LabelGit

Sas, Cezar; Capiluppi, Andrea

Sharma et al. [2] uses an combined solution of topic modeling and genetic algorithms called LDA-GA [3]. They apply topic modeling 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. They release a list of 10,000 examples annotated by their model (half belonging to 'Others' category), which was evaluated using 400 manually labeled projects.

ClassifyHub [4] uses an ensemble of 8 naïve classifiers, each using different features (e.g. file extensions, README, GitHub metadata and more) to perform the classification task. They use the InformatiCup 2017[3] dataset, which contains 7 categories.

HiGitClass [5] uses approach for modeling the co-occurence of multimodal signal in a repository (e.g. user, name of repository, tags, README and more). They perform the annotation according to a hierarchical taxonomy that is given as an input with keyword for each leaf node. The release a datset with taxonomies for two domains: a machine learning taxonomy with 1600 examples, and a bioinformatics one with 876 projects.

In [6], they use 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. They used 120 popular topics from GitHub, and released a dataset of around 10,000 distantly annotated projects in different programming languages.

[7] uses names, descriptions, READMEs, wiki pages, and file names concatenated together as input to BERT [8], a neural language model, that creates a dense representation of the input text. Then, to this representation, they apply a fully connected neural network to predict multiple labels. Their dataset is no currently available and contains 152K GitHub repositories in various languages classified using 228 topics.

However, all these solutions perform software classification by classifying proxies of the projects (e.g., README), which might not always be available. This problem gets more accentuated if we want to perform classification on smaller parts of

code like components, which require models trained on source code.

This work's contributions are a new dataset, called `LabelGit` [9], containing annotated GitHub projects for software classification. The classification task is performed given an attributed graph with continuous features representing the source code files. We hope this will act as a catalyst for further research on machine learning for software engineering. Moreover, this can also be used to develop and evaluate new deep learning methods for real-world graphs.

II. PRELIMINARIES

In the following section, we introduce the concept of creating document embeddings, a dense vector representation of documents used to capture the semantic and syntactic information carried by the specific data source.

A. Source Code Representations

The learning of a document's embedding is usually performed using learned representations of words first, then combining those to create a document's embedding. We used two different types of features to represent documents into a continuous space (i.e., its embeddings): the resulting embeddings of the source code files reflect two different interpretations. The first one is *semantic*, and based on natural text information. It is created using a neural language model architecture called *fastText*. The second solution, called code2vec, is trained on source code and learns representations by taking advantage of the *structural* information of source code.

a) fastText [10]: is a neural language model that creates word embeddings by learning subword information [10]. *fastText* words are split into *n*-grams, and for each it learns an embedding. The final word embedding is an aggregation of the embeddings of all *n*-grams that the word is made of. An advantage of using subword information like *n*-grams, is that the smaller information level means that there is no risk of out-of-vocabulary words when applying to more technical domains.

b) code2vec [11]: is a neural embedding model specifically designed for source code. The embeddings are learned using abstract syntax trees paths between variables in a method, and the task is self supervised by predicting the method name. The available pretrained model is learned from Java projects, therefore, the variables representations contain domain specific information.

III. DATASOURCES

We created our dataset using two sources, the first is GitHub, while the second is a list of curated GitHub Java projects called *Awesome-Java*.

a) GitHub: is a repository hosting service that provides an ecosystem for storing, development, and sharing of software. It is home to many open source communities and commercial software developers worldwide and contains more than 100 million repositories.

b) Awesome Lists: are curated repositories containing resources that are useful for a particular domain. In our case we use *Awesome-Java*[4], a GiHub project that aggregates several hundreds of curated Java frameworks, libraries and software. It contains 69 categories, and an overall 700 repositories. We removed tutorials, URLs to websites, and projects that failed to be analyzed, and obtained a grand total of 495 projects.

IV. LABELGIT

`LabelGit` [9] is our new dataset for classification of Java repositories that uses source code semantics and dependency graphs. It is based on a curated and annotated list of Java projects. The pipeline used to create the `LabelGit` dataset is represented in Figure 1, and can be divided in the following steps: 1) extraction of the dependency graph; 2) extraction of the features; 3) label remapping; 4) data augmentation.

We made our source code[5] and dataset[6] publicly available. It contains the annotations in CSV format, the zip file for the graphs, and the two other zip files for the features. The dataset size is around 15 GB compressed, and 52 GB when uncompressed.

A. Dependency Graph

The first step of our approach is the extraction of the dependency graph for each project in our sample. Using the Arcan [12] tool, we obtained the complete set of nodes and the edges describing the dependencies between classes. Edge weights were also extracted to describe the number of uses [13] of one class by the other. For example, the *antlr4* project contains a dependency between `ParserATNFactory.java` (from the `org.antlr.v4.automata` package) and `LexerGrammar.java` (from the `org.antlr.v4.tool` package), as the former imports the latter.

The distribution of the number of vertices and edges for the graphs in our dataset are shown in Figure 2.

The dependency graph is stored using *GraphML* (`.graphml`), a XML based graph storage file format. The nodes in the graph contain an attribute with the relative path of the file in the project folder, this is used to identify the source file in the feature files and perform alignment.

The `.graphml` file name allows to easily identify the project and the git commit, and it is structured as follows:

[project name]-[commit number]-[commit sha].graphml

where *[commit number]* is the index of the commit in the version history tree, and *[commit sha]* is the id of the commit.

B. Semantic Features

The extraction of features representing the documents is performed with two different language models: *fastText* and code2vec. The first one was trained with natural text, the second with source code in various programming languages.
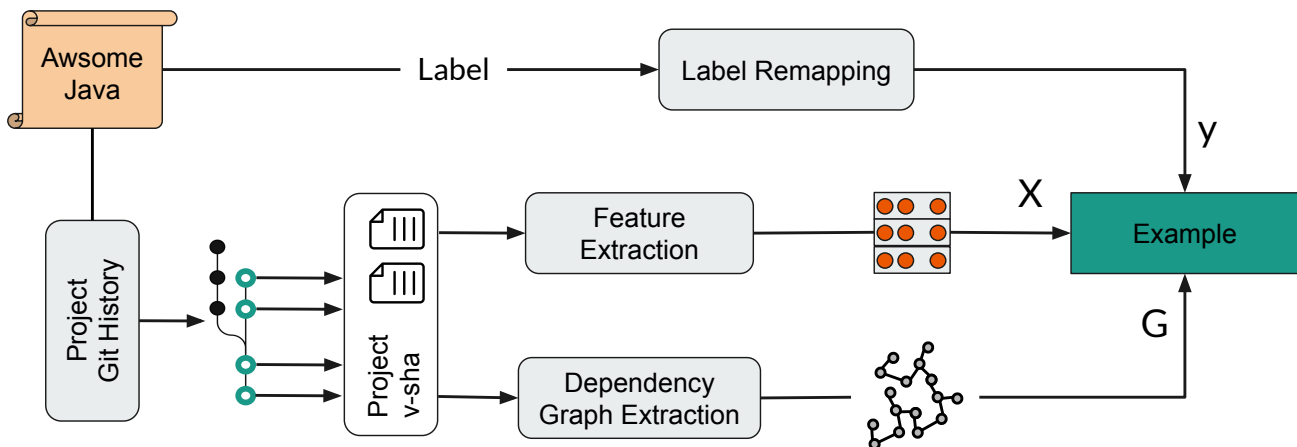

Fig. 1: Our dataset creation pipeline. We start with an annotated list from which the annotations are reduced to a smaller set using a manually created mapping. On the other side, we extract the git project history, and for each commit in the *master* branch (green nodes), we run the dependency graph and feature extraction methods. The final example is made by a graph G, a feature matrix X, and a label y.
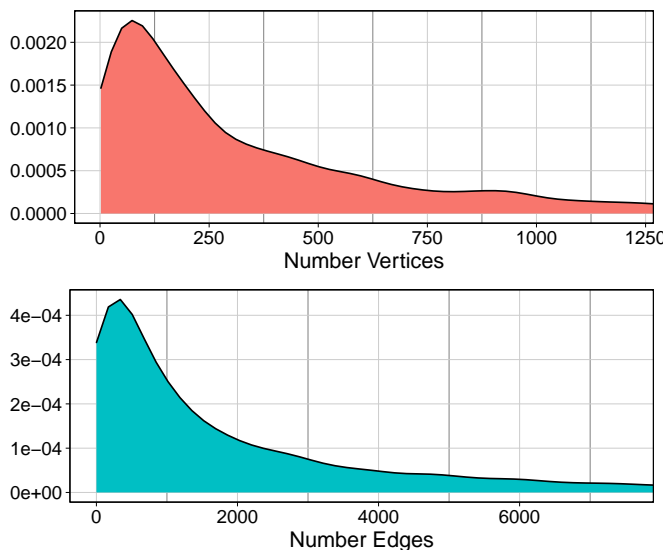

Fig. 2: Distribution of the number of vertices (top) and edges (bottom) in the augmented dataset's graphs.

We apply the two embedding methods on the *identifiers* extracted from the source code file. For the extraction of the identifiers contained in the source code, we used the *tree-sitter*[7] parser generator tool. It makes easy to get the identifiers, without keywords, from the annotated concrete syntax tree created using a grammar for Java code. We clean the identifiers by separating the camel case strings into words, lower case every token, and remove common Java terms that do not add much semantically (e.g., '*main*', '*println*', etc).

The features are stored in a textual file (`.vec`), with each file containing the embeddings for a particular project at a specific commit. The project's source code files are described

---

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

with a row containing the relative path of the file, and the vector of features with each feature separated by a space.

The filename follows the same structure as before, with the only difference in the extension:

[project name]-[commit number]-[commit sha].vec

C. Label Mapping

The *Awesome-Java* list contains 69 annotations (or categories), and on average, each category contains some 8 projects, making any classification task very challenging. Moreover, those categories have different granularity levels and can represent very general concepts or very detailed keywords.

For these reasons, we decided to manually reduce the original categories to a smaller set of 13 (see *Label* column in Table I). This mapping was performed manually, in a hierarchical fashion, by the authors.

The initial and final annotated labels are stored as a CSV file (*annotations.csv*). The file has the followings 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 *Awesome-Java* category into one of the labels of the smaller set.

D. Data Augmentation

Given the small amount of projects contained in the *Awesome-Java* list, and the complexity of the classification task, we decided to perform a 'data augmentation' [14] which has been shown to aid generalization in deep learning models in various domains [15], [16], including software engineering

[17] step. Given our data's nature, we can synthetically generate a large amount of data, while maintaining its original integrity. This is done by making use of the history of a project's repository: instead of taking a single snapshot of it, we take all the snapshots from its *master* branch. Therefore, for each annotated project, we are able to extract multiple different examples that can be used as added 'synthetic' projects in the training phase.

This augmentation approach allowed us to increase the initial 495 samples to 11,502, with an average increase of 23 times the amount of samples per project. In Table I we can see the number of examples before and after the augmentation separated per class.

TABLE I: Distribution of the number of examples before (Projects), and after (Samples) the data augmentation process.

Label	Projects	Samples	Increase (x)
Introspection	32	744	23
CLI	8	142	17
Data	49	1,088	22
Development	100	2,306	23
Graphical	11	226	22
Miscellaneous	59	1,729	20
Networking	25	503	20
Parser	41	935	22
Scientific/Engineering	39	915	23
Security	14	249	18
Server	37	727	20
Testing	42	974	23
Web	38	964	25
Total	**495**	**11,502**	**23**

To evaluate the overlap resulting from the augmentation approach, we measured the graph difference between two consecutive commits of a project. In order to do so, we take the absolute value of the difference of size (sum of the number of vertices and the number of edges) of the two graphs, and normalize by the maximum size of the considered graphs[8].

Formally, given two graphs representing consecutive analyzed versions of a specific project, $\mathbf{G}_i = (V_i, E_i)$, $\mathbf{G}_j = (V_j, E_j)$, their difference is:

$$diff(\mathbf{G}_i, \mathbf{G}_j) = \frac{abs((|V_i| + |E_i|) - (|V_j| + |E_j|))}{max(|V_i| + |E_i|, |V_j| + |E_j|)}$$

where $|\cdot|$ is the cardinality of the set.

The average difference between the graphs of two subsequent commits is around 12%, with a mean interval ($abs(i-j)$) between the analyzed consecutive versions of around 550 commits.

V. USES AND FUTURE WORK

a) Uses: the dataset that we have proposed will aid the development and evaluation of new deep learning models, especially those designed to work on non-euclidean data [18],

---

[8]We chose to evaluate this proxy because computing an exact measure would be extremely demanding, as graph edit distance on large graph requires substantial amounts of memory and time.

such as Graph Neural Networks (GNN) [19]. These models will allow to solve tasks such as classification of software projects, and more in general of real-world graph data. Moreover, by performing the classification, these models can also learn better graph representation: this, in turn, can be used for other downstream tasks, such as the identification of architectural smells [20], and software similarity [21]. Furthermore, the learned representation and the longitudinal nature of our dataset allow us to analyse how software semantics changes with the development and increase in the project's size. Lastly, we can transfer the model's knowledge learned from the coarse project to finer parts of software, like components, to classify them and aid their retrieval for reuse purposes during development.

b) Future Work: as future improvements to the dataset, we plan to extract a bottom-up *taxonomy* from the data and provide it with multi-label classification capabilities. This is because a single project might encompass different characteristics and contain components that belong to different categories. We foresee that this process will be based on GitHub's topics and that those will be reduced using hierarchical clustering, as shown in our process above. Moreover, we plan to expand the sample by increasing the number of projects and examples, allowing for a better generalization of models.

c) Threats to validity: the main threat to our approach is the use of the curated list for both the gathering of the projects, and as the starting point for creating our taxonomy. This choice can be considered arbitrary; however, we use this as a starting point and something to improve on; moreover, the development of various approaches will not be affected by the taxonomy.

Another threat is the data augmentation technique adopted as it might not allow for good generalization as they are still only 495 projects; however, this is reduced by the large difference between the graphs and the fact that various studies showed how data augmentation improves generalization in low data domains.

VI. CONCLUSIONS

In this work, we proposed `LabelGit`, a new dataset for software classification of Java repositories. Compared to previous solutions, ours is designed to be a direct representation of the source code by creating an attributed dependency graph, where each node is represented with semantic features. We expect that `LabelGit` will increase the interest in developing better solutions for software classification, and more in general, for real-world attributed graph classification.

ACKNOWLEDGEMENTS

REFERENCES

[1] S. Vargas-Baldrich, M. Linares-Vásquez, and D. Poshyvanyk, "Automated tagging of software projects using bytecode and dependencies (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 289–294.

[2] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo, "Cataloging github repositories," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 314–319.

[3] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "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)*, 2013, pp. 522–531.

[4] M. Soll and M. Vosgerau, "Classifyhub: An algorithm to classify github repositories," in *KI 2017: Advances in Artificial Intelligence*, G. Kern-Isberner, J. Fürnkranz, and M. Thimm, Eds. Cham: Springer International Publishing, 2017, pp. 373–379.

[5] Y. Zhang, F. F. Xu, S. Li, Y. Meng, X. Wang, Q. Li, and J. Han, "Higitclass: Keyword-driven hierarchical classification of github repositories," in *2019 IEEE International Conference on Data Mining (ICDM)*, Nov 2019, pp. 876–885.

[6] C. Di Sipio, R. Rubei, D. Di Ruscio, and P. T. Nguyen, "A multinomial naïve bayesian (MNB) network to automatically recommend topics for github repositories," in *Proceedings of the Evaluation and Assessment in Software Engineering*, ser. EASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 71–80. [Online]. Available: https://doi.org/10.1145/3383219.3383227

[7] M. Izadi, S. Ganji, and A. Heydarnoori, "Topic recommendation for software repositories using multi-label classification algorithms," *ArXiv*, vol. abs/2010.09116, 2020.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "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, Volume 1*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[9] C. Sas and A. Capiluppi, "LabelGit: A dataset for software repositories classification using attributed dependency graphs," Feb. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4459080

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://www.aclweb.org/anthology/Q17-1010

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290353

[12] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 282–285.

[13] L. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper, "The accuracy of dependency analysis in static architecture compliance checking," *Software: practice and Experience*, vol. 47, no. 2, pp. 273–309, 2017.

[14] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[15] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[16] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017.

[17] Q. Mi, Y. Xiao, Z. Cai, and X. Jia, "The effectiveness of data augmentation in code readability classification," *Information and Software Technology*, vol. 129, p. 106378, 2021. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584920301464

[18] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[19] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[20] I. Pigazzini, "Automatic detection of architectural bad smells through semantic representation of code," in *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ser. ECSA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 59–62.

[21] A. Capiluppi, D. Di Ruscio, J. Di Rocco, P. T. Nguyen, and N. Ajienka, "Detecting java software similarities by using different clustering techniques," *Information and Software Technology*, vol. 122, p. 106279, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058492030029X