DeepIaC: Deep Learning-based Linguistic Anti-Pattern Detection for Infrastructure-as-Code

Nemania Borovits[1], Indika Kumara[1], Parvathy Krishnan[1], Stefano Dalla Palma[2], Dario Di Nucci[2], Fabio Palomba[3], Damian Tamburri[1], Willem-Jan van den Heuvel[2]
[1]Jheronimus Academy of Data Science, Eindhoven University of Technology, The Netherlands
[2]Jheronimus Academy of Data Science, Tilburg University, The Netherlands
[3]University of Salerno, Italy

Abstract

Linguistic anti-patterns are recurring poor practices concerning inconsistencies among the naming, documentation, and implementation of an entity. They impede readability, understandability, and maintainability of source code. In this paper, we attempt to detect linguistic anti-patterns in infrastructure as code (IaC) scripts used to provision and manage computing environments. In particular, we consider inconsistencies between the logic/body of IaC code units and their names. To this end, we propose a novel automated approach that employs word embeddings and deep learning techniques. We build and use the abstract syntax tree of IaC code units to create their code embedments. Our experiments with a dataset systematically extracted from open source repositories show that our approach yields an accuracy between 0.785 and 0.915 in detecting inconsistencies.

CCS Concepts: **• Software and its engineering → Maintaining software**; **• Computer systems organization → Cloud computing**; **• Computing methodologies → Supervised learning by classification**.

Keywords: Infrastructure Code, IaC, Linguistic Anti-patterns, Deep Learning, Word2Vec, Code Embedding, Defects

ACM Reference Format:
Nemania Borovits[1], Indika Kumara[1], Parvathy Krishnan[1], Stefano Dalla Palma[2], Dario Di Nucci[2], Fabio Palomba[3], Damian Tamburri[1], Willem-Jan van den Heuvel[2]. 2020. DeepIaC: Deep Learning-based Linguistic Anti-Pattern Detection for Infrastructure-as-Code. In *2020 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), November 13, 2020, California, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With growing importance for the "need for speed" in the current IT market, the software development cycle is becoming shorter everyday. Development and IT operation teams are increasingly cooperating as DevOps teams, relying massively on automation at both development and operations levels. The software code driving such automation is collectively known as Infrastructure-as-Code (IaC), a model for provisioning and managing a computing environment using the explicit definition of the desired state of the environment in source code and applying software engineering principles, methodologies, and tools [14].

Although IaC is a relatively new research area, it attracted an ever-increasing number of scientific works in recent years [16]. Nevertheless, most research has been done on IaC frameworks, while only a few studies explored the notion of infrastructure code quality. Among others, the first steps in this direction focused on applying the well-known concept of Software Defect Prediction [7] to infrastructure code defining defect prediction models to identify pieces of infrastructure that may be defect-prone and need more inspection. In this perspective, previous works mainly focused on the identification of structural code properties that correlate with defective infrastructure code scripts.

However, this is only one of the possible proxies to identify defective code. Indeed, many problems can be rose by analyzing the plain text of software code. In particular, linguistic anti-patterns, that is, recurring poor practices concerning inconsistencies among the naming, documentation, and implementation of an entity, have shown to be a good proxy for defect prediction [3, 10, 15]. Therefore, while the existing literature mainly focuses on structural characteristics of defective IaC scripts, at the best of our knowledge, none exists that analyze linguistic issues. This motivation led to the research goal of this work:

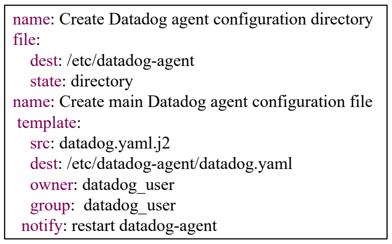
How accurately can we detect linguistic anti-patterns in infrastructure as code (IaC) using a Deep-Learning approach?

Boosted by the emerging trend of deep learning and word embeddings for software code analysis and defect prediction, we propose DeepIaC, a novel approach to detect linguistic anti-patterns in IaC, focusing on name-body inconsistencies in IaC code units. Our experiments on a dataset composed of open source repositories show DeepIaC yields an accuracy between 0.785 and 0.915 in detecting inconsistencies with AUC (Area Under the ROC Curve) metric between 0.779 and 0.914, and MCC (Matthews correlation coefficient) metric between 0.570 to 0.830. We deem our approach can contribute to step the current research up by tackling the problem of IaC Defect Prediction by a different perspective and provide a solid baseline for future studies focusing on linguistic issues.

The remainder of this paper is organized as follows. Section 2 describes IaC, Ansible, and the related works. Section 3 details the approach to identify linguistic anti-patterns with a focus on tasks name-body inconsistencies. Section 4 elaborates on the empirical evaluation of the proposed approach, its results, and limitations. Finally, Section 5 concludes the paper and outlines future works.

2 Background and Related Work

This section introduces Infrastructure-as-Code and the Ansible configuration management technology, and describes previous studies aimed at identifying defects and anti-patterns in infrastructure code.



```
name: Create Datadog agent configuration directory
file:
   dest: /etc/datadog-agent
   state: directory
name: Create main Datadog agent configuration file
 template:
   src: datadog.yaml.j2
   dest: /etc/datadog-agent/datadog.yaml
   owner: datadog_user
   group:  datadog_user
  notify: restart datadog-agent
```

Figure 1. A snippet of an Ansible role, showing two tasks

2.1 Infrastructure as Code and Ansible

Infrastructure-as-Code (IaC) is a model for provisioning and managing a computing environment using the explicit definition of the desired state of the environment in source code and applying software engineering principles, methodologies, and tools via a Domain Specific Language (DSL). IaC DSLs enables defining the environment state as a software program, and IaC tools enable managing the environment based on such programs. In this study, we consider the Ansible IaC language, one of the most popular languages amongst practitioners, according to our previous survey [6].

In Ansible, a *playbook* defines an IT infrastructure automation workflow as a set of ordered *tasks* over one or more *inventories* consisting of managed infrastructure nodes. A *module* represents a unit of code that a task invokes and serves a specific purpose, such as setting up a Datadog agent, creating a MySQL database, or installing an Apache webserver. A *role* can be used to group a cohesive set of tasks and resources that together accomplish a specific goal, such as installing and configuring MySQL. When the tasks are executed, the states of the resources in the target nodes change. To react to such changes, *handlers* can be configured per task using *notify* parameter.

Figure 1 shows an Ansible snippet for configuring a Datadog agent. The two tasks use the Ansible modules *file* and *template* to create a directory to keep the configuration file of Datadog and generate a configuration file from a template. Once the configuration file is created (i.e., a state change), the handler is triggered to ensure that the Datadog agent is restarted to make the new configuration effective.

2.2 Related Work

Most of the previous works describe infrastructure code quality in terms of smelliness [5] and defects-proneness of Chef and Puppet infrastructure components. From a smelliness perspective, Schwarz et al. [20], Spinellis et al. [21], and Rahman et al. [17] applied the well-know concept to IaC, and identified code smells that can be grouped into four groups: (i) *Implementation Configuration* such as complex expressions and deprecated statements; (ii) *Design Configuration* such as broken hierarchies and duplicate blocks; (iii) *Security Smells* such as admin by default and hard-coded secrets; (iv) *General Smells* such as long resources and too many attributes. From a defect prediction perspective, Rahman et al. [19] identified ten source code measures that significantly correlate with defective infrastructure as code scripts such as properties to execute bash and/or batch commands, to manage file permissions, and more.

In this work, we step this line of research up by proposing a novel automated approach that employs code embeddings (vector representation of IaC code) and deep learning techniques to detect linguistic anti-patterns, focusing on name-body inconsistencies in IaC code units. We focus on Ansible, rather than Puppet and Chef, because Ansible is the most used IaC in industry [6].

3 DeepIaC: Deep-Learning-based Linguistic Anti-Pattern Detection for Infrastructure-as-Code

This section presents DeepIaC, our approach to identifying inconsistencies between names and logic/bodies in IaC code units and, in particular, in Ansible. Figure 2 illustrates the workflow of DeepIaC as a set of steps, which can be categorized into the following phases:

- *Corpus Tokenization.* Given a corpus of Ansible tasks, this phase generates token streams for both task names and bodies. To tokenize a body of a task while considering its semantic properties, we build and use its abstract syntax tree (AST).
- *Data Sets Generation.* Since it is challenging to find a sufficient number of real buggy task examples that suffer from inconsistencies, we apply simple code transformations to generate buggy examples from likely correct examples. We perform such transformations on the tokenized data

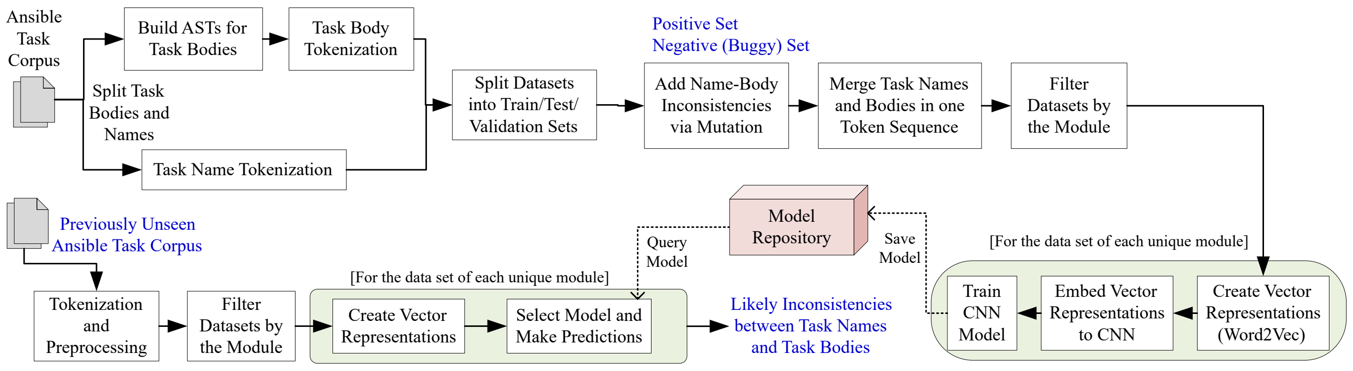

Ansible Task Corpus
Build ASTs for Task Bodies
Task Body Tokenization
Split Task Bodies and Names
Task Name Tokenization
Split Datasets into Train/Test/ Validation Sets
Positive Set
Negative (Buggy) Set
Add Name-Body Inconsistencies via Mutation
Merge Task Names and Bodies in one Token Sequence
Filter Datasets by the Module
Model Repository
Query Model
Save Model
[For the data set of each unique module]
Previously Unseen Ansible Task Corpus
Tokenization and Preprocessing
Filter Datasets by the Module
[For the data set of each unique module]
Create Vector Representations
Select Model and Make Predictions
Likely Inconsistencies between Task Names and Task Bodies
Train CNN Model
Embed Vector Representations to CNN
Create Vector Representations (Word2Vec)

Figure 2. Overview of the DeepIaC approach

set and assume that most tasks in the corpus do not have inconsistencies. Indeed, several previous studies [9, 15] in software defect prediction have successfully applied similar techniques to generate training and test data.

- *From Datasets to Vectors.* We employ Word2Vec [12] to convert the token sequences into distributed vector representations (code embeddings). We train a deep learning model for each Ansible module type as our experiments showed a single model does not perform well, potentially due to low token granularity. Thus, the tokenized data set is divided into subsets per module, and the code embeddings for each subset are separately generated.
- *Model Training.* This phase feeds the code embeddings to a Convolutional Neural Network (CNN) model [11] and train the model to distinguish between the tasks having name-body inconsistencies from correct tasks. The trained model is stored in the model repository.
- *Inconsistency Identification.* The trained models (classifiers) from the model repository are employed to predict whether the name and body of a previously unseen Ansible task are consistent or not. Each task is transformed into its corresponding vector representations, which can be consumed by a classifier.

3.1 Tokenization of Names and Bodies

This step converts the Ansible task descriptions (raw data units) to a stream of tokens, which can be consumed by our deep learning algorithms. The names of the tasks are generally short texts in natural language, and thus we tokenize them by splitting them into words. However, the body of a task has a structured representation. Hence, we use the abstract syntax tree (AST) of the task body to generate the token sequences while preserving the code's semantic information. In the research literature, ASTs are commonly used for representing code snippets as distributed vectors [2, 10].

A task body defines the configuration/instance of an Ansible module as a set of parameters (name-value pairs). It can also specify a conditional (*when*, loop, and notify action to inform other tasks and handlers about the changes to the

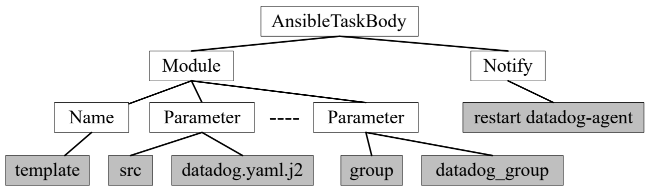


Figure 3. AST model for a task using *template* module

state of a resource managed by a module). We create an AST model that can capture these key information of a task body. To generate the token sequence from the AST, we use the pre-order depth-first traversal algorithm.

Figure 3 shows a snippet of the generate AST model for the task example in fig. 1. AST node types capture the semantic information such as modules and their parameters and notify action, and the raw code tokens capture the raw text values. The token stream generated from the AST will be *[AnsibleTaskBody, Module, Name, template, Parameter, src, datadog.yaml.j2, ..., Notify, restart datadog-agent]*

3.2 Generating Training, Test, and Validation Data

Our linguistic anti-pattern detection is a binary classification task and employs supervised learning. Thus, we need a data set that includes correct (name-body consistent) and buggy (name-body inconsistent) task examples. As the Ansible is a domain-specific language and is relatively new, it is non-trivial to collect a sufficient number of buggy examples from real-world corpus. By inspired by the training data generation in the defect prediction literature [9, 15], we generate the buggy task examples from a given corpus of likely correct task examples by applying simple code transformations.

Before applying code transformations, we divide the tokenized data set into training, test, and validation sets to avoid potential data leakage between three sets during transformations. Within each data set, we swap the body of a given task with the body of another randomly selected task to create inconsistencies. We consider two cases: (i) the tasks using the same module (e.g., two tasks with the *template* module)

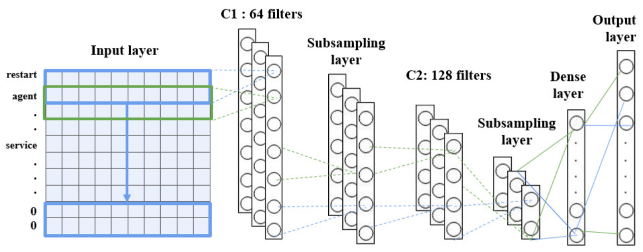


Figure 4. Architecture of CNN model used

and (ii) the tasks using different modules (e.g., one task with the *template* module and another with the *file* module).

3.3 Creating Vector Representations

To feed the token sequences into a learning algorithm, we need to transform them into vector representations. We use the word embedding techniques for the vector representation of the Ansible task names and the corresponding task bodies.

Word embedding techniques take a set of token sequences as inputs and produce a map between string tokens and numerical vectors [12]. They embed tokens into numerical vectors and place semantically similar words in adjacent locations in the vector space. As a result, the semantic information from the input text is preserved in the corresponding vector representation. We use Word2Vec to produce word embeddings. Word2Vec is a two-layer neural network that processes text by creating vector representations from words [12]. The input for the Word2Vec is a sequence of words (tokens), while its output is a set of feature vectors that represent these words. Word2Vec is used by several deep learning-based approaches to software defect prediction [4, 9, 10, 15].

We used the Continuous Bag of Words (CBOW) model of Word2Vec that can predict target words from the surrounding context words. The rationale is that IaC scripts (i.e., task names and bodies) are sequences of tokens. Let us assume to have a sequence of tokens t_1,... t_i ..., t_j) where t_i is a token of an task, DeepIaC considers a window of w tokens around t_i. For predicting the context of the token t_i, the two methods consider $\frac{w}{2}$ tokens before t_i and $\frac{w}{2}$ tokens after t_i.

3.4 Training Predication Models

We use Convolutional Neural Network (CNN) to build our binary classifier that can categorize the tasks into name-body consistent or not. CNNs are biologically-inspired variants of multi-layer artificial neural networks [11]. Although they are widely used in image classification tasks, numerous studies report their success in the domain of NLP [8, 23], and defect prediction of textual source code [1, 4, 10, 15].

Figure 4 shows the architecture of CNNs that our approach uses. The embedded token vectors of Ansible tasks generated by the trained CBOW (Word2Vec) model are used as an input to a CNN. The input for the CNN is a two-dimensional vector which dimension varies since each Ansible module has a different vocabulary. However, each vector representation is long 100 words since we use the same setup for training Word2Vec. We use two convolutional pooling layers to reduce data dimensionality and capture the tasks' local features, similarly to the previous work [10]. L2 regularization is used in each convolutional layer. Furthermore, a dropout layer avoids overfitting, and a dense layer combines the previously captured local features by the convolutional and subsampling layers. The dense layer's output vector predicts and detects inconsistent module use within a task. The output layer consists of neurons per one task of each module. The output neurons are 0 (inconsistent) or 1 (consistent). The measure for the loss function is the Mean Absolute Error (MAE), and the corresponding optimizer is the Stochastic Gradient Descent (SGD).

For the CNN training, we padded the input token sequences to comply with the fixed-width input layer on CNN. Motivated by Wang et al. [24], we appended zero vectors at the end of the token sequences to reach the size of the longest token sequence of the input tasks. To compute the maximum length of the input sequences s we used the equation: $max\_length_s = mean_s + standard\_deviation_s$. To avoid having long sequences with many padded zeros, we decided the max length of the input sequences should be within two standard deviations of the mean [13]. This way, we filtered outliers by reducing noise from the padded zeros, and only the 3% of the input token sequences were affected by this operation.

3.5 Inconsistency Identification

Inconsistency identification is a binary classification task since the test data are labeled in two classes: *consistent* (the negative class in this work) and *inconsistent* (the positive class in this work). Once the binary classifier is trained with a sufficiently large amount of training data, we can query it to predict whether unseen Ansible tasks (e.g., unseen test data sets) have name-body inconsistencies. To evaluate the performance of the trained models, we used the common metrics used in binary classification problems, namely *accuracy*, *precision*, *recall*, *F1 score*, *MCC* (Matthews correlation coefficient), and *AUC* (Area Under the *ROC* (receiver operating characteristic curve) Curve).

3.6 Implementation

To parse Ansible tasks and build ASTs for them, we developed a custom python tool. We tokenized the task names using the NLTK library[1]. We used the Word2vec implementation of the gensim library to generate vectors from tokens. We implemented CNN/deep learning models using TensorFlow and

[1] http://www.nltk.org/

Keras frameworks. We used PyGithub[2] and PyDriller [22] to locate repositories that contain Ansible IaC scripts. The complete prototype implementation of DeepIaC, including data set is available on GitHub[3].

4 Empirical Evaluation

We evaluate DeepLaC by applying it to a real-word corpus of Ansible tasks. We aim to answer the research question: *How effectively does DeepIaC identify inconsistent tasks?*

4.1 Data Collection

We collected the data set from GitHub. To ensure the quality of the data collected, we used the following criteria (adopted from Rahman et al. [18]) when searching for repositories that include Ansible scripts.

- **Criteria-1.** At least 11% of the files belonging to the repository must be IaC scripts.
- **Criteria-2.** The repository has at least 10 contributors.
- **Criteria-3.** The repository must have at least two commits per month.
- **Criteria-4.** The repository is not a clone.

We found 38 GitHub repositories that meet the above criteria. We extracted 18, 286 Ansible tasks from them. As we trained a CNN model per a unique Ansible module, our experiments only considered 10 most used modules, which account for 10, 396 tasks in the collected data set (57% of the data set). As discussed in Section 3, we split each task into its name and body and tokenized both.

4.2 Data Preparation and Model Tuning

We split the tokenized dataset as follows: 60% of the data was used for training, 20% was the test set during the training, and 20% was used for the evaluation of our model. For each data set, we applied the transformations described in Section 3.3 to create the corresponding buggy data set. The filtered token sequences were the input to the Word2vec. We tuned the Word2Vec parameters as: model(CBOW), vector size(100), Learning rate (0.025), Min word frequency(1), window size(6), and epochs(1000). Next, we embedded the vector representations to the CNN classifier. We tuned the parameters of the CNN as: convolution dimension (10), activation layer (ReLu), output layer (softmax), optimization algorithm (sgd), token sequence range (84-99), learning rate (1e-02), pooling type (max pool), and loss function (MAE).

4.3 Effectiveness in Identifying Inconsistencies

Table 1 presents the inconsistency detection results for the top 10 Ansible modules in our data set. Overall, our approach yielded an accuracy ranging from 0.785 to 0.915, AUC metric from 0.779 to 0.914, and MCC metric from 0.570 to 0.830. Our approach achieved the highest performance for detecting inconsistency for the *file* module, where the accuracy was 0.915, the F1 score for the inconsistent class was 0.92, and the F1 score for the consistent class was 0.91. We also observed that the ROC curve, the model loss, and the accuracy plots confirm the model's good performance. Due to the limited space, we do not present the corresponding visualizations, which are in the GitHub repository of this study.

4.4 Threats to Validity

Threats to construct validity. The collected repositories may not be relevant for the problem at hand. We mitigated this threat by applying the criteria used in previous works on IaC to ensure the quality of the collected data. Although the number of repositories may seem low, a small but relevant and representative dataset of active repositories is preferable. Another threat to construct validity concerns the mutation of scripts employed to generate inconsistent cases, which may not represent real-world bugs. Nevertheless, we tried to mitigate this threat by applying the existing approaches that have successfully used mutation for generating the training data [9, 15]. We plan to further mitigate this threat by gathering more real-cases of inconsistent tasks.

Threats to internal validity. The choice of the features used to train the CNN model could influence linguistic anti-patterns detection. We mitigated this threat by training the model using a high number of features (obtained by transforming each task to a vector space of words) extracted from more than ten thousand Ansible tasks. The feature engineering for the classification task depends on the quality of the code base, including naming conventions, use of typos, and abbreviations. This aspect poses a threat to validity, and advanced NLP techniques can be employed to overcome this.

Threats to external validity. The conclusions are derived only from a subset of modules in Ansible (i.e., the ten most used), which might not be reproducible for other modules and languages. However, we used both generic modules (such as *command* modules) as well as more specific modules. Specific modules (e.g., the *copy* module) do focus works, but general modules can execute ad-hoc OS commands. We believe that using a mix of generic and specific modules may mitigate, at least partially, this threat. Finally, we analyzed only Ansible projects, and the results could not generalize to other IaC languages (e.g., Chef, Puppet). Extend our approach to such languages is part of our agenda.

5 Conclusion and Future Work

DeepIaC is an approach to detecting linguistic anti-patterns in IaC scripts by leveraging word embedding and deep learning. In particular, DeepIaC provides automated support to the users to debug inconsistencies in the names and bodies of IaC code units. Our experimental results show that our

[2] https://github.com/PyGithub/PyGithub
[3] https://github.com/SODALITE-EU/defect-prediction/tree/master/linguistic-ap

Table 1. Classification results for the top 10 used Ansible modules

Evaluation Metric/Module		**shell**	**command**	**set_fact**	**template**	**file**	**gather_facts**	**copy**	**service**	**debug**	**fail**
Inconsistent	Precision	0.880	0.790	0.770	0.820	0.900	0.900	0.860	0.870	0.870	0.820
	Recall	0.810	0.840	0.900	0.940	0.940	0.830	0.810	0.760	0.770	0.690
	F1 score	0.843	0.814	0.830	0.876	0.920	0.864	0.834	0.811	0.817	0.749
Consistent	Precision	0.810	0.820	0.890	0.930	0.930	0.905	0.82	0.800	0.750	0.760
	Recall	0.890	0.770	0.750	0.800	0.890	0.770	0.870	0.900	0.860	0.870
	F1 score	0.848	0.794	0.814	0.860	0.910	0.870	0.844	0.847	0.801	0.811
	Accuracy	0.847	0.805	0.819	0.868	0.915	0.817	0.838	0.833	0.809	0.785
	MCC	0.697	0.610	0.649	0.744	0.830	0.685	0.678	0.669	0.625	0.570
	AUC	0.848	0.804	0.822	0.868	0.914	0.848	0.838	0.830	0.814	0.779

approach's performance achieves an accuracy between 0.785 and 0.915 in detecting inconsistencies. We plan to extend the DeepIaC approach to detect name-based bugs [15] and misconfigurations in IaC code scripts. We also aim to apply the DeepIaC to multiple widely used IaC languages.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825480 (SODALITE project).

References

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, Article 40 (Jan. 2019), 29 pages.

[3] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y. Guéhéneuc. 2013. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*. 187–196.

[4] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol. 2018. Keep it simple: Is deep learning good for linguistic smell detection?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 602–611.

[5] Martin Folwer. 1999. Refactoring: Improving the Design of Existing Programs.

[6] Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 580–589.

[7] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2011), 1276–1304.

[8] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1746–1751.

[9] G. Li, H. Liu, J. Jin, and Q. Umer. 2020. Deep Learning Based Identification of Suspicious Return Statements. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*. 480–491.

[10] K. Liu et al. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1–12.

[11] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. 2003. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks* 16, 5-6 (2003), 555–559.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[13] David S Moore, William I Notz, and Michael A Fligner. 2015. *The basic practice of statistics*. Macmillan Higher Education.

[14] Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.".

[15] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[16] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2018. Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research. *arXiv preprint arXiv:1807.04872* (2018).

[17] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering*. 164–175.

[18] Akond Rahman and Laurie Williams. 2018. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 34–45.

[19] Akond Rahman and Laurie Williams. 2019. Source code properties of defective infrastructure as code scripts. *Information and Software Technology* 112 (2019), 148–163.

[20] Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 220–228.

[21] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 189–200.

[22] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 908–911.

[23] Peng Wang et al. 2015. Semantic clustering and convolutional neural network for short text categorization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 352–357.

[24] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308.