University of Groningen

Antipatterns in software classification taxonomies

Sas, Cezar; Capiluppi, Andrea

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

[Link to publication in University of Groningen/UMCG research database](#)

Antipatterns in software classification taxonomies[☆]

Cezar Sas *, Andrea Capiluppi

Bernulli Institute, University of Groningen, Nijenborgh 9, Groningen, 9747 AG, Netherlands

ARTICLE INFO

ABSTRACT

Empirical results in software engineering have long started to show that findings are unlikely to be applicable to all software systems, or any domain: results need to be evaluated in specified contexts, and limited to the type of systems that they were extracted from. This is a known issue, and requires the establishment of a classification of software types.

This paper makes two contributions: the first is to evaluate the quality of the current software classifications landscape. The second is to perform a case study showing how to create a classification of software types using a curated set of software systems.

Our contributions show that existing, and very likely even new, classification attempts are deemed to fail for one or more issues, that we named as the 'antipatterns' of software classification tasks. We collected 7 of these antipatterns that emerge from both our case study, and the existing classifications.

These antipatterns represent recurring issues in a classification, so we discuss practical ways to help researchers avoid these pitfalls. It becomes clear that classification attempts must also face the daunting task of formulating a taxonomy of software types, with the objective of establishing a hierarchy of categories in a classification.

1. Introduction

In the context of empirical software engineering research, the main goal of empirical papers is to achieve the generality of the results. The most common approach in doing so is to analyze projects having different application domains to decrease threats due to the generalizability of the results: as only a few examples, the work in Mojica et al. (2013) analyzes a collection of 200,000 mobile systems, (Wen et al., 2019) examples 1500 GitHub systems based on their popularity, while (Zhao et al., 2017) is based on 165,000 GitHub projects based on the Travis CI. As a side effect, the domain, context and uniqueness of a software system have not been considered very often by researchers as driving factors for detecting similarities or differences between software systems.

In parallel, there has been a call for 'context-driven software engineering research' (Briand, 2012; Briand et al., 2017): in the testing and verification fields, for example, the set of assumptions is obviously specific to the systems under test, and those assumptions are based on the type of system, the development process, and other factors. Although the diversity and context of software systems have received some attention in the past (Vassallo et al., 2018; Easterbrook et al., 2008), contemporary research

in the computing field is almost entirely application-independent. This has not always been the case — early in the computing era, '*there were totally separate application domains (for example, scientific and data processing) and the research focus was often application-specific*' (Glass and Vessey, 1995).

From the practitioners' point of view, categories and types of software systems are an important aspect to consider. Well known collaborative platforms like GitHub, that contain very large amounts of software repositories, show an increasing need to search and retrieve repositories based on their semantics. As a solution, GitHub has started to propose a service called *Topics*,[1] that allows developers to annotate their projects manually, and other users to search software via these topics. GitHub also provides the means to create *Collections*[2] (previously named Showcases), that is a curated list of topics where good quality repositories are grouped and showcased under the same umbrella. However, both these solutions have various issues: for example, *Topics* are only optional features to a hosted software project, while GitHub does not suggest or restrict its usage in any way. As a result, there are plenty of similar (or identical, with a different morphological form) topics, making the search less effective. On the other hand, the *Collections* list is manually curated; therefore, it is not scalable to all topics, reducing the effectiveness of finding repositories, especially those annotated

[1] https://github.com/topics
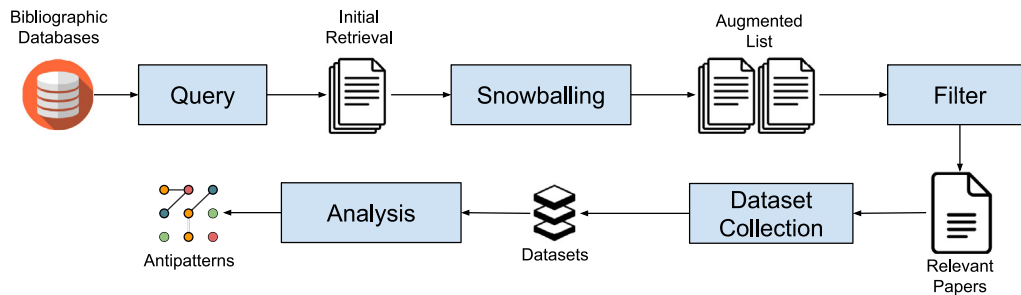[2] https://github.com/collections

Fig. 1. Pipeline used to identify the antipatterns in software application domain classification datasets.

with non-popular topics. *Furthermore, developers tend to not use these tools, or using topics that are not helpful to retrieve their code (e.g., using programming languages).*

The call for context-driven software engineering research, easier retrieval of relevant projects using semantics, and the extra burden put on the developers to label their project with all the correct labels requires a more automated way to label software projects.

From past research and efforts, there have been several approaches to perform software classification, and depending on what seed classification has been used as a stepping stone. In some cases, the seed was initiated with a *top-down* approach, i.e., using an external classification (Linares-Vásquez et al., 2014; Soll and Vosgerau, 2017): researchers would then use the categories (or labels) of the given classification to fit a sample of software projects. In other cases, categories were generated by the researchers (Borges et al., 2016), and the software projects assigned to the categories using again a top-down approach. Finally, a *bottom-up* approach was used when researchers used, as categories, the labels assigned by developers to their own software (Capiluppi and Ajienka, 2020).

Moreover, there are various artifacts that can be used to perform the classification, from README files, to the source code. These two approaches are very different in terms of difficulty, for example the README might be lacking or containing irrelevant information about the repository content like information regarding the building of the code. Whereas the source code, there are hundreds or thousands of files, each containing some relevant semantic information that needs to be aggregated keeping track not only about the frequency but also about the interactions of the files.

The misclassification, or lack of it, has various implications both on the repository, and also on the research that makes use of the labels. The developer might struggle to find contributors for their new and less popular repositories, as these are unable to discover and use the code. Furthermore, research that uses poorly labeled projects might infer wrong patterns and give bad advice to practitioners.

In this work, we evaluate several existing software classifications proposed in the literature. The selection criteria of these works comprise (i) research papers that attempt a classification of application domains, and (ii) research works that made their data available. While analyzing the resulting body of research works, we came across a number of recurring issues that researchers struggled with. These represent the most common *antipatterns* in classifying software systems.

Similarly to the work in Kalliamvakou et al. (2016) that highlighted the pitfalls of mining software repositories, the goal of our work is to analyze existing classifications from past datasets, and to present a list of common antipatterns that researchers encountered when creating them.

In this work we focus on the following research questions:

RQ1 — *How* is the *quality* of existing software classification datasets?

RQ2 — *What* are the *antipatterns* of creating a software classification dataset or a taxonomy of software application domains?

RQ3 — How can we *improve* software classifications and move towards a universal taxonomy that can be actively shared and used?

This paper presents two main contributions: first, we perform a case study attempting to create a classification for software systems that minimizes common issues present in current datasets' classification. Second, using the acquired experience and inductive analysis, we distil a set of 7 common *antipatterns* that researchers have encountered while attempting to classify software systems. These antipatterns might have happened (and are likely to happen again) when the researchers (i) create their own labels, (ii) use a pre-defined classification of labels, or (iii) use the labels manually entered by software developers. *A visual representation of our pipeline is presented in* Fig. 1. For the sake of replication, we have made all our data[3] and code[4] publicly available.

The rest of this work is structured as follows: in Section 2 we will give an overview of previous work with a focus on the used classifications. Using the evidence obtained from the analysis of the datasets, in Section 3 we summarize the antipatterns when creating a software classification. In Section 4 we present a case study on creating a software classification and resolving some issues. In Section 5 we discuss the threats to validity to our work. Finally, we present our conclusions and discuss future developments for our work in Section 6.

2. Related work and existing taxonomies

There have been several attempts in the literature focusing on software classification: in our paper we choose to only focus on those performing a classification of application domains. In general, all those previous works use their own datasets and different classifications. This generates an even more broad issue: it becomes hard to have a real applicability of these approaches, or an agreement on a shared benchmark.

While this paper is not a systematic literature review, the analyzed works have been selected using a similar approach to a systematic literature review. We retrieved the past works that (1) focused on the classification of software into application domains, and that (2) are *proposing a new dataset*. In order to perform this query, we performed an initial search *on computer science bibliography services like dblp, Google Scholar, and Arxiv. We used the* following terms: '*software categorization*', '*software classification*', '*github repository classification*', and '*software similarity*'. We perform a first stage to validate the relevance of each work,

---

Table 1
List of works divided by the different data source used.

Data source	Works
Source code	Kawaguchi et al. (2004), Tian et al. (2009), Linares-Vásquez et al. (2014), LeClair et al. (2018), McMillan et al. (2012), Vásquez et al. (2016), Altarawy et al. (2018), Theeten et al. (2019) and Ohashi and Watanobe (2019)
Other project data	Vargas-Baldrich et al. (2015), Sharma et al. (2017), Soll and Vosgerau (2017), Nguyen et al. (2018), Zhang et al. (2019), Di Sipio et al. (2020), Izadi et al. (2020) and Borges et al. (2016)

we filter on the results using the title and abstracts. The works that passed the first filtering were subsequently used to perform a manual forward and backward snowballing for further relevant papers. *Works that are case limits, were kept if their method or dataset can be used to perform software classification (e.g., Theeten et al., 2019; Borges et al., 2016).*

The list of papers that form the result of our search are listed in Table 2, and it spans a window of 15 years (2006 to 2020). The approaches used to perform the classification task of software projects in the retrieved works varies: from project metrics (Liu et al., 2018), to source code (Linares-Vásquez et al., 2014), and binary data (Escobar-Avila et al., 2015). In this paper, we focus on the approaches based on:

(A) source code; and
(B) other project data (e.g., *README files*),

as we are interested in the classification task using semantic information, and structural (can be extracted from source code). *Table 1 contains a list of the works divided by their approach.*

Below we provide, for each of the used information source a summary of the representative works with a focus on the work's dataset. A more in detail review of the software classification task landscape is presented in Auch et al. (2020).

2.1. Source code approaches

One of the initial works on software classification is MUD-ABlue (Kawaguchi et al., 2004), which applied information retrieval techniques to classify software into 6 SourceForge categories. In particular, the authors used Latent Semantic Analysis (LSA), on the source code identifiers of 41 projects written in C.

Following MUDABlue, Tian et al. proposed LACT (Tian et al., 2009), an approach based on Latent Dirichlet Allocation (LDA), a generative probabilistic model that retrieves topics from textual datasets, to perform the classification task from the identifiers and comments in the source code. In addition, the authors use a heuristic to cluster similar software. The authors use a dataset of 43 examples divided in 6 SourceForge categories. The list of projects is available in their paper.

A different approach was adopted in Linares-Vásquez et al. (2014), the authors used API packages, classes, and methods names and extracted the words using the naming conventions. Using the example in Ugurel et al. (2002), the authors use information gain to select the best attributes as input to different machine learning methods for the task of classifying 3286 Java projects into 22 SourceForge categories. Their dataset is not available anymore.

LeClair et al. (2018) used a neural network approach. The authors use the project name, function name, and the function content as input to a C-LSTM (Zhou et al., 2015), a combined model of convolutional and recurrent neural networks. Their dataset is made of 9804 software projects, with the annotations from the Debian packages repository. The authors only analyzed

programs containing C/C++ source code, divided into 75 categories: many of these categories have only a few examples, and 19 are duplicate categories with different surface form, more specifically 'contrib/X', where X is a category present in the list.

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, the authors 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, then, the similarity across all applications is computed. Lastly, the authors combine the similarities from the packages and classes by summing the entries. The data is not available. In Vásquez et al. (2016), the authors propose CLANdroid, a CLAN adaptation to the Android apps domain, and evaluate the solution on 14,450 Android apps. Their dataset is not available.

Another unsupervised approach was adopted by LASCAD (Altarawy et al., 2018), a language agnostic classification and similarity tool. As in LACT, the authors used LDA over the source code, and further applied hierarchical clustering with cosine similarity on the output topic terms matrix of LDA to merge similar topics. The authors also proposed two datasets: an annotated one consisting of 103 projects divided in 6 categories (from GitHub Collections) with 16 programming languages (although many languages have only 1 example), and an unlabeled one which is not available.

Taking a more Natural Language Processing (NLP) inspired approach, based on the distributional hypothesis: '*A word is characterized by the company it keeps*' Firth (1957), Theeten et al. (2019) proposed a neural network solution to create dense representation (i.e., embeddings) of libraries. The authors used the co-occurrences of import statements of libraries to learn a semantic space where libraries that appear in the same context are close (similar) in the space. The authors do not perform classification, therefore, their dataset is not annotated, however, the learned representation can be used to compute similarity and also train a classification model.

Differently from the previous works, Ohashi and Watanobe (2019) used the C++ keywords and operators, represented as a binary matrix, as input to a convolutional neural network to assign the correct category out of 6 in the computer science and engineering field. The dataset is made of 40,023 students written source code for assignment/exams (short, single file, programs). Their dataset is not publicly available.

2.2. Other approaches

The following is a review of the works that have been based on software artifacts other than source code. Following MUD-ABlue, Vargas-Baldrich et al. (2015) used an approach based on bytecode, the external dependencies of the project and information from Stack Overflow to generate a tag cloud. Their dataset is no longer available.

Sharma et al. (2017) used a combined solution of topic modeling and genetic algorithms called LDA-GA (Panichella et al., 2013). The authors apply LDA topic modeling on the README files, and optimize the hyper-parameters using genetic algorithms. While LDA is an unsupervised solution, humans are needed to annotate the topics from the identified keywords. The authors release a list of 10,000 examples annotated by their model into 22 categories, which was evaluated using 400 manually annotated projects. It is interesting to notice that half of the projects eventually end up in the 'Other' category, which means that they are not helpful when training a new model.

ClassifyHub (Soll and Vosgerau, 2017) used an ensemble of 8 Naïve classifiers, each using different features (e.g. file extensions,

Table 2
Summary of the different datasets used in literature.

Work	Year	Data source	Available	Task	Dataset stats				
					Examples	Categories	Balance	Min	Max
MUDABlue (Kawaguchi et al., 2004)	2006	Source code	List only[a]	Classification	41	6	0.91	2	13
LACT (Tian et al., 2009)	2009	Source code	Yes	Classification	43	6	0.97	4	9
CLAN (McMillan et al., 2012)	2012	Source code	No	Similarity	8310	–	–	–	–
Linares-Vásquez et al. (2014)	2014	Source code	No	Multi-label class.	3286	22	0.96[b]	303	1115
Borges et al. (2016)	2016	–	List only	Classification	2500	6	0.88	103	837
CLANdroid (Vásquez et al., 2016)	2016	Multimodal	No	Similarity	14,450	–	–	–	–
Sharma et al. (2017)	2017	README	List only	Classification	10,000 (5360)[c]	22	0.60 (0.91)[c]	85	670[c]
ClassifyHub (Soll and Vosgerau, 2017)[d]	2017	README	List only	Classification	208	5	0.88	95	19
LeClair et al. (2018)	2018	Source code	Yes	Classification	9804	75	0.73	1	3534
LASCAD (Altarawy et al., 2018)	2018	Source code	Yes	Classification	103	6	0.95	7	26
CrossSim (Nguyen et al., 2018)	2018	Multimodal	Yes	Similarity	582	–	–	–	–
Import2Vec (Theeten et al., 2019)	2019	Imports	Embedding	Repr. Learning	–	–	–	–	–
Ohashi and Watanobe (2019)	2019	Key and Op	No	Classification	40,023	23	0.93	4713	10769
HiGitClass (Zhang et al., 2019) — AI	2019	Multimodal	Yes	Hierarchical class.	1596	3–13[e]	0.58–0.87[e]	48–1213[e]	21–361[e]
HiGitClass (Zhang et al., 2019) — Bio	2019	Multimodal	Yes	Hierarchical class.	876	2–10[e]	0.87–0.91[e]	261–27[e]	615–210[e]
Di Sipio et al. (2020)	2020	README	Yes	Multi-label class.	12,060	134	1	100	100
Repologue (Izadi et al., 2020)	2020	Multimodal	No	Multi-label class.	152,000	228	–	–	–
Awesome-Java	2021	–	Yes	Classification	495	69	0.93	1	39
Reduced AJ	2021	Source code	Yes	Classification	495	13	0.93	8	100

[a] A reproduced dataset is available in LACT (Tian et al., 2009)
[b] Multi-labels, and also numbers in the paper table do not sum to the number of examples used for the measure
[c] After removing the 'Others' class that contains almost half of the examples in the dataset
[d] InformatiCup 2017 Dataset
[e] Two level hierarchy. First value is for the first level, the other for the second level of the taxonomy.

README, GitHub metadata and more) to perform the classification task. The authors use the InformatiCup 2017[5] dataset, which contains 221 projects unevenly divided into 7 categories.

Nguyen et al. (2018) proposed CrossSim, an approach that uses the manifest file and the list of contributors of GitHub Java projects: this data is used to create a RDF graph where projects and developers are nodes, and edges represent the use of a project by another or that a developers is contributing to that project. The authors used SimRank (Jeh and Widom, 2002) to identify similar nodes in the graph. According to SimRank, two objects are considered to be similar if they are referenced by similar objects.

HiGitClass (Zhang et al., 2019) used an approach for modeling the co-occurrence of multimodal signals in a repository (e.g. user, name of repository, tags, README and more). The authors performed the annotation according to a taxonomy (hierarchical classification) that is given as an input with keyword for each leaf node. The authors released a dataset with taxonomies for two domains: an artificial intelligence (AI) taxonomy with 1600 examples, and a bioinformatics (Bio) one with 876 projects.

Di Sipio et al. (2020) used the content of the README files and source code, represented using TFIDF, as input to a probabilistic model called Multinomial Naïve Bayesian Network to recommend possible topics. Given its premises, the work is defined as a multi-label classification. The authors used 120 popular topics from GitHub, and released a dataset of around 10,000 annotated projects in different programming languages.

Repologue (Izadi et al., 2020) also adopted a multimodal approach. The authors used project names, descriptions, READMEs, wiki pages, and file names concatenated together as input to BERT (Devlin et al., 2019), a neural language model, that creates a dense vector representation (i.e., embeddings) of the input text. Then, a fully connected neural network was applied to these embeddings to predict multiple categories. Their dataset (currently unavailable) contains 152K repositories in various languages classified using 228 categories from GitHub's *Collections*, which should be similar as the ones from Di Sipio et al. (2020).

Finally, Borges et al. (2016), albeit not performing a classification of software repositories, made a list of 2500 projects (annotated in 6 domains/categories) available for other researchers.

[5] https://github.com/informatiCup/informatiCup2017

2.3. Summary of related work

Table 4 presents a summary of the datasets used in the literature. A single file with all the categories and the number of examples for each of the analyzed works is available in the replication package for inspection or further analysis. We use the following attributes to describe and analyze each dataset:

- **Work**: the publication details where the dataset is proposed or used for the first time;
- **Year**: the publication year;
- **Data Source**: the type of information used to perform the classification task from the software systems. This was further coded into the following:

  - *Source Code*: when the authors of the research directly used the source code of a system to infer (i.e., bottom up) or assign (i.e., top down) the software categories;
  - *README*: when the authors used the textual description of a software project of the README file to infer or assign one or more categories;
  - *Imports*: when the authors focused on what external libraries have been imported into a software project to infer or assign a category;
  - *Key and Op*, when the authors used the predefined or reserved words of a specific programming language (e.g., C++), along with the operators used in the source code;
  - *Multimodal* (Baltrusaitis et al., 2019): when the authors used a combination of several sources (e.g., *Source Code* and *Wiki pages*).

- **Available**: whether the dataset is available or not. We distinguish whether the available part is the list of annotated projects, or the list and the used files are;
- **Task**: the type of task that can be performed using the dataset. This attribute has the following possibilities:

  - *Classification*: assign one of n mutually exclusive categories to the input project;
  - *Multi-Label Classification* (Tsoumakas and Katakis, 2007): assign to a project one or more categories from set of n;

– *Hierarchical Classification* (Gordon, 1987): assign m of n categories as for the Multi-Label problem, however there is a hierarchy among the categories;
– *Similarity*: the task is to retrieve software that are similar to one given as input;
– *Representation Learning* (Bengio et al., 2013): a more general case of the Similarity, in this case the goal is to create a dense representation (embedding) that preserves the similarities among projects, and it can also be used for downstream tasks.

• **Examples**: the total amount of examples in the dataset;
• **Categories**: the number of different categories used to classify the software into, higher scores are not always better as we will see later on;
• **Balance**: the level of class balance in terms of examples. It is computed using the Shannon Diversity Index (or Normalized Class Entropy Kalousis et al., 2004 in Machine Learning), a normalized entropy (Shannon, 1948) value:

$$\text{Balance} = \frac{-\sum_{i=1}^{k} \frac{c_i}{n} \log \frac{c_i}{n}}{\log k}$$

where the numerator is the entropy for a dataset of size n with k categories each of size c_i, and the denominator is the perfect case of a dataset with balanced categories, used to normalizes the values. The results range between 0 (e.g., a completely unbalanced dataset with only one category containing all the examples) and 1 (e.g., a perfectly balanced dataset containing categories with the same amount of examples). A low score means that the dataset contains a large number of categories that are not well represented in examples, and therefore more difficult to perform the classification task when encountered. *This measure is not suitable for cases where there is a large amount of classes with many examples, and only a few classes with a small number of examples;*
• **Min**: *the number of examples for the class with the least amount of representation;*
• **Max**: *the number of examples for the largest class in the dataset.*

2.4. Related work content analysis

The quantitative summarization of the previous section is not sufficient to give us a complete idea of the datasets. In this section we present the content of the categorizations in the selected datasets. We will give an overview of their intended application, inferred from the labels, and discuss in more detail the semantics of the labels using word embeddings.

We use fastText (Bojanowski et al., 2017), a neural language model, as the method for extracting the vector representation of the category: this is because it can handle out-of-vocabulary words, however, we obtained similar results also with BERT (Devlin et al., 2019) and Stack Overflow (Efstathiou et al., 2018) embeddings. In Fig. 2, we can see the distribution of similarities among categories, for each dataset. On the one hand, it is difficult to say anything definitive of the low similarity outliers, as terms from different domains might have low similarity; on the other hand, for the high similarity ones, these mostly result in categories that are in a hierarchical relationship or are highly related.

• **MUDABlue**: it has a very small categorization, with a focus on developers containing categories like 'Compilers', 'Editor', and a very specific one 'xterm'. However, it also contains labels that are not relevant to the others, in particular
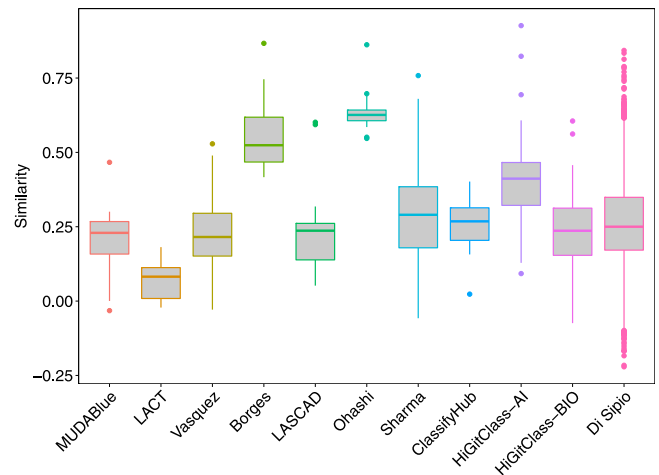

Fig. 2. Cosine similarity between labels using fastText embeddings.

'Boardgame'. Overall the terms are not too similar between themselves, with the only outlier being the pair 'xterm' and 'Compilers' of 0.46. And the lowest similarity being among the 'Boardgame' label, and 'Editor'. Given its small size, we can see the high spread as a lack of specificity.
• **LACT**: it has a similar domain as MUDABlue, but with more terms. It is also more general, as it contains several terms that are broader and less specific. We can find: 'Database' and 'Editor', as in MUDABlue, 'Terminal', which can be considered a more general version of 'xterm'. The other terms are more cohesive compared to MUDABlue, for example 'E-Mail' and 'Chat'. In this case, we do not find any outlier classes with a high similarity, and the distribution is quite narrow.
• **Vasquez**: it proposes a much more general taxonomy, as is a subset of SourceForge. Its labels span multiple fields in the Computer Science domain, some more general: 'Scientific', 'Networking', and 'Security': others more specific: 'Indexing', and 'Compilers'. Given its well defined focus, and a higher number of topics compared to previous dataset, we find some labels that have a high similarity: 'Compilers' and 'Interpreters' result in a similarity of 0.52 while 'Networking' and 'Communication' are at 0.48. The latter are co-hyponyms, so hyponyms that share the same hypernym, while the former are related, as communication software use networking technologies.
• **LASCAD**: smaller compared to Vasquez et al. while still using Computer Science labels, they are not as complete and their labels are more sparse and less related to each other. The labels are: 'Machine Learning', 'Data Visualization', 'Game Engine', 'Web Framework', 'Text Editor', and 'Web Game'. As expected from their surface form, the high similarity outliers pairs are: 'Web Game' and 'Game Engine', with a similarity of 0.60, and 'Web Game' and 'Web Framework' with 0.59. These pairs, beside being related by sharing a term, are also related in terms of usage.
• **Ohashi**: this is a very specific categorization, based on the domain of science courses. The label set includes: 'Combinatorial Optimization Problems', 'Number Theory Problems', 'Shortest Path Problems'. The overall high similarity between labels is due to the fact that all the labels contain the 'Problems' term.
• **Sharma**: it is a developers oriented classification. The terms cover various areas, labels include: 'Security', 'Music', 'Gaming and Chat Engines', and 'Blogging'. Furthermore, there

also some programming languages like 'Lua' and 'Ruby related'.

- **ClassifyHub**: as a more educational oriented dataset, its focus is not well defined, and it has high level labels in very loosely related domains: 'Homework', 'Documents', 'Development', 'Education', and 'Website'.
- **HiGitClass**: their two datasets are very specific, one focusing on AI subfields, while the other is focused on Bioinformatics. Labels in the AI dataset include: 'Computer Vision', 'NLP', and 'Speech' at level zero and 'Image Generation', 'Super Resolution', 'Language Modeling' at the first level. In Fig. 2, we can see the similarity among the labels at all levels. The outliers are due to surface similarity among the labels (e.g., 'Text Classification' and 'Image Classification'). As expected, the average similarity is higher, given the very specific domain which as mentioned, also means that some words are present in multiple labels, increasing this score. In the Bioinformatics dataset, labels include: 'Computational Biology' and 'Data-Analytics' at level zero, and 'Sequence Analysis', 'Database and Ontology', and 'System Biology' at level one. This dataset contains some labels that are representing two distinct concepts (e.g., 'Database and Ontology'), these labels are less informative when used as we are unsure of which of the two concepts the annotated project belongs to. The outliers show similar characteristics as for the AI dataset.
- **Di Sipio**: their categorization is the most general as it is a subset of the most common GitHub Topics. The topics include application domains like 'Machine Learning', 'Database', and 'Operating System'. Moreover, we find programming languages like 'Python' and 'Java', and also companies and services like 'Google' and 'AWS'. Given the large variety in labels, we also have many that are highly related to others. We start with 'Cryptocurrency' and 'Bitcoin' with similarity 0.84; followed by a list of database-related labels with similarity in the range $0.75 - 0.80$, these are 'PostgreSQL', 'SQL', and 'MySQL', 'NoSQL', 'MongoDB'. And also 'Machine Learning' with 'Deep Learning' having a similarity of 0.77.

A complete list of the statistics and labels for each dataset is available in our data replication package.[6]

2.5. Discussion

We gathered several insights analyzing the results collected in Table 4: first, approximately one in three of the datasets are not publicly available; similarly, the authors have only released the list of categories in one in three of the datasets, which in most cases is a sub-sample of a larger classification. In both those cases, it is hard (if not impossible) to reproduce the steps that lead to the classification: the unclear pre-processing has in fact a direct effect on the performance of the empirical approach (Uysal and Gunal, 2014).

Second, we noticed the variance of the amount of examples and the resulting classifications: from 41 examples to 12K or 150K categories (although the latter are not publicly available), and from 6 to 134 or 228 (again, the latter are unavailable). The higher bound of these stats shows acceptable numbers for both the number of example and the number of different categories.

Furthermore, from the inspection of the categories shows some issues: in particular, they contain some categories that are not relevant to the intended use of the dataset: software application domain classification.

From the observations above, it becomes clear that most existing classifications have fundamental issues that prevent them from being further adopted by other researchers. While creating a new classification, one should not only be able to reproduce the steps performed by other researchers, but also annotate the aspects that might represent common antipatterns for others pursuing a similar goal.

Next, in Section 3, we collect all the practical insights gained from analyzing the datasets and systematically present issues that we found in other classifications.

3. Antipatterns

Using the evidence observed during our case study, and an inductive analysis of the state of the art classification and taxonomies that comprises the information summarized in Table 2, we highlight 7 antipatterns that researchers have faced so far while creating a taxonomy or a classification of software. We also add a discussion to each, and a suggested solutions to reduce the effect of each of these antipatterns.

The analysis was performed with a particular focus on the following characteristics:

- Coverage: each classification has its own domain, that can be more specific or general. With this requirement, we can evaluate if there are missing categories given the domain of the classification, and therefore evaluate its completeness and usability in its domain;
- Cohesion: this also relates to the domain of the classification, however, with this requirement, we try to assess if there are categories that do not belong to the domain;
- *Consistency*: lastly, we check if the classification has any other issue *that affect its consistency* like duplicate or overlapping categories, categories with a bad surface form (e.g. in LeClair et al., 2018 there is 'contrib/math' and 'math'), or any other abnormality.

The results of the analysis is a list of 7 common antipatterns. Below, we present a definition of each, and we will discuss them with instances of how each was observed in past literature and give some suggestions on how they can be fixed from a practical point of view.

MT – Mixed Taxonomies: this issue happens when a label set contains various categories from different taxonomies (e.g., Programming Languages and Application Domains).

MG – Mixed Granularity: this issue emerges when categories are found (or imposed) by researchers to a dataset, although those categories belong to different levels (e.g., family, species, etc.) of a taxonomy.

SC – Single Category: this is a very common issue in software classifications, and it is based on simplifying a complex artifact like a software system, and describing it with only one category.

NE – Non Exhaustive Categories: this issue is visible when the categories chosen (i.e., top-down) or extracted (i.e., bottom-up) by the researchers do not cover the entire spectrum of software categories.

NRC – Non Relevant Categories: this issue is visible when researchers choose or extract a subset of categories that is not representative of the domain of the classification.

UJC – Unnecessarily Joined Categories: this issue occurs when researchers arbitrarily join or use several categories as one, although those are a compound of two or more different domains.

SKC – Sink Category: this is another very common issue in software classification, and it manifests itself when researchers use a generic category for all the software systems that do not fit any of the available categories.

3.1. Mixed taxonomies — MT

Definition: The MT antipattern is defined as a label set consisting of a mixture of two or more different taxonomies, each covering a different domain.

Possible Root Cause: We hypnotize, that one of the causes of this, is the fear of the creators of the dataset to exclude labels, which might make it less appealing to the final users.

Potential Side Effects: This is problematic as the model has to perform two or more tasks at the same time with only a single label per project. Having multiple annotations for the same project is not a problem itself, the issue is having them mutually exclusive as there might be labels diverging from each other.

Concrete Examples: One very common additional part of the classification is based on having programming languages in the labels set. This is common to Sharma et al. (2017), LeClair et al. (2018) and Di Sipio et al. (2020), where, on average, 10% of the examples belong to programming language categories. It is also common to have language specific frameworks or technologies as part of the label set, as for example found in Di Sipio et al. (2020). In some cases, we might find a domain category like 'Deep-Learning', and specific framework or technology labels like 'Django' or 'AWS' that are part of the same classifications.

Potential Solution: A solution to this issue is to define the different classifications independently, and use them as separate tasks when training models. Having projects annotated with different classifications is useful as they can be used as auxiliary tasks to a multi-task learning (Caruana, 1997) model to improve generalizations (Ruder, 2017), and boost performance for all the tasks.

3.2. Mixed granularity — MG

Definition: Having a dataset where some labels are very specific for a field and others are more general, or worse, when labels are in an 'IS-A' relationship, without these relations being explicitly represented.

Possible Root Cause: This issue is caused by the difficulty in creating such relations among all the labels in the categorization.

Potential Side Effects: The former can make the model catch very specific terms, that are dependant on the sample in the dataset, to distinguish between categories. The latter causes overlap between classes, making the classification harder or even impossible when having a single annotation for the projects.

Concrete Examples: As examples of this antipattern in action, we observed that Di Sipio et al. (2020) contains the 'Cryptocurrency' and 'Bitcoin' categories, that have a similarity of 0.84. Similarly, Linares-Vásquez et al. (2014) contains the 'Compilers' and 'Interpreters' categories with a similarity of 0.52. Even in the label sets where we could not detect hierarchical relations among categories (for example in Linares-Vásquez et al., 2014), we also observed more general categories like 'Networking' and 'Web' along with very specific ones like 'Interpreters' and 'Indexing'.

Potential Solution: A solution to this antipattern is to perform a refinement of the categories, and try to aggregate them, in a hierarchical fashion, as we attempted in our case study. The benefits are visible in Fig. 2, we have a lower number of outliers with high similarity. Moreover, a more qualitative analysis to evaluate the extent of the issue, is to assign a position in a taxonomic ranking scale, like the ones used in biology, for each category and evaluate the number of different ranks covered by the used taxonomy.

3.3. Single category — SC

Software systems do not contain only one large feature or functionality, but they are rather composed of many other smaller parts, each with its own specific task. Most of the times software systems are labeled with only one category, that limits the extent to which researchers can learn from it.

Definition: We define the *Single Category* antipattern as the annotation of software with only one category, despite it being a mix of different ones.

Possible Root Cause: This is caused by the annotation process being time consuming and also by not being made by the developers, who will only need to annotate their own project.

Potential Side Effects: Different components of a project, each with its own domain, influence what a model learns, making it harder for it to assign a single category to all components, especially when their semantic contribution to the model is different. This antipattern gets more evident when having a Mixed Granularity (and also a Mixed Taxonomy) classification, where one category is contained in another, however the system is penalized by suggesting the other category.

Concrete Examples: This antipattern was detected in all datasets except for the one of Linares-Vásquez et al. (2014) that performs multi-label classification, and Di Sipio et al. (2020) and Izadi et al. (2020) that performs recommendation of GitHub *Topics*.

Potential Solution: While the solution for this is obvious (e.g., 'annotate each sample with multiple categories'), this is not that easy to achieve as it requires extra effort from researchers and developers during the annotation phase. A less demanding approach to achieve the goal would be to adopt the annotation approach as in the GitHub *Topics*, however those topics are highly noisy as mentioned previously. Therefore, this antipattern requires more attention in future works.

3.4. Non exhaustive categories — NE

Definition: A taxonomy where there are terms that have a common parent, but one of them is lacking, is considered to be suffering of this anti-pattern. For a classification to be usable, it needs to cover the entire range in the domain. This is dependant on the actual domain, and also changes over time, and therefore has to be considered in the domain behind the classification.

Possible Root Cause: There are various possible causes, one is that the missing term was not existent, or very uncommon, at the time that the taxonomy was created (e.g., the Deep Learning was not very common 20 years ago). Another cause is the how the taxonomy was created, if a subsampling of another one, then some terms might have been excluded by the process, if the taxonomy is defined top-down, then the knowledge of the domain by the authors has a big impact on the presence or not of this antipattern.

Potential Side Effects: Having a classification that is too small or with missing relevant categories is an issue as the classification model performance can be affected based on what category is missing, as the category can be easily differentiated if no similar ones are present. Moreover, will make the approach less useful for general uses.

Concrete Examples: Some example from the previous work are the classifications of Kawaguchi et al. (2004), Tian et al. (2009) and Altarawy et al. (2018) that are too small and lack many categories, and in LeClair et al. (2018), where they have 'Interpreters' but not 'Compilers', they are also missing a 'Security'/'Cryptography' related categories. In Di Sipio et al. (2020), a

GitHub Topics based classification, we have 'NLP' but not 'Computer Vision' in spite of it having 134 categories.

Potential Solution: Solutions to this are limited, as mentioned previously, application domains change over time with new categories appearing. A possible solution is to have a coarser granularity, however, this might not be a possibility in some cases, and will also reduce the utility of the classification.

3.5. Non relevant categories — NRC

Definition: This antipattern is based on assigning very fine-grained categories to a project. This means that researchers have in the past added categories to their taxonomies that are too specific and non relevant.

Possible Root Cause: The presence of these categories can be caused by a lack of a specific usage for the taxonomy; or a lack of refinement of categories, when subsampling them from a larger pool.

Potential Side Effects: This antipattern has the effect of making the classification task too simple, since the categories have very few shared terms (or even none) with the others. This can be viewed as a special case of Mixed Taxonomies: however, the categories that would be more relevant to those are usually one or two, and, differently from Mixed Taxonomies, they are not related to specific technologies or programming languages.

Concrete Examples: Examples of non relevant categories were found for instance in Kawaguchi et al. (2004), where categories are very different between each other (e.g., 'Boardgame' and 'Editor'); and in Soll and Vosgerau (2017), where there are a 'Development' category and a 'Homework' one. Another example is from Di Sipio et al. (2020), where we find 'Minecraft' (i.e., a popular videogame) as a category.

Potential Solution: A possible workout for this antipattern would be to just remove these categories and either discard the examples along with it, or try to reassign them to a relevant one that belongs to the domain that is being modeled.

3.6. Unnecessarily joined categories — UJC

Definition: This antipattern manifests itself in categories that join several categories using the "and" conjunction (e.g., 'Gaming and Chat Engines'). While this is a less common antipattern, having a category that is a conjunction of two unrelated categories is something to pay attention to.

Possible Root Cause: One cause for this is the high similarity between the terms, and the low number of examples each of them have, therefore joining them will make the number of examples higher.

Potential Side Effects: The joint categories do not provide as much of an information to the final user as having a single term: which category of the two in the conjunction does the project belong to?.

Concrete Examples: In Sharma et al. (2017) there are many examples of these joined categories: while some might be considered acceptable (for example, 'Data Management and Analysis') others form a weak combination (for example, 'Gaming and Chat Engines' or 'Build and Productivity tools'), since they join labels that belong to very different categories.

Potential Solution: An easy solution for this antipattern would be for researchers to avoid using conjunctions, or to use them only when the categories are related. However, if the categories are indeed related, there should be a more general (single) label to group them under, which is a more appropriate solution.

Table 3
Summary of the antipatterns in previous works.

	MT	MG	SC	NE	NRC	UJC	SKC
MUDABlue (Kawaguchi et al., 2004)	✓	✓	✓	✓	✓		
LACT (Tian et al., 2009)		✓	✓	✓			
Linares-Vásquez et al. (2014)			✓				✓
Borges et al. (2016)		✓	✓	✓			✓
LeClair et al. (2018)	✓	✓	✓				✓
LASCAD (Altarawy et al., 2018)		✓	✓	✓			
Ohashi and Watanobe (2019)			✓	✓			
Sharma et al. (2017)	✓		✓	✓		✓	✓
ClassifyHub (Soll and Vosgerau, 2017)	✓		✓	✓	✓		
HiGitClass (Zhang et al., 2019)	✓		✓			✓	
Di Sipio et al. (2020)	✓	✓		✓			

3.7. Sink category — SKC

Definition:. This is a very common antipattern to fall in, when dealing with large classifications. This antipattern manifests itself with a category, used as a super-label, that is applied to any software that does not fit any another category in the classification, but that still needs an annotation. The most common one is to have a category named 'Others', or other synonyms. However, there are other categories that might not be that obvious, like 'Frameworks', 'Libs' and so on.

Possible root cause:. While the 'Other' category is needed for the classification, its abuse and presence of the other *Sink Categories* is eased by the difficulty of annotating some projects, hence labeling them with a *Sink Category* makes it easier.

Potential side effects:. This sink category adds extra noise, as it might get applied to the majority of projects contained in the pre-existing classification. This category might be also applied to projects that actually belong to other categories, but that were not originally contained in the classification; and also can be used as a backup for harder to classify projects.

Concrete examples:. Examples from previous work include: in LeClair et al. (2018) has three of these which are 'Libs', 'Utils', and 'Misc' which total to 30% of the dataset size; in Borges et al. (2016) they have a category called 'Non-web libraries and Frameworks' containing 25% of their dataset's examples. Lastly, Sharma et al. (2017) has a category 'Others' containing 50% of the dataset examples.

Potential solution:. This antipattern is a harder to avoid, and it was commonly found in our survey: the works that do not suffer from this were usually dealing with small classifications, or very domain-specific ones.

3.8. Summary

In Table 3 we summarize, for each work, the antipatterns they have in their classification. We can notice the least easy to fall in antipatterns are the NRC and UJC, and the most common are NE and SC which are also the hardest to avoid as they require extra work in the annotation phase. The most problematic issues, MT and MG are also quite common, with the former being present in most of the larger and more general taxonomies.

We can also see that there is no perfect taxonomy: if one only considered the amount of antipatterns contained in a dataset, they would select the works of Di Sipio et al. (2020), Zhang et al. (HiGitClass) (Zhang et al., 2019), and Ohashi and Watanobe (2019). However, the latter two have very specific and closed domains, that are more straightforward to create, but less useful to other researchers.

In Section 4, we present an attempt at creating a classification using a new, bottom up taxonomy: we will annotate all the

steps in doing so, and try and address the limitations of existing classifications presented above.

4. Case study

In this section we describe our attempt at creating an *example* classification, *with real world usages*, that minimizes the general issues noted above. This section describes the original source of the classification 4.1, the manual process that was used to reduce the categories in order to balance the number of examples in each 4.2. Finally, in order to evaluate how distinct the categories are from each other, we evaluated the lexical similarity between categories, described by their projects content 4.3.2.

4.1. Classification source

As a starting point (e.g., the 'seed') for the creation of our *case study* dataset, we picked a pre-existing classification, from a *Java Awesome List* hosted on GitHub. Awesome Lists are curated repositories containing resources that are useful for a particular domain: in our case we use *Awesome-Java*,[7] a GitHub project that aggregates overall 700 curated Java frameworks, libraries and software organized in 69 categories. In an initial phase of cleaning, we removed tutorials and URLs to websites, obtaining 530 examples; we also removed the projects that could not be analyzed *(e.g., gives errors in the pipeline, including: encoding, no keywords left to create an encoding, etc.).* The total of projects finally considered for our task was 495.

Using GitHub Topics[8] could be an alternative to the selected Java Awesome List: however, the categories for the same list of projects is larger in the former (around 1000 labels) than in the latter (69 labels). Also, the decision of using the Awesome Java list was to avoid using pre-existing classifications or taxonomies. Beside the previously mentioned issues, other have sporadically emerged in the past (e.g., in LeClair et al., 2018, where many examples of that dataset come from secondary code that is not relevant to the main projects. Moreover, *Awesome-Java* is an annotation of a closed ecosystem (Java development), making it the seed of a small, but realistic, classification. In fact this process, when improved and automated, could be applied to GitHub's *Topics* annotations to obtain an unlimited source of distantly annotated examples. *Lastly, the Awesome-Java repository is a collective effort of more than 300 contributors and continuous updates to the list, making it the go-to source for more than 31K (as stars) developers when looking for a library.*

However, the GitHub Topics, would be a better source for a more general list of categories and a larger scale source of projects. However, there are larger challenges as there are more than 100K GitHub Topics, hence, this will move the focus of our study to unrelated issues.

4.2. Label mapping

The *Awesome-Java* classification contains 69 categories: on average, each category contains 8 projects. Also, some of the categories represent either general concepts ('Science') or detailed keywords (e.g., 'Bean Mapping').

As a result, the Awesome-Java categories make classification tasks quite challenging: therefore we decided to manually reduce the original categories, in order to reduce the complexity, and avoid duplicates or synonyms. This mapping was performed manually, in a hierarchical fashion, by one of the authors, and resulted in a smaller set of 13 categories (*Reduced AJ*): the *Label* column of Table 4 lists the reduced categories that were obtained from the

Table 4
Distribution of the number of examples for each category in the *Reduced AJ*.

Label	Projects
Introspection	32
CLI	8
Data	49
Development	100
Graphical	11
Miscellaneous	59
Networking	25
Parser	41
STEM	39
Security	14
Server	37
Testing	42
Web	38
Total	**495**
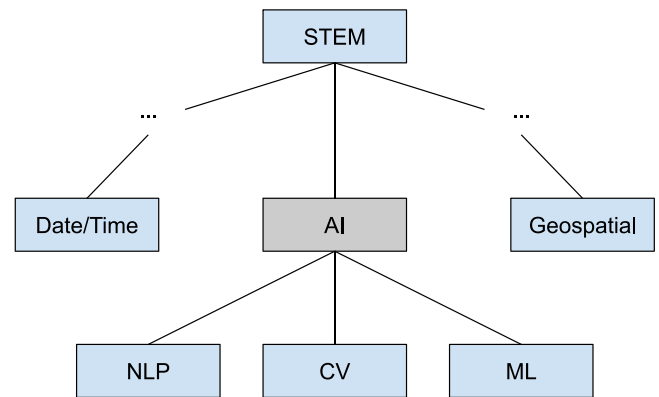
Fig. 3. Example of the reduction process. The blue rectangles are actual classes in the initial or final dataset, the gray one are intermediate logical classes used to aggregate labels. The dots represent not well defined intermediate classes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

original 69. *The reductions were evaluated by the second author, and disagreements on terms were resolved by discussion.*

This reduction, in addition to increasing the amount of examples per class, also helps with one of the issues that the original *Awesome-Java* presented, that is the lack of a hierarchical relation between categories. Fig. 3 shows a visual representation of the reduction, and how this helps the establishment of hierarchical links. Given three labels in the *Awesome-Java* taxonomy: Natural Language Processing ('NLP'), Computer Vision ('CV'), and Machine Learning ('ML'); we reduce those three to an intermediate conceptual label 'AI'. Together with two other categories, 'Date/Time' and 'Geospatial', we assign all these to the 'STEM' category.

The initial and final annotated labels are stored as a CSV file, and is available in our replication package. The file has the following schema:

- **project.name**: name of the project;
- **project.desc**: short description of the project from *Awesome-Java*;
- **project.link**: URL to the GitHub repository;
- **category**: *Awesome-Java* annotation;
- **category.desc**: short description of the category from *Awesome-Java*;
- **label**: mapping of the original category into one of the reduced set.

---

7 https://github.com/akullpp/awesome-java
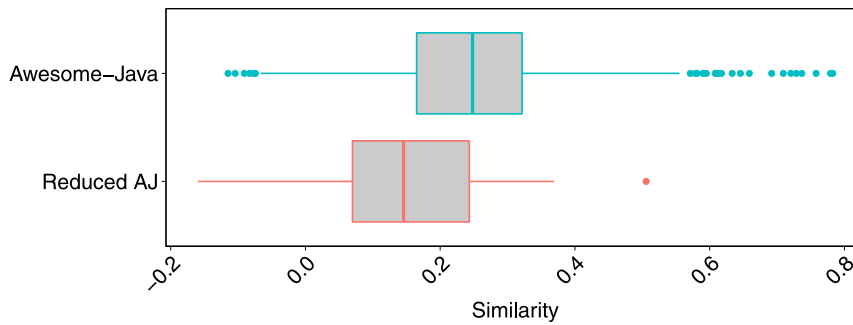8 https://github.com/topics

Fig. 4. Cosine similarity between labels using fastText embeddings.

4.3. Evaluation

We evaluate the quality of our approach of reduction on the *Awesome-Java* taxonomy using both qualitatively and quantitative measure. We first compare the original and the reduced taxonomies using the introduced perils and pitfalls. Furthermore, we measure the lexical similarity of classes.

4.3.1. Antipatterns

A summary of the antipatterns found in the original *Awesome-Java* and the *Reduced AJ* is present in Table 5. The original *Awesome-Java* presents several of the antipatterns identified in the taxonomies in previous works, from non exhaustive label set (NE), to mixed granularity (MG) and mixed taxonomies (MT). Therefore, we can be more confident that the processes we used to reduce these issues can be deployed to different taxonomies as well. Examples of the antipatterns found in *Awesome-Java* include:

- Mixed Taxonomy: examples of this antipatterns are the presence of technologies like 'Apache Commons' in the list;
- Mixed Granularity: for examples we find the label 'Science' with the label 'Configuration', or 'Development' and 'Compiler Compiler'. Moreover, there are labels that are in a 'IS-A' relationship, like 'Mobile development' and 'Development'.
- Non Exhaustive Categories: one examples is the lack of a 'Audio Processing' category, while there are for 'Computer Vision' and 'Natural Language Processing'.
- Sink category: the 'Apache Commons' label contains many projects that can be annotated with another label in the set, for example 'Commons CLI[9]' can be annotated with the 'Command Line Interface' label.

The two main benefits of the reduction process are the removal of the Non Exhaustive (NE) label set issue and the removal of the Mixed Taxonomy (MT). Another benefit, although not completely removed as seen in Table 5, is a marked decrease in the severity of the Mixed Granularity (MG) issue.

In our case study, most of these antipatterns have been resolved using the label reduction process. However better results require tackling Mixed Taxonomy (MT) issue: its resolution required manual annotations of the examples belonging to the problematic category (e.g., 'Apache Commons') as the mere reduction would just map everything to Sink Category.

Similarly for the other datasets, we also computed the similarity of the labels using fastText, in 4 we can see the similarity before and after the reduction. The lower average similarity is caused by a reduction in the terms in a hierarchical relationship, and also by a lower number of terms sharing a common subword.

Table 5
Summary of the antipatterns in the original *Awesome-Java* and our Reduced AJ.

	MT	MG	SC	NE	NRC	UJC	SKC
Awesome-Java	✓	✓	✓	✓			✓
Reduced AJ		✓	✓				✓

4.3.2. Lexical similarity between categories

To evaluate the quality of our process, we evaluated the lexical similarity between each category using the content of all projects belonging to that category. This step is of fundamental importance, since it helps to evaluate the quality of the mapping process into categories, and to give an empirical evaluation of how similar categories are, before and after reduction. In order to lexically represent the categories we used the TFIDF approach; in order to measure the similarity of two categories, we used the cosine similarity. *We did not opted for embedding solutions like fastText or BERT as they are not suited for our task. For example fastText is not designed for long document embeddings, as it performs a mean operation over the embeddings to create the final representation, meaning that all the documents will converge to a very similar embedding, resulting in very high similarities between all documents. With BERT, given the small amount of token it accepts as input (512) we will have a similar issue, as we will need to combine the embeddings of subset. Other, more code oriented solutions, like code2vec or CodeBERT have issue as well. Code2vec is trained with the objective of encoding structure and semantic of the code, and not semantic of the word. CodeBERT suffers of the issues of BERT.*

Extraction of the category documents. For each project belonging to a category, we created the category document using all the *identifiers* contained in the source code files of the project belonging to that category. For the extraction of the identifiers, we used the *tree-sitter*[10] parser generator tool. The identifiers, without keywords, are extracted from the annotated concrete syntax tree created using a grammar for Java code.

The identifiers were further processed by (1) separating the camel case strings into words, (2) lower casing every word, and (3) removing common Java terms that do not add much semantically (e.g., 'main', 'println', etc.). Lastly, we perform (4) lemming, which is a way to reduce amount of different terms in the vocabulary by removing the morphological differences in words with the same root (e.g., '*networking*' becomes '*network*').

Evaluation of the similarity between categories. These category documents were used as an input to TFIDF, a statistical vectorization of words based on the Bag of Words (BoW) model. Documents are considered as a collection of words/terms, and converted to a vector by counting the occurrences of every term. Differently from BoW, in TFIDF the words are weighted by their
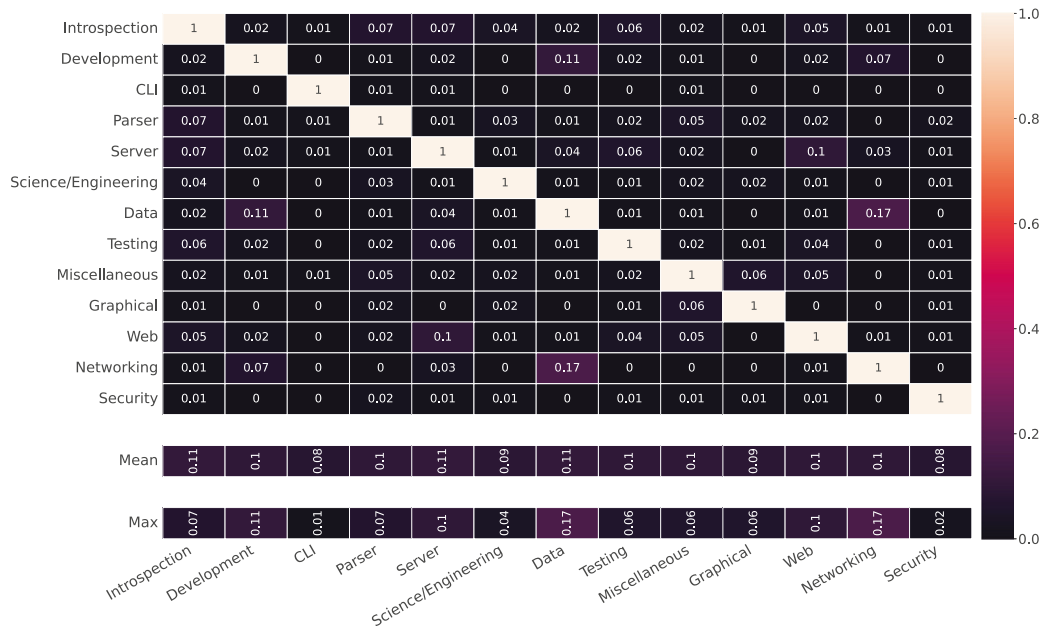

Fig. 5. Cosine similarities between categories in the *Reduced AJ*. The last two rows are the mean and max similarity per category.

total frequency in the collection of documents. This will result in a list of vectors representing the lexical content of that category. We limit the amount to the top 1000 terms that have a max document frequency lower than 0.8, hereby words that are present in less than 80% of the labels, therefore ignoring common words.

We adopted the cosine similarity, a measure of similarity between two vectors, in order to measure the similarity between all categories, and to evaluate possible overlaps or large differences between them. The cosine similarity, when using TFIDF, ranges from 0, different content, to 1 identical content. We compute the similarities between the categories of the original finer grained *Awesome-Java* classification, and the *Reduced AJ* as well.

Results. The results in Fig. 5 show the final similarities for the reduced classification, while Fig. 6 shows the similarities between the categories of the original *Awesome-Java*. The initial impression is that the overall similarity between categories is very low for both classifications: this is a clear effect of the pre-filtering of the terms that are very frequent in all documents. The second observation is that the mean similarity of the classification with 69 labels is higher and has more variance: in particular there is an average similarity of 0.0520 ± 0.0677, as compared to the reduced one of 0.0210 ± 0.0290. This is also visible graphically in the heat map of Fig. 6: the brighter spots, therefore higher similarity are much more frequent there than in the reduced classification in Fig. 5.

Discussion. The higher similarities of Fig. 6 are caused by a combination of different factors:

1. the first cause is the presence *of the Mixed Granularity antipattern* t. For example, the similarity between the categories 'Development' and 'Code Analysis' is 0.45. These two were mapped into the 'Development' combined category of the reduced classification.
2. the second cause of high similarity *is the Single-Label antipattern*. As a result, some projects are labeled with one category, but their features would require multiple labels. An example of this would be the high similarity (0.68) between 'Database' and 'Messaging' which in *Awesome-Java* is described as "*Tools that help send messages between clients to ensure protocol independency*". An example explaining

this high similarity is by considering '*Apache Kafka*', a distributed event streaming framework used for various tasks including data integration, being categorized as 'Messaging' while still containing a high amount of data management terms like '*query*'. This also remains in the reduced classification (Fig. 6) for the 'Database' and 'Networking', in which 'Messaging' is mapped.

3. Lastly, we also have to consider noise, given the smaller number of examples per category in the original classification, the documents used might not be very representative of the category.

4.4. Discussion: moving towards a complete taxonomy

The purpose of a classification, like the ones that we have summarized in Table 2 is to organize similar items (in this case, software systems) into categories for future reference. This could be for proactively recommending systems (Nguyen et al., 2018), in order to generate a list of alternatives from the same category; or for identification of common patterns into the found categories (Capiluppi and Ajienka, 2019).

On the other hand, the purpose of a taxonomy is to organize categories into levels: for instance, in a taxonomy for biology, 'class' (say, "mammals") is placed on a different level than 'order' (say, "carnivorous"). In the classification works that we analyzed, we never observed an attempt to define at which level the categories are (except in Zhang et al., 2019), or whether those should be considered more or less generic or specific in the more general terms of a taxonomy.

As a further analysis, and in order to evaluate how specific, general, or mixed level the taxonomy is, we asked a group of 10 people *belonging to our research group. The pool of annotators is* composed of PhDs, Post Docs, and Professors in the Software Engineering field to indicate whether the categories illustrated in Table 4 should be placed in a higher or lower level of a taxonomy. The questionnaire included *13 questions, one for each topic in the Reduced AJ, where the annotators were asked to perform a rating by assigning the topic into one of* 5 levels, from 1 (very generic) to 5 (very specific). We collected their responses and analyzed them to determine if any of the categories that were reduced from the
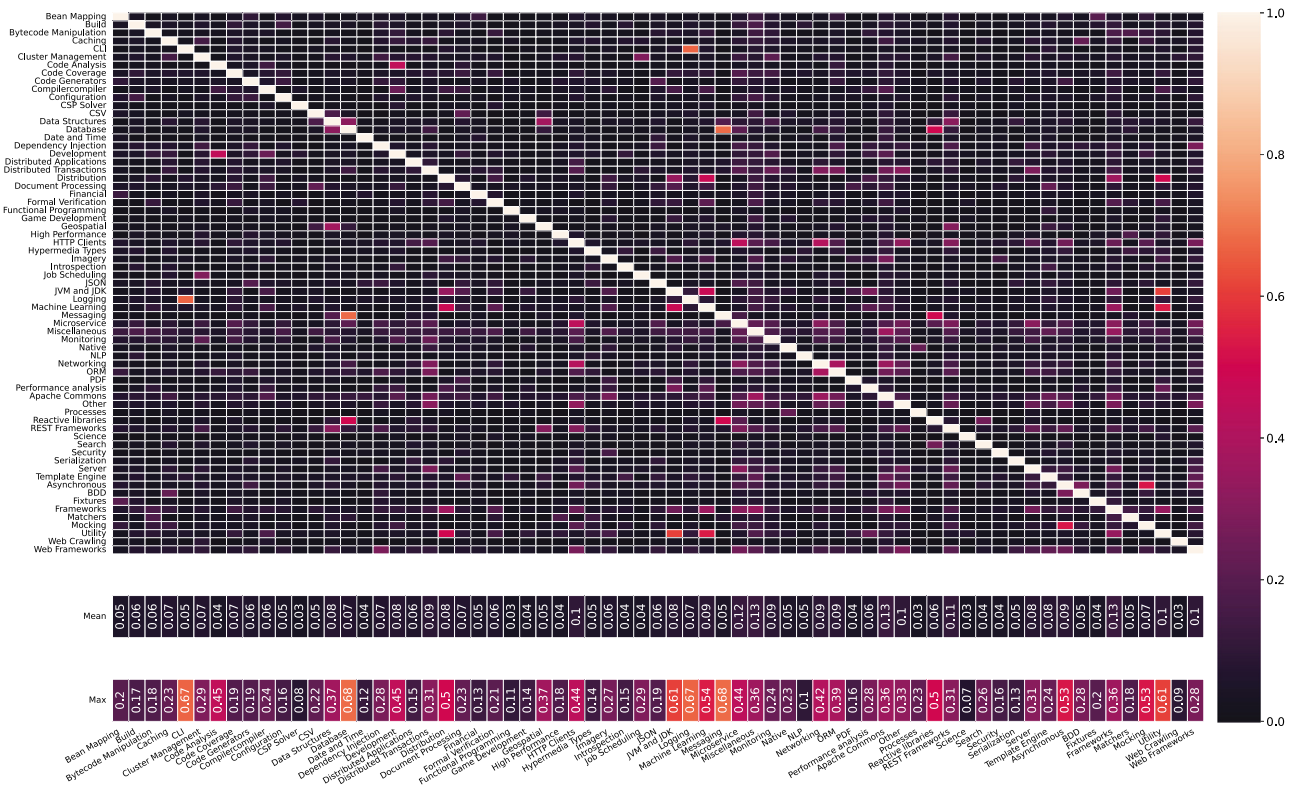
Fig. 6. Cosine similarities between categories in the original *Awesome-Java*. The last two rows are the mean and max similarity per category.

Awesome-Java sample should be considered a 'family', or 'group' or even a 'species' within a software taxonomy.

The results of this preliminary qualitative analysis showed that specific categories were placed fairly consistently among either the very general (e.g., the 'STEM' category), or the very specific level ('Introspection', 'CLI'). On the other hand, several other categories were assigned uniformly to levels 2, 3 and 4, therefore being placed to middle-ground levels of the taxonomy, depending on the assessor's point of view. Fig. 7 shows the visualization of what has initially emerged from the answers of our questionnaire. *The assignment of a topic to a level was performed using the majority voting, those without majority are not presented.*

This is further evidence that defining categories for software systems faces the challenging task of placing them in an over-arching map: the 'mixed levels' antipattern will always affect a classification effort, unless a more concerted research effort is conducted and shared, in order to build a taxonomy and to place the categories in its levels. *A magnifier of the uniform distribution of some topics can be imputed to our methodology, the rating task is more complex compared to a ranking to asses subjective characteristics (Ye and Doermann, 2014). Hence, the future work will focus on using better methods to rank topics.*

5. Threats to validity

We use the classification of Runeson et al. (2012) for analyzing the threats to validity in our work. We will present the *construct validity*, *external validity*, and *reliability*. Internal validity was not considered as we did not examine causal relations (Runeson et al., 2012).

5.1. Construct validity

A construct threat in our work is the choice of the classification for the case study. However, given the wide variety of datasets,
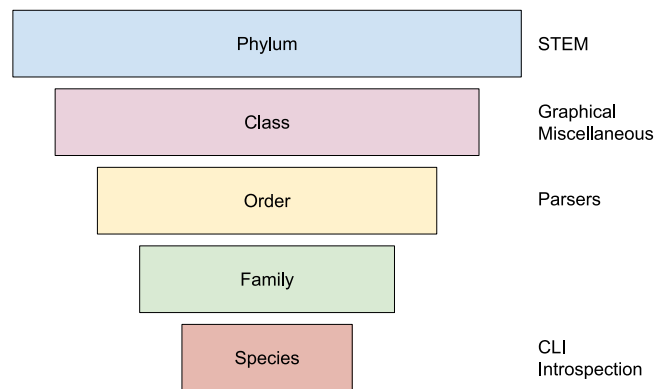

Fig. 7. Assignment of the categories to levels of a taxonomy.

and the similarity Awesome Java has regarding the issues with the state of the art classifications, this threat is mitigated.

Another threat regards the way the reduction was performed. Having a single annotator performing the reduction can increase the bias in the selection of the resulting categories. We mitigated this threat by having another author evaluate the resulting categories; furthermore, we collected feedback from other colleagues regarding the same resulting categories.

5.2. External validity

We reduce the external validity to a minimum by analyzing a large variety of datasets. We analyzed 12 datasets, with a different origin of the base classification: both bottom up, and top down classifications have been considered for study. Moreover, these classifications are based on different domains, some more specific (e.g., bio-engineering) other more generic: this should help alleviate this threat to validity.

5.3. Reliability

The analysis of the classifications and taxonomies is inherently subjective, as it involves natural language, and prior knowledge about the different application domains. We adopted objective tools, like semantic analysis, to aid with the subjective analysis.

6. Conclusions and future work

In this work we evaluated the different classifications used for the software classification task. The current classifications have issues that might compromise generalizability of classification models, moreover, there is no general classification that can be actively used (**RQ1**). We identified a list of 7 common antipatterns that researchers encounter when creating a software classifications for classifying systems into application domains (**RQ2**). While the ideal case would be to avoid those antipatterns when creating a classification, this is quite difficult, and a refinement stage helps with the reduction (but not the complete removal) of some of these issues. We presented a case study, using a real classification, in which we mitigated some of the antipatterns using a reduction of the categories (**RQ3**). The reduction was performed manually in a hierarchical fashion.

As future works we plan to perform the similarity between the categories content also for the other works in the literature. Furthermore, we plan to perform analysis similar to the work of Sen and Saffari (2020) for the Question Answering (QA) task in the Natural Language Processing field, where they look at what clues QA models actually use to answer questions. We are interested in checking if the models learn general terms of a specific domains, or they pick up dataset specific clues that are not transferable to others.

We are also planning to create a taxonomy induced from all the GitHub *Topics* given the variety of projects, and therefore application domains, that are hosted on the platform. Given the large amount of terms, the hierarchical aggregation process needs to be automated. First, we plan to create a ranking with a larger pool of annotators, and given the high disagreement in our ranking case study, use a different methodology: ranking from pairwise comparisons, as is less complex for annotators (Shah et al., 2016). Lastly, use the ranking to create links between levels, to group terms from different levels in the same domain.

CRediT authorship contribution statement

Cezar Sas: Conceptualization, Methodology, Software, Validation, Investigation, Data Curation, Writing – original draft, Visualization. **Andrea Capiluppi:** Conceptualization, Methodology, Validation, Investigation, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

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

Auch, M., Weber, M., Mandl, P., Wolff, C., 2020. Similarity-based analyses on software applications: A systematic literature review. J. Syst. Softw. 168, 110669. http://dx.doi.org/10.1016/j.jss.2020.110669, URL https://www.sciencedirect.com/science/article/pii/S0164121220301278.

Baltrusaitis, T., Ahuja, C., Morency, L.-P., 2019. Multimodal machine learning: A survey and taxonomy. IEEE Trans. Pattern Anal. Mach. Intell. 41 (2), 423–443. http://dx.doi.org/10.1109/TPAMI.2018.2798607.

Bengio, Y., Courville, A., Vincent, P., 2013. Representation learning: A review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. 35 (8), 1798–1828.

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

Borges, H., Hora, A., Valente, M.T., 2016. Understanding the factors that impact the popularity of GitHub repositories. In: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME. pp. 334–344. http://dx.doi.org/10.1109/ICSME.2016.31.

Briand, L., 2012. Embracing the engineering side of software engineering. IEEE Softw. 29 (4), 96.

Briand, L., 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.

Capiluppi, A., Ajienka, N., 2019. The relevance of application domains in empirical findings. In: Adams, B., Constantinou, E., Mens, T., Stewart, K., Robles, G. (Eds.), Proceedings of the 2nd International Workshop on Software Health, SoHeal@ICSE 2019, Montreal, QC, Canada, May 28, 2019. IEEE/ACM, pp. 17–24, URL https://dl.acm.org/citation.cfm?id=3355304.

Capiluppi, A., Ajienka, N., 2020. Towards a dependency-driven taxonomy of software types. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 687–694.

Caruana, R., 1997. Multitask learning. Mach. Learn. 28 (1), 41–75. http://dx.doi.org/10.1023/A:1007379606734.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the ACL: HLT. vol. 1, Association for Computational Linguistics, Minneapolis, Minnesota, pp. 4171–4186.

Di Sipio, C., Rubei, R., Di Ruscio, D., Nguyen, P.T., 2020. A multinomial Naïve Bayesian (MNB) network to automatically recommend topics for GitHub repositories. In: Proceedings of the Evaluation and Assessment in Software Engineering. EASE '20, Association for Computing Machinery, New York, NY, USA, pp. 71–80. http://dx.doi.org/10.1145/3383219.3383227.

Easterbrook, S., Singer, J., Storey, M.D., Damian, D.E., 2008. Selecting empirical methods for software engineering research. In: Shull, F., Singer, J., Sjøberg, D.I.K. (Eds.), Guide to Advanced Empirical Software Engineering. Springer, pp. 285–311. http://dx.doi.org/10.1007/978-1-84800-044-5_11.

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. http://dx.doi.org/10.1145/3196398.3196448.

Escobar-Avila, J., Linares-Vásquez, M., Haiduc, S., 2015. Unsupervised software categorization using bytecode. In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. ICPC '15, IEEE Press, pp. 229–239.

Firth, J.R., 1957. Studies in Linguistic Analysis. In: Publications of the Philological Society, Blackwell, URL https://books.google.nl/books?id=JWktAAAAMAAJ.

Glass, R.L., Vessey, I., 1995. Contemporary application-domain taxonomies. IEEE Softw. 12 (4), 63–76. http://dx.doi.org/10.1109/52.391837.

Gordon, A.D., 1987. A review of hierarchical classification. J. Roy. Statist. Soc. Ser. A 150 (2), 119–137.

Izadi, M., Ganji, S., Heydarnoori, A., 2020. Topic recommendation for software repositories using multi-label classification algorithms. ArXiv, arXiv:2010.09116.

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. http://dx.doi.org/10.1145/775047.775126.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., Germán, D.M., Damian, D.E., 2016. An in-depth study of the promises and perils of mining GitHub. Empir. Softw. Eng. 21 (5), 2035–2071. http://dx.doi.org/10.1007/s10664-015-9393-5.

Kalousis, A., ao Gama, J., Hilario, M., 2004. On data and algorithms: Understanding inductive performance. Mach. Learn. 54 (3), 275–312. http://dx.doi.org/10.1023/B:MACH.0000015882.38031.85.

Kawaguchi, S., Garg, P.K., Matsushita, M., Inoue, K., 2004. MUDABlue: An automatic categorization system for open source repositories. In: 11th Asia-Pacific Software Engineering Conference. pp. 184–193. http://dx.doi.org/10.1109/APSEC.2004.69.

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. http://dx.doi.org/10.1109/ICSME.2018.00056.

Linares-Vásquez, M., 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. http://dx.doi.org/10.1007/s10664-012-9230-z.

Liu, C., Yang, D., Zhang, X., Ray, B., Rahman, M.M., 2018. Recommending GitHub projects for developer onboarding. IEEE Access 6, 52082–52094. http://dx.doi.org/10.1109/ACCESS.2018.2869207.

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. http://dx.doi.org/10.1109/ICSE.2012.6227178.

Mojica, I.J., Adams, B., Nagappan, M., Dienst, S., Berger, T., Hassan, A.E., 2013. A large-scale empirical study on software reuse in mobile apps. IEEE Softw. 31 (2), 78–86.

Nguyen, P.T., Di Rocco, J., Rubei, R., Di Ruscio, D., 2018. CrossSim: Exploiting mutual relationships to detect similar OSS projects. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA. pp. 388–395. http://dx.doi.org/10.1109/SEAA.2018.00069.

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. pp. 194–200. http://dx.doi.org/10.1109/MCSoC.2019.00035.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshynanyk, D., De Lucia, A., 2013. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: 2013 35th International Conference on Software Engineering, ICSE. pp. 522–531. http://dx.doi.org/10.1109/ICSE.2013.6606598.

Ruder, S., 2017. An overview of multi-task learning in deep neural networks. arXiv preprint arXiv:1706.05098.

Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering - Guidelines and Examples. Wiley, URL http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html.

Sen, P., Saffari, A., 2020. What do models learn from question answering datasets? 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. 2429–2438. http://dx.doi.org/10.18653/v1/2020.emnlp-main.190.

Shah, N.B., Balakrishnan, S., Bradley, J.K., Parekh, A., Ramchandran, K., Wainwright, M.J., 2016. Estimation from pairwise comparisons: Sharp minimax bounds with topology dependence. J. Mach. Learn. Res. 17, 58:1–58:47, URL http://jmlr.org/papers/v17/15-189.html.

Shannon, C.E., 1948. A mathematical theory of communication. Bell Syst. Tech. J. 27 (3), 379–423.

Sharma, A., Thung, F., Kochhar, P.S., Sulistya, A., Lo, D., 2017. Cataloging GitHub repositories. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering. EASE'17, Association for Computing Machinery, New York, NY, USA, pp. 314–319. http://dx.doi.org/10.1145/3084226.3084287.

Soll, M., Vosgerau, M., 2017. ClassifyHub: An algorithm to classify GitHub repositories. In: Kern-Isberner, G., Fürnkranz, J., Thimm, M. (Eds.), KI 2017: Advances in Artificial Intelligence. Springer International Publishing, Cham, pp. 373–379.

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. http://dx.doi.org/10.1109/MSR.2019.00014.

Tian, K., Revelle, M., Poshyvanyk, D., 2009. Using latent Dirichlet allocation for automatic categorization of software. In: 2009 6th IEEE International Working Conference on Mining Software Repositories. pp. 163–166. http://dx.doi.org/10.1109/MSR.2009.5069496.

Tsoumakas, G., Katakis, I., 2007. Multi-label classification: An overview. Int. J. Data Wareh. Min. 3, 1–13.

Ugurel, S., Krovetz, R., Giles, C.L., 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. KDD '02, Association for Computing Machinery, New York, NY, USA, pp. 632–638. http://dx.doi.org/10.1145/775047.775141.

Uysal, A.K., Gunal, S., 2014. The impact of preprocessing on text classification. Inf. Process. Manage. 50 (1), 104–112. http://dx.doi.org/10.1016/j.ipm.2013.08.006, URL https://www.sciencedirect.com/science/article/pii/S0306457313000964.

Vargas-Baldrich, S., Linares-Vásquez, M., Poshyvanyk, D., 2015. Automated tagging of software projects using bytecode and dependencies (N). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE. pp. 289–294. http://dx.doi.org/10.1109/ASE.2015.38.

Vásquez, M.L., 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. http://dx.doi.org/10.1109/ICPC.2016.7503721.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., Gall, H.C., 2018. Context is king: The developer perspective on the usage of static analysis tools. In: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018. IEEE Computer Society, pp. 38–49. http://dx.doi.org/10.1109/SANER.2018.8330195.

Wen, F., Nagy, C., Bavota, G., Lanza, M., 2019. A large-scale empirical study on code-comment inconsistencies. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC. IEEE, pp. 53–64.

Ye, P., Doermann, D.S., 2014. Active sampling for subjective image quality assessment. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014. IEEE Computer Society, pp. 4249–4256. http://dx.doi.org/10.1109/CVPR.2014.541.

Zhang, Y., Xu, F.F., Li, S., Meng, Y., Wang, X., Li, Q., Han, J., 2019. HiGitClass: Keyword-driven hierarchical classification of GitHub repositories. In: 2019 IEEE International Conference on Data Mining, ICDM. pp. 876–885. http://dx.doi.org/10.1109/ICDM.2019.00098.

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B., 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE. IEEE, pp. 60–71.

Zhou, C., Sun, C., Liu, Z., Lau, F.C.M., 2015. A C-LSTM neural network for text classification. CoRR, arXiv:1511.08630.

Cezar Sas, M.Sc., is a Ph.D. student at the University of Groningen under the supervision of Prof dr Andrea Capiluppi. His research interest include Natural Language Processing (NLP) and Machine Learning (ML). Currently his work focuses on applying NLP and ML for Software Engineering (SE) problems including Software Classification and Domain Specific Taxonomy Induction. He is part of the SEARCH research group, and the NLPforSE subgroup.

Prof dr **Andrea Capiluppi** works as an Associate Professor in Software Engineering at University of Groningen since 2020. He specializes in Open Source software, effort estimation and software maintenance. He is part of the SEARCH research group, and he is the director of the NLPforSE subgroup within SEARCH. Prof Capiluppi has a strong connection to industry and a sustained collaboration with industrial partners, who act as the beneficiaries of software development for Regional and National impact. A list of the industry collaborators is available at http://www.cs.rug.nl/search/Collaborations/Collaborations.

The NLPforSE (Natural Language Processing for Software Engineering) is a newly formed research sub-group with a focus on machine learning techniques for software engineering. The focus of the NLPforSE sub-group is the application of natural language processing techniques, as well as machine learning and artificial intelligence, to the maintenance and evolutionary aspects of software engineering.