Multi-granular software annotation using file-level weak labelling

Sas, Cezar; Capiluppi, Andrea

Published in:
Empirical software engineering

DOI:
10.1007/s10664-023-10423-7

Empirical Software Engineering (2024) 29:12
https://doi.org/10.1007/s10664-023-10423-7

Multi-granular software annotation using file-level weak labelling

Cezar Sas[1] · Andrea Capiluppi[1]

Accepted: 14 November 2023 / Published online: 30 November 2023

Abstract

Context One of the most time-consuming tasks for developers is the comprehension of new code bases. An effective approach to aid this process is to label source code files with meaningful annotations, which can help developers understand the content and functionality of a code base quicker. However, most existing solutions for code annotation focus on project-level classification: manually labelling individual files is time-consuming, error-prone and hard to scale.

Objective The work presented in this paper aims to automate the annotation of files by leveraging project-level labels; and using the file-level annotations to annotate items at larger levels of granularity, for example, packages and a whole project.

Method We propose a novel approach to annotate source code files using a weak labelling approach and a subsequent hierarchical aggregation. We investigate whether this approach is effective in achieving multi-granular annotations of software projects, which can aid developers in understanding the content and functionalities of a code base more quickly.

Results Our evaluation uses a combination of human assessment and automated metrics to evaluate the annotations' quality. Our approach correctly annotated 50% of files and more than 50% of packages. Moreover, the information captured at the file-level allowed us to identify, on average, three new relevant labels for any given project. We can conclude that the proposed approach is a convenient and promising way to generate noisy (not precise) annotations for files. Furthermore, hierarchical aggregation effectively preserves the information captured at file-level, and it can be propagated to packages and the overall project itself.

Conclusions We can conclude that the proposed approach is a convenient and promising way to generate noisy (not precise) annotations for files. Furthermore, hierarchical aggregation effectively preserves the information captured at file-level, and it can be propagated to packages and the overall project itself.

Communicated by: Xin Peng

✉ Cezar Sas
c.a.sas@rug.nl

Andrea Capiluppi
a.capiluppi@rug.nl

[1] Bernoulli Institute, University of Groningen, Groningen, The Netherlands

Keywords File-level labelling · Weak labelling · Software classification · Program comprehension

1 Introduction

Large code bases are becoming more common, both open-source and private. This rapid increase in software development translates into many developers switching to new projects, which requires considerable time to familiarize themselves with their content (Xia et al. 2018).

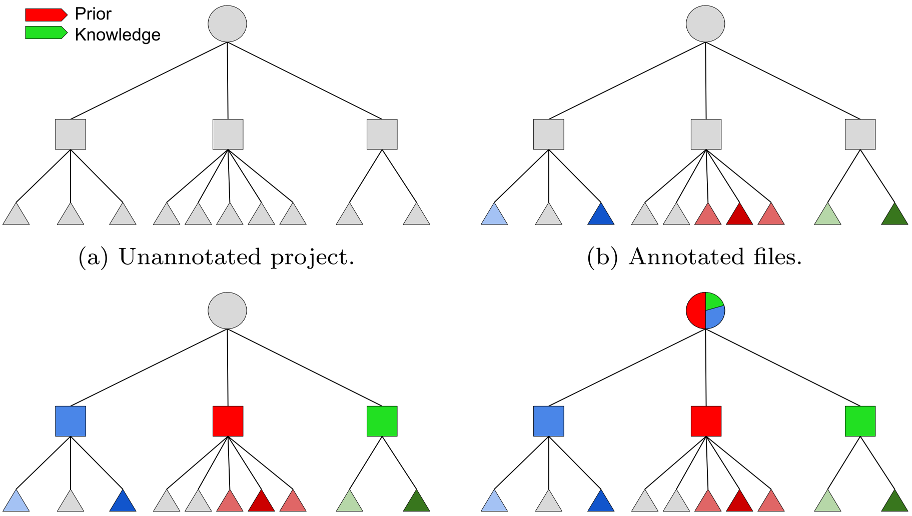
In past research, several approaches have been proposed for automatic software application domain classification (Izadi et al. 2021; Nguyen et al. 2020; Di Rocco et al. 2020; Sipio et al. 2020). Nevertheless, while showing promising results, past and current works have so far focused on classifying the project as a whole. Moreover, these approaches do not consider the compositionality of software, since a large system typically comprises several modules and components, each with its own functionality. As a result, several past and current approaches have only assigned a single label (Sas and Capiluppi 2022) to projects, and many rely on proxies like the `README` file to infer labels.

Although there are instances of prior work focusing on the use of source code identifiers to assign topics to files (Kuhn et al. 2007), they are based on the clustering of files with shared terms using Latent Semantic Analysis (LSA). This approach can be effective for a single project, but it still requires manual annotations of the clusters. Therefore, this solution is not scalable to large code bases, as it still requires substantive human intervention, in the form of manual annotations. When such annotations are unavailable, developers will still be required to understand the cluster, which can lead to ambiguity between developers due to the vagueness of natural language.

Performing manual annotation of files is time-consuming and expensive, and as such, it requires automated methods to annotate data. However, weak supervision is a rapidly developing field in machine learning (ML), and it is a research area that has so far shown interesting results (Zhang et al. 2022): as a method, it focuses on training ML models using imprecise, incomplete, or noisy labels. The labels are created through weak labelling, an automatic approach to data annotation based on heuristics.

Our paper proposes a weak labelling approach for annotating source code files in a code base. We use this file-level annotation strategy to aggregate annotations at different levels, including package-level and project-level, resulting in the ability to do multi-granular annotations. Figure 1 shows a case of how this approach works using an example project: we assume that the project has existing labels (e.g., the 'Prior Knowledge' at the top left) that developers assigned to the project. Using a weak labelling approach, it is possible to assign labels to each file (Fig. 1b), and those file-level labels can be lifted to annotate the packages containing the annotated files (Fig. 1c). All the annotated packages, in turn, can be used to generate labels for the project as a whole, potentially augmenting the existing, pre-defined labels with new labels extracted from the working code (Fig. 1d).

Figure 1 shows that multi-granular annotations can help developers to comprehend the software system's semantic content better and to quickly locate and understand the functionalities of each area in a repository (e.g. *Networking*, or *Database*). Moreover, the different levels of granularity can facilitate the identification of new labels not only for the project as a whole but also for the semantic subcomponents that have the potential for reuse.



(a) Unannotated project.

(b) Annotated files.

(c) Package-level aggregation.

(d) Project-level aggregation.

Fig. 1 Multi-granular annotation steps. The approach does not use the prior knowledge. Some nodes might not be labelled. Files are triangles, squares are packages, and the circle is the project as a whole. Colour is the label

An automatic method for creating weak labels can be useful and has already produced promising outcomes when used as a source of weak supervision (Zhang et al. 2022). As shown in various domains (in computer vision tasks Khoreva et al. 2017; Papandreou et al. 2015, or natural language tasks Mekala et al. 2020, 2021), weak and noisy labels can achieve good results in scarce data, or no annotated data. It is important to notice that our approach does not perform classification: instead, we focus on the prior step, creating annotated data.

To evaluate the effectiveness of our approach in performing multi-granular annotations, we used a combination of automated metrics and human annotators. We assessed the ability to correctly annotate at all levels of granularity, the confidence of the annotations, and the number of items that cannot be annotated.

For the purpose of this work, we are interested in answering the following research questions:

RQ1: *To what extent is weak labelling effective in capturing the semantic content of files for annotation purposes?*

RQ2: *Does aggregation at the package-level provide an effective means for capturing the content of packages?*

RQ3: *Does aggregation at the project-level provide an effective means for capturing the content of projects?*

RQ4: *Can file-level annotations allow the discovery of new topics within projects?*

This paper is a major extension of our preliminary work (Sas and Capiluppi 2023), which was focused on three projects only and performed an exploratory analysis of the feasibility of our approach. Besides expanding the dataset, this work evaluates different methods to annotate the data; it also does package-level annotation, and we perform an extended automatic analysis and human evaluation.

The contributions of this paper are summarized as follows:

- A scalable approach based on weak labelling to automatically annotate source code files;
- A framework for multi-granular labelling of software projects, which will allow developers to comprehend the code at different granularities;
- A dataset to train models for file-level software classification.

The paper is structured as follows: in Section 2, we present a motivating example for this work. Section 3, offers some background knowledge for techniques and methods used in this work. Section 4 presents the proposed approach, which is evaluated using the methods explained in Section 5. The evaluation results are shown in 6. The discussion of the results is presented in Section 7, and an overview of possible uses of our work is discussed in Section 8. We present the threats to validity in Section 9, and in Section 10, we discuss previous work related to our own. Lastly, Section 11 presents the conclusions and future works.

2 Motivating Example

A program is not a single, monolithic piece of code that performs a single function: instead, it is a set of modules, each contributing differently and interacting with others to create different functionalities in the software. Current solutions ignore this aspect and classify the software as a whole, using proxies like the `README` files. For example, if we consider a project like Weka [1], an ML desktop application and library, or Pumpernickel [2], a small UI library, we can see the downsides of this approach. The `README` files give pieces of information that are unrelated to the project content and its application.

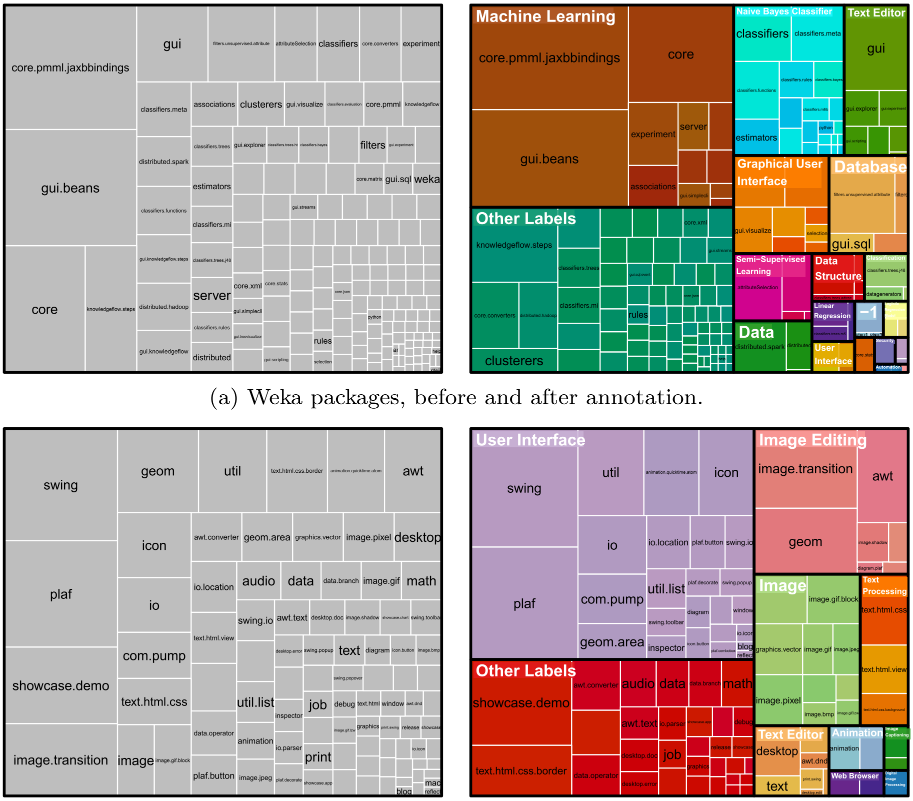
If we look at the Topics in GitHub, Weka reports only *Machine Learning* as a label; similarly, the Pumpernickel project only lists the *UI* topic. However, using the project's content, one can find more information for inferring labels. Using our approach, along with the *Machine Learning* label assigned by the developers, we can identify more specific instances of ML (e.g. *Naïve Bayes Classifier*) and parts that one might not be immediately aware of, like the *Graphical User Interface* parts. The composition of labels for the Weka project can be viewed in Fig. 2a, where packages are annotated with a label. Packages have been annotated as *Other Labels* when none of the most likely labels (in Fig. 2a we display 20) can be applied. Lastly, some wrong classifications are visible, like the `weka.gui` package being labelled as *Text Editor*.

The same file-level annotations can be used also to annotate the Pumpernickel project (Fig. 2b), and to complement the labels provided by the developers. As visible from the figure, we can identify parts responsible for *Image Editing* and *Text Editing*. These labels allow developers to gain an overview of the content of the projects quickly and reduce the time required to familiarize themselves with new unknown projects. Furthermore, identifying modules responsible for specific tasks can be helpful for software reuse.

In the remainder of this paper, we report the methodology used to extract file-level labels and annotate packages and projects, together with our results.

[1] https://github.com/Waikato/weka-3.8

[2] https://github.com/mickleness/pumpernickel



(a) Weka packages, before and after annotation.

(b) Pumpernickel packages, before and after annotation.

Fig. 2 Package annotations for Pumpernickel and Weka. Labels are identified using the approach proposed in this work. Texts in black are the package names, and texts in white are the labels. Colour shades are for the same labels. The *Other Labels* encompasses labels not at the top

3 Background

In this section, we provide a detailed description of the techniques that form the basis of our approach, particularly ‘weak labelling’ and ‘keyword extraction’. Weak labelling enables us to annotate the data with minimal effort, while keyword extraction allows us to extract domain-specific knowledge used in the annotation process.

3.1 Weak Labelling

Machine learning techniques have revolutionized research in many areas; however, these models depend on access to high-quality labelled training data. Yet, collecting human annotations is not always feasible and might be impractical (e.g. annotating pixels in images for pixel-level semantic segmentation Papandreou et al. 2015, or natural language classification Mekala et al. 2020).

Weak supervision (Zhang et al. 2022) is a growing area of research in machine learning that aims to train machine learning models using incomplete, noisy, or imprecise methods created by weak labelling. Weak labelling uses heuristics, rules, or domain-specific knowledge to automatically assign labels to the observed data based on their characteristics rather than relying on manual annotation.

The pipeline for generating weak labels consists of using different Labelling Functions (LFs), each with a different source of supervision. However, there is also a need to combine these LFs as they might have different characteristics.

One approach combines these LFs using a label model (Ratner et al. 2019), a weighted ensemble of the LFs, where the weights are learned in a probabilistic fashion using graphical models. One limitation is that the label models are currently only suited for a single prediction per LF; as a solution, we will use simple approaches like voting.

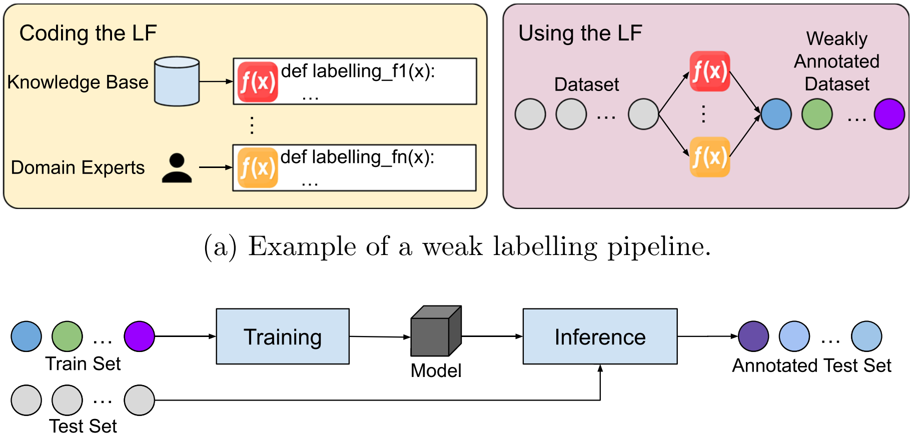
While the annotation task might seem like classification, in weak labelling, there is no training and no inference. Therefore, instead of using annotated examples and learn the parameters of a function defining a separating hyperplane, external knowledge is combined with heuristics (in the form of LF) to assign a noisy label to an example (Fig. 3).

3.2 Keyword Extraction

Keyword extraction is a critical step in text mining, particularly when the number of available documents or domains grows. Given the sheer volume of documents, it is impractical for a user to read them all in detail. The objective of keyword extraction is to identify the words that most effectively represent the document (Firoozeh et al. 2020).

There are several approaches to extracting keywords, including simple statistical methods, linguistic models, and machine learning models (Bharti and Babu 2017). However, since machine learning models require annotated data and linguistic models rely on external knowledge, we focus on statistical methods in this study, specifically for domain-specific documents.

We use a state-of-the-art unsupervised statistical approach called `YAKE!` (Campos et al. 2020) to extract keywords in this work. This algorithm tokenizes the text and removes English stopwords. It then calculates various statistics for each term, such as frequency,



```
def labelling_f1(x):
    ...
def labelling_fn(x):
    ...
```

(a) Example of a weak labelling pipeline.

(b) Example of a classification pipeline.

Fig. 3 Differences between labelling and classification

co-occurrences, position in the text, and the number of sentences it appears in. To identify n-gram keywords, a sliding window approach is employed. The final score of each keyword is a product of the scores of each term belonging to the keyword normalized by the keyword frequency. This method effectively enables us to extract relevant and informative keywords from domain-specific documents without prior knowledge.

3.3 Word Embeddings

Word embeddings are a popular technique in natural language processing (NLP) and machine learning to represent words as numerical vectors in a high-dimensional space. Word embeddings are typically learned from large amounts of text data using neural network models that model the probabilities of words in textual data. As such, they capture the intricate semantic and syntactic relationships between words. Word embeddings are helpful as they can be trained on some data and then applied for other downstream tasks. In our case, we use word embeddings to model textual data to measure similarities between two texts.

One example of such models is Word2Vec (Mikolov et al. 2013), a neural network model that learns the embeddings of words by using the context (e.g., their neighbouring words) in which the word occurs. These embeddings can then be averaged to compute the embedding of a sentence. One adaptation of this approach specifically for the software engineering domain is the Stack Overflow embeddings SO-W2V (Efstathiou et al. 2018).

One issue with current Word2Vec approaches is their limited vocabulary, making it impossible to model words that have not been seen during training; one way to address this issue is to use subword information like n-grams. On the other hand, FastText (Bojanowski et al. 2017) embeddings use n-grams as their building blocks and average their embeddings to create the word and sentence embeddings.

While models that use subword information effectively solve the out-of-vocabulary issue, they do not consider the specific context in which a word appears. Contextualized Language Models (LMs) like BERT (Devlin et al. 2019) create word vectors that also contain the information of the context, making it easier to disambiguate the meaning of a word.

Lastly, there are also code-specific LM Alon et al. (2019); Feng et al. (2020); Allal et al. (2023); however, in this paper, we are interested in the natural language meaning of the terms present in the code, rather than their code-specific syntactic and semantic information. On the other hand, code-specific LMs are generally trained for code completion and generation tasks: as a result, even using them to extract features will require some fine-tuning. Nevertheless, adaptations of these models could be used in future work for training models to perform classification using the annotation created from this work.

4 Methodology

Our methodology is illustrated in Fig. 4, which shows the various steps of the pipeline. In the following sections, we provide a detailed description of each step. Furthermore, to promote reproducibility and enable future research, we have made our code[3] and data[4] publicly available.

[3] https://github.com/SasCezar/CodeGraphClassification

[4] https://zenodo.org/record/7943882

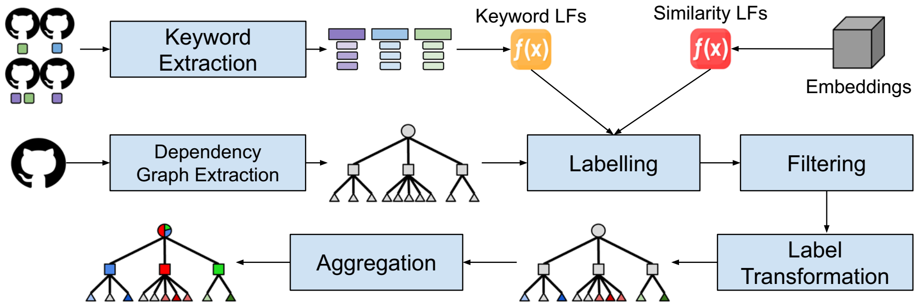


Fig. 4 Pipeline for the proposed approach

4.1 Dataset

The dataset adopted for our experiments is a subset of our previous work GitRanking (Sas et al. 2023); however, we restrict our analysis to solely Java projects. The dataset uses a subset of GitHub Topics that have been manually checked to ensure their relevance as software application domains, including categories such as *Networking* and *Database* while excluding others like *Google*. Furthermore, the dataset's labels are linked to Wikidata (Vrandečić 2012), an external knowledge base for disambiguation purposes. The subset used for this work contains only the Java projects of the dataset (an overall 2,795 projects) accompanied by a set of 267 unique labels. A subset of such labels can be seen in Table 1: as seen from the excerpt, all the labels are linked to a software-related application domain (Glass and Vessey 1995).

4.2 Dependency Graph Extraction

For the multi-granular approach to work, we need to know the structure of each project. We used the Arcan tool (Fontana et al. 2017) to extract the dependency graph, obtaining the complete set of nodes and edges describing the dependencies between classes and packages. Furthermore, Arcan also extracts dependencies between files, which can be used as extra information in future works.

4.3 Keyword Extraction

The idea behind using keyword extraction algorithms is to identify the most important terms in each project, and then assign these terms to the developers' assigned labels. In this work,

Table 1 Subset of labels used to annotate the projects in our dataset

Label
Machine Learning
Graphical User Interface
Database
Animation
Linear Regression
Software Engineering
...

we used a state-of-the-art unsupervised statistical approach called `YAKE!` (Campos et al. 2020) to extract the keywords.

In our case, a document is a software project, and we extracted the project's content using the file names. With Java syntax, we used a simple camel case tokenizer to split the composed words into individual terms. This approach, compared to extracting keywords from the source code, reaches a high throughput as less text is being analysed, while maintaining enough information to describe the content (Ajienka and Capiluppi 2016) and reduced noise from non-informative identifiers in the code (e.g. like *i*, abbreviations, or typos). In Table 2 we can see some keywords for the Weka and Pumpernickel projects.

The project extracted keywords are then assigned to all the labels that the project has been annotated by the developer on GitHub.

Once we extracted the keywords for all labels, we compute a weight for each keyword; our choice to assign weights to keywords was based on TF-IDF. The *document* is the label, the extracted *terms* are the words, and the *frequency* of the keyword is the number of occurrences in the documents annotated with that label.

We adopted this weighing as a mechanism to reduce terms that appear in multiple labels, as each keyword for a project is assigned to all the labels the project is annotated with; therefore, some keywords are likely to appear in more labels but will have a higher weight in the correct label. In Table 3, we see an example of keywords for two labels. The table also shows some unrelated keywords (e.g. *Database* in the *Machine Learning* label) that are present in the lists: as mentioned above, this is because it is not possible to separate what label each keyword belongs to when multiple labels are available per project.

4.4 Labelling

Our weak labelling approach employs two distinct types of LFs for annotation. The first type of LF is based on keyword matching (Section 4.4.1), while the second type uses semantic features (Section 4.4.2). Our LFs do not return a single prediction; instead, they produce, as an output, a vector that represents the probability distribution over a set of *m* variables (in our case, the labels). We discuss the two types of LFs below.

4.4.1 Keyword-based Labelling Functions

The keyword labelling function uses the keywords extracted from the file names and checks if the analyzed documents contain these keywords.

For example, for the label *Machine Learning*, we can see some terms in Table 3. For Weka, the file `../classifiers/meta/ClassificationViaClustering.java`, if we

Table 2 Project keywords extracted by `YAKE!`

Weka	Pumpernickel
classifier	gif
data	css
jaxbbindings	image
tree	button
regression	shape
bayes	renderer
...	...

Table 3 Subset of keywords of two labels with the respective TFIDF

Machine Learning		Graphical User Interface	
Keyword	TFIDF	Keyword	TFIDF
ELKI	0.67	Editor	0.41
Clustering	0.06	Swing	0.17
Classification	0.03	Refactoring	0.15
NLP	0.02	Scene	0.12
Database	0.02	Widget	0.08
...	...	...	...

use the name, will result in the document with the terms: *classifiers, meta, classification, via, clustering*. Combining the terms in the document, and the ones in the *Machine learning* label, an overall probability of 0.0825 will result, being the third most likely label for the file (see Table 4).

Formally, the label scores of a node (file) n given a label l are defined as:

$$LS_{n,l} = \sum_{t \in terms(n)} freq(t,n) \times weight(t,l) \tag{1}$$

where:

- $terms(n)$: gives us all the terms in the source file n;
- $freq(t,n)$: represents the frequency of the term t in the file;
- $weight(t,l)$: is a weight computed for each keyword (in our case, TF-IDF).

Lastly, we normalize the label score by dividing each score by the sum of the scores of all labels.

We apply the *keyword*-based LFs on two different modalities: the file **name** itself and the **identifiers** in the source file. A Java identifier can be a class, method or variable name. We use the tree-sitter[5] library for the parsing of source code files and extract the identifiers.

4.4.2 Similarity-based Labelling Functions

For the *similarity*-based LFs, the labels' distribution is computed using the semantic similarity between the label and source code file name. An example can be seen in Table 5, with the probability (normalized similarity) for various labels for the file `../classifiers/meta/ClassificationViaClustering.java` in the Weka project.

The label score of a node with name n and a label l is defined as:

$$LS_{n,l} = sim(n,l) \tag{2}$$

for a given semantic similarity function sim. Our choice for $sim()$ is the cosine similarity. Since the cosine similarity is bounded between $[-1, 1]$, we normalize the vector by summing the absolute value of the minimum score and then performing normalization, turning it into a probability vector, with the values in the [0, 1] range, and $norm = 1$.

We use **fastText**, **BERT**, and **W2V-SO** embeddings models on the name.

[5] https://github.com/tree-sitter/tree-sitter

Table 4 Probabilities for the top labels using the *keyword*-based LF on the name

Label	Prob
Naive Bayes Classifier	0.1192
Classification	0.1100
Machine Learning	0.0825
Data Mining	0.0550
Data Analysis	0.0550
…	…

The file is `../classifiers/meta/ClassificationViaClustering.java` and belongs to the Weka project

4.5 Filtering

The LFs we used can always annotate a file, even when highly uncertain, making the annotation very noisy. The noise is expressed as a very uniform distribution in the probability vector. We adopt the Jensen-Shannon Distance (JSD) (Endres and Schindelin 2003) to measure how close the prediction is to the uniform distribution. The JSD is a symmetric and bounded metric to compute the distance between two probability distributions. The JSD is the square root of the average of the forward and backward Kullback-Leibler divergence (Kullback and Leibler 1951), a distance measure between distributions. The JSD ranges from 0 (the distributions are identical) to 1 (dissimilar). We test different thresholds to mark the files with a JSD lower than a threshold as '*unannotated*'. Along with no filtering, two thresholds (i.e. 0.25 and 0.5) were tested in this work.

Figure 5 presents a visual example of the JSD and why it is an effective filtering approach. When measured against the uniform distribution (grey), the high JSD distribution (red) exhibits a high probability for a few labels, whereas the low JSD one (blue) has low probabilities overall. The thresholds effectively help to select the probability peaks with more or less confidence.

In the case of the annotations examples presented in Tables 4, and 5, the JSD score is respectively 0.74, and 0.20, therefore the *keyword*-based approach (Table 4) will not be filtered in any case, while the W2V-SO LF annotation will be marked as *unannotated* for both filtering settings.

4.6 Label Transformation

The labelling functions used are returning distributions, which are soft labels, meaning each label has a non-zero probability attached: Fig. 5 shows how the probability varies for each

Table 5 Probabilities for the top labels using the W2V-SO model

Label	Prob
Classification	0.5327
Naive Bayes Classifier	0.4766
Cluster Analysis	0.4205
Data Mining	0.3364
Machine Learning	0.2243
…	…

The file is `../classifiers/meta/ClassificationViaClustering.java` and belongs to the Weka project

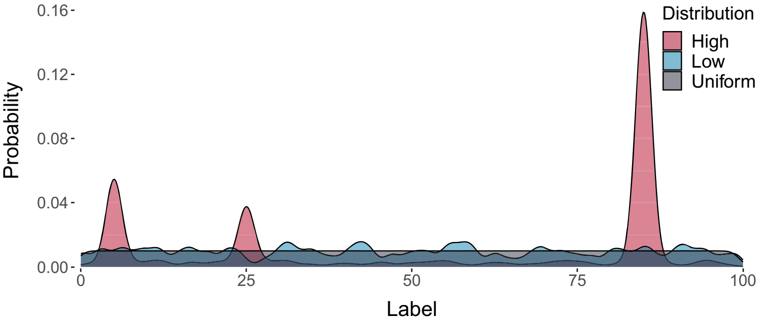


Fig. 5 Example of two distributions that have different JSD w.r.t. the uniform distribution (grey)

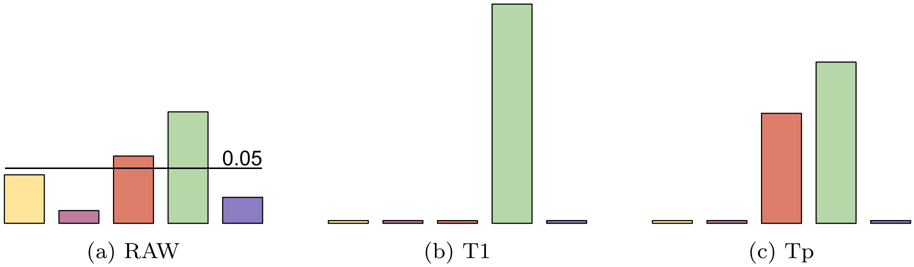
label. Along the raw output without any transformation (**RAW**), we also investigate different transformations to the distributions to improve the performance (Fig. 6). One obvious transformation is to pick only the highest probability label (**T1**) as the only label, as displayed in Fig. 6b. Another approach is to pick only the labels with a probability higher than a threshold (**Tp**); we pick 0.05 as a 12x over the uniform probability to keep only the confident predictions, shown in Fig. 6c. The results are normalized to maintain the annotations as probabilities.

An example of transformation can be seen in Table 6, where we apply the **Tp** transformation to the example in Table 5. As we notice the original **RAW** probabilities are changed, with an increase for the top labels, and the labels with lower probabilities (from the sixth) are suppressed to 0.

4.7 Aggregation

We obtained the multi-granular annotations (for a package, or the project as a whole) by aggregating the file-level annotations. We chose a naïve solution as an initial approach, the final probability vector for the package, and project are computed using the average over the files vectors. Figure 1 shows an example of aggregation; the information from the files is averaged over for the package-level annotations (Fig. 1c) and project-level (Fig. 1d).

For the aggregation at the project-level, we computed the mean over all vectors of the annotated files and picked the top K labels. The resulting labels can be used to evaluate the file-level annotations indirectly, as project-level annotations are the only source of supervision available.



(a) RAW (b) T1 (c) Tp

Fig. 6 Results of applying different transformation functions. Figure 6a, is the raw annotation with a line indicating the threshold used for Tp

Table 6 **Tp** transformation applied to the Weka project file `../classifiers/meta/ClassificationViaClustering.java`

Label	Prob
Naive Bayes Classifier	0.6022
Classification	0.5588
Machine Learning	0.4169
Data Mining	0.2779
Data Analysis	0.2779
…	0.0

The package-level annotations were predicted similarly to the project-level; however, we only considered the files that belong to that package when aggregating. Using the dependency graph we get all the annotated files and average their probabilities vector to get the package vector. Furthermore, we filtered out the labels not in the top K for the project to reduce noise and avoid having too many labels for a project. Therefore, the main label used for visualization is the label with the highest probability, also in the top K at the project level. If there is no label in the top K labels in the package, the package is marked as *Unannotated*.

4.8 Ensemble

Different labelling functions might have different strengths and weaknesses. Combining their predictions into an aggregated one (e.g., an *ensemble*) can assist with reducing the individual weaknesses, and obtaining a better result than individual LFs.

For this task, we compared two distinct ensemble techniques: cascade (**CSC**) and voting (**VT**). The cascade method takes the annotations from the first LF that annotated each node, and follows an ordered list of LFs: therefore, putting first the LFs with a higher JSD score, but lower annotated percentages, combines the high-quality annotation of these LFs with the ones with higher coverage, but lower quality.

On the other hand, the voting ensemble technique involves each LF casting a weighted vote for their top 10 labels. The weight of the vote is inverse to the position of the label (Table 7). Consequently, the label with the highest probability is awarded a score of 10, while the label in the second position is given a score of 9, and so on, until the 10th label, which gets a score of 1, after they are all 0. Finally, the votes are summed, and the vector is normalized (Table 7b). Only LFs that annotate the node can cast a vote. This ensemble method removes a lot of the noise since most of the labels will have a probability of zero.

The LFs used in these two approaches are manually selected by taking into account the individual characteristics of each LF. We use recall, percentage of unannotated nodes, the agreement between LFs, and variety to decide which LF to use in the ensemble.

5 Evaluation

To evaluate the quality of the annotation generated by the LFs, we use both the project-level labels assigned by the developers (i.e. the ground truth) and human evaluations of the generated labels at each level of granularity (project, packages, and files). Furthermore, we also use two automatic metrics (polarity and agreement) to get a general view of the characteristics of each LF.

Table 7 Example of the VT ensemble approach for the Weka file `../classifiers/functions/SimpleLogistic.java`

Label	Weight	Prob
(a) Weighted predictions for the *keyword*-based LF on the identifiers		
Random Forest	10	0.0561
Information Extraction	9	0.0337
Anomaly Detection	8	0.0337
Software Design Pattern	7	0.0337
Facial Recognition	6	0.0337
…	…	…
(b) Final predictions on the voting ensemble		
Random Forest		0.3603
Logistic Regression		0.3603
Information Extraction		0.3243
Linear Regression		0.3243
Anomaly Detection		0.2882
…		…

Lastly, we also present the results for a baseline approach (**Rand**). We picked the random baseline as it is the only approach that does not require other data or manual annotation to generate. The random baseline consists of sampling a label from a uniform distribution over all labels.

5.1 Annotators Instruction

For the manual evaluation, we used a total of 6 annotators, with four being PhD students, and two industry developers. Their background varies, with the majority having a software engineering background, while there are a couple with a more machine learning and natural language processing background.

The annotators were instructed to familiarize themselves with the project by checking the GitHub page for the project, the website, or documentation. For the project-level annotations, assign 1 whether they thought that the predicted label was correct for the corresponding project; 0 otherwise. For the package-level and file-level, since for each package/file, there were three predictions, we asked the annotators to assign 1, 2, or 3 based on which of the labels was the correct one. They were instructed to use the package/file name as a way to reduce the complexity and time required by reading the file's content. If they do not think that any of the labels are correct, then they could assign 0.

5.2 Project-level Annotations

The evaluation of the labelling functions was initially performed at the project-level, since we already have access to the ground truth data (i.e., the project labels available on GitHub Topics): this enabled us to estimate, at the project level, the overall performance of each LF. It has to be mentioned that the ground truth data available at the project-level is imperfect, as it contains noise (e.g., irrelevant labels) and incomplete annotations. Therefore, instead of precision, we focused on evaluating the recall measurement, which indicates how well the LFs capture the developer-assigned labels. We evaluated the recall@k, with k = 3, 5 and 10 labels.

As explained above, and in order to better understand how the LFs perform, we used the Jensen-Shannon distance (JSD) of the predictions against the uniform distribution, which allowed us to evaluate the confidence of the project-level predictions. A higher JSD value indicates that the annotations are less noisy (i.e., the peaks are more clearly distinguishable): this, when combined with recall, provides a general idea of the effectiveness of the labelling process.

Lastly, in order to evaluate the LFs' ability to capture *new* application domains for the projects, we manually assessed a sample of new labels for 100 projects. The projects selected for this assessment were chosen based on their popularity, as this can increase the annotators' familiarity or reduce the time required to get familiar with the project. We pick the top-10 recommended labels for each project and discard the ones that matched those already assigned by developers, resulting in 817 pairs (project, new label) being evaluated.

This evaluation is only performed on the best method, the voting ensemble. We utilize Cohen's kappa (Landis and Koch 1977), a widely used measure for intra-rater reliability.

5.3 Package-level Annotations

Similarly to the project-level annotations, we used the JSD algorithm to measure how confident the LFs are in their annotation at the package level. Unlike the pre-existing project-level labels, however, we could not access ground truth labels for packages. To address this limitation, we leveraged a characteristic that software packages should embody: all the source files contained within a package should be related to a specific functionality, and share a high cohesion within the package they belong to.

In order to evaluate the package annotation, we calculated a cohesion score to asses how differently the files within a package were annotated (by the previous step) compared to the others. This 'label cohesion' score was computed by taking the average pairwise JSD values for all annotated files within a package. A higher score indicates that the LFs' annotations are more cohesive within the package, a lower score, on the other side, indicated that the labels assigned to the package files are different.

Lastly, we performed a human evaluation over a randomly selected set of 1, 000 annotated packages, i.e., 10 for each of the 100 projects selected above. We presented the annotators with the top 3 labels for each package and asked them whether the correct label was available in the presented list.

We evaluated whether the annotators could agree on any of the first three predicted labels. If, for example, the predicted labels for a package were A, B and C, the 2 annotators would get those three to choose from. If the human evaluations returned as C, C, we would consider the package correctly labelled (with the label in position 3). If there was a disagreement, like the annotators marked A, and C, a third annotator would perform a disagreement by picking the best or marking both as wrong.

As for the project-level labels, we utilized Cohen's kappa to measure the intra-rater reliability for these package-level labels (this is done before the disagreements are resolved).

5.4 File-level Annotations

As for the package-level label predictions, we did not have ground truths to leverage to annotate source code files; furthermore, we do not have other information that can assist us as for the previous levels (e.g. files in the package). Therefore, we only conducted a human evaluation to assess the quality of the file-level annotations. To this end, we randomly selected $1,000$ annotated source files, 10 from each of the 100 projects, and asked the human annotators to evaluate the proposed annotations. This evaluation was achieved with the same procedure as the package-level, by showing the annotators the top 3 labels for each source and asking which one is correct. As for previous levels, we utilized Cohen's kappa to measure the intra-rater reliability.

5.5 Labelling Function Statistics

Besides measuring the performance of annotation produced by the LFs, we can also evaluate their characteristics. One measure we used to evaluate their performance is the *polarity*, i.e., the number of unique labels the LF outputs. This is an indicator of the LF's ability to capture the labels' features in the documents.

Another metric we used to evaluate LFs is based on measuring the *agreement* between two LFs, i.e., the number of labels that the two LFs agree upon, using a pool of the top 10 predicted labels. We computed these metrics at the project-level.

6 Results

The results obtained from our study are presented in this section, providing a detailed analysis of both the automated metric and human evaluation results across all levels: project (Section 6.1), package (Section 6.2) and file-level (Section 6.3). Furthermore, we also present the statistics of the considered labelling functions (Section 6.4).

6.1 Project-level annotations

We start with evaluating the project-level annotations as they allow us to assess the effectiveness of our LF automatically, making it easier to pick the best one and perform manual evaluation only on it.

The project-level evaluation will allow us to answer **RQ3**:

> ***RQ3****: Does aggregation at the project-level provide an effective means for capturing the content of projects?*

The recall of the three labelling functions, computed at various thresholds (i.e., using 3, 5, and 10 labels) is shown in Fig. 7 for the project-level annotations.

Overall, we noticed that the LFs with the highest recall are the keyword-based ones: all the *similarity*-based LFs score noticeably worse regarding their recall. The general higher recall for *keyword*-based LFs is due to how they predict more labels that are not in the first (more likely) positions. For the similarity-based ones, while their performance is not optimal, they are still able to pick the best label in the first positions; therefore, a reduction of noise is more beneficial.

Considering the *keyword*-based LFs, the RAW predictions are better than the transformed ones (T1, and Tp). In contrast, when a transformation is applied, the *similarity*-based LFs show an improvement for all except one case (T1 for W2V-SO).

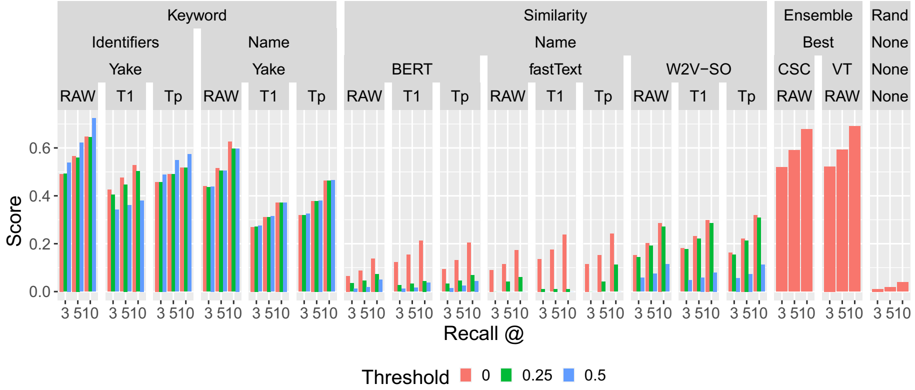


Fig. 7 Recall scores for project level annotations considering different numbers of top labels

The high noise for the *similarity*-based LFs is also visible from the filtering (using the threshold at 0, 0.25 or 0.5), where their performance suffers noticeably. Filtering strongly affects the *identifiers*-based LFs without any transformation, whereas filtering with a threshold of 0.5 achieves the best recall score.

Among the *similarity*-based LFs, the best recall is achieved by the Word2Vec model trained on Stack Overflow (W2V-SO), which suggests that domain knowledge is needed to achieve good results.

Lastly, when considering the ensemble, we see similar results between the two approaches (CSC and VT), with a slightly higher score for VT with 10 labels. These results align with the *keyword*-based LFs with a filtering of 0.5.

All the methods outperform the random baseline, however, for the *similarity*-base approaches, when increasing the threshold, the performance reaches the one of the random baseline.

While having similar recall scores, using this as the only indicator to decide which LF is the best is insufficient. As mentioned earlier, increasing the filtering threshold might remove some nodes based on the annotations' noise. Therefore, we also need to consider the number of nodes that are not being labelled. Figure 8 shows us the percentage of unannotated nodes for each LF. As clearly visible, using a threshold of 0.5 negatively affects the amount of annotated nodes. In most cases, the number of unannotated reaches 90%, except the *name*-based LF, where it only reaches 50%. A threshold of 0.25 also negatively affects the *similarity*-based LFs, but not the keyword-based ones, indicating higher confidence in the predictions for the *keyword*-based LFs.

We can measure this confidence with the JSD of the prediction against a uniform distribution (highest entropy), as shown in Fig. 9. In most cases, a high filtering threshold is beneficial; however, it is not ideal given the large number of unannotated nodes. A more conservative threshold of 0.25 slightly increases the JSD and does not significantly affect the amount of annotated nodes. Overall, the filtering, at the project level, while improving the recall, does not seem to be a crucial aspect, given the downside of fewer annotated nodes.

We can see that the random baseline has a very high variance in the JSD, indicating that the labels are all over the place, however, given the fact that there is only one label for each file, the score is high.

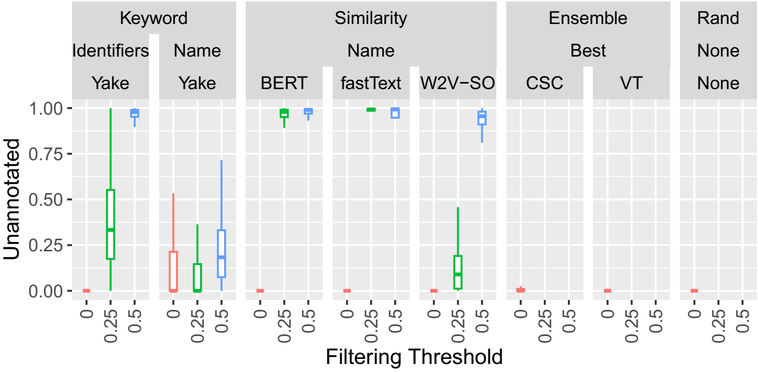


Fig. 8 Distribution of the percentage of unannotated nodes

Focusing on the ensemble, we can notice that for the recall, it performs similarly to the *keyword*-based LFs. However, if we also consider the number of unannotated nodes, we can see that, with the *ensemble*-based LFs, we achieve a near zero percentage, while the *keyword*-based ones present more noise. This higher noise is also captured by the lower JSD. Therefore, considering all these metrics, we pick the voting ensemble (VT) as the best LF for human evaluation.

Therefore, we can answer our **RQ3**:

Finding 1

The voting ensemble LF achieves a recall for the developer-assigned labels between 50% (recall@3) and 70% (recall@10). This shows the effectiveness of the LF in capturing the pieces of information at the file-level, and that the signal is strong enough not to get suppressed by the aggregation.

Now that we measured the ability to discover the developers' assigned labels, we are interested in answering **RQ4**:

***RQ4**: Can file-level annotations allow the discovery of new topics within projects?*

Using human annotators, we evaluated the ability of the models to find new topics for the project. Figure 10 presents the human evaluation results on the newly discovered top-

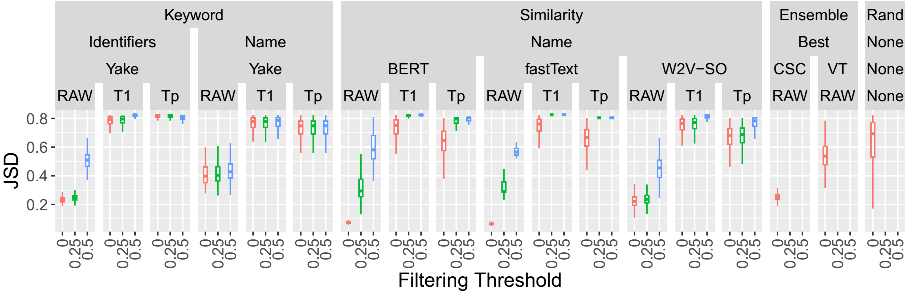
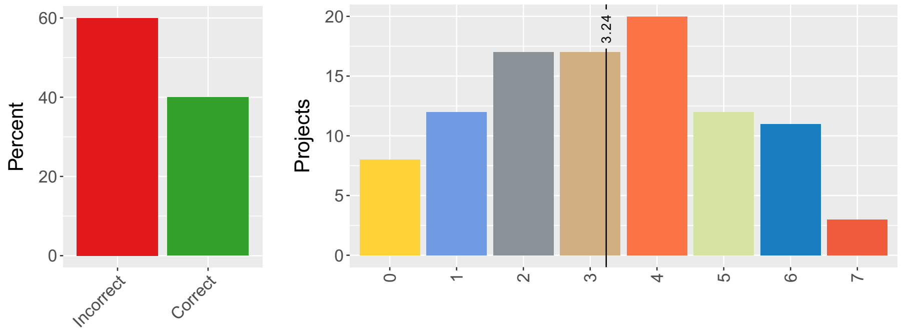


Fig. 9 Project-level JSD distribution for the different LFs



(a) Percent of correct and incorrect new labels.

(b) Distribution of the number of new topics discovered. The mean is shown with a vertical line.

Fig. 10 Results of the human evaluation at the project-level

ics. We first measure the intra-rater agreement, shown in Table 8. We can see a moderate agreement (Landis and Koch 1977) of 0.55 between the annotators, with 21% of the labels requiring resolution of the disagreement. The disagreement was resolved using a third annotator before computing the metrics. Figure 10a shows that 40% of the topics identified are correct, while the remaining 60% are not. The number of newly identified topics varies across the projects (Fig. 10b), with, on average, three new topics being found for each project, and in a couple of cases, we reach seven new topics.

These results show the ability of our approach to not only find the developer-assigned topics, as we saw using the recall (Fig. 7), but also find new relevant topics for the project using the file-level information.

An example of newly predicted project-level labels can be seen in Table 9. While the examples are above average regarding the number of new topics identified, they give us an idea of what the predictions look like. For Weka, we can see that while the wrongly predicted labels are not relevant, we can argue that both *Data Structure* and *Database* can be present. Similarly, for Pumpernickel, we have topics that, while incorrect, are closely related to the domain of the application (e.g. *Animation*).

We can summarize the findings and answer **RQ4**:

Finding 2

File-level annotations can play a crucial role in discovering new project topics. The results suggest that around 40% of new predictions are correct. It is estimated that three new topics (besides those already set by the originating developers) can be discovered for every project, on average.

Table 8 Cohen's Kappa for the intra-rater agreement at the various levels, and the percentage of examples annotators disagree on

Level	Kappa	% Disagreement
Project	0.55	21%
Package	0.46	35%
File	0.50	32%

Table 9 Top predicted project-level labels with the human evaluation for Weka and Pumpernickel

Weka		Pumpernickel	
Predicted	Eval	Predicted	Eval
Semi-supervised Learning	1	Image Editing	1
Database	0	Image	1
Data Structure	0	Text Editor	0
Naive Bayes Classifier	1	Image Captioning	1
Big Data	1	Digital Image Processing	1
Data Binding	0	Animation	0
Logistic Regression	1	GUI	1
Data	1	Text Processing	1
User Interface	1	Web Browser	0

6.2 Package-level Annotations

We are now moving to the evaluation of the annotations of packages, which will allow us to answer **RQ2**:

> ***RQ2****: Does aggregation at the package-level provide an effective means for capturing the content of packages?*

Given the inability to use ground truths, we focus on the cohesion of the annotation in the package. Figure 11 shows that the variance is high in most LFs; however, the average scores are also high. The random baseline scores are around 0.20, indicating almost no cohesion, as expected when assigning labels in a random fashion. Moving to the *keyword*-based LFs, we can see that they perform on average better than all the others. In particular, *name*-based LF

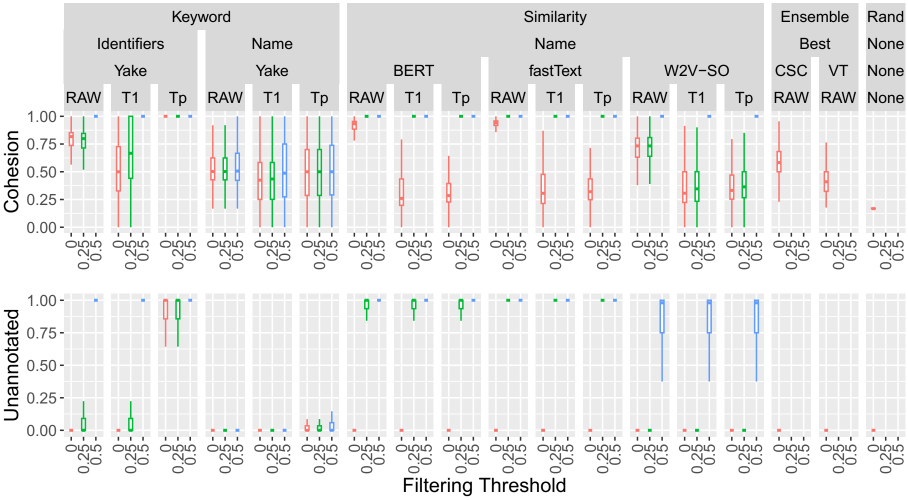


Fig. 11 Package stats. Cohesion, average pairwise JSD among files in the same package. And percentage of unannotated packages

has an average cohesion of 0.5; in contrast, the *identifiers*-based LF performs much better, averaging around 0.8. On the similarity side, the best case is the W2V-SO without filtering, with the other performing below 0.5 on average. Lastly, the cohesion for the filtering threshold of 0.5 is almost always at 1; however, in all cases, it is due to the high amount of unannotated packages.

The number of unannotated packages follows an interesting pattern. In all cases except for the W2V-SO, we have either a very low percentage of unannotated packages or almost all unannotated. Again, high thresholds are not optimal, but compared to the project-level, a moderate one is ideal for slightly better cohesion and little effect on the number of unannotated packages.

Moving to the ensemble approaches, while there is a higher cohesion for the cascade approach, the difference with the voting method is not substantial enough to make the cascade approach a better strategy when considering the much higher difference in JSD at the project-level.

Moving to the human evaluation, we have a decrease in the disagreement (Table 8) compared to the project-level; however, with a kappa of 0.46 we can consider it still moderate agreement. The package level has the highest level of disagreement, with 35% of the considered samples requiring resolution from a third annotator.

After resolving disagreements, we can see in Fig. 12 that most predictions are correct for the package-level annotations, with only 43% of examples being incorrectly labelled. Moving to the correct instances, most of the accurate labels are in the first position, with 34% of cases having the correct label in the first position. The aggregation is also effective at capturing the semantics of a package. An example of the predictions and the human evaluation is presented in Table 10.

One aspect to consider for the results is that when evaluating, we assume that the package content has high semantic cohesion, which is not always the case. The lack of cohesion also affects the annotation results at the package level.

Summarizing the results, we can answer **RQ2**:

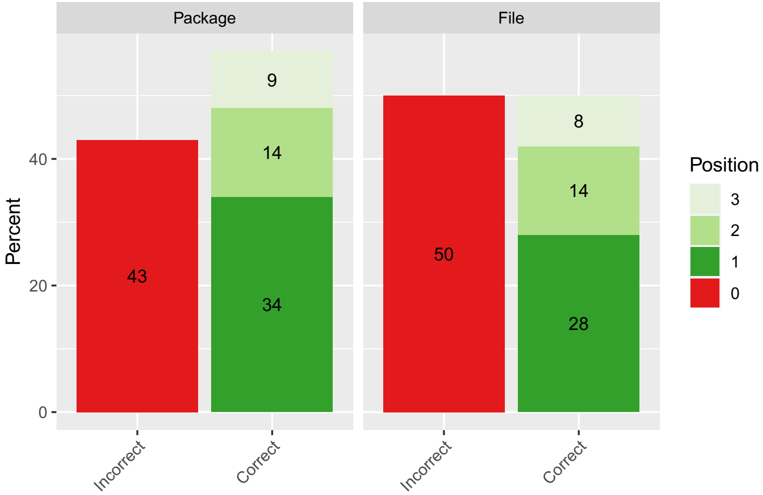


Fig. 12 Results of human evaluation at the package and file-level. Percentage of incorrect and correct labels, the correct labels are separated by their position in the prediction

Table 10 Example of 5 Weka packages with the human-assigned evaluations and their position in the sorted prediction list

Package	Prediction	Pos
classifiers.evaluation.output.prediction	Data	1
classifiers.timeseries.core	Machine Learning	3
datagenerators.classifiers.regression	Logistic Regression	1
core.matrix	-	0
datagenerators.classifiers.classification	Machine Learning	3

Finding 3

Aggregation at the package-level can indeed provide an effective means for capturing the content of packages. Furthermore, the results suggest that combining the file-level annotations obtained from the voting ensemble LF (VT) makes it possible to accurately annotate at least 50% of the examples at the package level.

6.3 File-level Annotations

In this section we are going to evaluate the file-level annotation, which will allow us to answer our **RQ1**:

> ***RQ1****: To what extent is weak labelling effective in capturing the semantic content of files for annotation purposes?*

At the file-level, we already had a view of the number of unannotated nodes in Fig. 8. Therefore, this section will only present the human evaluation results.

In Fig. 12, along with the package results, we can also see the file-level results. In this case, the percentage of incorrectly labelled files is higher, reaching 50%. However, the most likely prediction, in position 1, is accurate in most correct cases.

A qualitative view of the file-level annotations can be seen in Table 11. In this subset, most of the examples are correct; however, when examining the specific case of the .../`gui/beans/Note.java` file, given the file path, it is easy to say that it should be labelled as *UI*. This raises the idea that using extra information from the package might benefit the approach.

Table 11 Example of 5 Weka files with the human-assigned predictions and their position in the sorted prediction list

File	Prediction	Pos
.../clusterers/NumberOfClustersRequestable.java	Machine Learning	2
.../gui/beans/AbstractTestSetProducerBeanInfo.java	User Interface	2
.../pmml/jaxbbindings/BoundaryValueMeans.java	Data	2
.../gui/beans/Note.java	-	0
.../classifiers/functions/SimpleLogistic.java	Logistic Regression	1

Lastly, we should also consider that some files might not have enough information for proper classification or contain a mix of topics in a single file. For example, while the file `gui/beans/AbstractTestSetProducerBeanInfo.java` has been correctly labelled, its label is in position two. However, the best prediction, in position one, is *Machine Learning*. As the filename suggests, there is a mix of ML and GUI terms; however, the file is responsible for *UI* for a *ML* application. Therefore, it is also important to understand that some files might be misclassified due to this overlap or cases where a file is concerned with more than one responsibility.

In conclusion, the effectiveness of weak labelling in capturing the semantic content of files for annotation purposes can be evaluated based on the positive rate achieved during manual evaluation. In the case mentioned, the weak labelling approach captured the semantic content of the files with a positive rate of 50%. Furthermore, the proposed approach has a near-zero amount of unannotated files, while maintaining a high JSD.

Finally, we can summarize the results and answer **RQ1**:

Finding 4

Our weak labelling approach achieved a 50% positive rate in capturing the semantic content of the files during manual evaluation. This indicates a moderate level of effectiveness and has the potential to be useful for annotation purposes, albeit with certain limitations.

6.4 Labelling Function Statistics

We can use the *polarity* and the *agreement* measurements to understand better the used LFs' behaviour.

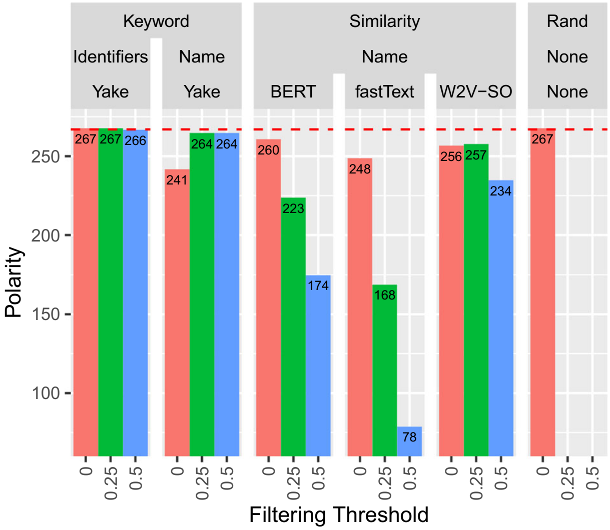


Fig. 13 Set of unique top 10 labels each LF outputs with the raw annotations at the project-level. The dashed line is the number of total labels (267)

Polarity – Figure 13 presents the polarity score: we can see that the *keyword*-based approaches produce better results than the *similarity*-based approaches. The *identifiers*-based LF can return almost all the labels at all levels of filtering, indicating its ability to identify all the classes. For the *name*-based LF, we noticed something interesting: the unfiltered annotations (threshold = 0) have the lowest amount of labels predicted, and that can be a symptom of bias for specific labels in the more uncertain cases (i.e. label A has a slightly higher similarity in the majority of cases when all other labels are low as well).

For the *similarity*-based LFs, we noticed a significant decrease in the polarity with an increasing threshold for all functions. Furthermore, when using a *similarity*-based LFs, no LF can predict all the labels independently of whether there is a filtering of uncertain nodes.

Lastly, for the *name*-based LF, we noticed an increase in the polarity when the filtering threshold increased, in contrast to the other LFs. This can be due to some labels that are favoured when uncertain, similar to the *similarity*-based LFs; however, in this case, it can be due to the lack of significant keywords in the file, and one label having a general one that will boost its probability.

Agreement – To check the similarities in annotation behaviour between the LFs, we also evaluated the *agreement* metric: Fig. 14 shows the ratio of labels predicted between the LFs.

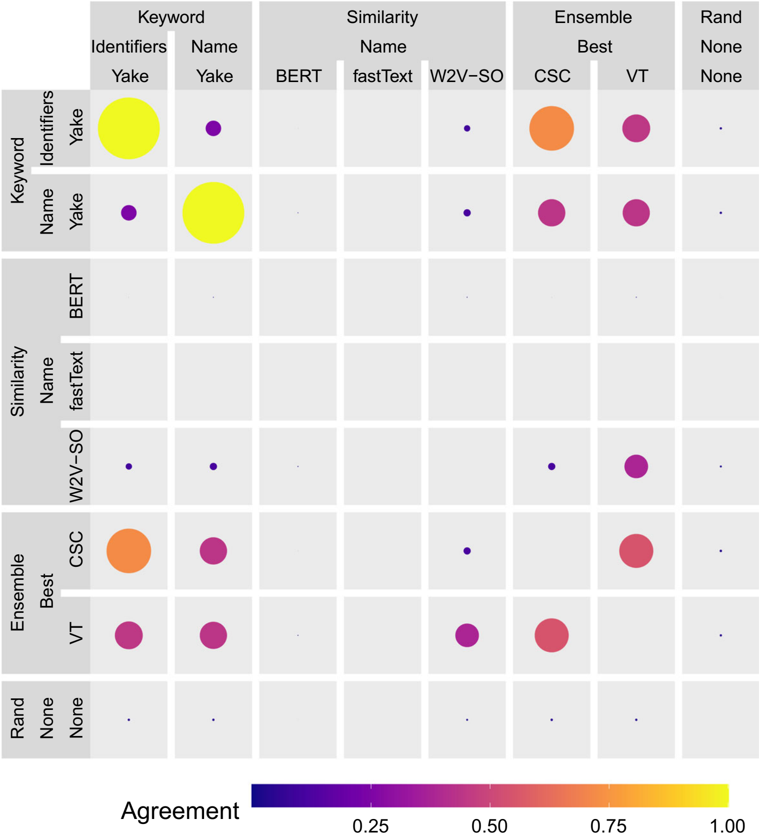


Fig. 14 Agreement in annotation between LFs

The scores show a minimal overlap between the LFs, excluding the ensemble methods. Albeit low, this overlap can boost the better labels, which is the idea behind using the ensemble method. The *similarity*-based LFs have the slightest overlap, while the keyword-based approaches share a higher agreement. Concerning the ensemble, we notice that the CSC approach is biased towards the first LF, the identifiers-based one in our case. In contrast, the VT approach considers all LFs predictions equally (note that BERT and fastText were not used in the ensembles due to their low recall and high noise). Lastly, another reason why we chose the VT ensemble can be seen by checking the agreement between the LFs, and the ensemble methods. As respected in the cascade, most agreement is found in the first LF in the cascade, the *identifier*-based LF. However, we can see that the agreement between the *keyword*-based LF, and the W2V-SO overall is minimal, indicating that they identify different labels. This difference and the reasonable results in terms of recall for the W2V-SO LF, suggest that using this information can be beneficial. The VT ensemble has a higher agreement, indicating that the labels are considered. While there is no gain in recall, it might help discover new labels. Lastly, as expected, the random baseline has almost no intersection with the other approaches.

7 Discussion

In this section, we will discuss qualitative stances on the proposed approach.

The results showed the effectiveness of automating the annotation of files in software repositories. However, an important aspect to report, and that can be noticed in both Tables 10 and 11, is the lack of specificity exhibited by some assigned labels. One such example is the Weka source code file:…`/clusterers/NumberOfClustersRequestable`, for which a more suitable label would be *Clustering*. This is also noticeable in other instances of the evaluated examples: this phenomenon is likely because more examples feature general labels, making them more probable since they contain a broader list of keyword terms. Building a taxonomy incorporating explicit hyponymy and hypernymy relationships[6] within the labels would address this issue.

Another aspect we can notice from the results is the significant difference between the *keyword*-based LFs, and the *similarity*-based ones. One reason we can point to this large difference is the domain. The *keyword*-based approaches use domain information, while the *similarity*-based ones are more general approaches, except for the W2V-SO, which also performs better. Finding better models that encode domain knowledge can help the results for the *similarity*-based LFs.

Furthermore, while not part of our RQs, the filtering and transformations of the predictions were important aspects evaluated in this work. While the final method mostly uses raw predictions, filtering and transformations can be helpful in some cases, especially with very large code bases. As seen from the previous metric, filtering, by removing noisy annotations, can preserve (fewer) high-confidence annotations. Therefore, an optimal filtering threshold allows only confident annotation; however, the downside of reducing the amount of annotated nodes for very large-scale datasets can be less of an issue. Furthermore, the transformation of the annotations aids with removing some noise while preserving a good amount of information.

[6] In natural language, a hypernym describes a broader term, whereas a hyponym is a more specialised word. For example, ‘Deep Learning’ is the hypernym, while ‘Convolutional Neural Network (or ‘CNN’) is the hyponym.

Moreover, one issue with our approach is that the average gives more weight to frequent topics in files, not central to the software's functionality. Therefore, a software application domain might end up being a secondary, or lower, label while backend functionality might get the main label spot. This issue can be addressed in future work by incorporating the structural information from the dependency graph.

Lastly, currently, we are using a naïve ensemble approach for the combination of the labels; however, this does not take into account that some LFs might have better performance on different labels, which will also allow the use of per domain (label) specific language models (e.g., a biology LM for the biology labels, and a finance one for finance labels). Further research could explore this direction.

8 Uses

From a practitioner's point of view, multi-granular annotations can be leveraged to automatically generate semantically labelled graphs that depict a software system's internal semantic content, as seen in Fig. 2. This can significantly reduce the time spent on software comprehension, which typically accounts for around 58% of the development time (Xia et al. 2018). This is particularly beneficial in industry settings, where newly recruited developers must be trained to understand the business process and relevant code, which can be time-consuming.

Additionally, this information can assist with automatic documentation generation, similar to Software Architecture Reconstruction (SAR) in microservices architectures (Rademacher et al. 2020; Walker et al. 2021). This has significant practical implications as it can help to facilitate the retrieval of components from open-source platforms like GitHub, promote software reuse, and improve overall development efficiency.

From a research perspective, multi-granular annotations can be employed to investigate context-driven research in the software engineering domain. This approach is consistent with recent studies emphasising the importance of considering the context in software engineering studies (Briand 2012; Briand et al. 2017).

9 Threats to Validity

We will present the *construct validity*, *internal validity*, and *external validity* that we encountered during our study, and we discuss how we addressed them.

9.1 Construct Validity

The recall is used to measure the quality of the file-level annotations indirectly; therefore, we do not have a direct measure of the quality of the file and package-level annotations. We mitigate this issue by implementing human validation on a representative subset of examples from both files and packages. By doing so, we can obtain a more accurate evaluation of the quality of annotations at these levels.

The JSD is a valuable metric for measuring the noise in assigned labels, although it is not necessarily an indicator of quality. However, high JSD values (i.e. low noise) can indicate that the LFs generate fewer high-likelihood predictions. This desirable behaviour suggests that the LFs provide fewer and more specific candidates. As a result, even though the JSD score may implicitly favour the voting ensemble, this behaviour ultimately leads to prediction with very few candidates and nearly no noise.

9.2 Internal Validity

Analysing the labelling is inherently subjective since it involves natural language and requires prior knowledge of various application domains. Moreover, manual evaluations were conducted solely based on the names, which can make it more challenging due to the limited information available. However, we have mitigated this potential issue by ensuring that two annotators evaluate each example and a third annotator resolves any discrepancies.

9.3 External Validity

We obtained a comprehensive list of terms for our labelling functions by extracting keywords from a large pool of projects. However, it's worth noting that the keywords were only taken from a sample of Java projects, which may limit their generalizability. One approach to address this limitation is expanding the project pool to include more programming languages. Fortunately, language-specific parsers can be used to quickly adapt our approach to different programming languages, which can help improve the LFs' generalizability.

10 Related Work

In the context of our research, we have identified two closely related areas of study: software classification and similarity, and program comprehension. This section comprehensively reviews the relevant prior work in these areas, highlighting their contributions and limitations.

10.1 Software Classification and Similarity

One of the initial works on software categorization is MUDABlue (Kawaguchi et al. 2004), which applies information retrieval techniques to categorize software into six SourceForge categories. In particular, they use Latent Semantic Analysis (LSA) on the source code identifiers of 41 projects written in C.

Following MUDABlue, Tian et al. propose LACT (Tian et al. 2009), an approach based on Latent Dirichlet Allocation (LDA), a generative probabilistic model that retrieves topics from text datasets, to categorize software from identifiers and comments in source code. In addition, they use a heuristic to cluster similar software.

Altarawy et al. expand LACT into LASCAD (Altarawy et al. 2018), by replacing the heuristic in LACT with hierarchical clustering using cosine similarity over the LDA vectors.

Another approach that uses topic modelling is proposed by Sharma et al. (2017), using a combination of topic modelling and genetic algorithms called LDA-GA (Panichella et al. 2013). They apply LDA topic modelling on the `README` files, and optimize the hyperparameters using genetic algorithms. While LDA is an unsupervised solution, humans are needed to label the topics from the identified keywords.

A different approach was adopted in Vásquez et al. (2014); they take API packages, classes, and methods names and extract the words using the naming conventions. Following Ugurel et al. (2002), they use information gain to select the best attributes for the classification and then apply different machine learning methods.

CLAN McMillan et al. (2012) provides a way to detect similar apps based on the idea that similar apps share some semantic anchors. Given a set of applications, they create two terms-document matrices, one for the structural information using the package and API calls, the other for textual information using the class and API calls. Both matrices are reduced using LSA, and the similarity across all applications is computed. Lastly, they combine the similarities from the packages and classes by summing the entries. In Vásquez et al. (2016), they propose CLANdroid, a CLAN adaptation to the Android apps domain.

Nguyen et al. (2018) propose CrossSim, an approach that uses the manifest file, project files, and the list of contributors of GitHub Java projects to create an RDF graph. Projects and developers are nodes, and edges represent the use of a project by another or that a developer is contributing to that project. They use SimRank (Jeh and Widom 2002) to identify similar nodes in the graph. According to SimRank, two objects are considered similar if similar objects reference them.

Recent research has focused on utilizing GitHub as a primary source for classification. In Sipio et al. (2020), a multi-label classifier is proposed to predict a curated list of topics based on the `README` of a GitHub repository. The content of the `README` files is encoded using the TF-IDF weighting scheme as a preprocessing step. A probabilistic model called Multinomial Naïve Bayesian Network (MNB) is then utilized to recommend new potential topics for the project. This work has been extended with the development of TopFilter (Di Rocco et al. 2020), which combines the MNB network with a collaborative filtering engine to incorporate non-featured topics in the recommendation list. The system represents repositories and topics in a graph-based structure, and the underlying recommendation algorithm computes cosine similarity using featured vectors to suggest the most similar topics. Moreover, the authors extended TopFilter, and proposed HybridRec (Rocco et al. 2023), which deals with the issues of unbalanced data by using a combination of stochastic and collaborative filtering recommendation strategies. The stochastic part uses Complement Naïve Bayesian Network, similar to TopFilter. The Collaborative part encodes the projects' topics and looks for projects with similar topics. The final recommendation is a joined list of topics.

Several approaches have been proposed that utilize neural networks for code classification. LeClair et al. (2018) and Ohashi and Watanobe (2019) both use convolutional neural networks (CNN) as their model. LeClair et al. use a C-LSTM, a combination of CNN and recurrent neural networks, with the project name, function name, and function content as input. Ohashi et al. use a binary matrix representation of C++ keywords and operators to classify short, single-file programs into six computer science and engineering categories.

Another approach utilizing a CNN as its classifier is HiGitClass (Zhang et al. 2019). HiGitClass uses a heterogenous graph to model the co-occurrence of multimodal signals in a repository, including user, repository name, topics (labels), and README. They use a topic modelling approach to learn word distribution and generate documents to train a CNN for classification. The word embeddings are created using ESIM (Shang et al. 2016), a meta-path guided heterogeneous network embedding.

Taking a more NLP-inspired approach, based on the distributional hypothesis: '*A word is characterized by the company it keeps*' (Firth 1957; Theeten et al. 2019) proposes a neural network-based approach for generating embeddings of libraries by leveraging import statement co-occurrences. Their method involves training a semantic space that captures the proximity between libraries that appear together in a given context. While they don't directly perform any classification, the resulting embeddings can be utilized for measuring similarity or training classification models.

With the popularity of large language models like BERT (Devlin et al. 2019) in various domains, in Repologue (Izadi et al. 2021), they exploit its ability in the software classification task. They use a multimodal approach that uses project names, descriptions, READMEs, wiki pages, and file names concatenated together as input to BERT. Then, they apply a fully connected neural network to predict multiple labels. Their dataset has also been used in Widyasari et al. (2023), to evaluate the performance of extreme multi-label (Wei et al. 2022) classification models. Furthermore, the authors expanded their work (Izadi et al. 2023) by builing a custom knowledge graph (SED-KGraph) with extra information like whether the label is a '*field*', a '*event*', a '*programming-language*'. Lastly, they apply a recommender system for the suggestion of the topics.

Similarly, GHTRec (Zhou et al. 2021) has been proposed to recommend personalized trending repositories, i.e. a list of most starred repositories, relying on the BERT language model and GitHub Topics. Given a repository, the system predicts the list of topics using the preprocessed `README` content. Afterwards, GHTRec infers the user's topic preferences from the historical data, i.e. commits. The tool eventually suggests the most similar trending repositories by computing the similarity on the topic vectors, i.e. cosine similarity and shared similarity between the developer and a trending repository.

Another multimodal approach has been proposed in Repo2Vec (Rokon et al. 2021). They use a concatenation of three embeddings created respectively from the repository metadata (title, topics, description, and `README`), the tree structure of the directory, and the source code. For the metadata, they use (Le and Mikolov 2014), while for the directory structure, they use node2vec (Grover et al. 2016). The source code embedding is obtained using the approach proposed in Compton et al. (2020), which uses code2vec (Alon et al. 2019) on each method in each file, and they aggregate the embedding using the mean up to the repository level.

In a similar concept to CrossSim, in Qian et al. (2022), they build a heterogeneous graph of various repositories, developers, and topics from GitHub and perform repository embedding. Each node in the graph is then represented using an embedding. Topics are embedded using BERT; developers are a combination of the metadata of their repositories and their profile data. The repository node is obtained by embedding source code and metadata using BERT. Lastly, the graph and features are used as input to a graph neural network that will refine the embeddings by using the information of neighbouring nodes.

10.2 Program Comprehension

Various approaches focus on assisting developers with program comprehension; while we focus on classification, there are intersections between the two research areas. We, therefore, present the relevant work in this section.

One initial approach is proposed by Kuhn et al. (2007). They propose an unsupervised approach for extracting terms from source code files. Furthermore, they perform clustering using LSA and Singular Value Decomposition. However, this approach requires human annotation for the identified topics in the code.

Various approaches similar to Kuhn et al. have been proposed. For example, TopicXP (Savage et al. 2010) is an approach to identify topics in source code based on LDA instead of LSA. In Ieva et al. (2019), they propose a method for software comprehension on large code bases that uses keywords, similar to Kuhn et al. (2007). They also add structural information from the call graph for various analyses, including feature location, semantic clone detection, summary generation, and more. Lastly, in Sun et al. (2017), they use LDA to extract topics to annotate packages to assist developers during software maintenance and evolution. As in Kuhn et al. (2007), these approaches do not perform classification.

11 Conclusions and Future Works

This study has presented an automated method for labelling source code files through weak labelling. We also achieved multi-granular labelling with a hierarchical aggregation, expanding the file-level labels to both package- and project-level annotations.

We evaluated our approach using a mix of automated metrics, and human evaluation. Results from both assessments have shown that weak labelling is able to capture the application domain of the files effectively. Furthermore, hierarchical aggregation preserves the information captured at the file-level and allows for correctly annotating packages and the project as a whole. Moreover, we have shown how the proposed approach enables the identification of new topics for the projects, not previously included by the originating developers.

Overall, while being an initial step towards file-level classification, our approach demonstrates promising results and has the potential to be extended and further optimized for more accurate and efficient source code labelling.

As the main future work, we will use these annotations to train ML models that perform classification at all levels. Some approaches include using large language models designed for code to extract features from the files; then, neural networks can be trained to perform the classification. Methods that use the structure can also be used: examples like node classification using graph neural network (Bronstein et al. 2017) would easily apply to our case.

While our study provides valuable insights, several areas could be explored to improve the proposed approach. For example, instead of aggregating the results from the file to annotate the packages, one could apply the LF directly to the package and then find the best approach to combine this information, with the annotations of the files belonging to the package.

Moreover, to address the issue of which label refers to core functionalities or side functionality, since we have the dependency graph, we can use community detection algorithms and centrality measures. These metrics can be used to boost topics that are in the more central files.

Another critical aspect that would make our approach even better is the creation of a hierarchy among the terms in our taxonomy. This will improve the labelling as it can consider relations between labels, reducing cases where a more generic label is preferred given a higher pool of terms due to training data. Again, domain-specific information, like SED-KGraph (Izadi et al. 2023) and natural language approaches, can be used to achieve this.

Along with creating a taxonomy, developing a strategy to better assign the keywords extracted from each project to the labels is something to explore better. One approach could use semantic information of the terms to decide to which label the term belongs.

Furthermore, future work will focus on expanding to more programming languages, like C# and Python, which have similar project structures to Java projects, expanding the number of projects and increasing the terms list to suit differences between the languages and their communities (e.g. Python being the preferred data science, and machine learning programming language). This extension will allow more data to be used in the training of ML models.

Lastly, future work could explore the use of more label-specific LFs, for example, the use of specific LM, or knowledge bases to create even more LFs that target subsets of labels. This in combination with better ensemble approaches could improve the annotation of files.

Data Availability Statement The dataset used and generated artefacts are available in a *Zenodo* repository: https://zenodo.org/record/7943882. The code is available at the following repository: https://github.com/SasCezar/CodeGraphClassification

Declarations

Conflicts of interest The authors declared that they have no conflict of interest.

References

Ajienka N, Capiluppi A (2016) Semantic coupling between classes: Corpora or identifiers? In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, ESEM '16. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2961111.2962622

Allal LB, Li R, Kocetkov D, Mou C, Akiki C, Ferrandis CM, Muennighoff N, Mishra M, Gu A, Dey M, Umapathi LK, Anderson CJ, Zi Y, Lamy-Poirier J, Schoelkopf H, Troshin S, Abulkhanov D, Romero M, Lappert M, Toni FD, del Río BG, Liu Q, Bose S, Bhattacharyya U, Zhuo TY, Yu I, Villegas P, Zocca M, Mangrulkar S, Lansky D, Nguyen H, Contractor D, Villa L, Li J, Bahdanau D, Jernite Y, Hughes S, Fried D, Guha A, de Vries H, von Werra L (2023) Santacoder: don't reach for the stars! https://doi.org/10.48550/arXiv.2301.03988

Alon U, Zilberstein M, Levy O, Yahav E (2019) Code2vec: Learning distributed representations of code. Proc ACM Program Lang 3(POPL). https://doi.org/10.1145/3290353

Altarawy D, Shahin H, Mohammed A, Meng N (2018) Lascad?: language-agnostic software categorization and similar application detection. J Syst Softw 142:21–34. https://doi.org/10.1016/j.jss.2018.04.018

Bharti SK, Babu KS (2017) Automatic keyword extraction for text summarization: a survey. arXiv:1704.03242

Bojanowski P, Grave E, Joulin A, Mikolov T (2017) Enriching word vectors with subword information. Trans Assoc Comput Linguist 5:135–146. https://doi.org/10.1162/tacl_a_00051. https://www.aclweb.org/anthology/Q17-1010

Briand L (2012) Embracing the engineering side of software engineering. IEEE Softw 29(4):96–96. https://doi.org/10.1109/MS.2012.86

Briand LC, Bianculli D, Nejati S, Pastore F, Sabetzadeh M (2017) The case for context-driven software engineering research: Generalizability is overrated. IEEE Softw 34(5):72–75. https://doi.org/10.1109/MS.2017.3571562

Bronstein MM, Bruna J, LeCun Y, Szlam A, Vandergheynst P (2017) Geometric deep learning: going beyond euclidean data. IEEE Signal Proc Mag 34(4):18–42. https://doi.org/10.1109/MSP.2017.2693418

Campos R, Mangaravite V, Pasquali A, Jorge A, Nunes C, Jatowt A (2020) Yake! keyword extraction from single documents using multiple local features. Inf Sci 509:257–289. https://doi.org/10.1016/j.ins.2019.09.013. https://www.sciencedirect.com/science/article/pii/S0020025519308588

Compton R, Frank E, Patros P, Koay A (2020) Embedding java classes with code2vec: improvements from variable obfuscation. In: Kim S, Gousios G, Nadi S, Hejderup J (eds) MSR '20: 17th international conference on mining software repositories, Seoul, Republic of Korea, 29-30 June, 2020, ACM, pp 243–253. https://doi.org/10.1145/3379597.3387445

Devlin J, Chang M, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), Association for Computational Linguistics, pp 4171–4186. https://doi.org/10.18653/v1/n19-1423

Di Rocco J, Di Ruscio D, Di Sipio C, Nguyen P, Rubei R (2020) Topfilter: an approach to recommend relevant github topics. In: Proceedings of the 14th ACM / IEEE international symposium on empirical software engineering and measurement (ESEM), ESEM '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3382494.3410690

Efstathiou V, Chatzilenas C, Spinellis D (2018) Word embeddings for the software engineering domain. In: Zaidman A, Kamei Y, Hill E (eds) Proceedings of the 15th international conference on mining software repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, ACM, pp 38–41. https://doi.org/10.1145/3196398.3196448

Endres DM, Schindelin JE (2003) A new metric for probability distributions. IEEE Trans Inf Theory 49(7):1858–1860. https://doi.org/10.1109/TIT.2003.813506

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) Codebert: a pre-trained model for programming and natural languages. arXiv:2002.08155

Firoozeh N, Nazarenko A, Alizon F, Daille B (2020) Keyword extraction: Issues and methods. Nat Lang Eng 26(3):259–291. https://doi.org/10.1017/S1351324919000457

Firth J (1957) Studies in linguistic analysis. Publications of the Philological Society. Blackwell. https://books.google.nl/books?id=JWktAAAAMAAJ

Fontana FA, Pigazzini I, Roveda R, Tamburri DA, Zanoni M, Nitto ED (2017) Arcan: a tool for architectural smells detection. In: 2017 IEEE international conference on software architecture workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, IEEE Computer Society, pp 282–285. https://doi.org/10.1109/ICSAW.2017.16

Glass RL, Vessey I (1995) Contemporary application-domain taxonomies. IEEE Software 12(4):63–76. https://doi.org/10.1109/52.391837

Grover A, Leskovec J (2016) node2vec: Scalable feature learning for networks. In: Krishnapuram B, Shah M, Smola AJ, Aggarwal CC, Shen D, Rastogi R (eds) Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, San Francisco, CA, USA, August 13-17, 2016, ACM, pp 855–864. https://doi.org/10.1145/2939672.2939754

Ieva C, Gotlieb A, Kaci S, Lazaar N (2019) Deploying smart program understanding on a large code base. In: IEEE international conference on artificial intelligence testing, AITest 2019, Newark, CA, USA, April 4-9, 2019, IEEE, pp 73–80. https://doi.org/10.1109/AITest.2019.000-4

Izadi M, Heydarnoori A, Gousios G (2021) Topic recommendation for software repositories using multi-label classification algorithms. Empir Softw Eng 26(5):93. https://doi.org/10.1007/s10664-021-09976-2

Izadi M, Nejati M, Heydarnoori A (2023) Semantically-enhanced topic recommendation systems for software projects. Empir Softw Eng 28(2):50. https://doi.org/10.1007/s10664-022-10272-w

Jeh G, Widom J (2002) Simrank: a measure of structural-context similarity. In: Proceedings of the Eighth ACM SIGKDD international conference on knowledge discovery and data mining, July 23-26, 2002, Edmonton, Alberta, Canada, ACM, pp 538–543. https://doi.org/10.1145/775047.775126

Kawaguchi S, Garg PK, Matsushita M, Inoue K (2004) Mudablue: an automatic categorization system for open source repositories. In: 11th asia-pacific software engineering conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea, IEEE Computer Society, pp 184–193. https://doi.org/10.1109/APSEC.2004.69

Khoreva A, Benenson R, Hosang JH, Hein M, Schiele B (2017) Simple does it: weakly supervised instance and semantic segmentation. In: 2017 IEEE conference on computer vision and pattern recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017, IEEE Computer Society, pp 1665–1674. https://doi.org/10.1109/CVPR.2017.181

Kuhn A, Ducasse S, Gîrba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49(3):230–243. https://doi.org/10.1016/j.infsof.2006.10.017. https://www.sciencedirect.com/science/article/pii/S0950584906001820

Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22(1):79–86

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1), 159–174. http://www.jstor.org/stable/2529310

Le QV, Mikolov T (2014) Distributed representations of sentences and documents. In: Proceedings of the 31th international conference on machine learning, ICML 2014, Beijing, China, 21-26 June 2014, *JMLR* workshop and conference proceedings, vol 32, pp 1188–1196. JMLR.org. http://proceedings.mlr.press/v32/le14.html

LeClair A, Eberhart Z, McMillan C (2018) Adapting neural text classification for improved software categorization. In: 2018 IEEE international conference on software maintenance and evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, IEEE Computer Society, pp 461–472. https://doi.org/10.1109/ICSME.2018.00056

McMillan C, Grechanik M, Poshyvanyk D (2012) Detecting similar software applications. In: Proceedings of the 34th international conference on software engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, ICSE '12, IEEE Computer Society, pp 364-374. https://doi.org/10.1109/ICSE.2012.6227178

Mekala D, Gangal V, Shang J (2021) Coarse2fine: fine-grained text classification on coarsely-grained annotated data. In: Moens M, Huang X, Specia L, Yih SW (eds) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics, pp 583–594. https://doi.org/10.18653/v1/2021.emnlp-main.46

Mekala D, Zhang X, Shang J (2020) META: metadata-empowered weak supervision for text classification. In: Webber B, Cohn T, He Y, Liu Y (eds) Proceedings of the 2020 conference on empirical methods in natural language processing, EMNLP 2020, Online, November 16-20, 2020, Association for Computational Linguistics, pp 8351–8361. https://doi.org/10.18653/v1/2020.emnlp-main.670

Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. In: Bengio Y, LeCun Y (eds) 1st international conference on learning representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings. arXiv:1301.3781

Nguyen PT, Rocco JD, Rubei R, Ruscio DD (2018) Crosssim: exploiting mutual relationships to detect similar OSS projects. In: Bures T, Angelis L (eds) 44th Euromicro conference on software engineering and advanced applications, SEAA 2018, Prague, Czech Republic, August 29-31, 2018, IEEE Computer Society, pp 388–395. https://doi.org/10.1109/SEAA.2018.00069

Nguyen PT, Rocco JD, Rubei R, Ruscio DD (2020) An automated approach to assess the similarity of github repositories. Softw Qual J 28(2):595–631. https://doi.org/10.1007/s11219-019-09483-0

Ohashi H, Watanobe Y (2019) Convolutional neural network for classification of source codes. In: 13th IEEE international symposium on embedded multicore/many-core systems-on-chip, MCSoC 2019, Singapore, Singapore, October 1-4, 2019, IEEE, pp 194–200. https://doi.org/10.1109/MCSoC.2019.00035

Panichella A, Dit B, Oliveto R, Penta MD, Poshyvanyk D, Lucia AD (2013) How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Notkin D, Cheng BHC, Pohl K (eds) 35th international conference on software engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society, pp 522–531. https://doi.org/10.1109/ICSE.2013.6606598

Papandreou G, Chen L, Murphy K, Yuille AL (2015) Weakly- and semi-supervised learning of a DCNN for semantic image segmentation. arXiv:1502.02734

Qian Y, Zhang Y, Wen Q, Ye Y, Zhang C (2022) Rep2vec: Repository embedding via heterogeneous graph adversarial contrastive learning. In: Zhang A, Rangwala H (eds) KDD '22: The 28th ACM SIGKDD conference on knowledge discovery and data mining, Washington, DC, USA, August 14 - 18, 2022, ACM, pp 1390–1400. https://doi.org/10.1145/3534678.3539324

Rademacher F, Sachweh S, Zündorf A (2020) A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan S, Reinhartz-Berger I, Soffer P, Zdravkovic J (eds) Enterprise, business-process and information systems modeling - 21st international conference, BPMDS 2020, 25th International Conference, EMMSAD 2020, Held at CAiSE 2020, Grenoble, France, June 8-9, 2020, Proceedings, *Lecture Notes in Business Information Processing*, vol 387. Springer, pp 311–326. https://doi.org/10.1007/978-3-030-49418-6_21

Ratner A, Hancock B, Dunnmon J, Sala F, Pandey S, Ré C (2019) Training complex models with multi-task weak supervision. In: The thirty-third AAAI conference on artificial intelligence, AAAI 2019, the thirty-first innovative applications of artificial intelligence conference, IAAI 2019, the ninth AAAI symposium on educational advances in artificial intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 4763–4771. AAAI Press. https://doi.org/10.1609/aaai.v33i01.33014763

Rocco JD, Ruscio DD, Sipio CD, Nguyen PT, Rubei R (2023) Hybridrec: a recommender system for tagging github repositories. Appl Intell 53(8):9708–9730. https://doi.org/10.1007/s10489-022-03864-y

Rokon MOF, Yan P, Islam R, Faloutsos M (2021) Repo2vec: a comprehensive embedding approach for determining repository similarity. In: IEEE international conference on software maintenance and evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021, IEEE, pp 355–365. https://doi.org/10.1109/ICSME52107.2021.00038

Sas C, Capiluppi A (2022) Antipatterns in software classification taxonomies. J Syst Softw 190:111343. https://doi.org/10.1016/j.jss.2022.111343. https://www.sciencedirect.com/science/article/pii/S0164121222000826

Sas C, Capiluppi A (2023) Weak labelling for file-level source code classification. In: Zhang T, Xia X, Novielli N (eds) IEEE international conference on software analysis, evolution and reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 698–702. https://doi.org/10.1109/SANER56733.2023.00074

Sas C, Capiluppi A, Sipio CD, Rocco JD, Di Ruscio D (2023) Gitranking: a ranking of github topics for software classification using active sampling. Practice and Experience, Software. https://doi.org/10.1002/spe.3238. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3238

Savage T, Dit B, Gethers M, Poshyvanyk D (2010) Topicxp: exploring topics in source code using latent dirichlet allocation. In: Marinescu R, Lanza M, Marcus A (eds) 26th IEEE international conference on software maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, IEEE Computer Society, pp 1–6 . https://doi.org/10.1109/ICSM.2010.5609654

Shang J, Qu M, Liu J, Kaplan LM, Han J, Peng J (2016) Meta-path guided embedding for similarity search in large-scale heterogeneous information networks. arXiv:1610.09769

Sharma A, Thung F, Kochhar PS, Sulistya A, Lo D (2017) Cataloging github repositories. In: Mendes E, Counsell S, Petersen K (eds) Proceedings of the 21st international conference on evaluation and assessment in software engineering, EASE 2017, Karlskrona, Sweden, June 15-16, 2017, ACM, pp 314–319. https://doi.org/10.1145/3084226.3084287

Sipio CD, Rubei R, Ruscio DD, Nguyen PT (2020) A multinomial naïve bayesian (MNB) network to automatically recommend topics for github repositories. In: Li J, Jaccheri L, Dingsøyr T, Chitchyan R (eds) EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020, ACM, pp 71–80. https://doi.org/10.1145/3383219.3383227

Sun X, Liu X, Li B, Li B, Lo D (2017) Liao L (2017) Clustering classes in packages for program comprehension. Sci Program 3787053(1–3787053):15. https://doi.org/10.1155/2017/3787053

Theeten B, Vandeputte F, Van Cutsem T (2019) Import2vec: learning embeddings for software libraries. In: Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, pp 18–28. https://doi.org/10.1109/MSR.2019.00014

Tian K, Revelle M, Poshyvanyk D (2009) Using latent dirichlet allocation for automatic categorization of software. In: Godfrey MW, Whitehead J (eds) Proceedings of the 6th international working conference on mining software repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings, IEEE Computer Society, pp 163–166. https://doi.org/10.1109/MSR.2009.5069496

Ugurel S, Krovetz R, Giles CL (2002) What's the code?: automatic classification of source code archives. In: Proceedings of the Eighth ACM SIGKDD international conference on knowledge discovery and data mining, July 23-26, 2002, Edmonton, Alberta, Canada, ACM, pp 639–644. https://doi.org/10.1145/775047.775141

Vásquez ML, Holtzhauer A, Poshyvanyk D (2016) On automatically detecting similar android apps. In: 24th IEEE international conference on program comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016, IEEE Computer Society, pp 1–10. https://doi.org/10.1109/ICPC.2016.7503721

Vásquez ML, McMillan C, Poshyvanyk D, Grechanik M (2014) On using machine learning to automatically classify software applications into domain categories. Empir Softw Eng 19(3):582–618. https://doi.org/10.1007/s10664-012-9230-z

Vrandečić, D (2012) Wikidata: a new platform for collaborative data collection. In: Proceedings of the 21st international conference on world wide web, WWW '12 Companion, Association for Computing Machinery, New York, NY, USA, pp 1063-1064. https://doi.org/10.1145/2187980.2188242

Walker A, Laird I, Cerny T (2021) On automatic software architecture reconstruction of microservice applications. In: Kim H, Kim KJ, Park S (eds) Information Science and Applications, Springer Singapore, Singapore, pp 223–234. https://doi.org/10.1007/978-981-33-6385-4_21

Wei T, Mao Z, Shi J, Li Y, Zhang M (2022) A survey on extreme multi-label learning. https://doi.org/10.48550/arXiv.2210.03968

Widyasari R, Zhao Z, Le-Cong T, Kang HJ, Lo D (2023) Topic recommendation for github repositories: How far can extreme multi-label learning go? In: Zhang T, Xia X, Novielli N (eds.), IEEE international conference on software analysis, evolution and reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 167–178. https://doi.org/10.1109/SANER56733.2023.00025

Xia X, Bao L, Lo D, Xing Z, Hassan AE, Li S (2018) Measuring program comprehension: a large-scale field study with professionals. IEEE Trans Softw Eng 44(10):951–976. https://doi.org/10.1109/TSE.2017.2734091

Zhang J, Hsieh C, Yu Y, Zhang C, Ratner A (2022) A survey on programmatic weak supervision. arXiv:2202.05433

Zhang Y, Xu FF, Li S, Meng Y, Wang X, Li Q, Han J (2019) Higitclass: keyword-driven hierarchical classification of github repositories. In: Wang J, Shim K, Wu X (eds) 2019 IEEE international conference on data mining, ICDM 2019, Beijing, China, November 8-11, 2019, IEEE, pp 876–885. https://doi.org/10.1109/ICDM.2019.00098

Zhou Y, Wu J, Sun Y (2021) Ghtrec: a personalized service to recommend github trending repositories for developers. In: Chang CK, Daminai E, Fan J, Ghodous P, Maximilien M, Wang Z, Ward R, Zhang J (eds.) 2021 IEEE international conference on web Services, ICWS 2021, Chicago, IL, USA, September 5-10, 2021, IEEE, pp 314–323. https://doi.org/10.1109/ICWS53863.2021.00049

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.