings and brainstorming sessions were conducted until all 20,206 comments were annotated, with re-annotation or edits needed for many. On the final set, considering only the quality scores, we obtain a kappa (κ) of 0.62 (Cohen's metric [71]). For the 27 annotation labels only, κ=0.89 (almost perfect agreement). Considering the 3 quality scores and the 27 annotation labels, we got a κ=0.734 (substantial). The annotation bias is characterised in Table 21, wherein for *Case 3* (31.03%) and *Case 4* (7.23%), there are differences in quality score or annotation and quality score both. For these cases, we manually analyse and group the findings into three types – (a) annotation labels and overall quality scores agree with manual analysis, (b) only annotation labels agree with manual analysis, and (c) only overall quality score agrees with manual analysis (examples in Table 22). In most cases the annotation labels matched manual analysis but the quality scores were incorrectly provided by the annotators.

c) *Validation by Experts*: We set up a review team of 6 software developers, who participated in our company-based surveys. We sampled comments to represent the findings from the manual analysis (like the ones in Table 22) to obtain feedback from the review team and arrange for re-annotation wherever necessary. In the end, we developed a set of 25 rules based on annotation labels (almost perfect inter-rater agreement of 0.89, Table 21) to derive a final quality score (Useful, Partially useful, Not Useful). We present an example rule below based on the labels in Table 30 (Appendix): $(L2 \vee L1) \wedge (L9 \vee L10 \vee L11 \vee L12) \wedge L17 \wedge L19 \wedge L21 \rightarrow U$. Semantic interpretation would be that a comment having – *High* $\vee$ *low density of information* $\wedge$ *contains information related to important company categories* $\wedge$ *low scope* $\wedge$ *long* $\wedge$ *not available in associated code* $\rightarrow$ (is) Useful(U). Based on the calculated score, we obtained a 70.6% match with the manually annotated quality scores for the 20,206 comments.

The proportion of Useful comments are higher for all projects and hence may have a wider interpretation and can be sub-classified further in future (distribution of comment classes for 20,206 after ground truth analysis, Figure 10). The distribution of the important categories in Figure 11 can be used for future automated classification tasks.

7 | AUTOMATED CLASSIFICATION OF COMMENT QUALITY

We experiment with multiple classifiers (Decision Tree, Support Vector Machine (SVM), Random Forest, Artificial Neural Networks (ANN), vanilla Recurrent Neural Networks (RNN), and Long Short Term Memory (LSTM)) to understand how our data behaves and also to understand which architecture learns the quality classes most effectively. We conduct a total of 37 experiments[2] in total (hyperparameters and metrics for some representative ones in Tables 23, 24, 25). We observe that an architecture combining LSTM and ANN provides the best prediction metrics (experiments 6, 7, 8 in Table 23 and experiments L-N1 - L-N4 in Table 24).

Design of the LSTM / ANN Architecture: Liu et al. [31] analyse comment classification by either using pre-computed features or a supervised learning setup using vector representation of comment text as features. We extend this to design an architecture combining features based only on pre-trained embeddings (using `SWvec`, Section 5) along with pre-computed 20 features (comment categories and code-comment correlation, Section 5). This architecture has better precision and recall over any other classifier we experimented with. Each LSTM cell i,[72] has two inputs: the output from the previous cell c_{i-1}, which is called the short term state and h_{i-1}, which is the long term state (the output of each of the previous cells). The gates in a cell are denoted as g_r, g_m, g_n, g_p and the sigmoid and tanh activation functions as σ and tan respectively. We save h_i, the long term state, and concatenate this with the other 20 pre-computed numerical features. The final set of features then serve as input to a two layer ANN network. The output of one of the classes ($class_i$) is calculated as $output(class_i) = g_o . \sum_{n \in N} (g_n . \sum_{j \in J} I_j * W_j + 1) * W_n + 1$, where J denotes total number of input, N denotes total number of neurons, g_o and g_n denotes activation function of every neuron of the output layer and hidden

[2]Compute power: 2 GPU cores: Tesla P100-PCIE-16GB (UUID: GPU-3b616060-59ef-dc83-8aea-667ca0bba599), average runtime - 16 hrs

layer respectively, W_n denotes weight matrix for class i for every neuron and W_j denotes weight matrix for neuron i for every input (Architecture in Figure 12).

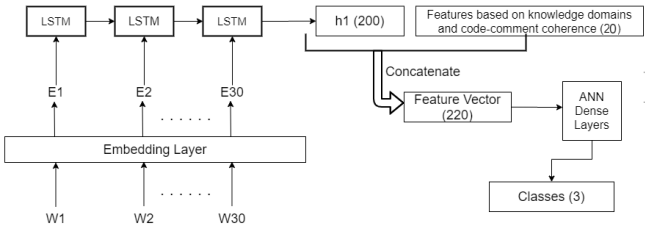


FIGURE 12 LSTM and ANN Combined Architecture

Results and Analysis: For every classifier, we experimented with various loss and optimisation functions, attention models and learning rates to select optimal hyperparameters (a manual grid search). The data distribution among classes is Useful (approx. 10,500), Partially Useful (approx. 4,500) and Not Useful (approx. 5,500). The total dataset is split 80%:20% for training and testing respectively, and validation (5 fold cross-validation), train, and test metrics (precision, recall, F1 score) are reported. For example, for a useful comment label, precision is the total number of predicted useful comments in the total predicted set of comments. Similarly, recall specifies how many of the useful labels have been correctly predicted from the total set of useful comments (ground truth).

Experiments have been conducted with hand-coded decision rules to derive the labels. However, it was difficult to manually build rules based on the range of values of the features to obtain correct inferences for a large set, and this led to significant overfitting (Experiment 1 in Table 23). Using decision tree classifiers (Gini Index and ID3,[73]) improved the overfitting with pruning. We experimented with various splitting strategies and tree depths to improve the results (top result in Experiment 2 in Table 23). We also employ non-linear kernels for the SVM[74] and tune margins to control overfitting by varying the regularisation cost C, from 1 to 10^1, 10^2, and γ from 0.3 to 0.9 (rbf kernel performs better than polynomial, experiment 3 and 4, Table 23). We observe that class separability increases with rbf kernel and retrieves more correct comment labels as the precision is significantly higher for projects compared to earlier results. For ANNs we tested with different sizes of hidden layers, neuron count and activation functions. Combining leaky relu with 3 layers dense network predicts the labels optimally (Experiment 6 in Table 23).

Analysing results of the LSTM-ANN combined architecture: Adding a pre-trained embedding layer to the pre-computed features helps capture the semantics of the concepts better in cases where the enumerated ontology is not sufficient. We experiment with various different loss functions, attention models, learning rates over all the features. The experiments (top 4 prediction wise) are summarised in Table 24 and the best one (overall precision and recall of 86.27% and 8.42%) is analysed further with confusion matrix in Table 25, Figure 13 and Figure 14. The Area Under the Curve (AUC)(Figure 14) is high as a result of the Precision-Recall curve (Figure 13) being high. False Positives could be due to incorrect correlation between the code and comments, or incorrect interpretation of concepts (see examples in Table 26). False Negatives could be due to a failure to identify relevant concepts in $\texttt{Comment}Probe$. Metrics improve slightly by applying feature optimisation techniques such as Principle Component Analysis (PCA) (Table 25). We show the separability of data based on the first four principal components in Figure 14.

Role of Features: We have designed two distinctive feature sets – textual features based on knowledge domains (Set I) and code correlation features (Set II) to predict the usefulness of comments. LSTM investigation – ANN architecture (Experiment 8 in Table 23) show that features related to code comment correlation are a better predictor. However, prediction using only code comment correlation will be limited as it detects only redundant and inconsistent comments based on the conceptual similarities of the comment tokens and code with metadata. However, analysing the usefulness of comments based on the relevant categories can be enforced using the textual features, but results in poorer metrics. Hence, adding the two sets of features helps to predict usefulness based on all the proposed quality parameters, provides improved results, and possibly justifies our feature design.

8 | DISCUSSION

To the best of our knowledge, $\texttt{Comment}Probe$ is the first attempt to implement an *automated comment quality assessment* framework based on the semantic analysis of code-comment pairs in context of their usefulness for software maintenance. We collected a dataset of over 20,000 comments and obtained prediction accuracy of greater than 80% to distinguish useful, not useful or partially useful comments. This could assist companies with de-cluttering source code, we all as develop guidelines for future commenting practices. Furthermore, although $\texttt{Comment}Probe$ has been developed for C, it can be easily extended for C-family programming languages, such as C++, C#, Lite-C and Rust.

TABLE 23 Results (on Test Set) from various experiments (actual experiment numbers not shown)

Dataset	Test Precision (%)	Test Recall (%)	Test F1 Score %
Data Dimension: 20 precomputed features (Feature Set 1 + Feature Set 2), Total set: 20,206 comments, train set: 16, 1642, test set: 4,042, 5 fold cross validation			
Experiment 1: Rule based manually coded decision trees			
cURL [39]	72.24	68.19	70.16
Libpng [46]	70.15	67.03	68.55
PLplot [67]	61.03	63.45	62.22
MariaDB [65]	59.15	59.47	59.31
Dealii [68]	57.21	55.55	56.37
overall	63.956	62.738	63.32
Experiment 2: Decision trees - Gini Index with pruning, splitter = best, depth = 26			
cURL [39]	76.12	78.99	77.52
Libpng [46]	78.89	75.03	76.91
PLplot [67]	67.88	62.33	64.98
MariaDB [65]	68.45	67.71	68.07
Dealii [68]	66.01	66.69	66.34
overall	71.47	70.15	70.77
Experiment 3: Support Vector Machine, rbf kernel, C = 100, gamma = 0.5, hinge loss, 5-fold cross validation			
cURL [39]	76.01	74.54	75.27
Libpng [46]	69.1	70.13	69.61
PLplot [67]	70.11	71.36	70.73
MariaDB [65]	78.2	73.21	75.62
Dealii [68]	72.1	73.02	72.56
overall	73.10	72.45	72.78
Experiment 4: Support Vector Machine, poly kernel, C = 100, gamma = 0.1, hinge loss, 5-fold cross validation			
cURL [39]	61.9	61.78	61.84
Libpng [46]	59.12	58.34	58.73
PLplot [67]	64.45	65.08	64.76
MariaDB [65]	67.13	66.98	67.05
Dealii [68]	68.02	66.27	67.13
overall	64.12	63.69	63.91
Experiment 5: 2 layer neural network (ANN), loss = logistic, optimiser = rmsprop, activation = tanh, leaky Relu with softmax, 5-fold cross validation			
cURL [39]	70.13	72.34	71.24
Libpng [46]	69.56	70.31	69.94
PLplot [67]	72.11	72.08	72.10
MariaDB [65]	70.18	70.76	70.47
Dealii [68]	71.78	70.00	70.89
overall	70.752	71.098	70.93
Experiment 6: 3 layer neural network (ANN), loss = categorical cross entropy, optimiser = adam, activation = tanh, leaky Relu with softmax, 5-fold cross validation			
cURL [39]	78.14	79.03	78.58
Libpng [46]	81.12	80.15	80.63
PLplot [67]	79.27	78.33	78.80
MariaDB [65]	80.02	79.43	79.72
Dealii [68]	78.98	78.00	78.49
overall	79.50	78.98	79.25
Experiments with ANN and LSTM architecture, using either Feature Set I or Feature Set II with 200 sized embeddings			
Experiment 7: LSTM embedding layer of size 200, 2 layer neural network, loss = categorical cross entropy, optimiser = rmsprop, activation = tanh with softmax, 5-fold cross validation, **only textual features on comment categories**			
cURL [39]	69.67	69.99	69.83
Libpng [46]	77.12	75.25	76.17
PLplot [67]	76.14	79.15	77.62
MariaDB [65]	70.24	69.14	69.69
Dealii [68]	71.35	72.48	71.91
overall	72.90	73.20	73.05
Experiment 8: LSTM embedding layer (size 200), 3 layer neural network, loss = categorical cross entropy, optimiser = adam, activation = tanh with softmax, 5-fold cross validation, **only code correlation features**			
cURL [39]	76.54	78.76	77.63
Libpng [46]	79.13	79.58	79.35
PLplot [67]	77.79	80.12	78.93
MariaDB [65]	74.42	68.11	71.12
Dealii [68]	70.02	68.65	69.32
overall	75.58	75.04	75.31

TABLE 24 Top 4 results using the LSTM-Neural Network Architecture (Figure 12)

Common Parameters: Dataset: 20,206 comments, 20 features (textual + correlation), LSTM embedding layer (size 200), HyperParameters: Batch Size - 100, Num Epochs - 600, Loss - Categorical Cross-Entropy, Output Layer of size 3 with Softmax activation, 5-fold Cross Validation

Experiment L-N1

2 Hidden Layers(size, activation) - [(64, LeakyReLU), (64, relu), Dropout: 0.4, Learning Rate: 0.0001, Optimiser: adam

Metric	Useless	Partially - Useful	Useful	Avg = macro
Validation Precision	82.86	80.83	91.79	85.16
Validation Recall	84.26	87.21	87.52	86.33
Validation F1-score	83.55	83.90	89.60	85.69
Test Precision	84.94	80.70	91.59	85.74
Test Recall	84.56	84.98	89.21	86.25
Test F1-score	84.75	82.78	90.39	85.97

Experiment L-N2

2 Hidden Layers(size, activation) - [(64, LeakyReLU), (128, tanh), Dropout: 0.5, Learning Rate: 0.0005, Optimiser: adam

Metric	Useless	Partially - Useful	Useful	Avg = macro
Validation Precision	81.57	82.30	90.83	84.90
Validation Recall	84.16	84.39	88.51	85.69
Validation F1-score	82.84	83.33	89.66	85.28
Test Precision	82.44	84.29	90.69	85.81
Test Recall	85.56	83.65	89.57	86.26
Test F1-score	83.97	83.97	90.13	86.02

Experiment L-N3 – Best Results (obtained empirically)

2 Hidden Layers(size, activation) - [(64, LeakyReLU), (64, LeakyRelu), Dropout: 0.5, Learning Rate: 0.00005, Optimiser: rmsprop

Metric	Useless	Partially - Useful	Useful	Avg = macro
Validation Precision	85.03	85.14	89.47	86.54
Validation Recall	82.30	85.40	90.64	86.11
Validation F1-score	83.65	85.27	90.05	86.32
Test Precision	84.40	83.46	90.94	86.27
Test Recall	83.69	85.24	90.33	86.42
Test F1-score	84.04	84.34	90.63	86.34

Experiment L-N4

3 Hidden Layers of (size, activation) as - [(128, LeakyReLU), (64, tanh), (16, LeakyReLU)], Dropout: 0.5, Learning Rate: 0.00005, Optimiser: rmsprop

Metric	Useless	Partially - Useful	Useful	Avg = macro
Validation Precision	84.31	83.78	90.38	86.16
Validation Recall	86.52	84.22	89.14	86.63
Validation F1-score	85.40	84.00	89.76	86.39
Test Precision	84.27	80.57	89.96	84.93
Test Recall	79.79	85.40	89.54	84.91
Test F1-score	81.97	82.91	89.75	84.88

TABLE 25 Analysis of the best obtained metrics – from Experiment L-N3

Total data Dimension: 20167 X 221, 3 classes (embedding dimension), result for **Test Data**

Dataset (%)	Precision (%)	Recall (%)	F1 Score %	MCC Score %
cURL [39]	85.12	83.21	84.15	76.54
Libpng [46]	91.46	92.67	92.06	81.82
PLplot [67]	92.88	92.14	92.51	82.31
MariaDB [65]	80.72	81.23	80.97	79.19
Dealii [68]	80.96	83.24	82.08	74.2
overall	86.22	86.49	86.36	78.81
overall $_{PCA}$	87.06	86.99	87.02	79.02

Confusion Matrix

	Not Useful	Partially Useful	Useful
Not Useful	395	32	45
Partially Useful	27	439	49
Useful	46	55	943

TABLE 26 Analysis of False Positives in `Comment`$Probe$, Examples taken from Libpng [46]

Example	Human Interpretation	`Comment`$Probe$ Interpretation
/* Check the buffer size (file signature / header / data / crc) is as expected */ `switch(io_state & PNG_IO_MASK_LOC) {case PNG_IO_SIGNATURE: if (data_length > 8 && err != 0) ... png_error(png_ptr);}`	Constructs in Scope for the comment = {io_state, PNG_IO_MASK_LOC, data_length, err }	All constructs till next comment, but `png_error` → global catch function; extern `png_ptr`, not related to checking buffer size. Different scope → different semantic analysis → different quality score
/* Calculate a reciprocal - used for gamma values. This returns * 0 in **order** to maintain an undefined value; * there are no warnings. */ `PNG_INTERNAL_FUNCTION (png_fx_pt, png_reci);`	order is not picked up as a concept (which is correct)	order is picked as the 8th most similar word to sort (4th most similar) using the embeddings from `Sd2Vec`, Section 5.

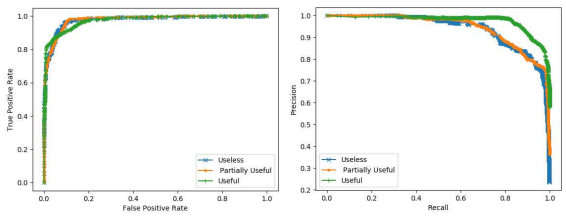


FIGURE 13 Test Data | Best Configuration | Precision Recall Curve | ROC AUC Curve

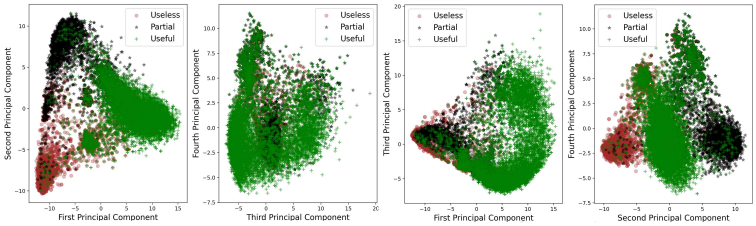


FIGURE 14 Class Separability Plots

TABLE 27 Comparison of Comment$Probe$ with existing comment classification and quality assessment approaches

Parameters	Comment$Probe$	Steidl et al. [11]	Pascarella et al. [9]	Liu et al. [31]	Padioleau et al. [27]	Rahman et al. [30]
Comment Classification	21 categories on knowledge domains	16 categories – task, purpose, metadata	Documentation and Inner comments	×	15 categories for linux OS's – memory, control flow, todo, etc	×
Quality Analysis	Useful, Partially Useful, Not Useful	non-informative comments	×	Informative and Non-Informative	×	Useful, Not Useful
Aspects for Quality Detection, features detailed in Table 2	Knowledge Domains of Software Maintenance + redundancy, inconsistency with code	redundancy with code, comment structure	×	comment placement in code, redundancy with code	×	comment authorship, comment evolution, redundancy with code
Comment Analysis Approach	Syntactic, Semantic, code and comment knowledge graph	Syntactic, Semantic	Syntactic	Syntactic, Semantic	Syntactic	Syntactic, Semantic
Automated Analysis	Neural Networks	Decision Trees	Manual Analysis	Bert, FastText, CNN	Manual Analysis	Random Forest, Naive Bayes
Developers Involved for Quality Aspects	Interviews: 16 Developers, 5 Companies; Crowdsourcing: 72 developers, students, 4 companies	Interviews: 16 developers, 2 research labs	×	×	×	interviewer: 7 developers, 1 company
Language [No. of Comments]	C, [20,600]	C++, Java, [1330]	Java [2000]	Java [1194]	C [1050]	Review Comments of Codeflow, Microsoft[1200]

TABLE 28 Evaluation and comparison of classification results using Comment$Probe$, work by Rahman et al.[30], Steidl et al.[11] and work by Liu et al.[31]

The features proposed by Rahman et al.[30], Steidl et al.[11] and Liu et al.[31] are used to classify Useful and Not Useful comments of Comment$Probe$, Partially Useful comments are not used			
(F1 Score %) Rahman et al.[30]	**(F1 Score %) - Steidl et al.[11]**	**(F1 Score %) - Liu et al.[31]**	**(F1 Score %) - Comment$Probe$**
72.19	66.21	70.21	89.10

Comparing Comment$Probe$ with existing approaches (Table 27), we highlight some of the main differences. Steidl et al.[11] define 7 comment categories mostly based on syntax and placement of comments, such as header, method level, inline, copyright, section, code, and task. Pascarella et al.[9] define a taxonomy of 16 categories that are related to purpose, IDE style and syntax, metadata (e.g., ownership and license), task-based, exception, or alert based on the analysis of comments from 56 Java open source projects. Some of the categories are logically similar to the ones proposed in Comment$Probe$, but the parameters to categorise them are different. For example, for the exception category, Pascarella et al.[9] use explicit tags (e.g., @exception and @throws) as identifying features, However, we use conceptual similarities with various exceptions in C (such as bad allocation, memory corruption and buffer overflow), along with syntax (throws or catch). Furthermore, Comment$Probe$ contains different categories related to application specific entities, or descriptions of the library includes and dataset description, which are based on the concepts and types that developers actively refer to whilst understanding existing code. On the other hand, Pascarella et al.[9] provide an exhaustive view of the various types of comments based on their general structure as followed by Java programmers, irrespective of their utility in understanding the design and development aspects of a codebase. Padioleau et al.[27] highlight categories, such as memory and data structure specific to C projects, and hence there are few categories in common with Comment$Probe$.

Empirical Evaluation: The features proposed Steidl et al.[11], Liu et al.[31] and Rahman et al.[30] (Table 2) for comment quality detection have been developed, and evaluated on our datasets to classify useful and not useful comments in Table 28. Some of the features proposed by Rahman et al.[30] are applicable to code review comments, such as the count of commits and authorship by developers, which we have omitted. We observe that the structural features based on comment text or similarity of concepts between code and comments in the existing comment quality approaches[31,11,30] do not perform well to interpret the semantics of the comments in the context of their relevance in software maintenance. Hence the features in Comment$Probe$ has been designed to interpret the concepts related to relevant knowledge domains which can aid in software maintenance coupled with the semantic and structural aspects.

In the subsequent text we further assess and compare the existing approaches using scenarios and examples various examples. Broadly, the approaches attempt to infer quality using a limited set of features, over a smaller dataset that cannot be scaled for real life applications.

a) Comment quality assessment by Steidl et al. [11] : The quality parameters to detect redundant comments defined by Steidl et al.[11] have been extended in Comment$Probe$ with more parameters related to inconsistencies and type of concepts based on developer study. For example, the comments below are labeled as not useful due to redundant concepts in [11] and also in Comment$Probe$.

```
/** removes all defined markers **/
public void remove All Markers ( ) {...}

// merge
CloneGroup merged = new CloneGroup ( ) ;
```

However, ontologies and vector representations used to develop features in Comment$Probe$ enable the semantic and conceptual analysis of code and comment concepts. Words like “buffer” and “memory” are conceptually similar, “union" and “structure" are related (the same type of datastructure), but are not detected in the work by Steidl et al.[11] which uses the Levenshtein distance (less than 2) to find similarity. Hence in the following comment, the approach in Steidl et al.[11] would detect no similarity between the comment and method name and report that the comment is useful and the method name can be improved to reflect some of the facts in the comments. However Comment$Probe$ will be able to detect the similarity between the words “buffer" and “memory" or “assign" and “allocate" using the ontology and vector representations, and classify the comment as Not-Useful.

```
/** allocating buffer **/
void memory_assign ( ) {...}
```

Further, Comment$Probe$ also defines a new taxonomy and quality parameters based on the relevance of information. Hence a comment containing details of an algorithm or data structure is considered more useful than comments mentioning hardware details, which are not considered in the quality model of Steidl et al.[11]

b) Detecting informative and non-informative comments (Liu et al. [31]): Liu et al.[31] mostly focus on the modular programming style of Java (dataset of 1,311 comments) to define parameters for quality based on only a project *jabref*[32] and hence cannot be applied effectively to various other Java codes. In the code snippet below, the comment `create buttons` is considered informative as these are indicators for mapping application-specific operations. The comment does not explain how the operation is modeled into the code structure, as it can be understood well from the object oriented programming style of calling relevant classes and API's.

```
// create buttons
ButtonType replaceEntries = new ButtonType(Localization.lang("Merge entry"), ButtonBar.ButtonData.DONE);
```

Comment$Probe$ would label this comment as partially useful, as it maps one application entity, but does not specify how. Considering the varied semantic expressions of C code, comments with more insight into mapping details or data structure state and algorithm help to comprehend code.

In terms of features, the set defined by Liu et al. in[31] to represent comment structure and code correlation, are also part of the set of 21 features in Comment$Probe$. However, Liu et al.[31] use cosine similarity between comment tokens and construct names and do not consider code metadata which we do in Comment$Probe$ `(correlated code knowledge graph)`. The pre-trained embeddings used in the work by Liu et al.[31] are also not trained on software development words and hence are not able to effectively find conceptual similarities between words.

c) Quality assessment of Code Review comments (Rahman et al. [30]): In this work, the authors automate quality analysis for code review comments in the context of how much code change is triggered for a comment and hence the notion of usefulness differ. Some features developed by Rahman et al.[30] and in Comment$Probe$ are similar in function, but their interpretations are different. For example, if the code element in review comments is higher, then it is considered useful as it triggers more change in developed code. Comments such as `I don't think we need 2 ways to call get_partner_whitelabel_config as market_id is None by default` is tagged as useful and `only check postable services?` as Not useful in the work by Rahman et al.[30] (examples from [30]). However, in Comment$Probe$, we focus on assessing how a comment provides extraneous information which is not relevant from the associated code snippet. If a comment contains code constructs that are also in its scope, Comment$Probe$ renders it redundant, and based on other parameters, the comment is labeled as either partially useful or not useful.

Limitations: In any scientific study, *dataset validity* is always a concern. In this work we have created a dataset of 20,206 comments from open source C applications (extracted from `Github`) belonging to multiple domains that are widely used and are actively maintained. However, it would have been good to use code bases from industry , along with open source projects, to gain a more comprehensive picture of commenting practices and enable the training of more robust models. We also faced various issues with distinctness and detected overlaps in case of comment categories whilst conducting the online survey. In the following code and comment extract some developers preferred to merge, F_{c3} - Algorithm Outline / Working Summary and F_{c5} - Description of the Dataset $\rightarrow$ into a single Working Summary category:

```
/* function returns a pointer to STATIC memory, converts the binary dump to hex format string for logs
using lookup tables */  char * data_to_hex(char *data, size_t len){..}
```

In this study we also work at the level of a single comment. However, often developers span their commenting process over multiple individual comments that are contextually interrelated. These interrelationships between the *precede* and *following* comments have not been analysed.

For the example below, comments C1 belong to categories F_{c3} (Working Summary) and F_{c4} (Mapping to Application Entities); C2 belongs to F_{c5} (Description of Dataset); and C3 belongs to F_{c8} (Exceptions). However C1, C2, C3 together are connected semantically and provide a complete working summary of the if block. Being able to group comments based on their context and make inferences for usefulness of contextually grouped comments would significantly add value to this work.

```
// C1; converting graphics matrix to EPS formats using row major transformations
if ( lax ) {
    //C2; y0 is a 2 dimensional array, converted to vector
    yp0 = plP_wcpcy(y0); plP_movphy(vppxmi, (PLINT) yp0); plP_draphy(vppxma, (PLINT) yp0);
    if ( ltx && !lxx ) {
        // C3; value of ltx should be greater than 20
        tp = xtick1 * floor( vpwxmi / xtick1 ); for (;; ){tn = tp + xtick1; { .. }}}}
```

Apart from relating the information from various comments in a source file to source code and metadata, it is necessary to relate them to knowledge assimilated from analysing runtime traces. We propose Smart-KT, a knowledge assimilation and transfer framework based on multiple sources of a software project in the work [50,75] and we plan to integrate with Comment$Probe$ to help build up the complete functional requirement of a project from comments.

9 | CONCLUSIONS AND FUTURE WORK

In this paper we propose Comment$Probe$, an automated comment quality classification framework for C codebases, based on using static instrumentation, natural language processing and machine learning techniques. We conduct a detailed study with software developers to identify a set of 21 categories based on the common concepts which manifest in comments and are relevant for software maintenance. We develop a comprehensive set of features based on code comment correlation and the comment categories. We generate a ground truth based on validation by industry experts for 20,206 comments collected from 5 Github open source projects. We develop an architecture combining LSTM and ANN resulting in a precision score of 86.27% and recall of 86.42% for classifying the overall quality score of a comment (Useful, Partially Useful, Not Useful). The relevance, repetitiveness and inconsistency of the information content in a comment have been semantically analysed, in an attempt to define a robust and effective comment quality model. The source code, dataset of 20,206 annotated comments and other related documents and responses are available at [22]. The semantic analysis framework in Comment$Probe$ can be used in other comment analysis tasks (Table 29), and also improves on existing approaches to analysing comments that are predominately based on syntactic methods (e.g. [6]).

TABLE 29 Comment$Probe$ – improving inferences in existing work on comment analysis

Example from Tan et al. [6] and modified	Available Approaches	Comment$Probe$
drivers/scsi/in2000.c: /* Caller holds instance lock! static int in2000_bus_reset (barrier_wait(& var); ... reset_hardware (. . .); }	**Tan et al. [6]**: Cannot locate lock phrases before reset_hardware; declares mismatch between code and comment	SD2Vec embeddings and SD ontology relate barrier with lock.

The industry survey conducted to set the context to assess the quality of code comments in software maintenance can also be re-used in further research to analyse and classify comments. There is still much work to be done to analyse comment quality and develop methods to assist with software tasks that could utilise automated techniques, such as code maintenance. In future work we plan to carry out the following. Firstly, we plan to enrich the vector space model using transformer based models, such as BERT [57]. Secondly, we plan to extend the annotation set with more comments for better classification performance. Finally, we plan to extend Comment$Probe$ with newer categories, knowledge domains and also features, such as interpreting correlation to runtime behaviour, to analyse the usefulness of groups of related comments.

Funding Agency: This work has been funded by the National Digital Library of India, Ministry of Education, Government of India. The surveys conducted at companies has been approved by the Ethical Committee, Sponsored Research and Industrial Consultancy, Indian Institute of Technology, Kharagpur, India

Acknowledgements: We thank and acknowledge the support of participants from Mentor Graphics (India), Peak Indicators (UK), Tata Consultancy Services (UK), Happy Wired (UK) and the National Digital Library of India (India) and our team of annotators.

References

1. Erlikh L. Leveraging legacy system dollars for e-business. *IT professional* 2000; 2(3): 17–23.
2. Roehm T, others . How do professional developers comprehend software?. In: International Conference on Software Engineering (ICSE). IEEE. ; 2012: 255–265.
3. Souza dSCB, Anquetil N, Oliveira dKM. A study of the documentation essential to software maintenance. In: Conference on Design of communication. ACM. ; 2005: 68–75.
4. Freitas JL, Cruz dD, Henriques PR. A comment analysis approach for program comprehension. In: Annual Software Engineering Workshop (SEW). IEEE. ; 2012: 11–20.
5. Jiang ZM, Hassan AE. Examining the evolution of code comments in PostgreSQL. In: Workshop on Mining software repositories (MSR). ACM. ; 2006: 179–180.
6. Tan L, Yuan D, Krishna G, Zhou Y. iComment: Bugs or bad comments?. In: Association for Computing Machinery's Special Interest Group on Operating Systems Review (SIGOPS). ACM. ; 2007: 145–158.
7. Ratol IK, Robillard MP. Detecting fragile comments. In: International Conference on Automated Software Engineering (ASE). IEEE. ; 2017: 112–122.
8. Van Kemenade E, Pupius M, Hardjono TW. More value to defining quality. *Quality in Higher education* 2008; 14(2): 175–185.
9. Pascarella L, Bacchelli A. Classifying code comments in Java open-source software systems. In: International Conference on Mining Software Repositories (MSR). IEEE. ; 2017: 227–237.
10. Haouari D, Sahraoui H, Langlais P. How good is your comment? a study of comments in java programs. In: International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE. ; 2011: 137–146.
11. Steidl D, Hummel B, Juergens E. Quality analysis of source code comments. In: International Conference on Program Comprehension (ICPC). IEEE. ; 2013: 83–92.
12. Bosu A, Greiler M, Bird C. Characteristics of useful code reviews: An empirical study at microsoft. In: Working Conference on Mining Software Repositories. IEEE. ; 2015: 146–156.
13. Rajlich V, Wilde N. The role of concepts in program comprehension. In: IEEE. ; 2002: 271–278.
14. Von Mayrhauser A, Vans AM. Program comprehension during software maintenance and evolution. *Computer* 1995; 28(8): 44–55.
15. Rugaber S. The use of domain knowledge in program understanding. *Annals of Software Engineering* 2000; 9(1): 143–192.
16. Krancher O, Dibbern J. Knowledge in software-maintenance outsourcing projects: Beyond integration of business and technical knowledge. In: Hawaii International Conference on System Sciences. IEEE. ; 2015: 4406–4415.
17. Zhang Y, Liu Q, Song L. Sentence-state lstm for text representation. *arXiv preprint arXiv:1805.02474* 2018.
18. Beale HD, Demuth HB, Hagan M. Neural network design. *Pws, Boston* 1996.
19. Martin J, Muller HA. Strategies for migration from C to Java. In: European Conference on Software Maintenance and Reengineering. IEEE. ; 2001: 200–209.
20. Prechelt L. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer, IEEE* 2000; 33(10): 23–29.
21. Beer B. *Introducing GitHub: A non-technical guide*. " O'Reilly Media, Inc." . 2018.
22. Majumdar S. The CommentProbe Tool – Quality Analysis and Visualisation (Source Code, Feature Datasheets, Installations and examples). Open Source; 2022. https://github.com/SMARTKT/CommentProbe, Last Accessed: April 01, 2022.
23. Majumdar S. Pre-trained Word Embeddings for Software Development related Concepts. Open Source; 2022. https://github.com/SMARTKT/WordEmbeddings, Last Accessed: April 01, 2022.
24. Tan L, Yuan D, Zhou Y. Hotcomments: how to make program comments more useful?. In: ACM. ; 2007: 20–27.
25. Ying AT, Wright JL, Abrams S. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. *ACM SIGSOFT software engineering notes, ACM* 2005; 30(4): 1–5.
26. Storey MA, Ryall J, Bull RI, Myers D, Singer J. TODO or to bug. In: International Conference on Software Engineering (ICSE) . IEEE. ; 2008: 251–260.
27. Padioleau Y, Tan L, Zhou Y. Listening to programmers taxonomies and characteristics of comments in operating system code. In: International Conference on Software Engineering (ICSE). IEEE. ; 2009: 331–341.
28. Aman H, Amasaki S, Yokogawa T, Kawahara M. Empirical Analysis of Words in Comments Written for Java Methods. In: Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE. ; 2017: 375–379.
29. Smith TC, Frank E. Introducing machine learning concepts with WEKA. In: Springer. 2016 (pp. 353–378).
30. Rahman MM, Roy CK, Kula RG. Predicting usefulness of code review comments using textual features and developer experience. In: International Conference on Mining Software Repositories (MSR). IEEE. ; 2017: 215–226.
31. Liu M, Yang Y, Peng X, et al. Learning based and Context Aware Non-Informative Comment Detection. In: International Conference on Software Maintenance and Evolution (ICSME). IEEE. ; 2020: 866–867.
32. Alver M, Batada N. JabRef: Cross-platform citation and reference management software. Open Source; 2003. https://github.com/JabRef/jabref, Last Accessed: December 12, 2020.
33. Lattner C, Adve V. The LLVM compiler framework and infrastructure tutorial. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC). Springer. ; 2004: 15–16.
34. O'brien MP. Software comprehension–a review & research direction. Tech. Rep. Technical Report, Department of Computer Science & Information Systems University of Limerick; Ireland: 2003.
35. Gonzalez-Perez C, Jablonski S. *Evaluation of Novel Approaches to Software Engineering*. Springer . 2010.
36. Ko AJ, Myers BA, Coblenz MJ, Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering, IEEE* 2006; 32(12): 971–987.
37. Brooks R. Using a behavioral theory of program comprehension. In: International Conference on Software Engineering (ICSE). ACM. ; 1978.
38. Team CDT. Chromium: The official GitHub mirror of the Chromium source. Open Source; 2019. https://github.com/chromium/chromium, Last Accessed: December 12, 2020.
39. Stenberg D. cURL: A command line tool and library for transferring data with URL syntax. Open Source; 2020. https://github.com/curl/curl, Last Accessed: April 25, 2020.
40. Weiss RS. *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster . 1995.
41. Tongco MDC. Purposive sampling as a tool for informant selection. *Ethnobotany Research and applications* 2007; 5: 147–158.
42. Majumdar S. 1600 comment examples, Candidate Comments. Open Source; 2022. https://github.com/SMARTKT/CommentProbe/tree/master/Comment_Examples, Last Accessed: April 01, 2022.
43. Kvale S. Planning an interview study. *Doing interviews* 2007; 1: 34–51.
44. Braun V, Clarke V. Thematic analysis.. *APA handbook of research methods in psychology, American Psychological Association* 2012; 2.
45. Basit T. Manual or electronic? The role of coding in qualitative data analysis. *Educational research* 2003; 45(2): 143–154.
46. Schalnat GE, Dilger A. libpng: Portable Network Graphics support, official libpng repository. Open Source; 2020. https://github.com/glennrp/libpng, Last Accessed: April 25, 2020.

47. Joaquim L. Typeform. Online Surveys; 2012. www.typeform.com, Last Accessed: April 25, 2020.
48. De Marneffe MC, Manning CD. The Stanford typed dependencies representation. In: Proceedings of the workshop on cross-framework and cross-domain parser evaluation. Association for Computational Linguistics. ; 2008: 1–8.
49. Singh A, Ramasubramanian K, Shivam S. Natural Language Processing, Understanding, and Generation. In: Springer. 2019 (pp. 71–192).
50. Majumdar S, Shakti P, Das PP, Ghosh S. SMARTKT: A Search Framework to Assist Program Comprehension using Smart Knowledge Transfer. In: International Conference on Software Quality, Reliability and Security (QRS). IEEE. ; 2019: 97–108.
51. Noy NF, McGuinness DL, others . Ontology development 101: A guide to creating your first ontology. Tech. Rep. Technical Report, Stanford knowledge systems laboratory; Stanford University: 2001.
52. Kernighan BW, Ritchie DM. *The C programming language* . 2006.
53. Majumdar S. Software Development Ontology. Open Source; 2022. https://github.com/SMARTKT/CommentProbe/tree/master/Comment_Examples/SD_ONTOLOGY, Last Accessed: April 01, 2022.
54. Castells P, Fernandez M, Vallet D. An adaptation of the vector-space model for ontology-based information retrieval. *IEEE transactions on knowledge and data engineering, IEEE* 2006; 19(2): 261–272.
55. Efstathiou V, Chatzilenas C, Spinellis D. Word embeddings for the software engineering domain. In: International Conference on Mining Software Repositories. IEEE. ; 2018: 38–41.
56. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. In: MIT Press. ; 2013: 3111–3119.
57. Peng Y, Yan S, Lu Z. Transfer learning in biomedical natural language processing: An evaluation of BERT and ELMo on ten benchmarking datasets. *arXiv preprint arXiv:1906.05474* 2019.
58. Lample G, Ballesteros M, Subramanian S, Kawakami K, Dyer C. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360* 2016.
59. Pakray P, Bandyopadhyay S, Gelbukh A. Textual entailment using lexical and syntactic similarity. *International Journal of Artificial Intelligence and Applications* 2011; 2(1): 43–58.
60. Majumdar S. Visualisation of knowledge graphs in Comment-Probe. Open Source; 2022. https://github.com/SMARTKT/CommentProbe/blob/visualization/Visualization/README_VISUALISATION.md, Last Accessed: April 01, 2022.
61. Fay MP, Proschan MA. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 2010; 4: 1.
62. Lee DK. Alternatives to P value: confidence interval and effect size. *Korean journal of anesthesiology* 2016; 69(6): 555.
63. Allamanis M, Barr ET, Devanbu P, Sutton C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys, ACM* 2018; 51(4): 1–37.
64. Arafat O, Riehle D. The commenting practice of open source. In: Object oriented programming systems languages and applications. ACM. ; 2009: 857–864.
65. AB MC. MariaDB: MariaDB server is a community developed fork of MySQL server. Open Source; 2020. https://github.com/MariaDB/server, Last Accessed: April 25, 2020.
66. Kotrlik J, Higgins C. Organizational research: Determining appropriate sample size in survey research. *Information technology, learning, and performance journal* 2001; 19(1): 43.
67. Irwin A, Ross A, Geoffrey F, Babcock H. PLplot: Cross-platform, scientific graphics plotting library. Open Source; 2020. https://github.com/PLplot/PLplot, Last Accessed: April 25, 2020.
68. Bangerth W, Heister T, Kanschat G, MaieSchalnat M. Deal.II: an open source finite element library. Open Source; 2020. https://github.com/dealii/dealii, Last Accessed: April 25, 2020.
69. Amraii SA. Observations on teamwork strategies in the ACM international collegiate programming contest. *XRDS: Crossroads, The ACM Magazine for Students* 2007; 14(1): 1–9.
70. Halkude SA, Awasekar DD. Learning by Competing. In: International Conference on Interactive Collaborative Learning. Springer. ; 2019: 946–956.
71. Gisev N, others . Interrater agreement and interrater reliability: key concepts, approaches, and applications. *Research in Social and Administrative Pharmacy* 2013; 9(3): 330–338.
72. Graves A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* 2013.
73. Peng W, others . An implementation of ID3-decision tree learning algorithm. *International Journal of Advanced Information Science and Technology, CiteSeerX* 2009; 13(1): 20-27.
74. Lin HT, Lin CJ. A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods. *Neural Computation, MIT Press* 2003; 11(2): 1–32.
75. Majumdar S, Papdeja S, Das PP, Ghosh SK. Comment-Mine—A Semantic Search Approach to Program Comprehension from Code Comments. In: Springer. 2020 (pp. 29–42).

10 | APPENDIX

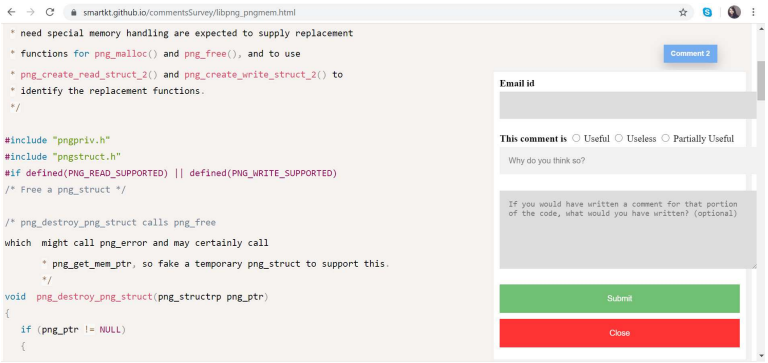


FIGURE 15 Example survey page on `github.io`

TABLE 30 Annotation labels

Labels	Motivation
A count has been decided though based on feedback from industry surveys	
L1: High SD Concepts **L2**: Low SD Concepts	high would represent SD (Software Development) Concepts with a count $>$ 5 in a comment
Labels to capture important comment categories – Inference from Survey with Developers (Section 4)	
L3: Mapping to AD Concepts (High) **L4**: Mapping to AD Concepts (Low)	high would represent AD (Application Domain . Sepcific) Concepts with a count $>$ 3 in a comment
L5: Developer Details	Name, email or contact details of developers involved in developing or enhancing the code
L6: Build	Pre-requisites and installation instructions
L7: System Specifications	Server specs, RAM size, OS requirements, compilers, big endian or small endian requirements, etc.
L8: External Libraries / Imports	Standard library header files like ctime.h, or library functions or links to external documents.
L9: Data	Dataset description – type, size, value, allocation type etc.
L10: Working Summary (Design)	Algorithm details
L11: Working Summary (Interactions)	Control flow, or data flow or operation
L12: Parameters / Return Type	Descriptions of inputs and outputs for a method
L13: Performance	Application running time, complexity of algorithm
L14: Possible Exceptions	Application running time, complexity of algorithms
L15: Project Management Details	Bug id, Fix descriptions, commit references
Labels to capture comment scope and length	
L16: High Scope, **L17**: Low Scope	High scope (sparsely spaced comment) (construct count $>$ 6 and construct distance $>$ 10 lines)
L18: Long, **L19**: Short	(half line to one line comments) with words (with POS tags in ($NN*$, $VB*$, $JJ*$)) $<$ 10
Labels to capture Code-Comment Coherence	
L20: Semantic Matches of Code-Comment (All)	Complete Conceptual equivalence with in-scope program construct & easy to understand from structure of constructs
L21: Semantic Matches of Code-Comment (Partial)	Partial Conceptual equivalence with in-scope program construct & easy to understand from structure of constructs
L22: Semantic Matches of Code-Comment (None)	No Conceptual equivalence with in-scope program construct & cannot be understood from structure of constructs
L23: Concepts Match Type / Datatype	Semantic equivalence to type / data type of in-scope program constructs
L24: Concepts Match Structure	Concepts from comments can be easily understood from the program structure
Labels to capture Additional Information	
L25: Code Comment	codes which are commented out
L26: Copyright	copyright / ownership details
L27: Junk	only symbols